Seamless Visual Object-Oriented Behavior Modeling for Distributed Software Systems

Holger Giese, Jörg Graf and Guido Wirtz
Institut für Informatik, Westfälische Wilhelms-Universität
Einsteinstraße 62, 48149 Münster, GERMANY
guidow@math.uni-muenster.de

Abstract

To ease the development of distributed systems, the visual notions for the structural aspects of object-oriented analysis and design should be combined with techniques handling concurrency and distribution. A novel approach and language for the visual design of distributed software systems is introduced and illustrated by means of an example. The language of OCoNs (Object Coordination Nets) is integrated into the structuring mechanisms of the UML standard for object-oriented analysis and design. Such an object-oriented notation is crucial for handling complex software systems and can be extended with the graphical expressive power of Petri nets to also describe concurrency and coordination. The same visual language is used to specify the interfaces and contracts of software components, the resource handling within a component as well as the control flow of services.

Keywords: visual language, object-orientation, contract, coordination, concurrency, Petri nets

1. Introduction and Background

Due to an increasing demand for enterprise wide cooperating software and web based solutions as well as the progress in workstation technology and networking, distributed systems have gained much importance recently. Moreover, middleware technology like, e.g., CORBA [26] helps to overcome classical low-level problems of heterogeneity in hardware, networks and languages and makes distributed systems a commonly accessible option for a wide range of applications in manufacturing, process control, banking, multimedia etc.

The development of such systems, however, is not yet well understood. Being at least as complex as many non-distributed business software systems, distribution adds its share to complexity by introducing – among others – concurrency and complex resource conflicts. These cannot be handled in the well-known centralized fashion which used to work well in operating systems for decades because distributed systems lack global access rights or even consistent global knowledge about their state. Nowadays well accepted object-oriented development methods, e.g., [5, 19, 29] deliver the basic features for specifying the static structure of such a model and provide a good point to start with. The real challenge with distributed systems, however, is their *behavior*. Here, the focus of OOA and OOD on business applications has resulted in a high emphasis on complex static structures and a really poor support for dynamic aspects and behavior. For example, among the many notations for dynamic modeling offered by the UML (Unified Modeling Language) [27] like, e.g., message sequence charts [17], collaboration and activity diagrams, only STATECHARTS [13] come close to sufficient expressive power. Their state-machine-based nature, however, makes concurrency not a central issue within the formalism and there are more natural models for handling concurrency like, e.g., Petri nets [7].

We utilize (parts of) the visual notations of the UML when designing the *static structure* of software systems and emphasize the *software architecture* [30] aspect to achieve a suitable design. In the very center of our approach is an object-oriented, visual design language (Object Coordination Nets) for describing all relevant dynamic aspects of coordinating the concurrent use of components and resources. OCoNs are not only a notation but a real visual language with syntactical rules, a polymorphic type system integrated with the underlying object-oriented structural model. The real advantage of OCoNs is the rigorous integration of object-oriented concepts into a high-level net formalism while preserving the benefits of simple net models. This is accomplished by using nets in a manner which carries the essential information in the visual part of an OCoN and not by means of complex textual formulas annotated to graphical symbols and arcs.

In the rest of this paper, we give a short characterization of our style of design (section 2), illustrate the basic elements of OCoNs (section 3) and their different usages by means of an example that is discussed in detail in section 4. Animation and validation aspects are considered in section 5, related work is discussed in section 6. We end up with some remarks on the project status and future work in section 7.

2. Contract-based Design Style

The focus of our behavioral specification is restricted to coordination aspects as discussed in [14] which are essential for the design of distributed systems and allow to delay other aspects (see [20]) to later phases of the design or even the implementation. We extend the usual interface notion and combine them with a behavioral description describing the availability of services for an interface (see [25]). These *contracts* can on a coarse grain level of design be interpreted as architectural connectors [1]. Our contract notion is in contrast to [18] restricted to cover only a single interface like [24]. We do not consider any functional aspects and specify only the coordination aspect by adding a behavior description. Because the details of our work towards requirements and a clear contract-based method for designing components, interfaces and their interaction have been described elsewhere [36, 12], we put our focus here on the OCoN visual design language and its different usages and benefits in the context of the systematic design of components and contracts.

We distinguish three views for behavioral aspects for specifying classes and their abstract interfaces/contracts in a distributed system. For each of the views we use OCoNs to model the dynamic aspects on the level of detail best suited for the view at hand:

- *externally visible interface/contract*: services accessible from the outside as well as the externally visible aspects of the state of instantiations of the class w.r.t. the availability of a service (*protocol net*).
- *object-wide coordination*: overall resource usage of the external as well as internal services of an object (*resource allocation net*)
- *a single service within its class context*: detailed implementation of a service concerning the resource coordination. This may be a so-called *service net* or a textually coded method.

Hence, the schematic view of a class looks like outlined in figure 1 where all variants of visual representations are OCoNs. Moreover, the continuous embedding of the three behavioral views providing a seamless behavior modeling is visualized. We can abstract from each implementation using protocol nets. In a resource allocation net, the request

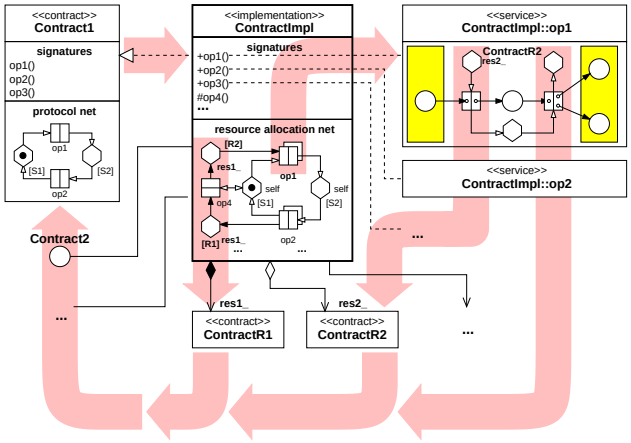

Figure 1. The seamless usage of OCoNs

processing actions are refined using service nets. Actions and resources represent contracts and their usage via service calls in service and resource allocation nets can be interpreted as embedding certain parts of the corresponding protocol net. The different kinds of nets are discussed in more detail in the next section.

Besides integrating our approach in a nowadays well-accepted methodology, object-oriented analysis and design of systems in general leads to a decomposition of a system into less complex entities. This decomposition and the separation of concern guarantees also the *scalability* in general and w.r.t. graphical complexity of our approach.

3. The Elements of OCoNs

Starting with the common Place/Transition net (P/T net) notion [7], there are only a few principal steps of further refining the general concepts in order to obtain a net model which fits into the object-oriented world for, e.g., specifying a single service in the context of a class. We assume the reader to be also familiar with the basic concepts of classes, methods and inheritance used in any object-oriented language like, e.g., in `Java` [2] to pick a popular example.

3.1. Contracts and Protocols

The first kind of net – *protocol nets* – is part of the contract and describes those restrictions which are essential to know for an external client when using the offered public services. A contract (≪contract≫) is specified using an extension of the UML stereotype ≪interface≫, where a compartment contains a protocol net. The `Order` contract shown in figure 2 is build by an interface containing a set of services that allow to initialize the order (`setData`, `addUser`), getting the order identifier (`getID`), test whether an order is overdue (`test`) and pro-

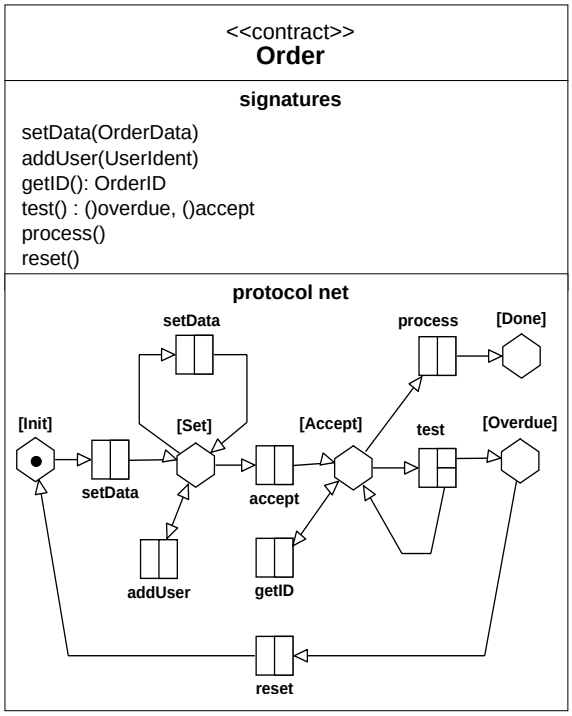

Figure 2. A contract with protocol net

cessing the order (`process`). Additionally, a protocol net is contained that describes the service availability based on abstract external states.

The *actions* (squares) in such a protocol net represent services and firing them is interpreted as a *service call*. An *action* is equivalent represented in usual P/T nets by a call and return transition. The well-known procedure call metaphor should ease the understanding of our nets in much the same way as the *remote-procedure-call* paradigm has gained much of its benefits from being similar to an ordinary procedure call. The state of the protocol net is represented by a couple of places (hexagons) representing the distinct states of the contract using a state token. Available services for a certain state are represented by actions which lead from this state to another or the same state. Besides the state, we distinguish two different levels of access modes:

- *potential parallel*: used for non-interfering accesses (two-way edge, see figure 2 action `addUser`)
- *exclusive*: used to avoid race conditions due to insecure intermediate states (seperate edges to/from action, e.g., `setData` in figure 2)

This may be an usage which does not change the state (get–use–put back) or a state changing use if an action changes the state of the contract. Note, that this distinction is only needed for aspects of the state of a contract which are important for its capability to execute services. In order to reduce the visual complexity, all services which are available in every state and cause no state changes are simply omitted, e.g., `getOrder` and `instructOrder` (see figure 8).

The protocol nets are restricted to finite state machines, but in contrast to STATECHARTS [13] which are used in the UML to specify the external protocol, extended features like and/or state decomposition are not supported to ensure that the resulting descriptions are simple enough to allow a seamless embedding into other behavioral specifications. To handle complex cases, the usage of multiple contracts (c.f. [34]) in a style supporting *separation of concern* as, e.g. role based modeling (see [28]), is applied.

3.2. From Protocols to Services

When using contracts, for example in a *scenario* as demonstrated in figure 3, parts of the specified usage protocol can be embedded as needed. In this scenario, an `Order` contract is obtained from a `Factory` resource and the order is initialized (`setData`), an user is added (`addUser`), the id is retrieved (`getID`) and the order is accepted (`accept`). An action is here again interpreted as a *service call*, but this time parameters have to be supplied and the termination will deliver a set of results. There

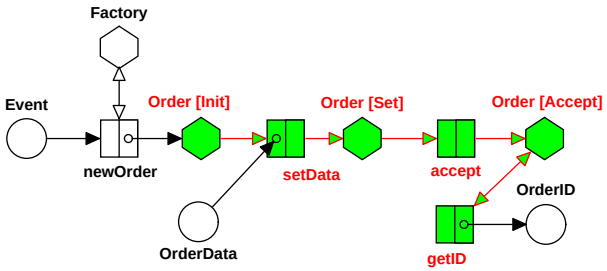

Figure 3. A scenario using the `Order` contract

has to be a contract of appropriate type which is *providing the activity* of an action as well as objects which act as *ingoing parameters* or *outgoing results* for each service call. The former category can be interpreted as representing the resources needed to fire an action which may be changed during that activity but tend to have a longer lifetime than a single activation. The latter represent most of the detailed flow of control and the objects which are to be processed and, hence, may be consumed during execution. W.r.t. a single action, two types of connected places, namely *event pools* (circles) and *resource pools* (hexagons) – already used for the states in protocol nets – are distinguished. Because a service may need additional resources besides the *unique single* carrier of activity, and an object may be used as a parameter in one action, but as a resource in another one, we partition edges into *enabling edges* (filled arrow head) for parameters/results and the *activation edge* (white arrow head) which is only used for resources. Parameter edges

have a *consume*, *produce* or *read* semantic concerning the pools in contrast to activation edges that describe execution access and thus do not really consume or produce the objects.

Based on the distinction between resources and objects 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. The *usage* and *status* of resources can be specified in detail. A single resource may be represented by more than one resource pool if the different pools stand for the same resource but different internal states. The state is annotated with the resource type, e.g., `Order[Init]`. This allows a rather detailed modeling of requirements: for an action to be enabled, the resource is required to be in a specific state, e.g., an `Order` should be in state `Accept` for activating a `getID` operation (see figure 2).

We distinguish *local* and *external* resources. The former are under exclusive access of the current context at hand whereas the latter are imported from the surrounding context and shared with other behaviors (visualized by double lines; see `SecManager` resource in figure 4 and 5).

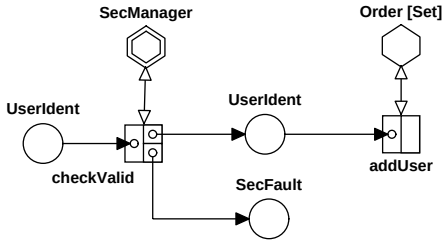

Figure 4. Check permission scenario

A second aspect of an order initialization should be a check whether the user has permission to do so. A scenario using a `SecManager` adds the user if the `checkValid` request to it succeeds (see figure 4).

The aspects discussed so far, are needed to bring the suitable modeling metaphors for distributed systems into the net language. The following aspects are requirements of a clear integration of object-oriented structural descriptions and nets by integrating the type system provided by the class hierarchy into the nets. The polymorphic type system defined by the class hierarchy provides an *isa* relation which as usually reflects the position of a class in its class hierarchy. Because an action is identified with a service, the carrier of activity has to be a contract of a type and state which provides that service. Moreover, the parameters and results have to be typed objects, too. Hence, pools have associated types and are restricted to markings with typed objects which fit into the pools, i.e. o fits into $p \in P$ iff $isa(o, typeof(p))$ is valid. Simple *token* are still in use to model the occurrence of events of a basic event type `Event` to describe pure control flow (see figure 5).

The nets can express calls via the provided actions with signatures. A signature defines – much in the sense of abstract data types [9] – simply the types of the parameters and resources connected to an action. It provides the information for static type checking. In and out are visualized by means of ports. The usage of traditional service signatures permits to interface our net formalism with textually provided code of the target language in order to reuse available class libraries, include legacy code or replace a net specification with a hand-coded or automatically generated piece of code. Due to the fact that the entire type system of OCoNs depends on the concept of interfaces/contracts and the underlying class hierarchy, it is a simple extension of the well-known mechanisms used in object-oriented languages. This has the additional benefit that a user who is familiar with such a textual language can use the same rules when working with OCoNs.

In our example, an `OrderCoord` object is identified to be responsible for the coordination described in the scenarios of figure 3 and 4. A *service net* that combines

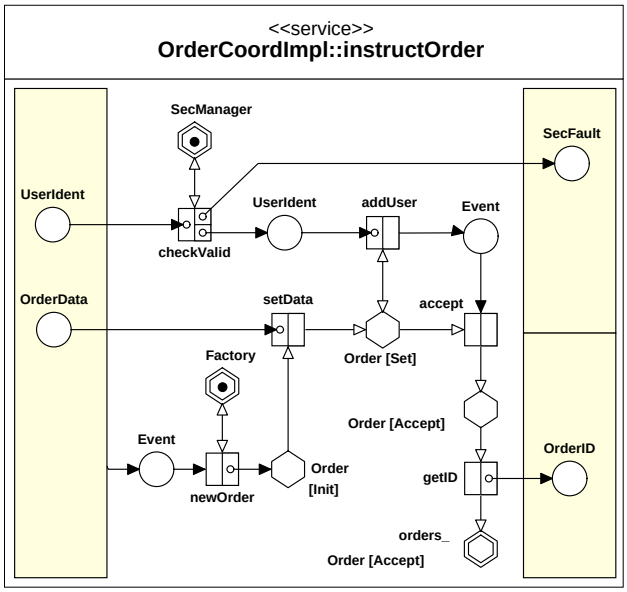

Figure 5. A service net to create an order

them and specifies how to create and initialize an order according to the specified arguments is presented in figure 5. The `instructOrder` service might fail if the `SecManager` detects a security fault. The nondeterministic output of the `checkValid` service usage in figure 5 is simply a short-hand notation for a behavior which can produce several results like illustrated for example in detail by the `instructOrder` service itself.

The number of incoming/outgoing arcs is visualized in the service net by (shaded) bars for in (left) and out (right). A single service specifies what kind and number of resources it needs by itself. This supports the concept of locality by avoiding the detailed treatment of complex resource interactions in each of the services.

3.3. Internal Resource Handling

In complex cases and for visualizing the initial demanded resources of each service, a description for the instance wide resource allocation and scheduling is needed. A stereotype ≪implementation≫ and a special resource

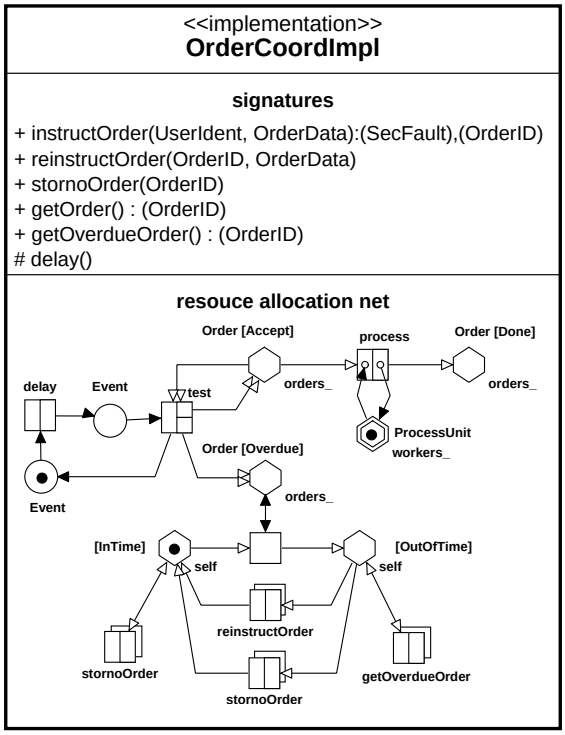

Figure 6. Resource allocation net

allocation net compartment is used to build an implementation for the `OrderCoord` contract presented later in figure 8. This implementation `OrderCoordImpl` in figure 6 contains a *resource allocation net* that combines the initial resource demands for all services of its class, provides the point for observing dependencies between services and for specifying design decisions w.r.t. overall resource handling and scheduling within a single class. Note, that only those services are explicitly represented in the resource allocation net which have either special resource requirements or which are not available in all resource states. For this reason, e.g., `instructOrder` has been omitted in the net of figure 6.

Here, resource pools which represent the state and number of the different associated objects and two different kind of actions are used. The state is represented by special state places (`self`) much like in the protocol nets; special *request processing actions* drawn with a shadow are used to specify the initial resource demands for services offered externally. This net represents the class internal view of its local resources as well as the external resources the class uses. A resource must always be in a single state iff present at all.

A principal design decision is that *all services are concurrent a priori*, i.e. as long as not stated otherwise, a service of a class may be called concurrently with itself as well as with other services of the same class. This does not imply that all calls really work in parallel because that may depend on the availability of resources. In our design, the aggregated elements of a class are interpreted to be *resources* which enable a class to implement its services. So, a parallel call of different services using, e.g., the same `OrderCoordImpl` object will be sequentialized due to conflicting resource requests.

Additionally, the *inner activity* of an object can be specified using the simple actions. In our example all created orders in state `Accept` are tested after a certain delay and transformed into state `Overdue` when their processing was not initiated in time. All orders in state `Accept` are processed by assigning a `ProcessUnit` to them if available.

4. The Order System Example

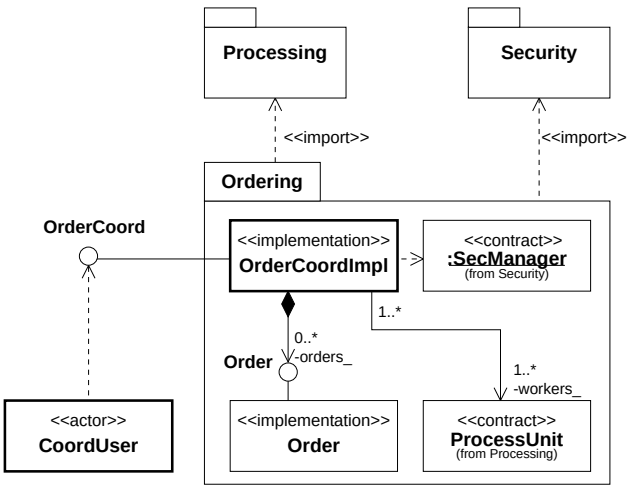

Figure 7. Class diagram of the example

We use the abstract requirements for an order processing system to demonstrate the combined benefits of our approach. Such an order system has to process orders and allow their management. The overall structure of the de-

sign in figure 7 (using a collapsed representation for the different entities) contains a special coordination contract for managing orders and controlling their processing named `OrderCoord` for each client, which is specified in figure 8. This contract is implemented by `OrderCoordImpl` (see figure 6) that contains a set of `Order`s represented by a resource `orders_` to handle received orders. A security manager `SecManager` and a resource pool `workers_` of `ProcessUnit`s to process the orders are also available for `OrderCoordImpl`.

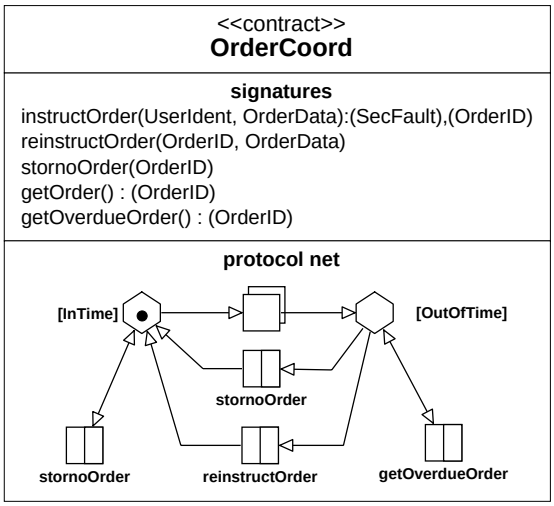

Figure 8. The external view on the system

The relative simple services `getOrder` or `getOverdueOrder` that simply retrieve information do not contain any coordination aspects and thus should better left for later implementation with the target language. The `instructOrder` service is used via a client to add orders. The analysis of the related coordination of an `Order` contract is described by the scenarios in figure 3 and 4 and its final specification in figure 5. The `OrderCoord` implementation `OrderCoordImpl` is an active object that triggers the correct order processing by repeating a `test` service call to each `Order` in state `Accept` until the order is either processed by assigning an `ProcessUnit` and thus changed to state `Done` or becomes `Overdue`. This is done using a cyclic event flow which is delayed by an internal `delay` method. A parallel call to all elements of the `orders_` resource pool in state `Accept` is specified using a parallel activation edge which is drawn with a double head. Each single call might transform the order either into state `Overdue` or the old state `Accept`. If an order is in state `Accept`, the processing can be initiated by assigning an available `ProcessUnit` to it via call `process`. For an `Overdue` order an exceptional handling is needed. Thus we switch the state of the `OrderCoord` contract using a local action in the resource allocation net which fires when at least one order in state `Overdue` is available (read arc) to signal a client that at least one of its orders has failed.

A client might react on the observable spontaneous change to a new state of a contract by using a simple precondition edge connected with a pool representing this new state. Thus using *call-backs* that may result in race conditions in the client behavior can be avoided and we still can use a simple unidirectional contract.

The client can either use the `stornoOrder` service to simply delete the order or set a new processing time via the `reinstructOrder` service. It resets the data and transforms the order to the state `Accept` again. If the client object does not care about failed orders, it can simply ignore the contract state and restrict its usage to `instructOrder` and `getOrder` which are available in each state.

5. Working with the OCoN Language

The interactive specification of behavior aspects with OCoNs is currently possible with a prototype of an editor with integrated simulator. This tight coupling of specification and validation improves the handling of behavior specification to a great extent.

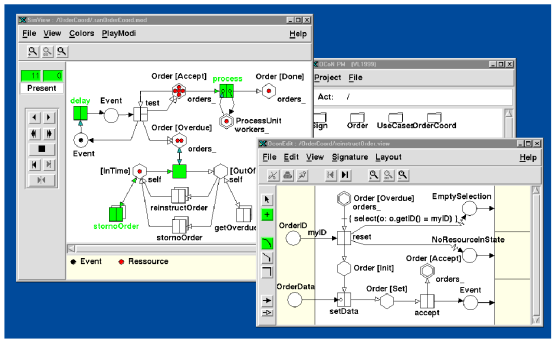

Figure 9. Combined Editing & Simulation Tool

As outlined in figure 9, a direct simulation of even only partial specified nets is possible due to an always valid abstraction that reduces object interaction to the non deterministic firing of transitions in the underlying Petri net model. Of course functional aspects can not be covered this way, but the coordination or at least its non deterministic abstraction can be explored.

Several relevant questions concerning the correct or compatible behavior of service, protocol and resource allocation nets can be analyzed based on available Petri net tools. Besides such qualitative analysis techniques, simulations with estimated time intervals can help to identify bot-

tlenecks or problems concerning the given performance requirements. Such quantitative analysis techniques are possible based on stochastic or timed Petri net models [22].

6. Related Work

There has been a lot of work on visual notations for OOA and OOD around for some years now. Most of the time, the many different notations have been seen to be a disadvantage for their overall use in serious business applications. With the UML, an approach to unify all these approaches is in the state of standardization. However, there are three major flaws with the proposed notation:

- Instead of unifying different notations, often the simple union of different notations has taken place which makes the UML an only hard to manage sampling of nearly 20 notations for the same or different aspects of modeling.
- Although working with meta-models, the UML is a notation but far from being a language with clear semantics at the moment.
- The support for dynamic aspects is – besides the STATECHARTS [13] – rather poor and not well-defined.

Message sequence charts (sequence diagrams) are essentially trace based and fail to express concurrency. They also provide a behavior description for a set of objects from an external point of view like collaboration diagrams. Thus they fail to describe behavior from the local perspective of a single object. State based notations like STATECHARTS or activity diagrams describe the behavior based on abstract states and support behavior integration only using state composition. Thus a suitable modeling based on the resource metaphor is not provided. Besides all these differences, none of these notations is covering external behavior descriptions (contracts), resource allocation/scheduling and service specification at the same time. Every notion only covers its special application area and thus can not provide a *seamless* integration as demonstrated in figure 1.

Other approaches like [10] try to combine common non object-oriented behavioral description languages like SDL [16] with object oriented concepts, the UML or other analysis and design notations. But these attempts map the characteristics of object-orientation on process based structures and thus can not provide a seamless integration.

Because we use a special kind of extended Petri nets, those projects which use a variety of high-level Petri nets to model complex concurrent systems, are closely related to ours. High-level nets have been proposed as a modeling language for about 15 years, e.g., [11], but the lack of abstraction and modularization concepts which are understandable for a non-expert restricts the acceptance of these approaches (cf. [15] for a discussion of five different abstraction mechanisms). The problem of integrating object-oriented concepts into the world of nets has been tackled based on algebraic specifications [3, 4] or through the extension/combination of colored Petri nets [21, 31]. The (low-level) concepts used there, however, provide no sufficient notion of abstraction yet. Our approach, as well as that of [23], takes the opposite way by integrating nets into the world of object-oriented modeling. Although it is demonstrated in [23] that traditional behavior modeling notations can be replaced by Petri nets, the model of interaction lacks abstraction.

In contrast to [33] and PNtalk [8] the OCoN approach integrates structural and behavioral aspects of object-oriented designs, contains an implementation scheme to specify object internal synchronization systematically and provides a seamless embedding of aggregated elements using the resource metaphor and contract concept. Besides our integration into an object-oriented modeling process, we focus on a visual specification of coordination aspects like [6]. In contrast to message passing based visual programming for distributed systems [32, 35], our approach is based on the remote procedure call as basic form of interaction.

7. Conclusions

We have presented a new visual formalism for the design of distributed systems which integrates object-oriented concepts with high-level Petri-nets. The object-oriented part permits a well-structured visual language obtaining a concept of hierarchy and encapsulation which is crucial for real-life modeling. The nets used for the visual part have been adopted to the metaphors of service calls and resources which makes it possible to specify distributed systems without relying too much on non-visual mechanisms. This is a big advantage compared to almost all other high-level net formalisms.

A working editor and simulator (see figure 9) currently supports the development using OCoNs. First experiences using the tool in software engineering courses and labs are promising. At the moment, an extension to support complex simulations and code synthesis are under development. The integration of the OCoN tools with a tool for static modeling using the UML including consistency checks between static notations and nets as well as the reuse of net analysis tools for OCoNs will be the next steps.

ACKNOWLEDGMENTS

The authors want to thank all students which have been involved in implementing and using the OCoN prototype tools.

References

[1] R. Allen and D. Garlan. A Formal Basis for Architectural Connections. *ACM Transactions on Software Engineering and Methodology*, July 1997.

[2] K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley, Reading, MA, 1996.

[3] E. Battiston, F. D. Cindio, and G. Mauri. Objsa Nets: A Class of High-Level Nets Having Objects as Domains. In G. Rozenberg, editor, *Advances in Petri Nets*, LNCS 424, pages 20–43. Springer Verlag, 1988.

[4] O. Biberstein and D. Buchs. An Object Oriented Specification Language based on Hierarchical Petri Nets. In *IS-CORE Workshop (ESPRIT)*, Amsterdam, Sept. 27-30 1994.

[5] G. Booch. *Object-Oriented Analysis and Design with Applications*. Addison-Wesley, Menlo Park CA, 1993. (Second Edition).

[6] P. Bouvry and F. Arbab. Visifold: A Visual Environment for a Coordination Language. In P. Ciancarini and C. Hankin, editors, *COORDINATION '96, Cesena, Italy*, LNCS 1061, pages 403–406. Springer Verlag, Apr. 1996.

[7] W. Brauer, W. Reisig, and G. Rozenberg [eds]. *Petri Nets: Central Models (part I)/Applications (part II)*. Springer LNCS 254/255, Berlin, 1987.

[8] M. Ceska, V. Janousek, and T. Vojnar. Pntalk - A Computerized Tool for Object Oriented Petri Nets Modeling. In *EUROCAST'97, Las Palmas de Gran Canaria, Canary Islands, Spain*, LNCS 1333. Springer Verlag, 1997.

[9] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification*. Springer, Berlin, 1985.

[10] J. Ellsberger, D. Hogrefe, and A. Sarma. *SDL, Formal Object-oriented Language for Communicating Systems*. Prentice Hall, 1997.

[11] H. J. Genrich and K. Lautenbach. System Modelling with High-Level Petri Nets. *Theor. Comp. Science*, 13:109 – 136, Jan 1981.

[12] H. Giese, J. Graf, and G. Wirtz. Modeling Distributed Software Systems with Object Coordination Nets. In *Proc. Int. Symposium on Software Engineering for Parallel and Distributed Systems – PDSE'98, Kyoto (JP)*. IEEE-CS Press, April 1998.

[13] D. Harel. On Visual Formalisms. *Science of Computer Programming*, 31:514–530, 1988.

[14] J. Hernandez, M. Papathomas, H. M. Murillo, and F. Sanchez. Coordinating Concurrent Objects: How to deal with the coordination aspect? In J. Bosch and S. Mitchell, editors, *Aspect-Oriented Programming Workshop ECOOP'97, Jyväskylä, Finland*, June 1997.

[15] P. Huber, K. Jensen, and R. M. Shapiro. Hierarchies in Coloured Petri Nets. In G. Rozenberg, editor, *Advances in Petri Nets*, LNCS 483, pages 313–341. Springer Verlag, 1990.

[16] International Telecomunication Union (ITU). *Specification and Description Language (SDL) Recommendation Z.100*, 1992.

[17] International Telecomunication Union (ITU) Study Groups CCITT. *Message Sequence Chart (MSC) Recommendation Z.120 draft version*, 1996.

[18] ISO/IEC. *Open Distributed Processing Reference Model - parts 1,2,3,4*, 1995. ISO 10746-1,2,3,4 or ITU-T X.901,2,3,4.

[19] I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process*. Addison Wesley Object Technology Series. Addison-Wesley, 1999.

[20] G. Kiczales. Aspect-oriented programming. *ACM Computing Surveys*, 28(4):154–, Dec. 1996.

[21] C. Lakos. From Coloured Petri Nets to Object Petri Nets. In G. D. Michelis and M. Diaz, editors, *Applications and Theory of Petri Nets 1995, 16th International Conference, Turin, Italy*, LNCS 935. Springer Verlag, June 1995.

[22] C. Lindemann. *Performance Modelling with Deterministic and Stochastic Petri Nets*. John Wiley & Son, 1998.

[23] C. Maier and D. Moldt. Object Colored Petri Nets - a Formal Technique for Object Oriented Modelling. Workshop PNSE'97, Hamburg, Germany, Sept. 1997.

[24] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1997. 2nd edition.

[25] O. Nierstrasz. Regular Types for Active Objects. In O. Nierstrasz and D. Tsichritzis, editors, *Object-Oriented Software Composition*, pages 99–121. Prentice Hall, 1995.

[26] Object Management Group. *The Common Object Request Broker: Architecture and Specification, CORBA/IIOP 2.2 Specification*, Feb. 1998. Revision 2.2: OMG Technical Document formal/98-07-01.

[27] Rational Software Corporation. *Unified Modeling Language 1.1*, Sept. 1997.

[28] T. Reenskaug. *Working with Objects: The OOram Software Engineering Method*. Addison-Wesley/Manning, 1996.

[29] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, 1991.

[30] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an emerging Discipline*. Prentice Hall, 1996.

[31] C. Sibertin-Blanc. Cooperative NETs. In R. Valette, editor, *Applications and Theory of Petri Nets 1994, 15th International Conference, Zaragoza, Spain*, LNCS 815, pages 471–490. Springer Verlag, June 1994.

[32] N. Stankovic and K. Zhang. Towards Visual Development of Message-Passing Programs. In *Proc. VL'97 - 13th IEEE Symposium on Visual Languages, Capri, Italy, 23-26 September, 1997, IEEE Computer Society Press, Los Alamitos, USA*, pages 144–151, 1997.

[33] M. M. Usher and D. Jackson. A Concurrent Visual Language Based on Petri Nets. In *1998 IEEE Symposium on Visual Languages (VL 98), Halifax, Nova Scotia, Canada*, Sept. 1998.

[34] R. Wirfs-Brock, B. Wilkerson, and L. Wiener. *Designing Object-Oriented Software*. Prentice Hall, 1990.

[35] G. Wirtz. Modularization and Process Replication in a Visual Parallel Programming Language. In A. L. Ambler and T. D. Kimura, editors, *Proc. Int. Symp. on Visual Programming, St. Louis, USA*, pages 72–79. IEEE, Sept. 1994.

[36] G. Wirtz, J. Graf, and H. Giese. Ruling the Behavior of Distributed Software Components. In H. R. Arabnia, editor, *Proc. Int. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA'97), Las Vegas, Nevada*, July 1997.