Software engineering for distributed systems is a notoriously hard problem. While common approaches for structural modeling like object-oriented analysis and design offer several benefits, their current support for distributed system design is rather limited. The visual notations for structural modeling can be adjusted to the demands of distributed systems. The behavior modeling notations fail w.r.t. the most crucial aspects. Often the domain-specific behavior notations that have been proposed in this context for the UML neither support concurrency as needed nor do they allow to describe scalable behavior. An object-oriented design technique based on UML notations and a special type of high-level Petri-Nets that overcomes these limitations is presented. It is demonstrated how a visual design language can support the crucial aspects for distributed system design as well as how these aspects can be smoothly integrated into a single language with multiple consistent views.

1. Introduction

The current boom to the internet in the form of e-commerce or supply chain support, etc., does further increase the demand for software engineering of distributed systems. Object-oriented technologies like CORBA, DCOM or Java provide suitable middleware support that allow to consider the design problem on a more conceptual level. The UML as a unification attempt for object-oriented analysis and design notations does further provide a standard instrument for conceptual modeling.

But software engineering for distributed systems is a notoriously hard problem and the common handling of structural modeling during object-oriented analysis has to be adjusted to support distributed systems. While this can be done straightforwardly for the visual structure modeling notations, the situation is less pleasant for behavior modeling. The provided notations fail to reflect distributed systems w.r.t. the most crucial aspects. These behavior notations that have been generally developed in a different domain-specific context do not support the essential elements and techniques for distributed systems and thus fail to describe a system in such a way that designs can be discussed.

The described approach extends the UML using a special type of high-level Petri-Nets. In contrast to state machine models, the underlying Petri-Net model is well suited to describe concurrency and distribution. The often irritating general concepts of
high-level Petri-Nets have been adjusted as well as hidden and the resulting language provides a rather simple and understandable notation that still preserves the ability of Petri-Nets to express concurrency and distribution.

In Section 2, the basic requirements that a visual language for distributed system design has to challenge are discussed. The basic concepts of the approach and a short overview are presented in Section 3. Then, an example is presented and several design alternatives are discussed in Section 4. The specific benefits are considered step by step in Section 5 using the given example. The pure UML, suggested extensions as well as other related work are compared in Section 6 and a final conclusion and outlook close the article.

2. Requirements

The nature of distributed systems results in a set of considerable additional problems not present in traditional sequential systems. Roughly stated these are the phenomena of distribution, heterogeneity, coordination, resource management and partial failures. A rather implicit resulting property is concurrency that adds its share to the complexity of handling distributed design problems. While several problems can be handled using the available technology, others remain inherently problematic and have to be tackled in a more explicit fashion. Distribution and system heterogeneity are handled by applying access and location transparency offered by today’s middleware systems. The resulting common interface typing does result in considerable restrictions and fails to support fully independent evolution; in practice it allows to abstract from these phenomena most often. But the remaining aspects have resisted all attempts to handle them in a comparable transparent way without resulting in unacceptable solutions (cf. [1]).

The coordination of the system processes on different nodes implies concurrent behavior and has to handle its additional complexity as well as exploit the potential benefit of parallel processing where necessary. The great difference between network latency and computational speed makes asynchronous operating as well as parallelism in several forms mandatory when scalability or higher throughput requirements are required.

The resource sharing is a rather economical option that distributed systems imply. In practice, a more explicit resource handling can result in considerable design overhead; on the other hand, abstracting from these resources can result in considerable problems. Resources and their coordinated access is one of the most important aspects that determines the possible system performance and scalability.

If the design incorporates the resource treatment, simulating the abstract model provides a suitable intuition whether the final resulting system operates as intended as well as whether for a given load scenario the necessary throughput is achievable. Thus, the simulation of abstract models can help to evaluate these aspects already during the design, while of course the difference between abstract model and the later resulting implementation may make such estimations worthless. The abstract model semantics has to allow to make estimations in such a way that a useful class of results holds for any correct realization if detected for the abstract model.

Although not a specific problem of distributed systems, modularity is a rather general problem for engineering software systems. Some support for separation that goes beyond the syntactical interfaces of the common middleware systems seems necessary here, too.
The problem of partial failures as well as system consistency is one of the really hard problems. Distributed transaction processing as well as reliable messaging are features supported by current middleware platforms. The rather big challenge is how to integrate them into the design. Especially for behavioral notations that incorporate several steps into an atomic transaction and roll-backs controlled externally are difficult to consider. When different concepts are occurring in intermixed forms the problem becomes even harder (cf. [2]).

A suitable language for the design of distributed systems has thus to support the design of these aspects in a more or less explicit fashion, because abstracting from them has been observed to result in a high risk that the resulting design does not fulfill its requirements.

### 3. The OCoN Approach

The object-coordination-net (OCoN) approach introduced in [3,4] tries to overcome the drawbacks of the UML by means of providing appropriate visual formalisms based on Petri-Nets [5] for behavioral descriptions which are seamlessly integrated with the concepts of object orientation, concurrency aspects and resource coordination in distributed systems (see [6]). Moreover, the very basis of the approach puts its focus on a contract-based system design [7]. In order to integrate the approach with the suitable parts of the UML [8], we add only specific elements as well as new diagrams where necessary. When dealing with static structure this can be done using the UML extension mechanisms, but, as presented in [9], the UML notations for behavior modeling are not that suitable.

In Figure 1, the basic elements of the additional diagrams are presented. An action is visualized by a square and represents in its basic form an operation call with request and reply parts. We further distinguish two sorts of pools in the form of event pools (circles) and resource pools (hexagons). The former are used to denote the control flow while the latter represent the resource environment. If the resource is not controlled exclusively within the given net a shared pool indicated by the double border is used. The flow relation of the net is described using pre- and post-condition arcs that denote which resources are consumed and produced by each action. The carrier of activity and the action-executing entity is denoted by a special activation edge with white instead of black head. The actions representing an operation call are further equipped with a signature using ports that represent the parameters (white dot) as well as the replied results (black dot). If in contrast an action is drawn with a shadow and without signature it describes an externally initiated activity. This may be for example an operation call viewed from within an instance (call forward action) or an autonomous state change visible from outside (autonomous contract step).

The common interface notion is extended to contracts by the so-called protocol net (PN) to specify non-uniform service availability. They are used to specify externally visible behavior and to decouple systems by clearly distinguishing between an interface and its implementation which may again use protocols of other system parts. An instance of a class (visualized in Figure 2), for example, provides a contract to the outside which is fulfilled by implementing its public services and an instance-local description of which resources are needed and how they are coordinated. This is done by the so-called resource
allocation net (RAN). If a single service contains interesting internal parallelism or relevant resource allocation steps, this can be specified by the so-called service net (SN), otherwise a method implemented, e.g. using Java can be used which provides an interface to legacy code. We assume the reader to be familiar with the most important diagrams of the UML (or its predecessors) and with basic Petri-Nets [5]. The details of the OCoN language described, for example, in [4] will be provided informally wherever used in the following example.

4. Example

As an example we will consider the design for a component of the decision support system. Its functionality includes the management of a given set of information resources like mail, news and channels as well as client-initiated data query processing whose results are integrated into the information base. Besides the necessary data processing and data management, this component has to fulfill several complex coordination tasks: the client wants to be informed when new relevant information arrives and the component has to autonomously retrieve useful information sources on behalf of its clients. The agent [10] concept can be applied to fulfill the identified requirements resulting in a rather autonomous operating entity.
In Figure 3, the relevant structural details of the design are presented. The offered Agent contract provides the intended functionality to a client. It is implemented by the implementation class AgentImpl using several other structural elements. Notable at first are the subsystem-wide-known shared resources of a mail server, news server and a channel. They are used by the AgentImpl class to retrieve interesting information periodically. To handle specific user-initiated tasks a special Task contract as well as a realization class TaskImpl are given. To store and manage the retrieved information an ArchiveImpl class accessed via its Archive contract is used.

The provided Agent contract is presented in Figure 4. In its state [actual] the operations select, status as well as addTask are available. Via a select request the information chunk (message) determined by an Msg::Id can be selected and obtained from the archive. A uniquely identified information chunk can be virtually modified by a specific client.
using the status command. If, for instance, it is marked for deletion any further interaction of that client will ignore these chunks of information. The assigned flags for an information chunk are further used to estimate their relevance and thus messages deleted by many clients may be thrown out first when the agents information base has to be reduced. If a message is created or received by a specific client it is initially visible only for the client. The client can either make it visible for other users or erase it using the status command.

When a client wants to initiate specific queries or analysis tasks she can do so with the addTask operation using an appropriate task description (TaskDesc). The result will be added to the information base of the agent and thus made available even to other users later if wanted.

When a new message is received or a task is completed, the contract state autonomously changes to the state [updated]. This is represented in the protocol net using a gray action with shadow indicating that the action may occur arbitrarily (quiescent autonomous contract step).

In the state [updated] the client can synchronize its information base with the agent and obtain all new message id’s using the update operation.

The contract Task (see Figure 4) is internally used by the client to manage a task. The initial state [working] represents that a newly created task is actually processing. When the task is either processed successfully or aborted, this will be represented using the state

---

<table>
<thead>
<tr>
<th>AgentImpl1</th>
</tr>
</thead>
<tbody>
<tr>
<td>+update():(set\langle Msg::Id\rangle) --processTask():()</td>
</tr>
<tr>
<td>+select((Msg::Id):(Msg)) --query():()</td>
</tr>
<tr>
<td>+status(Msg::Id,Msg::Flag):() --delay():()</td>
</tr>
<tr>
<td>+addtask(TaskDesc):()</td>
</tr>
</tbody>
</table>

---

**Figure 5.** A first realization AgentImpl1
A progress autonomous contract step (white square with shadow) is used to denote that this will definitely happen in the future. Note that in contrast to the quiescent autonomous behavior, this time progress is guaranteed. Finally, an observer can obtain the result with the getResult operation when the state [done] is reached. After reading the result, the Task contract does not provide any further interaction.

The first solution presented in Figure 5 provides only a single exclusive Agent contract and essentially realizes three coordination aspects. The skeleton for the provided Agent contract and its processing is described by the anonymous [actual] and [updated] pools. Two pools are used to represent the observed tasks. Two event pools and a cycling event are used to trigger the periodical retrieving of the shared mail, news and channel server (query). The archive resource is only allocated by the different activities, but implies no specific coordination demand.

The right upper part of the resource allocation net builds a control loop initially filled with a single token. The private internal operation delay is initially enabled and will be terminated after a not-further-specified amount of time is elapsed. Then, the private method query is also started that describes how the subsystem infrastructure containing a mail server, a news server and an information channel is retrieved. The archive representing the agent’s information base is thus initially locked and filled with newly obtained information. As an additional side effect, the provided Agent contract is changed from [actual] to [updated] to indicate to the observing client that new information chunks have been obtained. Note that in the external protocol the internally triggered query operation is w.r.t. its effect on the Agent contract state represented by the quiescent autonomous contract step (see Figure 4, left).

In the left upper part the initiated tasks are observed and if terminated ([done]), their result is added to the AgentImpl information base by the private processTask operation as done for the query operation. Again, the state of the offered contract is also adjusted accordingly.

In the lower part containing the [actual] and [updated] pools as well as the call forward actions update, addTask, status and select the offered Agent contract is realized. A skeleton
token either in the [actual] or [updated] pool represents the current state of the Agent contract. The arbitrary autonomous state change depends on either the left upper or right upper part and the operation processTask or query may occur as described before. Note that for these private operations the instance itself is the implicit carrier of activity, while for external calls the target of the request has to be specified explicitly. While the status and select operations also demand exclusive access to the archive resource, the addTask operation simply adds an additional task instance to the tasks [working] pool.

The described solution combines the reactive serving of the Agent contract with the polling strategy for the retrieving of the mail, news and channel server. Also, the observation of autonomous Task contracts is described within the same context. These three aspects and their suitable coordination is solved using the provided resource allocation net. But besides the overall coordination aspects and their interference described for an instance in its RAN, also the specific behavior of a single operation has to be described.

The so-called service nets as presented in Figure 6 are used for this purpose. In the AgentImpl1::query service net the strength of Petri-Nets w.r.t. explicit parallel control flow can be seen. The internally initiated query operation further on splits into three parallel
Figure 8. The adjusted query service net

Figure 9. A multi-contract-serving version
threads each processing one of the mail, news and channel server. Their results are added to the overall archive exclusively locked in the resource allocation net and the operation is finally terminated. Note that the accessed context resources `mailServer`, `news` and `channels` are made visible by their shared resource pools.
To demonstrate the benefits of the OCoN approach we will further on consider different alternative designs for the AgentImpl class.

### 4.1. Alternative Designs

While the presented solution does ensure the intended behavior in a consistent way via locking the archive resource where ever needed, it results in a coarse grain locking scheme that is of rather limited scalability w.r.t. the performance for multiple requests and can result in considerable delays for client requests. If the query operation takes considerable time, the client requests will be blocked which is not acceptable when an interactive client application is considered. Even for non-distributed applications such stalls have to be excluded where possible. This problem can be avoided by further distinguishing the query operation and the archive update w.r.t. locking.

By splitting the core query processing of the mail, news and channel server and adding the results to the archive this problem can be circumvented. A query operation that returns a non-empty set of newly retrieved messages and a processUpdate operation that takes such a set are used to decouple the processing (cf. Figure 7). We also change the processTask operation accordingly to avoid doubling the code describing how to add
messages to the agent information base \((\text{archive})\). The locking of the archive resource has thus been restricted to the operations of the update part.

The query operation has also to be adjusted (see Figure 8). Instead of adding the retrieved results directly they are first united in a temporary archive \((\text{tmp})\) and this is returned if not empty. Otherwise no return parameter is provided.

While already showing an improvement, the coarse grain locking of solution \(\text{AgentImpl2}\) as well as that of \(\text{AgentImpl1}\) will fail to provide the needed behavior if instead of a single agent contract multiple contracts have to be served.

Such a version serving multiple contracts at once is presented in Figure 9. The locking scheme has to be further improved and some sort of fine grain locking allowing parallel access where suitable is provided.

Instead of a single \(\text{archive}\) token the information base may be described using a distributed data structure represented by many independent archive elements. Thus, a select or status operation will only lock the necessary element. In the resource allocation net a corresponding selection expression using the \text{id}\ parameter of the call is used to lock only the needed element.

To serve multiple instead of a single exclusive Agent contracts also demand that instead of a single contract all of them have to be adjusted when new information is available. Instead of usual arcs, set-valued arcs described using the UML multiplicity annotation \(0..*\) are used. The \(*\) denotes that all existing elements are consumed rather than only that

Figure 15. Hierarchy of information agents

currently available. In figure 9, all contracts in state [actual] as well as [updated] are consumed and set to the state [updated] to exclude parallel processed update calls with inconsistent behavior. Note that thus pending requests may be delayed until the update is processed.

When these selections cannot be made *a priori* when starting the service, additional locking steps in the service nets have to be used. In Figure 10, the corresponding processing is described. This time an internal atomic action (only a single square) and a set-valued post-condition arc are used to denote splitting of the set of messages into different archive elements each build using the ArchiveImpl constructor.

A far more suitable solution is to avoid external locking of complex objects and instead let the objects handle concurrent updates on their own. The specific structure can then be exploited by the object implementation to achieve parallelism where suitable and to protect the object-specific consistency and semantics.

The resulting resource allocation net of Figure 11 is further simplified, because the resource allocation w.r.t. the archive is not needed any more. Note, however, that consistent updates are now only possible w.r.t. the granularity offered by the Archive contract. If multiple requests have to be combined to a consistent update, we have again to implement them using external locks or *transaction concepts* considered later.

The processUpdate operation realization in the form of a service net has to be adjusted also (see Figure 12). Here, a provided operation addSet of the modified Archive contract is used to ensure the consistent update of the archive w.r.t. to a given set of new messages.

While the considered solutions are able to handle the requirements of a medium-size configuration, for handling larger volumes of information we need more advanced strategies.

The modified version of the InformationAgent subsystem permits more than 1 stage of agents that filters relevant data pre-processed and provided by another Agent (see Figure 13).
A corresponding realization AgentImpl5 for the Agent contract is shown in Figure 14. Instead of the global mail, news or channel service this time a set of Agent contracts and its update information is used to retrieve new relevant information. Thus, the right upper part contains two pools representing the different external states of the subagents contracts. They are observed and if one contract enters the [updated] state, the internal operation getUpdate is initiated.
A layered hierarchical composition of InformationAgent and InformationAgent2 subsystems can then be used to establish the necessary infrastructure for a company-wide decision support system that effectively shares common interests of several people in the form of shared used information agents within the layered structure (cf. Figure (15)). The overall processing might be optimized by further modifying the structure, e.g. increasing the hierarchy for very busy agents.

5. Discussion

After presenting some design alternatives for the considered example of an agent for decision support we want to focus on the visual aspects of the presented OCoN approach.

The coordination aspect and especially concurrency and parallelism are supported in multiple ways. Parallel processing can be made explicit in a service net when multiple independent servers can be concurrently used (see Figures 6 and 8). Also independent parts of a resource allocation net can describe explicit parallel processing. See, for example, the task and query processing in Figures 5, 7, 9, 11 and 14. But also some sort of implicit parallelism is expressed in a net when the available tokens enable actions multiple times. See the Task tokens of Figure 5 as well as the used Agent contracts in Figure 14. When different or the same action is occurring multiple times in parallel like, e.g. for the multiple provided Agent contracts in Figure 9, a number of concurrently executed service nets can be initiated. While these are independent net instances, the overall OCoN semantics still ensures a consistent overall behavior where of course conflicts are possible.

The explicit management of resources and its implication for concurrent or non-concurrent processing is expressed in all OCoN nets with resource pools. Instead of single associations their set-valued management and related operation is considered. For example the left upper part of Figure 5 describes that the processTask action is activated if any and not a specific task is done. This way the implicit parallelism possible due to independently used and modified resources results as a natural option.

The OCoN design models are executable and thus can be evaluated to detect performance bottlenecks, exclude synchronization anomalies or check if the behavior is the intended one. See [11] for a more in-depth example on this topic. In Figure 16 such a tool-based procedure is presented using a screen-shot of the tool. The design version of Figure 5 and 6 is presented and the shown instance state with one active query service net visualizes that no external or internal operation can occur in parallel. Thus, the identified coarse grain locking becomes apparent.

Whereas operational semantics are in practice often suitable, every abstract behavior description language does not also allow to detect problems that may also hold for a final implementation. The OCoN language has been designed with emphasis on this aspect to achieve a suitable abstract operational view even for high-level behavior descriptions. Most visual description techniques use some form of textual guards to describe conditional behavior using a well-defined annotation language or even plain text. The latter results in non-executable semantics, while the former enforces to even evaluate the guards and thus the referenced attributed values and related updates have to be considered, too. While possible, this also implies that such updates as well as all
guards are fully specified. Even worse, later changes are only correct refinements when the guards as well as updates are also refinements. Thus, the designer has to establish a more or less logical implication between general expressions, which is a rather complex task.

The OCoN language design, in contrast to non-abstract guards, enforces to use abstract decision actions that provide a set of alternative replies. See, for example, the *query* operation used in Figure 7 that either results in a set of new messages or does not. The realization via a service net is shown in Figure 8. The important difference w.r.t. executing the abstract model is, that independent of any concrete implementation the overall correct processing as well as possible performance problems can be considered. Even when the implementation actually always results in a non-empty set of new messages, the *non-deterministic* action semantics is still valid. Thus, a suitable refinement is obviously to reduce the number of alternative outputs, while adding a new alternative reply changes the fundamental behavior of the decision action and thus the embedding structure has to be evaluated again. In contrast to a textual guard and update languages, the notion of refinement is rather simple and intuitive. The final and sometimes even heuristic realizations for decision actions can vary without leading to in principle different coordination behavior.

*Modularity* is a rather general requirement that engineered software systems should fulfill in order to support their further evolution and change. The OCoN language supports this requirement in the form of the *contract* notion. The pure syntactical interface notion supported by most middleware approaches is not sufficient at the semantics level. The notion of *design by contract* [12] has been proposed while the provided semantical aspects are pre- and post-conditions which fail to provide the necessary synchronization information essential for distributed systems. Non-uniform service availability has been identified by several researchers as an essential extension of type system w.r.t. concurrency and distributed systems (cf. [13–15]). Especially, w.r.t. the coarse-grain structure and the overall *software architecture* [16], higher-level behavioral models and the consideration of *connectors* and related protocols [17] has been proposed. The OCoN contract notion integrates these concepts smoothly [6] into the language. This results in the ability to consider even system fragments using the border built by the contracts and its protocols to simulate the behavior w.r.t. the outside-world interaction.

System *reliability* in distributed systems is a really hard problem. While in sequential systems only total failures of the overall system have to be considered, we have to face the case that parts of the system fail independently and in any combination. Thus, an interaction, e.g., in the form of a basic remote procedure call might fail at any point. In practice, only *at-most-once* semantic is usually provided and thus repeating messages in case of possible failures seems possible. This again can result in considerable problems when the offered operations are not *idempotent* and the whole interaction is not stateless. As general solutions either *reliable messaging* or *distributed transactions* using the *two-phase commit* protocol are integrated in today’s middleware infrastructures, while their integration into the abstract behavioral model is quite difficult.

For the OCoN approach these strategies can be orthogonally applied. The single atomic steps given by an action request or reply including their resource consumption and production can be made a transaction if needed. Then, the resulting failure cases can exclude any loss of information. Also, we can group the different actions combined in
a single transaction using the so-called nested transactions to obtain more coarse-grain transactional behavior. When a resource contract supports distributed transactions this further ensures that also the related resource updates are rolled-back if needed. Note, however, that either explicit locking for service nets as described in Figure 5 or the usage of self-managed complex objects that also support nested transactions is needed.

Another more asynchronous model is possible when the middleware provides exactly once semantics in the form of reliable messaging. Thus, the local stepwise atomic processing can be combined with remote entities in a secure fashion without the problem of accidentally repeated messages.

Both extensions are rather expensive w.r.t. processing resources and most often in distributed systems the problem of detecting or ignoring repeated messages or events can be more efficiently solved on the application level itself. Imagine that in our example the considered messages (information chunks) are equipped with a unique identifier and the connection between a client and its information agent is arbitrarily broken. While in rarely occurring failure scenarios for distributed transactions even the client has to wait for a crashed coordinator after agreeing on a commit, reliable messaging excludes such coupling effects. But a rather simple and straightforward solution is simply to reestablish the connection to the agent and reread the old update. If messages have been considered before, but the related status update has failed to reach the agent, the client himself can correct this. That the agent is idempotent with respect to single repeated requests using the inherent unique identifiers can be achieved easily.

In more complex cases it is even more useful to establish another protocol level in between the agent and its client that solves specific problems. For example, the real network-related protocol between the client and an agent can be adjusted to further simplify or optimize the network processing w.r.t. the regular or failure case. The Jini [18] approach for distributed systems with Java emphasizes the usage of smart proxies instead of thumb network references to handle these cases.

6. Related Work

In the context of visual languages and the UML especially the notion of state diagrams is relevant. The underlying model of finite state machines and statecharts [19] leads to some restrictions not suitable for the domain of distributed software systems. The UML state diagram notation does not support implicit intra-instance-concurrency as required. Intra-object concurrency, e.g., in the form of multiple concurrent threads processing different requests in the form of methods has to be encoded explicitly into special method objects (cf. [20]). Multiple concurrent objects are possible, but the provided notations do not provide a sufficient view where common resources of the object are naturally shared. Instead, each thread object has to access them via their external interface just like any other object. The state diagrams and the related notion of activity graphs have been extended with some Petri-Net-related extensions like sync-states while their interpretation in terms of the object context is missing. They are like the other elements used to construct the stepwise behavior and the related state space explicitly. The integration of resources and operations is rather minimal. The so-called flow objects can be used in activity graphs to describe their flow, but the concrete position in the object
context determining whether they are an association of the object or an element of the current execution context is left open.

The importance of executable specifications has also been identified by the UML community and an approach to standardize an annotation language as well as an executable model [21] are currently developed. But as argued before, defining an annotation language and using textual guards as done by state diagrams and activity graphs does exclude the needed flexibility w.r.t. the final realization and cannot provide a simple and manageable notion for correct refinements.

Another prominent approach for behavior modeling in the telecommunication domain are SDL [22] and MSCs [23]. The latter are actually observations or a folded set of observations and are the paragon for the UML sequence diagrams. For both it holds that deriving an executable model using a given set of observations is only possible in very restricted cases. And, even more critical, the important question of how to handle concurrent cases is usually not defined. SDL is in contrast a message-passing-based approach describing the control flow in a flow chart like notation containing also message send and receive elements. Note that also the UML activity diagrams have been extended with some of these constructs and that SDL also integrates some compatibility with the UML [24]. While the reactive behavior in the form of message receives, and related activity can be described straightforwardly the notion only supports explicit parallel processing. Additional structural extensions and multiple parallel processes have to be specified explicitly like in the case of state diagrams when implicit parallelism is needed. Also, any support for resource modeling as well as behavior depending on the available resources is missing.

UML and SDL fail already to model parallel processing as needed. Only the explicit parallel processing is supported while implicit forms like those described by multiple enabled actions in a net are not possible. This further excludes that the behavior description itself can be considered w.r.t. different resource scenarios without having to change the structure, which is simply done for nets by adjusting the initial marking. Thus, the resulting behavior descriptions are not scalable. An executable UML is under development and SDL models and tools support execution, but the use of textual guards and annotations results in rather concrete specifications which are more of the programming language level than abstract designs. Thus, they can be used to evaluate the final realization in combination with code synthesis. Early evaluation as intended by our approach is not possible.

7. Conclusion and Future Work

The presented OCoN approach has been developed in an ongoing project for four years and has been proven appropriate in a number of projects. Besides the core domain of distributed software systems [4,11] also distributed workflow [25,26] and embedded systems [27] have been approached. Also, extensions towards the component-based design of distributed systems have been developed [7].

While conceptually matured, the tool support is still rather limited. Besides the improvements in this field, the further development of run-time system support in the form of a framework that simplifies the mapping of constructed nets onto suitable run-time elements is actually under way. With respect to the language itself the tighter integration of distributed transactions and reliable messaging is currently considered.
Providing different failure models that are integrated into the semantics itself, the evaluation of a single design w.r.t. different failure models is a promising goal. Then, the impact of such design decisions on a system's scalability and performance can be estimated.

Acknowledgements

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

References


