Peer-Based Programming Model for Coordination Patterns

Eva Kühn, Stefan Crass, Gerson Joskowicz, Alexander Marek, Thomas Scheller

Peer-based Programming Model for Coordination Patterns

eva Kühn, Stefan Craß, Gerson Joskowicz,
Alexander Marek, and Thomas Scheller

Vienna University of Technology, Institute of Computer Languages
Argentinierstr. 8, 1040 Vienna, Austria
{eva,sc,josko,amarek,ts}@complang.tuwien.ac.at

Abstract. Modern distributed software systems must integrate in near-time parallel processes and heterogeneous information sources provided by active, autonomous software systems. Such lively information sources are e.g. sensory data, weather data, traffic data, or booking data, operated by independent distributed sites. The complex integration requires the coordination of these data flows to guarantee consistent global semantics. Design, implementation, analysis and control of distributed concurrent systems are notoriously complex tasks. Petri Nets are widely used to model concurrent activities. However, a higher-level programming abstraction is needed. We propose a new programming model for modeling concurrent coordination patterns, which is based on the idea of "peer workers" that represent re-usable coordination and application components. These components encapsulate behavior, structure distributed data and control flow, and allow integration of pre-existing service functions. A domain-specific language is presented. The usability of the peer-based programming model is evaluated with the Split/Join pattern.

1 Introduction

A common problem in software projects is the tightrope walk between design and implementation. While the designer's focus is to keep the system clean and verifiable under all possible operating conditions, the developer's main intent is getting things done while dealing with exceptional conditions in later refinements. The *Peer Model* that we introduce in this paper tries to bridge this gap, at least within a specified problem domain, by providing a domain-specific modeling language to design, analyze, and implement system integration patterns in distributed environments using predefined and application-specific components.

The need for methodologies that connect verifiable designs and implementations occurs especially in systems where a multitude of autonomous systems are cooperating to fulfill a specific task. One example that illustrates these problems is in the realm of distributed firewall configuration: firewalls belong to different organizations and the rollout of a new configuration must guarantee that no inconsistencies take place. Errors occur during the rollout process when sites are down, the network is unavailable, local configuration constraints are violated,

access rights are not granted etc. A modeling tool is required to specify the configuration process. It must allow modeling when a valid state is reached, and at which time and under which conditions new configurations may become active in the entire network or in parts of it. This will be the case if e.g., a certain number or a majority of firewall sites reported back that they have implemented the updates. Network configuration is a dynamic and quite complex task with many different error situations. Failure impact analysis shall be possible for all different configuration possibilities. It is therefore desirable that this process can be modeled, verified, and executed. A service-oriented security concept for the coordination of such firewalls has been proposed in [6], whereas in this paper, we investigate a new model for specification and implementation of complex coordination and integration patterns.

A further example that motivated the design of the Peer Model is from the rail traffic management domain. Here, measurement systems – like wheel sensors or RFID sensors – along the tracks generate information about approaching trains. This data is transported over mesh networks. A use case is for example a traffic control center that queries distributed sensors, collects answers, and aggregates them to a gain a sufficiently consistent view about the technical status of a train. As sensors are low cost components, wireless communication is error prone, and field conditions are harsh, many failure situations must be coped with.

Both examples have a basic pattern in common that consists of the following steps: 1) splitting of a task into individual subtasks to be executed by different nodes (firewalls resp. sensors), and 2) joining of received answers according to complex rules and under the assumption of failures, in order to derive a global decision. We term this recurring situation the *Split/Join pattern* and will use it as an example throughout this paper.

Many modeling methodologies for the purpose of designing concurrent software systems have been proposed, such as Petri Nets [1], Reo [3] and UML MARTE [2]. These methodologies are general and not domain-specific, as they are used to solve almost any software problem. On the one hand, that is what makes them powerful as the designer *can model everything*, but on the other hand, the designer also *does have to model everything* and cannot assume any functionality to be present.

As the Peer Model is targeted at modeling coordination patterns in distributed environments, it can make several assumptions on the target system. For example, it assumes a tuple space-based communication middleware to be present and can therefore resort to transactional transmissions of data between communicating components and to coordination mechanisms for accessing distributed data structures. Additionally, it follows a component-based approach and assumes several predefined, reusable components to be present. Designers can also structure their scenario-specific coordination logic into new reusable components, rearranging the resulting models more clearly and providing better scalability when additional logic is added. New components can also be created by combining existing components and using them as sub-components, thus leading to a nested component structure. It is important to note that only coor-

dination logic is modeled, not business logic. One weakness in Petri Net models of concurrent systems is that the resulting models mix business logic and coordination logic, thus reducing their readability and maintainability. For that reason our model strictly forbids defining arbitrary business logic within the model. Rather, it integrates and connects external services, thereby separating clearly concurrency, parallelism and distribution mechanisms of the collaborating components from the business logic.

The contribution of this paper is a higher-level programming and design model termed "Peer Model", or "PM" for short, aiming to ease the design and implementation of complex integration patterns in distributed environments. We introduce its Domain Specific Language (DSL), meta-model and graphical notation, but not its implementation itself and prove our claims by using its DSL to model a well-known pattern and comparing the resulting model to equivalent ones using Reo [3] and Colored Petri Net [10] notation.

In Section 2 we introduce the Peer Model along with its graphical notation, Section 3 introduces the pattern used for evaluation purposes, Section 4 gives an overview of the Peer Model's meta-model and DSL. In Section 5 we compare our resulting model of the pattern with those of other modeling languages and Section 6 finally points out our conclusion and future work.

2 Peer Model

The Peer Model is a novel programming model targeted at developers of highly concurrent applications. Its design is inspired by asynchronous message queue or tuple space communication, staged event-driven architecture, and data-driven workflow. In the Peer Model, *space containers* realize stages; a construct termed *peer* is introduced that represents a structured, re-usable, addressable component. It is modeled by means of two stages for input and output. Between them the internal logic of the peer takes place. Inter-peer collaboration occurs between output stage and a foreign peer's input stage. Stages thus mark end-points of distribution. *Wirings* model the flow between stages within and between peers. Data belonging to the same flow is marked with a unique *flow identifier*.

The conception of the Peer Model assumes a tuple-space [8] to bootstrap the model. Space-based middleware lets autonomous components coordinate themselves in a highly decoupled way. We use a space as described in [7], which provides shared *containers* that support configurable coordination mechanisms [11], and a flexible and extensible API for accessing them, although other spaces with similar functionality may also be used to bootstrap the Peer Model.

Data and requests are modeled as *entries* maintained in containers. An entry has an *application*- and a *coordination*-specific data part, termed *app-data* resp. *co-data*. The latter holds meta-information for system-internal mechanisms like entry selection and transactions, and is identified by labeled properties. The entry type is explicitly modeled by a *type property*, which is also queryable co-data. Note that the entry type differs from the type of the app-data transported by this entry. The entry type serves for coordinating the flow in the system.

Fig. 1. (a) Peer. (b) Space Peer.

A container is referenced by a URI in the network. It provides (1) *put* operations that *write* new entries into a container, and (2) *get* operations that *read* or *take* entries. The coordination principle (e.g. first-in-first-out order, key, template matching, SQL-like query etc.) by which entries are administrated and queried in the container is controlled by the entries' meta information. Read and take operations select entries according to the chosen coordination principle. They wait within the given timeout for the query to complete. In addition, take also removes the entries from the container. Put and get operations are carried out in transactions; both support bulk data processing. The write operation puts entries into a container in a single step. Read and take retrieve a certain amount of entries which is controlled by a counter specification as explained below.

The container is a basic building block for the specification of the Peer Model. Special capabilities of the container implementation like persistency and authorization [5, 6] influence the quality of the Peer Model with respect to reliability, consistency and security. However, these issues are out of the scope of this paper. For the following, we only assume two very basic coordination laws: (i) selection of entries by means of their type property, and (ii) specification of the amount of entries required to fulfill a query, otherwise the operation will block. The amount of required entries is expressed as a relation that represents the exact number ("="), a minimum (">", ">=") or a maximum ("<", "<=") of required entries. The semantics is to always take as much entries as possible.

2.1 Peer

A *peer* P is the main resource of the Peer Model. The graphical notation for the basic form of a peer is shown in Figure 1.a. It controls the execution of services. A peer possesses a name (URI), an input space termed *peer-in-container* (*PIC*), and an output space termed *peer-out-container* (*POC*). Via the PIC it receives request entries, takes them out of the PIC, processes them, and finally puts replies into the POC. The PIC restricts the entries it accepts to pre-defined types. The behavior of a peer is modeled by means of nested *sub-peers*, *wirings* and *services* (see Section 2.2). A sub-peer is a peer that is created in the scope of a peer. A wiring allows incorporating zero or more services into a peer. It waits until a defined selection of entries is available, invokes services, and moves or copies entries between the peer's PIC, its POC, and/or PICs and POCs of direct sub-peers. Therefore, wirings are the only active part of the system.

Major peer derivatives are: A *space peer (SPA)* as shown in Figure 1.b is a specialized peer that models the functionality of a space container. Its PIC and POC are melted into one place where all incoming entries are stored using the

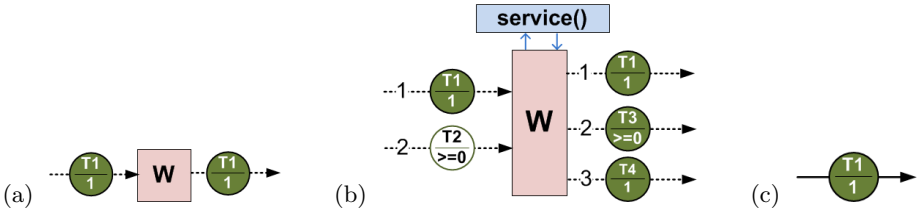

Fig. 2. Wirings.

write operation, and from which they can be read or taken. A SPA is used to store data that are shared between concurrent threads and/or processes, i.e. it serves as medium for communication and synchronization. In the Split/Join pattern example (see Section 3, Figure 5.b) it will be used to collect the workers' answers to be merged by a join peer. A *coordination peer (COP)* is a predefined system peer encapsulating re-usable coordination logic; examples are lookup, routing and filtering peers. An *application peer (APP)* includes application-specific logic provided by developers in form of service methods. Together, all peers of a site form a "Peer Space", whose runtime environment is bootstrapped via a *runtime peer (RTP)*, the name of which refers to the URI of the local site. It enables the dynamic creation and deletion of peers and their composition through wirings and services at the local site.

2.2 Wiring

Wirings are the active part of the system: They are the transport system within the Peer Model, controlling the movement of entries between PIC and POC containers of peers with the help of get and put space operations. A wiring has a name and consists of three sections: a *guard*, which defines the conditions under which the wiring activates, optionally followed by *service calls*, and an *action* that defines how to dispose the resulting entries. A guard resp. action section consists of 0, 1, or more *link operations*. As an extra condition, a guard must contain at least one consuming get operation (i.e. take).

The graphical notation of the control part of a wiring named "W" is represented in Figure 2.a, which illustrates a wiring with one input and one output link that are represented by dotted arrows. Figure 2.b shows a wiring with multiple input and output links that also calls a service. The notation for entries consists of a type query in the upper part and a counter query in the lower part of the circle. Filled circles on an input link denote the take operation, whereas unfilled ones – like the entry with query "$type = T2 \mid count >= 0$" on the second input link in Figure 2.c – model the read operation. This query selects all available entries of type $T2$. Details on the query mechanism are explained in [7]. The operator "|" behaves like the pipe operator in the UNIX shell in that it streams the results of the query at its left side to the query at its right side. Entry specifications on output links involve the write operation, i.e. these entries are written to the specified destination container (neither source nor destination

containers are shown in Figure 2). Therefore, in Figure 2.a, exactly one entry of type T1 is removed from some container and written into another one. This simple "move" behavior is the default behavior of a wiring and Figure 2.c depicts the graphical short cut for it, which omits the wiring control symbol and uses a filled line. By using bulk operations on containers, get and put operations can transport more than one entry across one link in a single step.

For simplicity we assume that all sections of W are committed in one atomic step at the end of the wiring, but may optionally already commit after the guard or the service phase. Early commits are beneficial for unblocking other wirings that wait for the same resource. Considering a long running service, the wiring could remove read locks on get links before the service completes, so that other wirings requiring the same entry are not blocked for the complete execution time of the service. A wiring implicitly is associated with a transaction, dubbed *wiring transaction*. Each wiring transaction possesses a configured timeout, after which the transaction automatically aborts. In addition, the RTP configuration specifies how often and in which interval a failed wiring transaction retries to process entries of the same flow.

The functioning of a wiring is: 1) Fulfill all input links sequentially in the specified order, leading to an *entry collection (EC)* which can be understood as an internal space container only visible to this wiring[1]. All input links must succeed. If one cannot be fulfilled, the entire guard blocks. 2) Call all services in the given order and pass them the current entry collection from which the services may read and take entries and into which they may write other entries which represent the results of the service. These operations resemble the get and put operations of a space with the difference that blocking mode is forbidden on EC. After service execution has completed, EC has been filled with all results of the services. 3) Finally execute the output links. The task of the output links is to distribute the entries of EC to PIC and/or POC containers of the own peer or to PIC containers of sub-peers. In contrast to the input links, not all output links must succeed. The semantics is to execute one after the other in the given order: if it is satisfiable then perform it, otherwise skip it and proceed with the next one. E.g. in Figure 2.b the wiring collects one entry of type T1 and all available entries of type T2, which it passes to the service that in turn may change the wiring's internal EC. Let us assume that the service consumes all entries of type T2, and adds two of type T3. So at this point in time, EC is a multi-set $EC = \{E^{T1}, E^{T3}, E^{T3}\}$ of three entries E^{type} (note that the app-data of the two entries E^{T3} might differ). Output link 1 gets the entry of type T1 from EC and writes it to some container (not shown in the picture). Output link 2 delivers both entries of type T3. Output link 3 is tried, but its query cannot be fulfilled and thus it is skipped. This link only works if an entry of type T4 is emitted by the service. After the action section, the effects of the wiring are automatically committed, and remaining entries are removed from EC.

[1] EC access need not be modeled explicitly by Peer Model users, which provides an abstraction of this mechanism. However, it is modeled explicitly in the formal model.

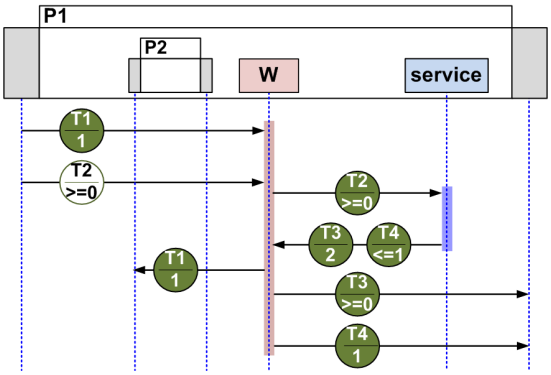

Fig. 3. Example: Wiring modeling with UML-like sequence diagram.

The functionality of wirings is modeled via UML-like sequence diagrams. The objects between which messages are exchanged are: PICs and POCs of peers, the wiring box itself, and service methods. These objects are quoted by means of their respective graphical notation plus a life line. Messages between objects represent link operations. Figure 3 shows a sequence diagram for the example wiring in Figure 2.b. It assumes a sub-peer P2, and lets W get entries E^{T1} and E^{T2} from its own peer's PIC, i.e. from P1/PIC, call the service, put E^{T1} to P2/PIC, and finally all entries E^{T3} and E^{T4} to P1/POC.

Wirings are the necessary mechanism to integrate pre-existing service functionality. The entry collection models a flexible interface for the invocation of arbitrary services. By using the features of the underlying space-based middleware, wirings dynamically connect peers and services in a data-driven way that provides high decoupling. Conceptually, all wirings whose guards are fulfilled run concurrently. The model allows for an arbitrary number of instances of the same wiring in parallel. This means that within a peer, the same service could run many times in concurrent threads. The coordination law of the containers can also be used to determine the concurrency. By default, arbitrarily many instances of W may be active at the same time, but it could also be beneficial to enforce a strict sequential execution. However, this requires slightly more complex coordination mechanisms than simple type queries. To control the concurrency level, our Java implementation of the Peer Model – termed "Peer Space" – provides system configurations for maximum thread pool sizes.

2.3 Flow Identifier

The entirety of all wirings that constitute an integration pattern is called a *flow*. In terms of enterprise systems it refers to a workflow. A flow involves a number of wiring executions that together solve a global task. A flow is identified by a unique *flow identifier (FID)* that is generated at its creation time by the Peer Space. A flow is started by emitting a first entry into the Peer Space. The initial status of a flow is "active". A flow can end under different circumstances:

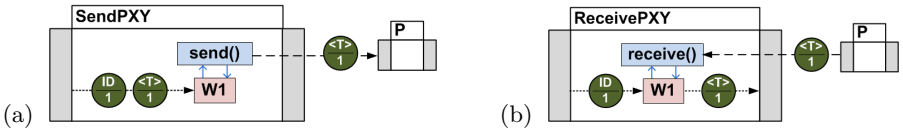

Fig. 4. (a) Send Proxy Peer. (b) Receive Proxy Peer.

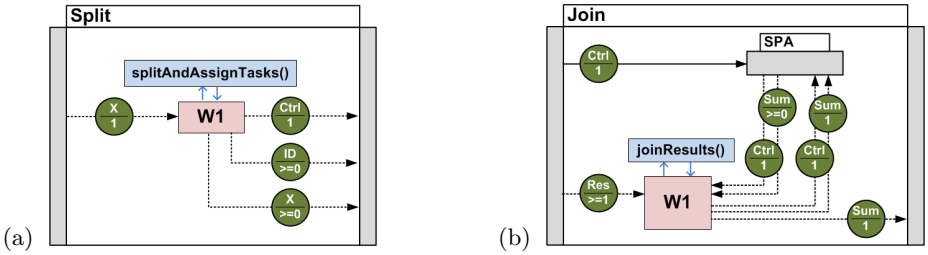

Fig. 5. (a) Split Peer. (b) Join Peer.

through an explicit success or failure result, or after the flow has reached a maximum time-to-live (TTL). Eventually all entries belonging to this flow will be recognized by wirings and automatically removed. If the developer creates an entry, s/he indicates to which flow this entry belongs. The entry automatically carries the TTL with it. A wiring will not treat an entry when its TTL has expired, but will wrap it into an error entry and put this entry into its own POC. For error entries, automatic wirings exist that let the error "roll back" up in the system until it ends at the client who originated the flow. Each wiring must only fetch entries in its guard that belong to the same flow.

3 Split/Join Integration Pattern

As an example use case, the Split/Join integration pattern [15] is chosen. The pattern's objective is to split a task of type X into several sub-tasks, distribute the execution of the sub-tasks to an arbitrary number of workers, wait for all executions, and aggregate and return the results. Developers must provide the logic how to split the original task, which worker to assign and how to join the results. All other parts are pure coordination logic. The challenge is to clearly separate application from coordination logic.

The Split/Join problem is decomposed into several patterns: Two of them are predefined by the system, termed *send* and *receive proxy* patterns. They are pure coordination patterns and modeled as two coordination peers (COPs) termed SendPXY (see Figure 4.a) and ReceivePXY (see Figure 4.b). SendPXY assumes a built-in service called send() that takes as input the ID of a peer, and an entry E^T of a configured type T. Let P be a local or remote peer to which the ID resolves. SendPXY writes E^T into P/PIC. ReceivePXY is the analogous

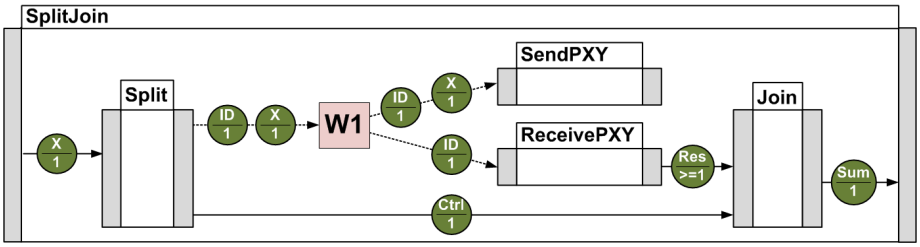

Fig. 6. Split/Join Peer.

counterpart: It uses the built-in system service receive() to take one entry from a remote peer's P/POC, where P's ID is the input entry of the pattern.

Two further patterns are Split and Join in Figure 5, both modeled as application peers (APPs) that require application specific logic. The Split peer has a wiring W1 with a guard that expects one task entry of type X as input, and passes it to the splitAndAssignTasks() method which is a user-programmed service that produces an arbitrary amount of sub-tasks and adds them to the internal entry collection of this wiring instance together with a multi-set of peer worker IDs that shall execute sub-tasks. If the same ID is generated twice the same worker must process two sub-tasks; however, which ones is irrelevant. Sub-task entries are also added to the entry collection. Finally, in the action section, the wiring puts a control entry, all IDs, and all sub-tasks into Split/POC. The control entry holds in its application data part specific control information needed for the join, e.g. how many results are needed, which quality is required etc. The Join peer's first wiring is a default wiring that moves one control entry of type Ctrl into a local space peer termed SPA, which serves as local memory for the joinResults() service. Its wiring W1 waits until at least one intermediary result of a worker peer has arrived. It then takes all already available result entries of type Res from worker peers as input, takes the control entry from SPA and also an entry of type Sum (if here) and passes all these entries to the joinResults() service method provided by the developer. The method checks control and result entries, and merges them into the sum entry (or generates one after the first invocation of W1 for this flow). If the service decides that enough results are here, it writes Sum to Join/POC. Otherwise, it writes Sum and Ctrl back into its local SPA, commits and continues its next iteration.

In addition, the user must provide the implementation of worker peers which are not detailed in the example. A worker peer takes one task of type X as input, then performs its specific application logic, and finally generates one output of type Res that it puts into its POC.

All these peer components are composed towards the Split/Join pattern depicted in Figure 6. It starts with applying the Split pattern on input entry X, which outputs many peer IDs, one control entry and many sub-tasks of type X. Wiring W1 is an inter-peer wiring and is executed for pairs of ID and X entries. Its guard requires one ID and one X entry and uses the ID entry to initialize

the SendPXY peer as well as the ReceivePXY peer. In addition, the sub-task entry is written into SendPXY/PIC. SendPXY writes the entry of type X to the corresponding worker peer. In parallel, ReceivePXY waits for the result of this worker peer, takes the results from the worker peer's POC and returns it in ReceivePXY/POC. From there a default move wiring transports it to the Join peer, which also gets the control entry from Split/POC into its Join/PIC as input. The Join peer now waits until enough results are received, merges them and returns the final output which is the aggregated sum of all obtained results. This sum entry is transported to the SplitJoin/POC and from there it is delivered to the originator of the flow. Many flows may run concurrently. Their entries are automatically correlated by the unique identifier (FID) of the flow.

The design of the pattern is resistant against adding more or different, local or remote worker peers. The pattern flexibly supports new requirements and challenges like changed logic for how to join the results: e.g., use only the best results, or stop after a certain amount of results, or stop if a certain condition is met. Only the corresponding joinResults() service must be rewritten. If the FID is invalidated, all still running processes of the flow will eventually stop. All other parts and components of the Split/Join pattern remain unaffected.

4 Meta Model and Domain Specific Language

The intention of the Peer Model is to provide a sound mechanism for modeling integration patterns. We have implemented the system with Promela [9] to achieve a runnable specification and lay the basis for future work on verifying certain properties by means of the SPIN model checker. In addition, a Java version has been implemented to get first experiences with an object-oriented Peer Model API. A presentation of this API is beyond the scope of this paper. We will instead outline the Domain Specific Language for the Peer Model (PM-DSL) that was implemented with Promela. First a meta-meta-model is built, consisting of data structures that model peer types. Then the meta-model that describes the functionality of all peers in a specific Peer Space is derived from the meta-meta-model. The space container operations are implemented by means of (blocking) channel operations. The PM-DSL provides the following operations:

`defPeerType(`*PeerTypeName*`,` *PeerTypeID*`);`
`addSubPeer(`*PeerTypeID*`,` *PeerTypeName*`,` *PeerName*`);`
`addWiring(`*PeerTypeID*`,` *WiringName*`,` *WiringID*`);`
`addGuardLink(`*WiringID*`,` *FromPeerName*`,` *FromContainerName*`,`
ReadOrTake`,` *TypeQuery*`,` *Count*`,` *CountDetails*`);`
`addService(`*WiringID*`,` *ServiceName*`,` *NArgs*`,` *Arg*1`,` *Arg*2`, ...);`
`addActionLink(`*WiringID*`,` *ToPeerName*`,` *ToContainerName*`,`
TypeQuery`,` *Count*`,` *CountDetails*`);`
`createPeer(`*PeerTypeName*`,` *PeerName*`,` *PeerID*`);`
`startPeer(`*PeerID*`);`
`containerWrite(`*PeerID*`,` *ContainerName*`,` *EntryType*`,` *AppData*`,`
CoData`);`

First the meta-meta-model must by initialized. For this, PM-DSL provides the following operations: defPeerType() specifies a new peer type of given type name and in the second argument returns the id by which this type is referenced in the following; addSubPeer() adds a peer given by a peer type to another peer type (given by id) and names it; addWiring() adds a named wiring to a peer type and returns its id; addGuardLink() adds an input link to a wiring by saying from which peer's PIC or POC with which query entries shall be read or taken; addService() adds a service to a wiring – there exist some built-in services with defined names resp. the developer may program own services and refer to them by their name – and here one can configure static arguments (e.g. the number of sub-tasks to be produced by the Split peer) of the service; addActionLink() adds one output link to a wiring and says to which peer's PIC or POC which entries shall be put. "`THIS_PEER`" refers to the own peer.

As an example, the SplitJoin peer pattern of Figure 6 is modeled with the PM-DSL. We show only parts of the specification of the meta-model for the SplitJoin peer; and we assume that the other peer types already exist.

```
/* SplitJoin peer type: */
defPeerType(SPLIT_JOIN_PEER_TYPE, ptID);
/* Split sub peer: */
addSubPeer(ptID, SPLIT_PEER_TYPE, SPLIT_PEER);
/* SendPXY sub peer: */
addSubPeer(ptID, SENDPXY_PEER_TYPE, SENDPXY_PEER);
/* ReceivePXY sub peer: */
addSubPeer(ptID, RECEIVEPXY_PEER_TYPE, RECEIVEPXY_PEER);
/* Join sub peer: */
addSubPeer(ptID, JOIN_PEER_TYPE, JOIN_PEER);
/* Wiring W1: */
addWiring(ptID, W1, wID1);
addGuardLink(wID1, SPLIT_PEER, POC, TAKE, X_TYPE, 1, EQUAL);
addGuardLink(wID1, SPLIT_PEER, POC, TAKE, ID_TYPE, 1, EQUAL);
addActionLink(wID1, SENDPXY_PEER, PIC, ID_TYPE, 1, EQUAL);
addActionLink(wID1, SENDPXY_PEER, PIC, X_TYPE, 1, EQUAL);
addActionLink(wID1, RECEIVEPXY_PEER, PIC, ID_TYPE, 1, EQUAL);
/* Wiring W2 (default wiring): */
addWiring(ptID, W2, wID2);
addGuardLink(wID2, THIS_PEER, PIC, X_TYPE, 1, EQUAL);
addActionLink(wID2, SPLIT_PEER, PIC, X_TYPE, 1, EQUAL);
```

For creation of the meta-model the PM-DSL offers the operation createPeer() which creates a new peer with given name and initializes its meta-model from the specified peer type, including all resolved wirings. It recursively also creates all sub-peers. Their names must be resolved in the right scope, i.e. the name of a sub-peer is only visible to its super and sibling peers. Only through lookup mechanisms these names become visible to other peers. Description of lookup peers is straight forward, as existing peer-to-peer algorithms can be applied. The Peer Model can now be started with Promela's run operation calling startPeer()

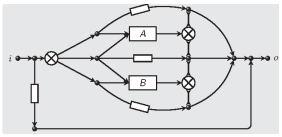

Fig. 7. Reo with 2 services: Synchronous Merge Connector, Fig. 1 from [4].

which starts the wirings of all peers as concurrently running processes. For each wiring multiple instances are assumed to exist concurrently (cf. "bang" operator in [12]). Creation of flows takes place through injection of entries into the system with containerWrite() whereby the FID is contained in the co-data part of the entry. Coming back to our use case, the final steps to start the pattern are:

```
createPeer(SPLIT_JOIN_PEER_TYPE, SPLIT_JOIN_PEER, peerID);
run startPeer(peerID);
containerWrite(peerID, PIC, X_TYPE, /* task */, /* FID etc. */);
```

5 Related Work

An approach related to Split/Join has been undertaken with Reo [4] termed Synchronous Merge Connector (see Figure 7). Flows are modeled by means of different connector types. The example starts with an exclusive router that non-deterministically selects one output which can be either service A or service B. From there three walkthroughs become possible: service A, service B, or both. The exclusive router is also used to control the joining of the execution. Reo supports composition of software components using a channel-based coordination model. In contrast to the Peer Model, Reo is oblivious to any coordination and concurrency inside the component instances; also the used communication mechanisms are of no interest [3], whereas the Peer Model builds upon tuple space-based middleware and allows for nesting of components.

Colored Petri Nets (CPN) [10, 1] are a graphical language for modeling concurrent and often also distributed systems. However, they can be applied in almost any thinkable domain[2]. Typically it is used for testing and performance analysis [10]. As the modeling is quite low-level and does not make any assumptions on the domain, it is very powerful; but on the other hand it does not provide any higher-level abstractions. We used CPN-Tools [14] to model the Split/Join use case of Section 3 in two variants: Figure 8.a without requiring the underlying Petri Net to allow for complex inscriptions on input and output arcs, and Figure 8.b making use of advanced features of the ML language used by CPN. For Figure 8.a the pattern was modeled with two services A and B, and every way through the network is controlled by an explicit transition. The other variant uses ML lambda expression to specify more complex arc conditions (Figure 8.b).

[2] Examples of Industrial Use of CP-nets: http://cs.au.dk/cpnets/industrial-use/

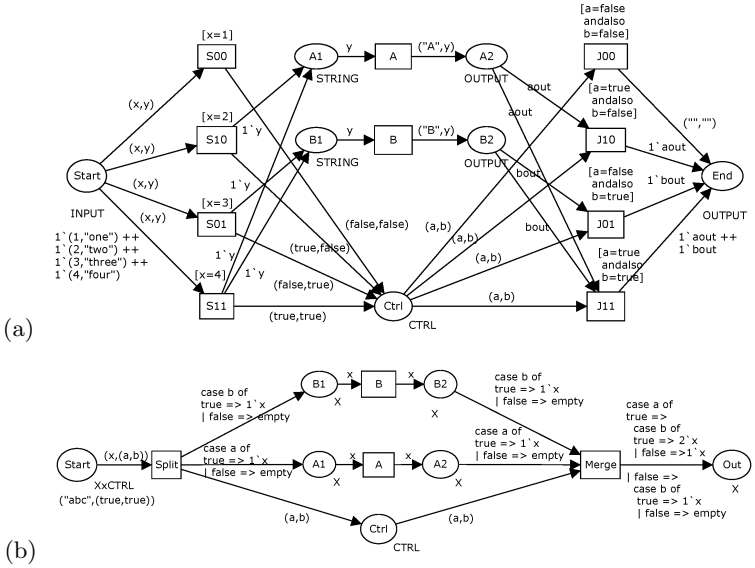

Fig. 8. CPN for 2 services with (a) weak, and (b) complex arc logic assumptions.

Next, we increased the design up to 4 services. Figure 9.a shows the basic variant, and Figure 9.b the one employing ML's expressiveness. Both figures demonstrate how complex these designs become. The latter needs less places, but the condition on the output link of the Merge transition becomes more complicated if more services are assumed. Moreover, the control logic is spread among the network. If services are added, the type definitions for the control token must be adapted, as well as for the Start and CTRL place. For each service two places and one transition, plus four arcs must be added explicitly, because the network is static. Two of these arcs must have a special condition. In summary, the comparison shows that Petri Nets do not scale because they are static. Every new component must be modeled explicitly.

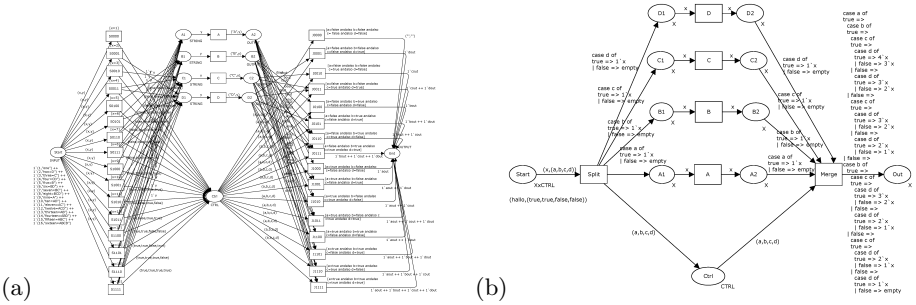

Fig. 9. CPN for 4 services with (a) weak, and (b) complex arc logic assumptions.

The main advantages of the Peer Model approach in contrast to more general and static tools like CPNs and REO are:

Design scalability. The Peer Model allows modeling of large patterns and scales naturally to any number of services, without exponential complexity. In the example this is accomplished by a send and a receive proxy peer, which (de-)multiplex sub-tasks to/from arbitrary worker peers addressed via their IDs. Dynamic addition of workers and increasing the number of sub-tasks is possible.

Dynamics. During runtime, new peers can be added on-the-fly. Also, peers can be changed in that, e.g., wirings are added dynamically.

Composability. The Peer Model design allows to break down a problem into smaller, re-usable patterns that can be composed to larger ones. CPNs also support hierarchically nested modules, but the composition requires global names for the "fusion" places.

Transparent remoting. The physical distribution of processes to other sites does not cause a difference in design compared to local processes.

Architecture agility. The model is robust against changes due to new requirements. We have termed this "architecture agility" in [13] and shown that the space-based architectural style helps to avoid costly architecture limiters or even breakers. The Peer Model extends this idea by not only considering passive data, but also modeling the active part, i.e. processes that access the data.

6 Conclusion

The Peer Model is a design tool for the specific domain of programming parallel and distributed applications. In contrast to other widespread modeling tools like Colored Petri Nets it is less general as it takes specific assumptions on its domain. More precisely, it requires a coordination middleware whose features have an impact on the quality of the Peer Model. At least, access to shared data by means of blocking operations, local transactions, and type-based queries with cardinality specification are required. The process space is structured into so-called peers which are framework components. They separate coordination from application logic to achieve software architecture agility. Application logic is integrated in form of service methods. Peers are reusable components that implement coordination patterns. Wirings control the entry flow between and within peers. Flow-identifiers automatically correlate entries of the same flow, thus allowing many instances of different flows to be processed concurrently.

We have presented the basic concepts of the Peer Model, its graphical notation, and domain specific language which is based on a formal model. Its conception is intended as foundation for real implementations. The usability of the Peer Model was evaluated with respect to design scalability, composability, remoting, architecture agility, and support of dynamics. Our future work will concern implementation of a designer tool, mappings to the Colored Petri Net formalism, and extending the mechanisms of how to model concurrency.

Acknowledgements. The work is partially funded by the Austrian Federal Ministry for Transport, Innovation and Technology, FFG-BRIDGE project no. 834162 "LOPONODE" (Coordination Middleware for Wireless Networks of Low Power Nodes).

References

1. Final Draft International Standard ISO/IEC 15909. High-level petri nets - concepts, definitions and graphical notation. Technical report, V. 4.7.1, Oct. 2000.
2. C. André, F. Mallet, and R. de Simone. Modeling time(s). In *10th Int. Conf. on Model Driven Engineering Languages and Systems (MoDELS)*, volume 4735 of *LNCS*, pages 559–573. Springer, 2007.
3. F. Arbab. Reo: a channel-based coordination model for component composition. *Mathematical Structures in Computer Science*, 14(3):329–366, 2004.
4. D. Clarke and J. Proença. Partial connector colouring. In *14th Int. Conf. on Coordination Models and Languages (COORDINATION)*, volume 7274 of *LNCS*, pages 59–73. Springer, 2012.
5. S. Craß, T. Dönz, G. Joskowicz, and e. Kühn. A coordination-driven authorization framework for space containers. In *7th Int'l Conf. on Availability, Reliability and Security (ARES)*, pages 133–142. IEEE, 2012.
6. S. Craß, T. Dönz, G. Joskowicz, e. Kühn, and A. Marek. Securing a space-based service architecture with coordination-driven access control. *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications (JoWUA)*, 4(1):76–97, 2013.
7. S. Craß, e. Kühn, and G. Salzer. Algebraic foundation of a data model for an extensible space-based collaboration protocol. In *Int. Database Engineering and Applications Symposium (IDEAS)*, ACM, pages 301–306, 2009.
8. D. Gelernter. Generative communication in Linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, 1985.
9. G. Holzmann, E. Najm, and A. Serhrouchni. Spin model checking: an introduction. *International Journal on Software Tools for Technology Transfer (STTT)*, 2(4):321–327, 2000.
10. K. Jensen, L. Kristensen, and L. Wells. Coloured petri nets and CPN tools for modelling and validation of concurrent systems. *Int. Journal on Software Tools for Technology Transfer (STTT)*, 9:213–254, 2007.
11. e. Kühn, R. Mordinyi, L. Keszthelyi, and C. Schreiber. Introducing the concept of customizable structured spaces for agent coordination in the production automation domain. In *8th Int'l Conf. on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 625–632. IFAAMAS, 2009.
12. R. Milner. The polyadic pi-calculus: a tutorial. Technical report, Logic and Algebra of Specification, 1991.
13. R. Mordinyi, e. Kühn, and A. Schatten. Towards an architectural framework for agile software development. In *17th IEEE Int. Conf. and Workshops on the Engineering of Computer-Based Systems (ECBS)*, pages 276–280. IEEE, 2010.
14. A. V. Ratzer, H. M. Lassen, M. Laursen, J. F. Qvortrup, M. S. Stissing, M. Westergaard, S. Christensen, and K. Kurt Jensen. CPN tools for editing, simulating, and analysing coloured petri nets. In *24th Int. Conf. on Applications and Theory of Petri Nets (ICATPN)*, volume 2679 of *LNCS*, pages 450–462. Springer, 2003.
15. W. M. P. van der Aalst, A. H. M. ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow patterns. *Distributed and parallel databases*, 14(1):5–51, 2003.