

On Actors and Objects - OOP in System Level Design

PHILIPP A. HARTMANN, JOACHIM K. ANLAUF
Computer Science Department, Division II
University of Bonn
{hartman, anlauft}@cs.uni-bonn.de

Abstract

The steadily increasing complexity of embedded systems requires comprehensive methodologies, that support the design process from the highest possible abstraction level. In most of the currently available frameworks, the joint description of hardware and software ends with the partitioning. Within this paper, we want to propose a new view of object-oriented systems, that allows the refinement of such systems from the abstract object-oriented description of the underlying algorithms down to the register transfer level within one single framework. The key contribution of this work is the separation of the representation of behaviour and its usage. We introduce a graph based model for the structural representation of concurrent object-oriented systems, that supports alternative behaviours as well as inheritance, polymorphism, reconfiguration and mobility.

1 Introduction

At the beginning of many system design approaches, the underlying algorithms already exist in a sequential, object-oriented description. There is an increasing demand for tools and methodologies that assist the system designers in the modelling process towards an implementation of a concurrent system based on such algorithms.

During the design process, several problems have to be faced, like refinement, hardware software partitioning, design space exploration, scheduling, simulation, verification, etc.. To provide tool support for these tasks from the highest abstraction levels down to RTL within one environment, a comprehensive methodology is needed. We are currently working on an internal representation to be used in system level design, that addresses these emerging challenges in such a design process.

The main motivation for the development of this new representation model has been the nearly complete distinction between hardware and software in common hardware software codesign flows today, especially in combination with object-oriented modelling. At the highest abstraction levels, hardware and software are usually treated equally, because only the abstract behaviour is studied. During the refinement, system designers or appropriate tools partition the design more or less irreversibly into hardware and software parts, since there is no comprehensive methodology to keep them together during the further development process. Although many codesign, cosimulation, cosynthesis and coverification tools are available today - if repartitioning becomes necessary, it is still a crucial task.

If, on the other hand, the starting point of a design process *is* an existing algorithm implemented in a sequential (object oriented) programming language like C++, there is not much tool support for design space exploration, modelling of resource consumption and automated transformation at early stages of the design process. Even though e.g. SystemC [OSC] allows the designer to include arbitrary C++ code, the benefits of object-orientation and explicitly modelled concurrency are not combinable without language extensions like SystemC-Plus [ODE, GO02] and a certain amount of redesign.

In this paper, we want to discuss the requirements for an internal representation for system-level design, that provides a common framework for the modelling of hardware *and* software. The proposed methodology supports various abstraction levels from sequential object-oriented algorithms down to RTL hardware in a series of small refinement steps. This implies of course the support of object-orientation at all levels of abstraction.

In section 2 we discuss the prerequisites of an object-oriented approach and give a brief overview of related work on the subject of high-level modelling, especially under object-oriented aspects. We reason about the main differences between hardware and software, components and objects, alternatives and inheritance and formulate basic principles, a methodology should adhere to. From this observations we conclude, that a novel approach is possible and reasonable.

Usually, concurrent systems are designed these days as a set of concurrent, active components, sometimes called *actors* [Agh86]. These actors exchange data through a certain communication model like abstract channels etc.. Object-oriented (software) designs are usually characterised by passive *objects*, whose behaviour is invoked externally through a well-defined interface. We propose a structural representation scheme for high-level system design in section 4, which combines both the modelling of interconnected components and the benefits of object-orientation. To achieve a reasonable combination of the above approaches, the representation of a system's component is split into the definition of an external interface, the representation of associated behaviour (alternatives) and the "usage" of a component. This separation is the key contribution of our work.

Prior to that, we introduce the basic communication concept of our approach in section 3. We concentrate on the aspects of communication that affect the structural modelling more or less directly. The representation of concrete behaviour, resource consumption and constraints, the definition of a reasonable set of primitive data types and related operations goes beyond the scope of this paper and will be published separately.

In section 5, we discuss the consequences of our approach on system level design and refinement. In addition, code and behaviour reuse, reconfiguration and mobility (the continuation of a computation at another location) are taken into account as well. We conclude the paper in section 6 and outline the directions of further research.

2 Prerequisites and related work

In this section, we want to provide a brief overview of existing specification models and languages for system level design and codesign. Beyond the "classical" models like CDFG's, State charts, Petri-Nets, FSM's (see for instance [Je99, Pan02]) and established hardware description languages like VHDL and Verilog, many other models and specification languages have been developed. There are many efforts underway to reduce the design gap in the design process of complex systems, even declarative programming is used in system design these days (e.g. Confluence [Haw04]). Since we want to propose an object-oriented approach, that supports alternative behaviours and reconfigurable systems, we want to focus on these aspects of system modelling.

Actors vs. Objects The majority of existing approaches can be called *actor-oriented* [LNW03]. This widely-accepted concept composes a (concurrent) system as a set of communicating components. Since the proposition of this approach by G. Agha [Agh86] in the mid eighties, many different communication frameworks have been developed to interconnect such components. The channel concept of for example SystemC [OSC], SpecC [STOC] and even signals from VHDL and Verilog are examples from hardware modelling. Within the PTOLEMY project [Le01], a complex framework covering the advantages of heterogeneous, interacting actors has been developed.

On the other hand, *object-orientation* is a widespread concept of modelling complex software systems. The main object-oriented aspects are data encapsulation and inheritance. In the field of parallel *software* development, a plethora of object-oriented languages and models have been developed. A survey on the different ways to include parallelism into object-oriented languages or object-orientation into parallel languages is given in [Phi00]. When modelling concurrent systems, the concepts of actors is usually already included in object-oriented approaches, since communication becomes an important issue.

In sequential software development, "objects" are passive components of a system, that expose an interface of "methods" to the outer world. Their state can only be changed through invocation of these methods, which results in an encapsulation of data and the related transformations. This is a major

difference to component based systems, where data tokens are often modelled to “flow” through the system and are modified on their way.

Hardware vs. software While object orientation is commonly used within the development of software, several approaches to apply these concepts to hardware were proposed. In addition, these models focus on the synthesis of object-oriented hardware descriptions (like [Rad00] as an extension to VHDL). Although the emerging industry-standard SystemC [OSC] is based on a C++ library, true support for object-oriented concepts had to be developed as language extensions like SystemC-Plus [GO02].

The difficulty to combine software *and* hardware lies in the implicit assumption, that components can access their attached behaviour at any time. In object-oriented software, this is achieved through function pointers, which means, that basically an infinite number of objects can share the same code within the memory. Contrary to that, when modelling “hardware objects”, the associated behaviour is assigned implicitly to every object again. Within the ODETTE project, the solution to this problem (for synthesis) lies in the separation of “data objects”, where the behaviour is synthesised at the current location of the contained data, and global “static objects”, which are accessible through a handshaking protocol (see e.g. [GO02]). This requires again a quite irreversible decision, which objects are what at early design stages. Therefore, we want to propose a combination of explicitly modelling “objects”, that can be refined to hardware and software within the same framework. As a result, the following paradigm is given:

Paradigm 1 *The usage of behaviour is to be modelled explicitly.*

This means, that an object is to be assigned to a concrete “location”, which holds the behaviour (and the “memory”) to provide the access to the object. We call this “location” an *instance* of a certain type and distinguish between an *object* (see 4.4), and an *instance* (4.3). Since in the software world, infinite numbers of objects can be bound (5.2) to one single instance (the program code) and in hardware usually one object maps one instance, we introduce the following:

Paradigm 2 *Every instance has a parameter $\text{size} \in \{1, \dots, \infty\}$, which denotes the number of different objects, that can be bound concurrently to this instance.*

As a result, the resolution of references (see 5.3) becomes easy and controllable during the refinement process. To the authors’ knowledge, these two paradigms or similar assumptions have not been applied to existing object-oriented hardware/software codesign frameworks, like ODETTE [ODE], BALBOA [DOSG02], ODYSSEY [GHM03] or OASE [SKKR01], yet. For details, please refer to the according references.

Behaviour alternatives vs. inheritance The modelling of behaviour alternatives is an important tool during the refinement of a system, since the definition of multiple alternative behaviours for a module leads to more flexibility and expressiveness. Therefore, common hardware description languages like VHDL have built-in support for this. On the representation side for example, an approach based on a hierarchical graph-model is introduced in [HTRE02]. On the other hand, sub-typing, generalisation and inheritance help to cope with the complexity of systems. But the modelling of alternatives can be achieved through inheritance as well (see e.g. [TAB03]), where different behaviours are expressed through inheritance and overriding of specific methods. The key to combine these two approaches lies in the following paradigm:

Paradigm 3 *The states/values of the members of a module are sufficient to represent the state of an object, regardless of its chosen implementation variant.*

Then alternative behaviours preserve the *type* of an object, and inheritance should be used mainly to specify additional behaviour (i.e. a different *type*).

Reconfiguration and mobility (Dynamic) reconfigurable systems are still a hot topic in recent research activities. Adhering to the above paradigms results in immediate support of the representation of reconfigurability, if the creation of instances is allowed during run-time. Mobility, i.e. the continuation of a computation at another location, is covered by dynamic binding, as described in section 5.2. [MNC⁺03] shall be mentioned as a prominent approach to model and synthesise dynamic reconfigurable systems.

We want to present a model, which satisfies the above described requirements in a simple way and allows us to mix them. Other important issues for the development of a comprehensive internal representation like the modelling of intra-object concurrency, resource consumption and constraints will be omitted here.

3 Communication

Modelling of communication between different parts of a concurrent system is one of the most important aspects during the design process. In this section, we want to outline the communication framework within our proposed methodology. Since the modelling of communication is not in the main scope of this paper, we want to introduce only those parts that are more or less connected to the structural representation.

During a system’s design, the modelling of the communication between the different components of the system is modelled at various abstraction levels. Prominent communication models are, message passing, discrete events, channels and signals, shared memory and others. Of course, when talking about (even object-oriented) software, synchronous remote procedure, method and function calls are to be mentioned as well. As with the structural representation, we have been looking for a way to support these (high level) concepts within a simple framework.

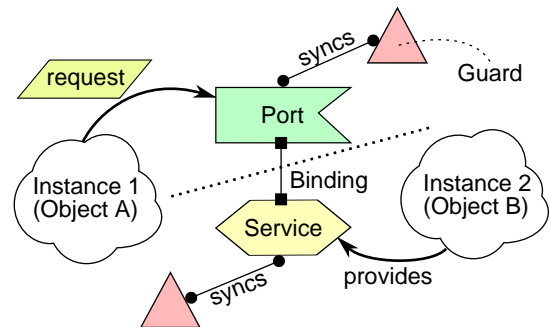


Fig. 1: Simplified service/request scheme

3.1 Service/request scheme

Our solution to this requirement is based on *service/request* pairs, comparable to object request broker architectures, like CORBA [OMG]: Objects are requesting *services* of other objects. But instead of talking to a centralised broker every time, which is quite far from hardware synthesis, we introduce additional *ports*, to model the structural connection of different components with our paradigm 1 in mind. The aspects of the communication mechanism is shown in Figure 1: if an object A wants to access a service *do()* of an object B, it issues a *request* via a *port*. There can be synchronisation constraints bound to the *port* and the *service*.

Services Every module (the basic parts of a system, see 4.1) defines a set of *services*. These services are the passive part of the component. The related behaviour is started through an external *request*, if the synchronisation conditions hold (see 3.2). Arguments of the services are directed (*in,out,inout*) and passed by value *or* by reference. For the modelling of references, see section 5.3.

Requests are started synchronously or asynchronously from within behavioural sections. These requests could be blocked due to the synchronisation conditions associated to this request. Requests can even fail, if the port, to which the request is bound, does not allow this request, no instance is currently bound to this port etc.. Some of these situations are discussed in section 5. We differentiate between two types of requests: Firstly, a request where the *instance* of an *object* is known, and secondly when we need to access a service of an object without knowledge of its current location. In the first case, the request can be directly sent to the instance and thus allows structural interconnection during design time. The latter concept is truly a “request” by the means of CORBA [OMG] and is needed to provide support

for higher levels of abstraction and the sophisticated modelling of references (see 5.3).

Ports are used to encapsulate requests and provide an additional layer between the requests (which are issued somewhere in the represented behaviour of a module) and the target of the request. These ports basically consist of a subset of *services* of a certain *type* (see 4.1), and describe a minimal set of required services. This allows the separation of the connectivity aspects of different components within the system from the behavioural representation. The behavioural model simply talks to a local port without caring of the actually connected component. Firstly, these ports can be bound to static *instances* during “compile-time” and provide therefore a way to model the (physical) interconnection of different instances. Secondly, these bindings are allowed to change during run-time, which might be necessary for the modelling of software and dynamically reconfigurable systems (see 5.2).

The other common communication mechanisms are representable within this framework. Remote procedure calls, method and function calls and message passing are supported out of the box, since these concepts are closely related to this concept. Channels are representable simply by defining own components with a certain interface (see section 4). For simple channels like signals, FIFO’s etc., we are working on reference implementations.

3.2 Synchronisation

The modelling of concurrent systems always needs to consider the synchronisation of the distributed computations. Constraints, when certain computations are allowed, e.g. the mutual exclusive access to shared resources or explicit serialization, are to be modelled at every phase of the design process.

Additionally, if we try to benefit from an object-oriented framework in a distributed world, we have to face a problem called “inheritance anomaly”. This phenomenon denotes the necessity of re-implementation of inherited behaviour due to changes within the synchronisation constraints. For instance Crnogorac et al. have classified these anomalies in [CRR98]. In [CRR97], they have shown that these anomalies are not completely avoidable. But the effort of re-implementation can be mitigated, if inheritance of computation and synchronisation code is possible independently of each other.

Therefore, we have adopted the commonly used concept of “method guards” within our framework to reduce the inheritance anomaly and model synchronisation issues separately from computational tasks. A *guard* consists of a *pre-condition*, a *pre-action* and a *post-action*. Since we are using the abstraction of *ports* around every *request* (i.e. method call), we extend this concept slightly: We do not only allow these guards to be applied to behaviour, but to ports as well (see 4.2).

The access to the guarded item is blocked, until the *pre-condition* evaluates to true. *pre-action* and *post-action* are executed atomically before, respectively after the access is performed. If multiple guards are assigned to a resource, they are all evaluated during the *request* of a *service*. The combination of multiple guards is basically a conjunction of the *pre-conditions*, and the sequential computation of the *pre-* and *post-actions*. There is a unique ordering implied (port guards on the request side are wrapped around guards on the service side, additionally see 4.3), this combined synchronisation is well-defined.

4 Structure

Our proposed representation of a system’s structure can be divided into four main parts: *Modules*, respectively *types*, *implementations*, *instances* and *objects*. These parts shall be described in this section.

4.1 Module and type

Every part of a represented system within our framework has an associated *module declaration* (see definition 4), that defines the visible interface of a system’s component. This declaration is performed incrementally, to provide sub-typing and inheritance.

Definition 4 A module declaration M is defined as a tuple $M = (T, \mathfrak{S}, N, P, S)$ with

- a set T of parameters,

- a set \mathfrak{S} of module declarations/types,
- and a set N of members.
- a set P of ports,
- a set S of service prototypes

Parameters The set of parameters $T =: \text{parameters}(M)$ of a module M are those configurable options, that do not change during runtime, or more precisely during the existence of an instance (see 4.3). They are introduced to increase the reusability of system descriptions, for example abstract signals which can contain arbitrary data tokens.

The concept is closely related to template parameters known from C++ or generics known from VHDL. Therefore, they are not inherited, but have to be included within the declaration every time. We call the combination of a module with values for its parameters a concrete *type*. Since we do not want to discuss the handling of the parameters and especially their influence on the inheritance mechanism here, we will use *module* and *type* as synonyms in the following.

Inheritance relationships between modules, i.e. the types, that are (directly) extended by a module M , are denoted through a set of super-modules $\mathfrak{S} =: \text{super}(M)$. To avoid inconsistencies like circular inheritance, it is necessary to specify, which relationships are considered as valid.

For two modules M, N from a set of modules \mathfrak{M} , we write $N \vdash M$, iff $N \in \text{super}(M)$. This defines a relation $\vdash \subseteq \mathfrak{M} \times \mathfrak{M}$. \vdash^* shall denote the reflexive, transitive closure of \vdash . For the set of (direct) sub-types of a module M , we write $\text{sub}(M) := \{N \in \mathfrak{M} : M \vdash N\}$. Similarly we define $\text{super}^*(M)$ and $\text{sub}^*(M)$ as the canonical extensions with respect to \vdash^* and $\text{types}^*(M) := \text{super}^*(M) \cup \text{sub}^*(M)$. Obviously, \vdash^* defines a preorder on \mathfrak{M} .

Definition 5 On a set of modules \mathfrak{M} , a relation $\vdash^* \subseteq \mathfrak{M} \times \mathfrak{M}$ defines a inheritance hierarchy, iff (\mathfrak{M}, \vdash^*) is a partial ordered set.

Members The Members in $N =: \text{members}(M)$ are module declarations, which are used to specify the persistent state space of the module's *objects* (see 4.4). As we will see in section 4.2, additional members can be defined, which are needed for the execution of a specific behaviour alternative. These members are not part of the *module declaration*, matching paradigm 3.


Additionally, N contains only those members, that are added with the declaration of M . The complete set of members that are used to describe an object's state are to be extracted from the inheritance hierarchy.


Services and ports The sets $P =: \text{ports}(M)$ and $S =: \text{services}(M)$ denote sets of (re-)declared *ports* and *service prototypes* of a module M . Similarly to the declaration of members, the complete set of *ports* and *services* declared for M are to be extracted from the inheritance hierarchy as well. If a port or a service is (re-)declared in different super-modules, it has to be redeclared within M , to provide a deterministic choice. Since these declarations do not contain any behaviour yet, this is no limitation to code reuse (see 4.2).


4.2 Implementation graph \mathfrak{I}


There is a layered, directed *implementation graph* \mathfrak{I} associated with every module M , which defines the modules' alternative behaviour implementations and inner structures as well as the related inheritance hierarchy. Strictly speaking, one single graph is associated with every disjoint set $\text{types}^*(M)$, since the inheritance hierarchy is represented through layers in this graph. Prior to giving a more formal definition of this concept, we want to motivate this idea and introduce the different node and edge types, that are used.


Nodes These aspects are modelled within our graph via different types of nodes and edges In the following, let M be a module and \mathfrak{I} the corresponding implementation graph. The images at the beginning of the next paragraphs show the corresponding graphical representation of the described nodes and edges.


 For each member of M , a corresponding *member node* exists in the graph. If a specific *implementation* (see below) requires additional variables, these are represented as nodes of this type as well. We call the set of member nodes $members(\mathfrak{I})$.

 Every port $p \in ports(M)$ has a corresponding *port node* $p \in ports(\mathfrak{I})$. For every member node $m \in members(\mathfrak{I})$, the implementation graph contains at least one port node, too. This is used to model the fact, that the access to members is always taking place via requests to ports.

 Similarly, every service prototype $s \in services(M)$ is represented as a *service node*. It should be mentioned, that for the overriding of services and ports multiple nodes exist in the graph. $services(\mathfrak{I})$ denotes the set of nodes of this type.

 The implementations of a service correspond to *implementation nodes*. Implementations contain the possible behaviour of a service prototype. We write $implementations(\mathfrak{I})$ to denote the set of all implementation nodes of \mathfrak{I} .

 In section 3.2, we introduced the concept of synchronisation through guards. *guard nodes* $g \in guards(\mathfrak{I})$ represent the available guards in the implementation graph.

 Last but not least, empty nodes are introduced to increase the capabilities to model alternatives. We combine these nodes to a set $alt(\mathfrak{I})$.

Edges The edges in the implementation graph are of different types, too. There are five types of directed edges to describe the relationships between the nodes.

\longrightarrow A *provides* edge is used to represent the capability of a node to provide the semantics of another node. This could be for instance the connection of an *implementation node* with a *service node*.

\blacklozenge Dependencies are modelled through *depends* edges. An edge $(v, w) =: d \in depends(\mathfrak{I})$ denotes, that the abstract semantics of v require the presence of w .

\longleftrightarrow *conflicts* edges are introduced as well. These are the opposite of the depends edges, since they mean for an edge $(v, w) =: c \in conflicts(\mathfrak{I})$, that the semantics of v and w are mutually exclusive. Although there are already restrictions, which nodes can be allocated within one instance (see 4.3), these edges are a way to model conflicts, that are implied by internal properties of the represented behaviour.

\rightsquigarrow As stated before, inheritance is modelled through different layers within the implementation graph. If a port or a service is overridden by a sub-module, there is an *overrides* edges between the corresponding nodes. These edges $o \in overrides(\mathfrak{I})$ are necessary for both inheritance and polymorphism.

$\bullet\text{---}\bullet$ A guard node $g \in guards(\mathfrak{M})$ can be connected through *sync edges* (all of which are combined within $syncs(\mathfrak{I})$) with nodes of arbitrary other types. This means, that the access to this note is “guarded” by g . Only one guard per node is allowed. Multiple guards can be introduced through the addition of appropriate, guarded empty nodes.

Every node is associated to exactly one module M . Therefore, the implementation graph \mathfrak{I} corresponding to the modules within $types^*(M)$ can be divided into *local implementation graphs*, denoted as $\mathfrak{I}(M)$. These local implementation graphs consist only of those nodes that correspond to the module M and the edges between them. We do not distinguish between $members(M)$, $ports(M)$, $services(M)$ and the related subsets of the nodes in \mathfrak{I} . For example, we write $members(M) \subseteq members(super^*(M)) \subseteq members(\mathfrak{I})$. With this notation, we can define the implementation graph as follows:

Definition 6 Given a set of modules \mathfrak{M} with an inheritance hierarchy \vdash^* and $M \in \mathfrak{M}$, the implementation graph of $types^*(M)$ is a directed, layered graph $\mathfrak{I} := (V, E)$, with the following properties:

(i) The external interface of every module M has a corresponding node in \mathfrak{I}

- $\bigcup_{M \in types^*(M)} members(M) \subseteq members(\mathfrak{I})$

- $\bigcup_{M \in types^*(M)} ports(M) \subseteq ports(\mathfrak{I})$

- $\bigcup_{M \in types^*(M)} services(M) \subseteq services(\mathfrak{I})$

(ii) All (even internal) member nodes have an associated port node:

- $\forall m \in members(\mathfrak{I})$ with $m \leftrightarrow M : \exists p \in ports(\mathfrak{I}) : (m, p) \in provides(\mathfrak{I}) \wedge p \leftrightarrow M$

(iii) All, except provides edges are directed “upwards” only:

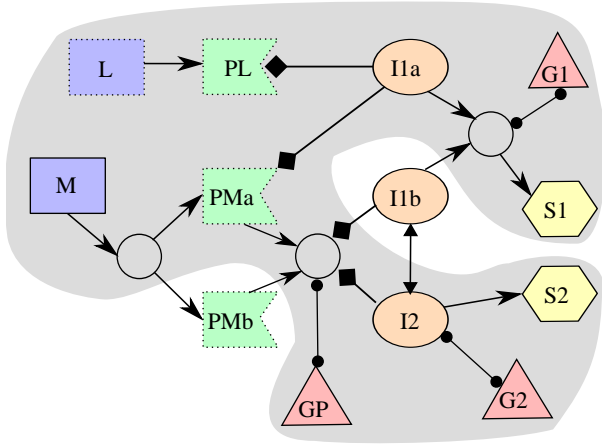


Fig. 2: An implementation graph, with a valid ...

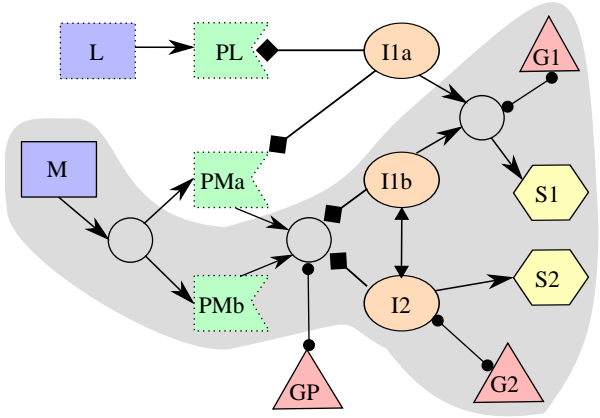


Fig. 3: ... and an invalid subgraph

- $\forall (v, w) \in (V \setminus \text{provides}(\mathfrak{S}))$ with $v \hookrightarrow M_1, w \hookrightarrow M_2 : M_2 \in \text{super}^*(M_1)$
- (iv) At most one syncs edge ends at every node:
 - $\forall (v, w), (v', w) \in \text{syncs}(\mathfrak{S}) : v = v'$
- (v) All overridden services and ports are connected, duplicates are resolved
 - $\forall M_S, M_T \in \text{super}(M), \forall s \in \text{services}(M_S) \cap \text{services}(M_T) : s \in \text{services}(M)$
 - $\forall M_S, M_T \in \text{super}(M), \forall s \in \text{ports}(M_S) \cap \text{ports}(M_T) : s \in \text{ports}(M)$
 - $\forall M_S \in \text{super}^*(M) : \forall s \in \text{services}(M) \cap \text{services}(M_S) :$
 $\exists v, w \in V, v \hookrightarrow M, w \hookrightarrow M_S : (v, w) \in \text{overrides}(\mathfrak{S})$
 - $\forall M_S \in \text{super}^*(M) : \forall p \in \text{ports}(M) \cap \text{ports}(M_S) :$
 $\exists v, w \in V, v \hookrightarrow M, w \hookrightarrow M_S : (v, w) \in \text{overrides}(\mathfrak{S})$

The formal treatment of additional restrictions like the consistent connection of *alternative* vertices shall be omitted here in favour of simplicity. A complete formal semantic is in preparation. But it should be noted, that *provides* edges are always directed (via empty nodes) either

- from implementation nodes *towards* service nodes,
- from member nodes *towards* port nodes,
- from port nodes *towards* empty nodes, on which implementation nodes depend.

As mentioned above, the handling of parameterised inheritance is not discussed within this publication, too. An example of an implementation graph is shown in the figures 4 and 5 and discussed in section 5.

4.3 Instance

Instances belong always to a certain *type* (M, t) , with t is a valid allocation of parameter values. Again, M shall denote a module and \mathfrak{S} the corresponding implementation graph. The main property of an instance is the concrete allocation of behaviour, which is selected as (basically the nodes of) a subgraph of \mathfrak{S} . Figures 2 and 3 show simple examples of a (single-layered) implementation graph with marked subgraphs.

We want to skip the formal definition of a *valid subgraph* here, and describe the requirements roughly. Most important is the fact, that *all* (even inherited) member nodes have to be included, but only *some* of the service and (external) port nodes need to be present. This allows fine-grained control of the allocated behaviour and corresponds to the possibility of allocating ports, that require only some of the possible services of a module. Of course, dependencies have to be fulfilled, conflicts must not occur, alternatives must be chosen deterministically and for every overridden port or service in $\text{super}(M)$, the corresponding behaviour has to be respected. Lastly, all guards, attached to nodes in the subgraph are forced to be present. If no nodes, except the (always required) member nodes are allocated, we call this the *empty*

instance of M , which is used, if only the storage of an *object* is needed. The creation of instances (even during runtime), the binding of *objects* to them and the extension of *instances* to *polymorphic instances* are discussed in section 5.

According to paradigm 2, we want to model an instance's ability to hold a specified number of objects at a time. Therefore, we introduce an optional parameter *size* for every instance, which does *not* affect the type of the instance.

4.4 Object

Lastly, the data, respectively the state in a temporal distribution is called an *object* of a certain *type*. At a given time, the object is solely determined by the state of the associated module's members, if we disregard running processes for the moment. The resulting state space $states(M)$ of M can be described in combination with the inherited members:

$$states(M) := states(N) \bigcup_{M_S \in super^*(M)} states(M_S)$$

An object can be bound to different instances at different times, but not simultaneously. The binding of objects to instances is discussed in section 5.2.

5 Consequences

After outlining the basic concepts of our framework, some fundamental considerations are presented and the usability of this model in system level design is discussed. To provide an illustration for the following remarks, we start this section with a simple example.

Example 7 Let $C = (\emptyset, \emptyset, N_C, P_C, S_S)$ be the module declaration of a simple counter with

- $N_C := \{val\}$, a member which holds the current value,
- $S_C := \{c()\}$, a service which increments val by 1, every time, it is invoked,
- $P_C := \{tick, out\}$, two public ports (for an implementation controlled through signals),
- and the (local) implementation graph shown in figure 4.

Additionally, $L = (\emptyset, \{C\}, N_L, P_L, S_L)$ shall be a “lockable counter”, which extends C through

- $N_L := \{is_l\}$, a member which stores the “locked” state,
- $S_L := \{c(), lock()\}$, the overridden service $c() \in P_C$ and an additional service $lock()$, which shall toggle is_l ,
- $P_L := P_C$, since both ports need to be overridden,
- and the implementation graph shown in figure 5. The inherited layer is marked (and not fully drawn).

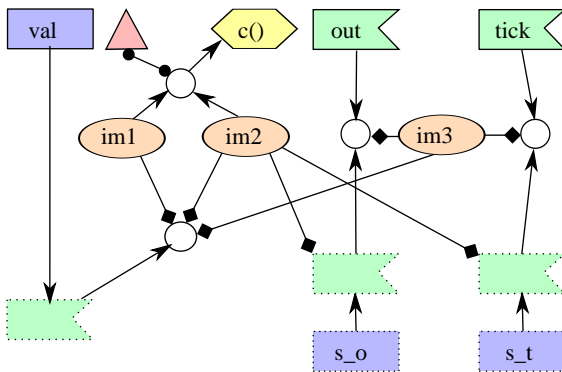


Fig. 4: Implementation graph \mathfrak{S} for the counter ...

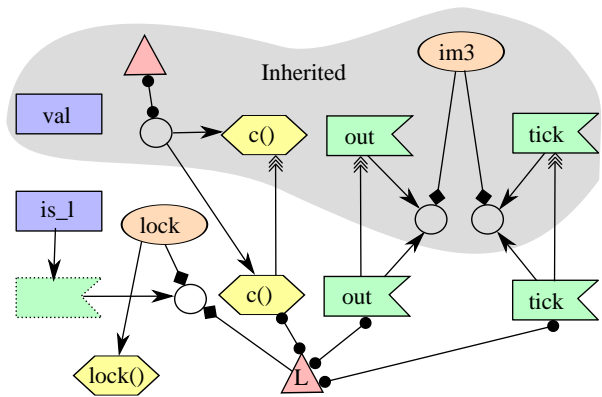


Fig. 5: ... and for the lockable counter in example 7

Note the different implementation nodes for the service $c()$, especially $im2$ and $im3$. It can be seen, that a refinement of $c()$ towards a signal controlled interface can be achieved through the addition of an appropriate behaviour $im3$ and alternative connections of it to newly introduced either the introduced external ports $tick$ and out or internally via signals s_t and s_o .

5.1 Construction and Initialisation

The explicit separation of behaviour and its usage requires the separation of *construction* of instances, where behaviour is chosen and “created“, and the *initialisation* of objects, where initial states are determined and concurrent processes are started.

During the *construction* of an instance I , we may need to allocate required resources and create instances for the contained members. The ports of the contained members might be bound statically at this time, too. This leads to some kind of “physical interconnection” of behaviour. The *initialisation* creates an *object* within this instance (maybe at one of the multiple slots). This is closely related to the binding of existing objects to an instance, since concurrent behaviour is to be started here, too.

5.2 Binding, mobility and polymorphism

The separation into *instances* and *objects* requires, that an object is associated to an instance to access it’s properties. This binding is modelled via specially marked services in an instance, that have been omitted so far. If no special $bind()$ service is defined for an instance, the binding of an object simply stores the values of the members and no further actions are invoked. Otherwise, within this service, this storage and arbitrary other actions can be modelled explicitly, even the modification of the instance itself is allowed. Mainly, this is necessary for the starting or resuming of concurrent processes within the instance. For the reverse operation, the $unbind()$ service can be defined accordingly.

Since *guards* can be used to restrict the use of services, even the delayed binding or unbinding can be modelled. Further details on the semantics of this concept will be published separately.

The binding of objects of a certain subtype of the instance’s type is allowed as well. The semantics are similar to the concept of “tagged types” in Ada95 [Eng97], which means that additional members are simply discarded and services, that might have been overridden are of course replaced by those services, that are available.

The binding and detachment of objects to instances containing running processes requires some additional information. Within our framework, *suspension points* can be included within the explicit modelling of behaviour. This means, that an $unbind()$ request can be blocked, until all processes are either stopped or reach one of those suspension points. According to paradigm 3, the object’s state is fully determined by the state of the members, preventing information loss.

Mobility and Reconfiguration A common situation, when run-time binding is unavoidable, is parameter passing through service requests. If these parameters are passed “by value”, they are not connected to an instance, yet. Therefore, there has to be a local instance (allocating the locally required services) to $bind()$ them to, prior to access their contents. For the “data objects”, as they are introduced in SystemC-Plus [GO02], a similar approach was chosen. This aspect is called mobility, since objects are moving from one instance to another.

Within the methodology, there is no restriction, which objects are allowed to change their instance. Usually, types that are using this feature are indeed “simple” types without an active process of their own. Additionally, the modification of instances during run-time is not restricted per se, as well. This allows the abstract modelling of dynamically reconfigurable systems.

Polymorphism The aspects of polymorphism are not covered by an instance, so far. To support the “dynamic” choice of an overridden service or port, we need to introduce *polymorphic instances*, which are usual instances with an associated subset $s \subseteq sub^*(M)$. If an instance belongs to type M . For these sub-types, those allocated services in the instance that are overridden, have to be included. These can be found by moving along the *overrides* edges within the implementation graph \mathfrak{S} . These ports and services

imply the allocation of certain other nodes, according to their dependencies. Additionally, all newly defined member nodes have to be allocated to allow the storage of a sub-typed object in this instance.

Afterwards, any object of a type within ε can be bound to this polymorphic instance. The corresponding behaviour can then be chosen statically during the binding. Therefore, during all of the following accesses to this object, there is no further need to determine the correct behaviour within the instance. This is one of the major advantages of the proposed framework. The selection of a reasonable subset $\varepsilon \subseteq \text{sub}^*(M)$ on the other hand, is to be determined automatically by tools or even manually by the designer. A formal discussion of this approach will be published separately.

5.3 Pointers and References

The modelling of references is necessary within a framework that claims to cover the aspects of object-oriented software. In general, references or pointers are considered to be not synthesisable. Within our framework, this is not entirely true. If there are accesses to objects, whose instance is known, the resolution of the access is easily modelled. Consider an instance of the counter module from example 7, which can hold n objects. We can refer all of those n counter objects dynamically, if we know their current position within the instance, which clearly is some kind of a pointer. Even some kind of pointer arithmetic becomes synthesisable, since the storage of multiple objects within one instance can be seen as a local memory.

To raise the abstraction level of representable systems, we do not want to limit the concept of references to the above mentioned case. It has been reasoned in [SBFNJ01], that the service/request level by the means of CORBA is a valuable level of communication abstraction. Within our model this means the access of an object's services, whose instance is currently not known. We need to consider two different situations: (i) service parameters "by reference" and (ii) reference members. (i) means, a parameter of a service is already bound to an instance, which we did not know in advance. In case (ii), members of an object do not actually contain data, but only a reference to an object (whose instance can change dynamically!). In those cases, we need another type of request, which resolves the current instance of an object implicitly. Possible ways to refine this locally cannot be introduced here.

6 Conclusion and further work

In this paper, we presented a new framework for the modelling of concurrent systems at various abstraction levels. The main advantage of the presented approach is the combination of object-oriented techniques with common component based frameworks. This combination is mainly achieved through the explicit representation of the usage of a certain behaviour.

Since it is possible and necessary to model the allocation of behaviour instances and binding of abstract data objects to it, the proposed methodology assists during the refinement of a system's structure starting from extensively object-oriented descriptions. While the concepts of object-orientation was introduced to increase the level of "code reuse" during the design of complex systems through inheritance and encapsulation, our framework focuses on the reuse of "behaviour" throughout all phases of the design process. This is important for tasks like high-level scheduling, design-space evaluation and partitioning.

We are currently working on the formal semantics, including the handling of polymorphism and detailed explanation how to map existing hardware and software representations into our framework.

Since the structural modelling is only one part of the representation of concurrent systems, we are continuing our work on a complete internal representation for high-level system design. This includes the development of an XML schema and an API for the inclusion of other tools as well as the development of a simulation environment.

Additionally, we are planning the development of translators from and to our internal representation. Aspects of synthesis approaches as they have been proposed by [ODE], [GHM03], or [SKKR01], will be considered as possible entry and exit points of our representation.

References

- [Agh86] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, 1986.
- [CRR97] L. Crnogorac, A. S. Rao, and K. Ramamohanarao. Inheritance Anomaly, A Formal Treatment. In *Proceedings of the 2nd International IFIP Workshop on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*. Canterbury, 1997.
- [CRR98] L. Crnogorac, A. S. Rao, and K. Ramamohanarao. Classifying Inheritance Mechanisms in Concurrent Object Oriented Programming. In *ECOOP'98*, vol. 1445 of *Lecture Notes In Computer Science*, pages 571–601. Springer-Verlag, 1998.
- [DOSG02] F. Doucet, M. Otsuka, S. Shukla, and R. Gupta. An Environment for Dynamic Component Composition for Efficient Co-Design. In *Proceedings of DATE'02*, March 2002.
- [Eng97] J. English. *Ada 95: The Craft of Object-Oriented Programming*. Prentice Hall, 1997.
- [GHM03] M. Goudarzi, S. Hessabe, and A. Mycroft. Object-oriented ASIP Design and Synthesis. In *Forum on Specification and Design Languages (FDL'03)*. Frankfurt, Germany, 2003.
- [GO02] E. Grimpe, and F. Oppenheimer. Aspects of Object Oriented Hardware Modelling With SystemC-Plus. In *System on Chip Design Languages. Extended papers: Best of FDL'01 and HDLCon'01.*, pages 213–223. Kluwer Academic Publ., 2002.
- [Haw04] Tom Hawkins. Confluence. <http://www.confluent.org>, 2004.
- [HTRE02] C. Haubelt, J. Teich, K. Richter, and R. Ernst. System Design for Flexibility. In *Proceedings of DATE'02*, pages 854–861. Paris, France, March 2002.
- [Je99] A.A. Jerraya et.al. Multilanguage Specification for System Design and Codesign. In *System-level Synthesis*, A.A. Jerraya and J. Mermet, Eds. Kluwer Academic Publishers, 1999.
- [Le01] E. A. Lee et.al. Overview of the PTOLEMY project. Tech. Rep. UCB/ERL M01/11, University of California, Berkeley, March 2001. <http://ptolemy.eecs.berkeley.edu/>.
- [LNW03] E. A. Lee, S. Neuendorffer, and M. J. Wirthlin. Actor-oriented Design of Embedded Hardware and Software Systems. *Journal of Circuits, Systems and Computers*, 12(3):231–260, 2003.
- [MNC⁺03] J-Y. Mignolet, V. Nollet, P. Coene, D. Verkest, S. Vernalde, and R. Lauwereins. Infrastructure for Design and Management of Relocatable Tasks in a Heterogeneous Reconfigurable System-on-Chip. In *Proceedings of DATE'03*, 2003.
- [ODE] Project ODETTE. <http://odette.offis.de>.
- [OMG] Object Mgmt. Group. Common Object Request Broker Architecture. <http://www.omg.org>.
- [OSC] The Open SystemC Initiative. SystemC. <http://www.systemc.org>.
- [Pan02] I. Panagopoulos. Models, Specification Languages and their Interrelationship for System Level Design. Iptool techreport, George Washington University, 2002.
- [Phi00] M. Philippsen. A Survey of Concurrent Object-oriented Languages. *Concurrency: Practice and Experience*, 12(10):917–980, 2000.
- [Rad00] M. Radetzki. *Synthesis of Digital Circuits from Object-Oriented Specifications*. PhD thesis, Carl von Ossietzky Universität, Oldenburg, 3 April 2000.
- [SBFNJ01] K. Svarstad, N. Ben-Fredj, G. Nicolescu, and A. A. Jerraya. A Higher Level System Communication Model for Object-Oriented Specification and Design of Embedded Systems. In *Proceedings of the 2001 Conference on Asia South Pacific Design Automation*, pages 69–77, 2001.
- [SKKR01] C. Schulz-Key, T. Kuhn, and W. C. Rosenstiel. A Framework for System-Level Partitioning of Object-Oriented Specifications. In *Workshop on Synthesis and System Integration of Mixed Technologies (SASIMI'2001)*. Nara, Japan, 2001.
- [STOC] The SpecC Technology Open Consortium. <http://www.specc.org>.
- [TAB03] A. Tsikhanovich, E.M. Aboulhamid, and G. Bois. Object-Oriented Techniques in Hardware Modeling using SystemC. In *Proceedings of the Northeast Workshop on Circuits and Systems*. Montreal, Canada, June 2003.