The OCoN approach for object-oriented distributed software systems modeling

Holger Giese* and Guido Wirtz†

*Software Engineering Group, Computer Science, University of Paderborn, Warburger Str. 100, 33098 Paderborn, Germany. Email: hg@upb.de
†Distributed Systems Group, Computer Science, Westfälische Wilhelms-Universität Münster, Einsteinstraße 62, 48149 Münster, Germany
Email: guidow@math.uni-muenster.de

There are many significant problems related to engineering distributed software systems that feature both control and data processing aspects. Besides software complexity, in general we also have to tackle issues of concurrency and distribution. A set of well-evolved formalisms particularly w.r.t. concurrency indeed exists, although the integration of such into a common software engineering framework is still lacking. Related attempts have often not achieved the desired level of acceptance. A fast growing market for distributed software does, however, effectuate a shift towards high-level behavior modeling. The OCoN approach as presented in this article provides such high-level behavior modeling as an extension to the UML de facto standard for object-oriented modeling by integrating an adjusted Petri net formalism with the software engineering reality.

Keywords: object-oriented, distributed software, systems modeling

1. INTRODUCTION

In the sixties, Carl Adam Petri developed the Petri net model as an extension of the state machine concept. These nets have been further generalized to those of place/transition nets [6]; the notion being one possible choice for behavior modeling in the context of structured analysis. There is a fairly long tradition of suggestions that propose the concept of Petri nets as a suitable tool for software engineering [32]. Unfortunately, the extent of the expressiveness of basic Petri net formalism is insufficient to handle real modeling problems and thus several extensions to high-level Petri nets (HLPN) [13, 6] have been proposed. As is common for formal methods and software engineering practice, however, there exists a trade-off between expressiveness and efficiently testable system properties. Although place/transition nets are generally recognized in software engineering [31], the situation is quite different for HLPN. Alongside the development of object-oriented analysis and design [34, 10], there has been a shift towards more high-level design views. In this arena, behavioral formalisms such as statecharts [19] gain more recognition than those formalisms which are based upon Petri nets.

Both logical and historical reasons serve to explain this state of practice. Petri nets are a conceptional extension of automata and thus they are inherently more complex than state machines. The adequate handling of concurrency is furthermore a complex problem. It is frequently the case that solutions applying database technology employing parallel access transparency in order to avoid related problems are more appropriate for building systems. The development of complex software system engineering incorporating sophisticated concurrency aspects had therefore been less influential and thus rendering suitable concepts for object-oriented concurrent behavior modeling outside of main stream technologies. To a great extent, the development of object-oriented methods and notations (including UML [30]) in fact neglects systems which contain elements of concurrency.

Compositional languages and modularity have been identified as essential in the successful design of software sys-
tem. Not only classical Petri net formalism but also initial approaches towards high-level concepts Most extensions instead emphasize the preservation of an analyzable model, while in practice it has emerged that clear semantics and support for embedding as based upon abstractions such as interfaces have become more important. Meyer [27, p. 979] summarizes the common critiques as follows:

Petri nets, in particular, rely on graphical descriptions of the transitions. Although intuitive for simple hardware devices, such techniques quickly yield a combinatorial explosion in the number of states and transitions, and make it hard to work hierarchically (specifying subsystems independently, then recursively embedding their specifications in those of bigger systems). So they do not seem applicable to large, evolutionary software systems.

In general, behavior modeling is in practice rather neglected, but several newer trends such as the shift towards software architecture [36] may alter this situation. Nowadays, software is evolving from isolated solutions for business or industrial applications towards distributed environments. Software will become increasingly pervasive and interlink information system structures which are currently regarded as being rather isolated. The software will often assume responsibility for considerable coordination tasks; indeed the demand for the improvement of business processes and process-centered technologies such as workflow [35] implies that this trend may render system behavior an essential aspect of design. The ever increasing complexity of current systems and their related requirements will boost the transition from programming towards high-level design notations for even behavioral aspects. In specific domains with emphasis upon behavior, such development has even begun already (cf. SDL [22]).

In cases in which behavioral modeling is required to provide some degree of modularity, the notions of abstraction and data hiding become mandatory. While several extensions to the classical interface notion to achieve a more suitable external specification have been suggested, the question of behavior in the context of subtyping and inheritance remains an active area of research. The phenomenon of non-uniform service availability for a class has to be considered for interfaces to provide the needed encapsulation and separation. For object-oriented design purposes, statecharts [19] for OMT [34] or path expressions for Fusion [10] have been proposed, both of which are applied in the specification of the external available operations of a class with respect to its history. In practice, the concept of interfaces (abstract classes) is applied in object-orientation in order to separate usage and implementation in relation to syntactical typing, but behavioral or even semantical aspects are in fact rarely specified. The design by contract concept [27] is a considerable exception. The specific context of distributed and open systems leads to a situation in which the support of abstraction only at the syntactical level will be fatal. The common practice of software testing usually identifies system flaws through both module and integration testing. In an open and distributed environment we cannot foresee all possible later system configurations and the combinatorial growth of the number of such will make related testing scenarios infeasible. Separation by means of syntactical interfaces is therefore insufficent for suitably modeling the behavior of distributed systems.

We believe that both formal methods and, in particular, Petri nets can achieve a higher level of acceptance when the described issues become a relevant factor. This is independent from the question of whether this trend leads to a common software engineering practice in which wholly analyzable system models will become the norm. The trend towards high-level abstraction has led to the success of visual notations in software engineering for structure modeling. Such greater level of acceptance can be achieved with a behavior modeling notation that is not only scalable but which also possesses intuitive semantics. The notation would have to cover concurrency as a specific aspect. Petri nets conceptually provide ingredients for all such issues.

The object coordination net (OCoN) approach [15] attempts to overcome the described problems of high-level Petri net formalisms. It has its origin and roots in an attempt to achieve a fair tradeoff between the requirements of modeling concurrency (by means of Petri nets) and object-oriented structure and behavior, while also considering the limits and demands which are implied by a suitable visual formalism.

In Section 2 of this article, the OCoN notation is introduced. The application of the approach is demonstrated in Section 3 through concrete examples. The semantics with emphasis upon the dynamic model is presented in Section 4 by defining the concept step-by-step. In Section 5, the advantages of the language for behavior specification and application as visual language are considered. Related research is later discussed in Section 6. We conclude this article with comments on project status and planned further work.

2. OBJECT COORDINATION NETS

We assume the reader to be familiar with place/transition nets [6] and basic concepts of object-orientation. Transitions in our context are interpreted to be actions which occur through calls to services as provided by objects. For places which are called pools, we distinguish between two main entities: simple data and control flow using untyped tokens and objects constructed according to the type system which are stored in so-called event pools. More permanent objects, which are (as result of the object-oriented view) elements of the instance context and which frequently operate as carrier of activity or provider of services, are interpreted as the resources used by the system in performing its tasks. Places which maintain resources are represented by resource pools.

An overview of the elements of an OCoN net is presented in Figure 1. The resource and event pool elements are represented by hexagons and cycles, respectively. We distinguish between them in order to portray the more transient character of parameters. Control flow and temporary events are depicted by event pools and parameter edges (black arrow head) alongside the more static resource character of associations and local variables as represented by resource pools. An action is represented by a square with one request and possible multiple reply parts and an annotated operation name (e.g. op1 in Figure 1(a)). The reply part in the righthand-side of the transitions named op1 and op2, for example, specifies that there is exact-
ly one outcome for these operations. In contrast, op3 has two different alternative results as visualized by the split reply part. Activation edges (white arrow head) are utilized to depict usage as the carrier of activity for an action. Activation together with event processing during computation describe the control flow of a net (see Figures 1(a) and (b)). Based upon the distinction between resources and objects as produced and consumed through the flow of data and control, the metaphor of resources which is crucial in distributed systems can be used to make resource handling explicit.

2.1 Decomposition, separation and contracts

In order to support a method for behavior modeling that fulfills the requirements of the previous section, we need to provide both the decomposition of a system into encapsulated subsystems and a well-defined and expressive formalism to describe interfaces and interface-based interaction. The OCoN approach fulfills such requirements by focussing upon a system design which is ruled by the contract principle [27]. Organizational aspects are often employed to obtain a coarse-grained structure of the context in which specific behaviors are to be modeled. Using UML structure diagrams, this results in a set of (nested) subsystems providing services specified in contracts to the outside and also possibly using services from other subsystems through contracts in order to implement the provided functionality.

A general notion of a contract covering both classical and additional aspects is presented in [4]; it also defines syntactical interfaces, behavior contracts, synchronization effects and quality of service negotiations. Although syntactical interfaces do not provide sufficient information to exclude semantical misusage, behavior contracts cannot be managed automatically in an efficient manner. Considerations about the quality of service are often dependent on the run time behavior, and thus can only be specified when instantiating a system on a specific platform. In contrast however, the synchronization aspect represents a viable design view which can be expressed through the use of Petri nets.

Within the OCoN approach, a single contract is therefore a traditional signature specifying the names and the parameters required for all provided services. This signature is extended with an optional state-machine-like protocol net (PN). The net is required if some services are sometimes not available and, hence, used to specify externally visible behavior. Such protocol nets describe only the distinct external visible states of the contract by means of resource pools with the state label in square brackets and the operations available in each state by means of a corresponding action and activation edges. This form of specification enhances the benefits of interfaces in the case of services which obtain non-uniform service availability. Synchronization contracts of that kind are often found in systems acting according to complex organizational or technical rules.

For example, a file handler protocol may imply a certain usage, e.g. a read request, which will not succeed before an open request has succeeded or if the end of file has already been reached. An appropriate protocol for a file handler with operations open, read and close is presented in Figure 2 using a protocol net. The initial state where only an open request is possible is named [closed]. A possible reply to an open request is a signal open1 as an acknowledgment for a successful opening, which results in state [opened]. If the request fails, a reply open2 occurs and the protocol remains in state [closed]. The general scheme is that alternative replies are denoted by consecutive numbers in accordance to their textual order within the signature. In a protocol net, the same ordering is applied in arranging the related reply parts from top to bottom. A file handler in state [opened] can further be used to read data. A successful read request is signaled by reply read1 whereas the reached end of file results in read2 and a state change to [eof]. If we are in either state [opened] or [eof], the close request can be used to close the file handler and once again reach the state [closed].
2.2 Describing behavior

In order to suitably decouple subsystems, we distinguish between the external information as provided by a contract and its internal implementation. We furthermore rigorously restrict the usage of services by means of provided contracts only. The internals of a subsystem itself may have a complex structure and complicated rules relating to the handling of control and data. In practice this leads to a hierarchical design. Besides the internal subsystem structure, there are two aspects of special importance inside a single system: the internal details of the implementation of the provided services and the overall management of resources to accomplish that goal. The concepts necessary to model such aspects include two additional variants of Petri nets, namely service nets (SN) for describing the detailed processing when performing a single service and resource allocation nets (RAN) for the resource management.

The interaction mechanism of the nets is visualized in Figure 3. The calling of a provided service as described in the corresponding PN activates a corresponding call forward action in the RAN of the instance. This action further provides such call with the required resources and delegates the call to a dynamically instantiated instance of the requested service (SN). If the service employs external services from other contracts, portions of the work are further delegated to instances of that used type. Therefore, the RAN implements the protocol as offered by the PN in the form of call forward actions. A shaded actions indicates that the RAN at hand does not have complete control over the call forward actions, because the related call is triggered externally.

The detailed processing methods that implement a service are specified with a service net. The illustrated interaction mechanism adopts the object-oriented view of method invocation by supporting an intuitive call semantics which is well-known from procedural programming languages and remote procedure calls. The hierarchical design of the service nets provides for calls to other services during its execution. As a result of the synchronous nature of call and return, this works across hierarchies and can also be used as a wrapper call to services implemented by legacy code. In order to support hierarchical abstraction, while also maintaining the essential elements at the abstract call level, calling actions visualize the signature of the called service. For example, in Figure 1(c), the action requires a single data parameter and produces a single output parameter.

Figure 4 visualizes this concept: a call action is interpreted to represent a call to a method-like service. The precondition for firing an operation like \texttt{op} is specified by the ingoing arcs. A carrier of activity (here of type \texttt{Res}) for processing the call is required alongside the parameters (here an object of type \texttt{E1}) and a local pre-condition in form of a simple \texttt{Event}. In addition, the carrier has not only to be of type \texttt{Res}, but in particular must be a resource in state \texttt{[S1]}. This is defined by the type specification for the resource, i.e. \texttt{Res[S1]}, which restricts the overall type \texttt{Res} to the state \texttt{[S1]} of \texttt{Res}. The firing of a call to \texttt{op} is composed of three elementary steps: (1) consume all pre-conditions, (2) perform some durable activity inside the service \texttt{op}, (3) produce all post-conditions. The effect is an object of type \texttt{E2} and a change of the state of the carrier, which enters state \texttt{[S2]}. Therefore, an action fires essentially twice during the processing of a request, once when the request is initiated and once when such action terminates. The two steps correspond particularly well to the input and output elements of the call action. Although the direct correspondence with classical Petri net transition is abandoned, this method better preserves the integrity of an operation request which includes sending requests and receiving replies.
The OCoN approach combines the strength of Petri nets with the structural modeling techniques of UML [30], the de facto standard for object-oriented modeling. By combining both techniques in an orthogonal manner, Petri net mechanisms can be applied to express coordination, concurrency and partial states, even though the structure is described in terms of objects, classes and associations. Orthogonal combination with the object-oriented structural model is both necessary and possible, because general Petri net concepts do not have an intuitive structural modeling concept. The integration of nets with the structural model is centered around the bus-like implementation structure for a class as presented in Figure 3. Using three different kinds of nets for different purposes ensures a clear separation between modeling request specific and overall instance behavior. Protocol nets (PN) are used to describe the guaranteed or assumed behavior of contracts. Service nets (SN) describe the behavior for a specific task such as a method with its own thread of control in a programming language. An instance-wide unique resource allocation net (RAN) describes the overall instance activities such as request acceptance and autonomous behavior. This contains the allocation of needed resources for request processing, including the creation of related service nets for a request.

The UML extension mechanisms in the form of stereotypes are utilized for integrating Petri nets into the structural model. Rather than interfaces, the stereotype <<contract>> is applied in either an exclusive or shared (shareable) mode. The example shown in Figure 8 presents the combined usage of both kinds in the form of a shared factory contract (AcceptorFactory) that offers exclusive contracts. The request processing and autonomous behavior of classes can be specified by the stereotype <<implementation>> which is extended to incorporate a resource allocation net (see, for example, MailTransAgent-Send in Figure 8). Such implementation types delegate request processing to service nets or textual services where the former can be integrated into the model using the <<service>> stereotype.

### 2.4 Subtyping and inheritance

Contracts are the essential elements in the separation of classes and subsystems and in providing for further independent evolution. Subtyping and contract inheritance are suitable concepts to support such efforts. The secure usage of contracts subtypes (substitutability [40]) is ensured by behavior subtyping [1], while most object-oriented languages only support interface subtyping. In cases in which multiple concurrent clients are possible, we also have to ensure view consistency [25] for each subtype.

For the OCoN approach, a contract subtyping notion has been developed [14] that provides the required substitutability and view consistency. Distinguishing between shared and exclusive contracts provides for a less restricted treatment of subtyping for exclusive contracts. Figure 10 contains an example of contract inheritance for an exclusive contract.

By additionally supporting a notion of inheritance, the OCoN language can be considered to fulfill the requirements for an object-oriented language (cf. [39]). The notion of inheritance for concurrent object-oriented languages is a critical design aspect. Inheritance can be employed to reuse sequential methods, but inheritance of the instance-wide synchronization is often impractical (cf. [1]). Therefore, in the current inheritance notion for OCoNs classes, a subclass obtains all structural properties and the associated service nets of its superclasses, while the resource allocation net is not inherited in this manner. Syntactical inheritance ensures that each subclass always contains all resources that may be required by a supertype service net. The overall resource allocation of a derived class has in contrast to be rewritten, yet the concept of reuse is not currently supported for resource allocation nets.

It is to be emphasized that in contrast to contract subtyping and inheritance, implementation inheritance is not particularly relevant for the intended purposes of OCoNs. The external visible contract or interface hierarchy should generally be better separated from implementation reuse strategies based on inheritance, otherwise when subsystems later evolve independently serious design problems may arise.

### 3. EXAMPLE

A simplified email system will be used in the following as example to describe the application of the OCoN approach. The basic entities are a user agent (UA) to denote an email client, and a set of connected mail transport agents (MTA), which provide the email transport.

While this example reflects several typical requirements and allows us to discuss suitable forms for their implementation, the example is far more abstract than common knowl-
edge of concrete mail systems (see Figure 5). A ‘naive’ view of this well-known domain is assumed to avoid a too complex example.

Starting with a vague idea of a mail system, we have to identify relevant properties. This can be done by means of use cases. Sending a mail from sender to receiver and local browsing of emails are the most essential use cases associated with our mail system. Using the concepts of mail transport agent (MTA) and user agent (UA), we can develop the structural analysis model presented in Figure 6 [18].

When further considering the system at the design level (see Figure 7), considerations related to the distributed nature of the system become relevant. As a result of ‘race conditions’, parallel usage of the same stateful shared protocol can be particularly problematic. Rather than a single shared contract, the protocol is assumed to be handled separately for each requesting client. This can be designed by using both the factory pattern presented in Figure 7, to provide a shared contract AcceptorFactory accessible to all, and the exclusive contract instances for each specific connection that can be obtained. It should be noted that this scheme also reflects the common practice for classical Unix daemons that tend to wait for connection at a specific port and then fork a new thread to serve each established socket connection. We further realize the two identified essential classes UA and MTA of Figure 6 using the UML subsystems UserMailSystem and MailTransportSystem to establish correct separated subsystems. The MailTransportSystem subsystem is further applied in discussing behavioral design with the OCaN approach.

The MailTransportSystem subsystem design as presented in Figure 8 depicts the combination of UML structure diagrams with the additional information as provided by our nets. The design employs special objects rather than threads to organize not only mail acceptance but also its further processing. The Acceptor contract and its protocol are specified by a protocol net. Commencing from state [Ready], first the header is submitted using the accept operation and then the data is submitted using the store operation. Both operations will result in a final state [Done], excluding any further usage. If an error occurs, the unusable state [Failed] will be reached. The MailTransAgent-Accept implementation is used to realize this contract and mail processing is decoupled into two steps. The MailTransAgent-Accept class checks incoming mails and creates an internal SendOrder for each mail which is further processed by MailTransAgent-Send. A subsystem-wide orders resource pool is used to store and hand over requests in form of SendOrder elements. Request processing is undertaken in the RAN of the MailTransAgent-Accept class by externally triggered accept and store service calls (shaded). During the accept call, the addresses are checked and if successful are finally stored in an instance local pool of type Pair(Address,Address). In this case, the next step is that a store request has to provide the contents of the email which results in creating a SendOrderImpl element for further processing and its storing in the orders resource pool. This shared import of the subsystem resource orders is described by a shared resource pool in the RAN (double border) in contrast to exclusive pools for instance associations (single border).

The same resource pool is also used by a non empty set of MailTransAgent-Send instances to process the buffered mails. The MailTransAgent-Send implementation has only autonomous behavior and processes the SendOrder elements in the orders pool. A contained mail is either in a state that demands further processing ([Sendable]), or in a state ensuring that it has been sent ([Done]) or in a state signalling that the send order has failed ([Failed]). In the RAN of MailTransAgent-Send no externally triggered service is

**Figure 5** Overall mail system architecture

**Figure 6** Class diagram for mail system

**Figure 7** Buffered overall design with protocols

**Figure 8** Structural analysis model presented in Figure 6 [18]. The design employs special objects rather than threads to organize not only mail acceptance but also its further processing. The Acceptor contract and its protocol are specified by a protocol net. Commencing from state [Ready], first the header is submitted using the accept operation and then the data is submitted using the store operation. Both operations will result in a final state [Done], excluding any further usage. If an error occurs, the unusable state [Failed] will be reached. The MailTransAgent-Accept implementation is used to realize this contract and mail processing is decoupled into two steps. The MailTransAgent-Accept class checks incoming mails and creates an internal SendOrder for each mail which is further processed by MailTransAgent-Send. A subsystem-wide orders resource pool is used to store and hand over requests in form of SendOrder elements. Request processing is undertaken in the RAN of the MailTransAgent-Accept class by externally triggered accept and store service calls (shaded). During the accept call, the addresses are checked and if successful are finally stored in an instance local pool of type Pair(Address,Address). In this case, the next step is that a store request has to provide the contents of the email which results in creating a SendOrderImpl element for further processing and its storing in the orders resource pool. This shared import of the subsystem resource orders is described by a shared resource pool in the RAN (double border) in contrast to exclusive pools for instance associations (single border).

The same resource pool is also used by a non empty set of MailTransAgent-Send instances to process the buffered mails. The MailTransAgent-Send implementation has only autonomous behavior and processes the SendOrder elements in the orders pool. A contained mail is either in a state that demands further processing ([Sendable]), or in a state ensuring that it has been sent ([Done]) or in a state signalling that the send order has failed ([Failed]). In the RAN of MailTransAgent-Send no externally triggered service is
Figure 8  Buffered MailTransportSystem subsystem design with protocols
provided and instead only internal processing in forms of the send, logSent and logFailed call actions are specified. No activation edge for assigning an explicit carrier of activity is required here, because the instance itself acts as the carrier.

The detailed behavior of the send service net is described in Figure 9. As described in the RAN of MailTransAgent-Send (Figure 8), the autonomous send activity is triggered when one sendable (i.e. state is [Sendable]) SendOrder exists in the orders resource pool. For each activated service net of type MailTransAgent-Send::send a SendOrder instance of the resource pool is locked for exclusive processing. The order is accessible in a local pool indicating that the processed item is subject to the total control of the service. The service determines firstly if the considered mail can be delivered further and, if so, which next transport node is the corresponding Acceptor for the mail. This is achieved by requesting getAcceptor from the MTALocator. Secondly, the email source (from) and target (to) are checked for acceptance by using the next MTA node in the form of the Acceptor contract in state [Ready]. If this succeeds, the contract switches to state [Accepted] and is ready for transferring the message body by the store command. If an error occurs, the SendOrder is transformed into the [Failed] state.

If the processing succeeds, the [Done] state is set by the done operation. For both cases, further internal processing routines logSent and logFailed in the RAN are given. While the logSent service deletes the SendOrder, the latter produces a different SendOrder, which either is used to attempt several retries or to send back an error report.

For the example contained within Figure 8, the subtyping notion for contracts can be applied to improve the interoperability and permit independent evolution for each separated subsystem. Several mail subsystem providers might use different implementations which may provide several more sophisticated schemes besides the basic mail transport concept as described in contract Acceptor.

In Figure 10, a simple extension of the Acceptor contract named Acceptor2 is presented. While the Acceptor contract supports only the acceptance of a single target address, this contract can also support delivering mails for multiple target addresses. It should be noted that each single request of the accept operation in state [Accepted] may result in an ok or fail response, whereas the contract state will in no way be affected. In this manner, a contract user can check whether a single target address is being supported, but may also continue with delivering even when a subset of target addresses...
fails. The case of subtyping for contracts results in a situation where any mail implementation using the Acceptor contract will also recognize an Acceptor2 contract as a suitable subtype. The extension of Acceptor by Acceptor2 is in conformance with this subtyping notion. The behavioral subtyping notion of the OCoN approach further guarantees that when employing the subtype rather than the supertype, no additional synchronization problems could occur.

4. **SEMANTICS**

The semantics for the OCoN approach must address the different net dialects and their integration into an overall framework. A layered architecture as visualized in Figure 11 is employed to achieve a workable semantics. Firstly, the basic notion of a contract and its formalization in terms of protocol nets is examined. Secondly, a flexible notion of port-passing nets named coordination nets (CoN) is introduced. Finally, such formalism is used to define the OCoN notation in combination with the UML. For the complete formal treatment of the dynamic model we refer to the language definition.

4.1 Protocol nets

In Figure 12, the most common operations that are supported by an OCoN protocol net (left-hand side of equation) are defined in terms of basic place/transition nets (right-hand side of equation). In order to determine whether a behavior has to occur (obligation) or may be utilized as required, we have to further distinguish between fair transitions that do guarantee a form of progress and quiescent (grey) transitions that do not (see [33]). The protocols are specified from the perspective of the client and thus a grey transition indicates free choice, otherwise a normal transition has to occur. The normal labels indicate a usual or one-way request, while a label $m$ corresponds to a reply or event. Only modification and synchronization concerning the protocol state itself are described in a protocol net, thus edges for parameter or reply values do not occur. The following operations are used in protocol nets: (a) usual operation requests employing request and reply, (b) operation requests with parallel reply, (c) requests with alternative replies, for example named replies or exceptions, and (d) one-way calls.

4.2 Coordination nets

While protocol nets are capable of describing the behavior related to a single connection in the form of a protocol, we have to provide a formalism that is also able to express the behavior of a dynamically changing set of instances that interact via multiple connections. The coordination nets (CoN) formalism [14] provides an additional abstraction layer for the specification of the OCoN semantics. The formalization is based upon the forthcoming ISO high-level Petri net standard [11], which provides a non hierarchical high-level Petri net model. Port passing capabilities to model instance and system behavior for even dynamically evolving structures are added. For the CoN formalism, Level Two conformance with the HLPN standard has been demonstrated.

As an extension to high-level Petri nets as defined within the standard, a concept to provide modularity is required. In order to achieve this, a system is constructed based upon a set of coordination net graphs which are able to interact. The formalism supports the modeling of multiple instances of one object type, each providing a set of interfaces with dynamically changing external protocol states. Several net instances may be employed to implement the object behavior together and thus a mechanism such as place sharing for such is necessary to successfully model the object environment as shared by all net instances of an object instance.

We do not provide an object notion with our coordination nets, rather an interface-based separation employing typed ports which consists of an interface and a protocol net restricting message occurrences. Our final net dialect object coordination nets will provide a suitable object and class notion based upon CoNs.

The HLPN standard is extended by mechanisms for com-
munication, place sharing and dynamic net and port creation. In Figure 13, the basic structure of a coordination net system is visualized. There do indeed exist multiple instances of the same net type possibly sharing places between a net instance and its child (‘real place’ arrow connection). The nets may communicate through a communication infrastructure. The basic idea for communication is to introduce ports \( \xi, \eta, \ldots \) which represent associated or exported interfaces (objects) as pairs of connected communication endpoints which are themselves represented by port tokens (see Figures 13 and 14 for port usage). These ports can be used to receive a message using the following annotation for a transition:

\[
\eta = \xi; (\langle \text{op}(a_1, \ldots, a_n) \rangle)
\]

\[
\eta = \xi; (\langle \text{op}(\ldots) \rangle)
\]

\[
\eta = \xi; (\langle \text{op}(\ldots) \rangle)
\]

in which \( \xi \) is the resulting port; \( \langle \ldots \rangle \) denotes a given marshalling function; and \( \text{op}(a_1, \ldots, a_n) \) stands for an operation call with operation name \( \text{op} \) and input parameters \( a_1, \ldots, a_n \). There may be several distinct return vectors for a call and thus we use \( \text{DL} \) as operation name for the return alternative \( i \) to an operation \( \text{op} \) and annotate \( \text{DL}_{\text{op}}(r_1, \ldots, r_m) \) for a reply with return vector \( r_1, \ldots, r_m \). A corresponding synchronous send can be specified using a port \( \xi \) and the synchronous send operator \( \langle \ldots \rangle \). Analogously, an asynchronous send can be specified using a different operator \( \langle \ldots \rangle \). We distinguish between provide ports \( \rho, \varsigma, \ldots \) for exported and usage ports \( \phi, \varphi, \ldots \) for associated interfaces. A provide port can receive operation calls \( \text{op} \) and sends replies \( \text{DL} \), whereas a usage port can be used to send requests \( \text{op} \) and receive replies \( \text{DL} \).

Asynchronous and synchronous interaction are distinguished, as the synchronous version provides a more sophisticated means of interaction. Asynchronous communication is in contrast more efficient and reduces coupling between two systems. If useful we do not specify further whether synchronous or asynchronous interaction is required and so the more general send operation \( \langle \ldots \rangle \) can be used.

To create ports of type \( P \) or net instances for a declared net type \( N \), corresponding annotation expressions are also supported as presented in Figure 13. A net creation expression \( (\langle \text{op} \rangle = @N) \) binds to \( \phi \) a usage port corresponding to a particular initially provided \( \text{standards port} (\langle \text{st} \rangle) \) which each new net instance contains by default. This initial connection permits the establishment of more connections by utilizing these port connections to publish others. After a port creation \( (\langle \rho, \phi \rangle = @P) \) a pair of new unique connected usage and provide port instances is bound to \( \phi \) and \( \rho \). In this way, the dynamic creation of active net instances as a basic formalism for modeling instances and multiple active threads of an instance can be provided.

By supporting the described annotations in net declarations, a dynamically changing set of net instances interacting via port instances can be specified. So as to achieve a better visual representation, we draw all transitions annotated with receive terms and imported places shaded as depicted in Figure 14. A request is received in transition 1, which is replied by a send expression in transition 2. Transition 3 creates a new net of type \( N \) and propagates the resulting usage port \( \phi \) together with its other pre-conditions \( a_1 \) and \( a_2 \) in form of a vector to a place. Transition 4 may then consume such. The additional annotations distinguish a single coordination net graph from a high-level Petri net as defined in the high-level Petri net standard. Such annotations add concepts of message send, message receive, port creation and net creation to a transition.

The single net graphs interact via send and receive annotations that utilize the already introduced ports as addresses. The resulting system consists of a number of net graph instances connected via ports and a marking for each, as presented in Figure 15. The left two nets interact via corresponding usage/provide ports and a synchronized send and receive transition pair. The synchronous send ensures that the message is received immediately. In the center, a net creation is presented and the resulting port pair is visualized. The imported place of the created Net \( N \) is linked to the corresponding local place of the creating net, and the standard port \( (\langle \text{st} \rangle) \) of the new net is connected to the resulting port \( \phi \) of the create expression. A port creation is illustrated in the right net. This technique is used to describe instance or subsystem-wide sharing of resources and is realized with a form of lexical scoping.

We have decided to provide both synchronous and asynchronous behavior for coordination nets in order to achieve a greater flexibility. Synchronous interaction is useful, because it provides high-level abstraction which describes explicit synchronization where necessary. The asynchronous interaction can in contrast be used to combine systems with FIFO queues. An example run of an asynchronous and synchronous interaction is presented in Figure 16. In the case of an asynchronous interaction (see left section), the FIFO queue decouples both transitions, while in the case of synchronous interaction (right section) both transitions fire atomically together.

A suitable typing should carefully distinguish usual types that describe a value domain (literals) and which represent passive data from ports which are handles or identifiers that permit the request of certain operations or attributes. A typing which supports subtype polymorphism is essential for ports. Ports represent connections to other entities in a manner that should ensure abstraction and autonomy – the essential characteristics of object-based systems (see [39]). The basic idea for port typing is to associate an interface (signature) and a behavior to each port connection. The object life cycle and possible interaction with an object therefore becomes a part of the usage contract.

A usage of a port in conformance with its protocol is presented in Figure 17. The port \( \phi \) is first transformed to \( \phi \) by sending \( m \). Later, it is transformed to \( \chi \) when receiving the

![Figure 14 A coordination net graph example](image-url)
reply \( m \). The port type and state is annotated using the shortcut \([\phi]_I\), where \( I \) denotes the interface and protocol and \( \phi \) the specific protocol state.

### 4.3 Object coordination nets

The OCoN approach combines the HLPN standard [11] with UML [30]. In Figure 11, the abstraction layers of the OCoN semantics and its foundation on the OMG metamodel and the UML are visualized. We avoid the considerable weaknesses of the UML by integrating our approach only with a subset that is consistent with respect to structural and behavioral semantics. The different behavioral specification aspects covered by protocol, service and resource allocation nets are embedded smoothly into the UML through the use of stereotypes. The stereotypes \[<<contract>>\] for either exclusive or shared contracts, \[<<implementation>>\] for implementation classes and \[<<service>>\] for methods are all employed.

OCoNs are defined on top of the CoN formalism so as to provide a formalism of a more high-level and restricted
nature. Structural aspects are realized with UML mechanisms and the usage of nets is restricted to model behavior. For a structural UML model consisting of this restricted set of types, a complete dynamic model can be derived. The resource allocation net of an implementation class results in a coordination net that describes the request processing and autonomous activities of each instance. To handle specific requests at runtime, instances of the service net that is assigned within the resource allocation net are employed. The mapping of high-level actions of the OCoN net dialects onto suitable CoN nets is described below.

In order to specify the semantics of OCoN constructs we use reentrant subnets in a macro style and a specific form of transition refinements, as presented in Figure 18. In this way, a call with either a contract blocking character, parallel reply or an one-way call, can be specified (see Figure 12).

In Figure 19, the corresponding CoN behavior for a regular call with alternative replies is specified. It is a generalization of the simple call described earlier in Figure 17. The pre-condition edge denotes the necessary port and a request \( m(a_1, ..., a_n) \) is sent by employing the usage port \( \phi \). For each possible reply, a ‘receive’ is immediately offered which may handle different return parameters. The different transitions can be used to specify different side effects within the embedding net.

For each contract usage port, a provide port exists. The owning instance of the provide port is required to provide the described services as guaranteed in the protocol net. This is to be achieved using a so-called call forward action as specified in Figure 20. A request is initially received and the set of resources exclusively required for the request processing is consumed. As a post-condition, not only the received parameters but also the allocated resources are forwarded to a new created net. Both the receipt port for results from the newly instantiated net and that for sending the response to the requesting party are stored locally as a pair. The created service net will initially receive the forwarded request. When this request terminates, it will return both the reply and the temporarily used resources. The reply will be forwarded and the resources returned to their original pools. A call action may occur in a service or resource allocation net, while a call forward action for the request acceptance is restricted and may occur only within a resource allocation net.

5. VISUAL LANGUAGE

The OCoN actions of Figures 19 and 20 represent the caller and callee role corresponding to the usual operation request with a reply presented in Figure 12(a). The language thus permits abstraction from the concrete and error-prone message passing style of modeling necessary with CoNs. The behavior can be described in terms of requests and simple contract usages. Explicit send and receive no longer require individual consideration, rather higher-level interactions such as call or one-way call are provided directly. The different types of high-level usage of a contract can therefore be visually described through the specific OCoN action symbols. The typing of the contracts using interfaces and protocol nets further enforces a disciplined usage. The creation of instances and subnets is furthermore achieved by using extra forms of actions.

For place/transition nets, the popular token game provides not only a suitable visual representation but also intuitive semantics. We thus have designed the OCoN formalism to provide a set of high-level action types that can be understood in the local context as a token game. It is therefore significant that the enabling does not depend upon textual guards. Methods or external operations with alternative replies (see the \( \text{op3} \) action in Figure 1(a)) are used in modeling to indicate the relevant alternative cases in a graphical rather than textual manner. As such, a useful additional abstraction for predicates is introduced and textual guards transferring the semantics from the transitions to the annotations can thus be avoided. This is in contrast to most HLPN approaches which make extensive use of textual annotations.

In Figure 1(c) the signature abstraction relating an action to a service net is depicted. Besides this embedding of a single service signature, fragments of a protocol net are also...
visually embedded when employing a contract. The visual embedding of the Acceptor contract (Figure 8) in the service net of Figure 9 is presented by the dashed line in Figure 21. After an Acceptor contract in state [Ready] has been obtained via the getAcceptor operation, the subsequent application of the accept and store operation and its resulting visual embedding of the contract is visualized. This seamless visual embedding [17] motivates our design decision to restrict protocol nets to state machines. The multiple places of a Petri net thus provide for the embedding of contracts using resource pools representing the different contract states. The object life cycle can also be modeled with non-state-machine-like Petri nets, e.g. with subtyping based upon branching-bisimulation and abstraction [38]. Such more general solution, however, excludes the intended visual embedding and a Petri net independent contract notion.

Associations represented by resource pools containing related contracts can be used in conformance with the specified protocol and thus their usage corresponds to the visual embedding shown in Figure 21. It is to be noted that call forward actions contained within a RAN additionally provide a corresponding visual embedding for the dynamically created service nets. In Figure 22, these multiple forms of visual integration of contracts described by protocol nets into resource allocation and service nets are portrayed (compare to Figure 3).
6. RELATED WORK

The OCoN language is influenced by and related to numerous software engineering aspects. Following the focus of this article we concentrate here upon the language design and its formal background as well as related Petri net approaches. Firstly, the related work concerning the coordination nets (CoN) formalism is addressed. Secondly, the OCoN approach is considered.

6.1 Coordination nets

In the case of colored Petri nets, the extension to hierarchical colored Petri nets and the composition mechanisms substitution of transitions or places, invocation transitions and fusion sets for transitions or places have all been proposed [21]. All such mechanisms, excluding those relating to invocation, are specific to a Petri net context and do not directly rely upon the natural notion of information exchange directly, but rather encode such into a net-specific view.

Consider, for example, a place fusion which might be a useful abstraction in an assembly line structure, but such would not correspond to a common software interface such as a procedure or stream. Nets with procedure calls as considered by Kiehn [23] result in considerable analysis problems and thus are usually provided as an add-on rather than basic concept; e.g. B(PN)² supports procedures only as extension [22]. The transition invocation can be considered as the procedural abstraction common in programming languages, thus providing the necessary general abstraction concept. To model object-orientation and dynamically evolving structures we even have to add references and port passing capabilities (see p-calculus [28]).

The net dynamics of the OCoN approach have been realized introducing not only the visual but also high-level Petri net-conform CoN formalism. Within the context of the p-calculus, a net-based calculus named Mobile Nets has also been developed [8]. It adds true concurrency to the p-calculus by extending the usual textual binding for p-calculus processes to cover a notion for places that can be accessed in parallel. Therefore, this approach does not provide the necessary visual net-related metaphor such as the CoN formalism essential for the construction of a visual language.

6.2 Object coordination nets

The OCoN approach provides a Petri net-based extension to the UML. However in practice, perfect orthogonality is seldom achieved. Both, UML behavioral description techniques and the underlying UML behavioral model present serious weaknesses, particularly in relation to distributed systems. The OCoN integration therefore combines the mature structural model of the UML with an appropriate Petri net dynamic model and overcomes several limitations and problems related to behavior modeling with the pure UML. As demonstrated in [16], the OCoN formalism results in a more accurate representation of the structural model within the formalism compared with Statecharts. We further refer to this work for a comparison with the UML concepts for behavior modeling itself and discuss here only other object-oriented Petri net approaches.

In the case of system design with Petri nets it is more promising to avoid a Petri net-specific mechanism and to integrate Petri nets into a common object-oriented decomposed system view. This approach should rely upon the successful mechanisms for abstraction and encapsulation. Most early approaches to combine object-orientation and Petri nets can be classified as either ‘Petri nets inside Objects’ [7] or ‘Objects inside Petri Nets’ [3]. More recent approaches advocate more dynamic and expressive models in which object references are controlled in nets related to classes, thus supporting both concepts (see [2]).

One crucial aspect for the design of an object-oriented Petri net formalism is its support for object interactions, including adequate support for interfaces and polymorphism. Several approaches propose place fusion as the main means for connecting Petri nets [24], but the use of places does not provide a satisfactory model for software interface. On the contrary, most solutions based on algebraic specifications model cooperation in terms of transition fusion [3, 5], allowing the related behavior to access all cooperating objects of that activity in an atomic action. These methods violate encapsulation and behavior becomes independent from objects, hence violating an important criterion for object-orientation. The support for message exchange or operations is essentially required to achieve encapsulation. The different approaches that support this vary with regard to the level of support for either message passing or the high-level interaction of a procedure call construct [37, 26, 9, 20]. Such approaches either explicitly provide an object state through one global net per instance [29, 24, 37, 20] or implicitly as composition of so-called method nets [26, 9]. A systematic separation into a resource-oriented scheduler describing the overall instance state and method related active method net instances is only realized within the OCoN approach. To achieve visual scalability the usage of several nets for specific tasks becomes necessary, while their simple visual separation using regions is insufficient.

In distributed system design, encapsulation must be guaranteed, thus rendering essential a type-secure notion of contract or interfaces. Non type-secure approaches [9] are therefore not appropriate. To our knowledge, no other approach integrates an external behavioral specification such as a protocol net in providing a contractual notion with behavioral subtyping, thus also supporting behavioral abstraction.

7. CONCLUSIONS AND FUTURE WORK

This paper presents the OCoN approach, which integrates an object-oriented approach with a high-level Petri net formalism, extended in a p-calculus style to cover dynamic aspects. The approach builds an orthogonal extension to a subset of the UML and adds powerful concurrency and contract modeling capabilities. A tight integration has been achieved and proprietary extensions to the UML itself have been circumvented.

The contract notion for the OCoN design approach supports the explicit specification of contractual relations and provides a notation for the specification of coordination aspects on an abstract level. Several design alternatives can be evaluated and compared in order to determine the qualities of each (see [18] for an example analysis). The OCoN
formalism is therefore a suitable notation that can be applied even during the early stages of the design process with emphasis upon software architecture [36]. A suitable visual notation is a crucial prerequisite for a successful approach. We applied useful Petri net visualization concepts and achieved preservation of such by applying object-oriented standard techniques in a systematic fashion [17].

Our initial application domain has been that of distributed software systems [15,18], although we have also explored embedded systems and are currently investigating the design of workflow applications [41]. Experience with student classes and courses indicates that even without experienced designers, our approach remains highly suitable. Although the results are promising, training for a specific net-based notation remains problematic for beginners. Future plans include work on a framework supporting the final implementation and in the evaluation of the OCoN tool support. We will seek an integration of our approach with a UML tool, with extensions towards consistency checks and complete simulation capability. In order to gain more experience with larger projects, we are currently studying the approach in an industrial environment.

ACKNOWLEDGMENTS

The authors want to thank all students involved in prototype implementations and in the evaluation of the OCoN formalism during classes and courses.

REFERENCES

22 International Telecommunication Union (ITU). Specification and Description Language (SDL) Recommendation Z.100, 1992


