Reusable Coordination Components: Reliable Development of Cooperative Information Systems

Eva Kühn

*Institute of Computer Languages
Faculty of Informatics, TU Wien
Argentinierstraße 8, 1040 Vienna, Austria
eva.kuehn@tuwien.ac.at*

Received 25 January 2016

Accepted 11 November 2016

Published 9 January 2017

Today's emerging trends like factory of the future, big data, Internet-of-things, intelligent traffic solutions, cyber-physical systems, wireless sensor networks, smart home, smart city and smart grid raise new challenges on software development. They are characterized by high concurrency, distribution and dynamics as well as huge numbers of heterogeneous devices, resources and users that must collaborate in a reliable way. The management of the interactions and dependencies between the participants is a complex task posing massive coordination problems. The here proposed approach is twofold: (i) to analyze similarities in the communication and synchronization behavior of such applications and to identify coordination patterns; and (ii) to give a precise specification of them by means of a suitable coordination model which enables the development of coordination pattern-based software components as solutions. The vision is to compose advanced cooperative information systems from proven, configurable, reusable, generic components that run on a suitable target platform, in order to reduce software development time, risks and costs. In this paper we delimit the idea of "coordination patterns" from other related pattern approaches and motivate the need for a well-defined model to specify them. Several coordination models to achieve this goal are discussed, and the advantages of a new coordination model termed the "Peer Model" are pointed out. The feasibility of the approach to identify coordination patterns, to model them and to provide generic components that can be reused in different scenarios through configuration and composition is evaluated by means of a coordination pattern found in several industrial use cases.

Keywords: Coordination pattern; coordination model; pattern solution; generic coordination component; space-based middleware.

1. Introduction

The application landscape is changing due to rapid advances in hardware and communication infrastructures. New application areas are emerging that are driven by innovations like new devices and commoditization of hardware with new possibilities

This is an Open Access article published by World Scientific Publishing Company. It is distributed under the terms of the Creative Commons Attribution 4.0 (CC-BY) License. Further distribution of this work is permitted, provided the original work is properly cited.

like virtual reality, wearable computing and implantable devices. These new applications must therefore deal with permanent changes of the underlying infrastructures. Beyond that, modern and future distributed cooperative information systems are complex, especially because they involve high concurrency, distribution and dynamics, as well as huge numbers of heterogeneous devices, resources and participants. They must coordinate and integrate highly concurrent and lively processes and information sources of active, autonomous software systems in real-time. Example use cases that rely on such lively data (e.g. sensory, booking, weather and traffic data from independent providers) can be found in many domains like factory of the future, Internet-of-things, cyber-physical systems, wireless sensor networks, smart energy systems, smart home, smart city and intelligent traffic management.

The technical challenge is to achieve common views on the highly concurrent and dynamic data and processes with an acceptable consistency level.

“*Coordination* is managing dependencies between activities”.¹ These activities are processes that need to collaborate or compete with each other in order to achieve a goal like for example, to balance dynamic loads among heterogeneous processes, replicate data under given consistency requirements, notify processes in real-time about events, achieve a consensus despite of distributed and dynamically joining and leaving partners or adaptively negotiate between interfaces and versions. In any case, communication between and synchronization of the different processes is needed. The quality of the solution will depend on how timely and consistently the information is coordinated and how well and fast members in large distributed system are aware of each other’s state in a timely and reliable manner.

Coordination therefore requires specialized expertise in distributed and concurrent systems to which a typical application developer might not be exposed, because the required skills differ from those required to program the domain-specific application business logic. Coordination is difficult to test and to monitor, and many failures occur only after a system is in production. Quite often the coordination complexity is not considered with enough emphasis in the requirements specification of a system, however, failures in the coordination parts of the software might compromise the entire system’s functionality: “Good coordination is nearly invisible, and we sometimes notice coordination most clearly when it is lacking”.¹ On the other hand, we become increasingly dependent on these systems. Depending on the use case, a failure might even be threatening, for example, if it occurs in a safety-critical application like a cyber-physical system or a train warning system.

Developers must adapt to new requirements, market standards and changes in networked partner systems; they must react quickly to feedback of end-users and cope with unexpected behavior in large complex systems. If applications are built on a low-level platform, developers must explicitly care for complex tasks like consistency, replication and load balancing in distributed systems. Time-to-market implies that software developers sometimes cut corners on design and code at a very early stage in development and trade an architected and engineered approach for fast implementation and sometimes even let the end-user test the system. They

therefore need efficient and pragmatic means to reduce the complexity²⁻⁴ through, for example, abstraction, decoupling, layering, standardization, modularization and reuse of documented experiences and solutions.

The approach suggested in this paper is to provide *coordination patterns* composed from “structural elements”, which are derived from a single concise model. These coordination patterns target similar coordination tasks found in cooperative information systems. Comparable to database systems that generalize the data management problem, we are seeking for a methodology that separates the complex problem of coordination from the application.

The paper is structured as follows: Section 2 summarizes pattern concepts and explains what is meant by a *coordination pattern*. Section 3 explains which features are required by a coordination model. Section 4 presents the Peer Model for the modeling of coordination patterns. Section 5 provides by means of the load balancing problem a proof-of-concept for a selected coordination pattern solution, and demonstrates that coordination can be decomposed into patterns and modeled with the Peer Model. Section 6 compares related work on coordination models. Section 7 gives an analysis of coordination patterns in several industrial use cases. Section 8 summarizes the results and points out our future research work in this area.

2. Patterns

Coordination¹ is required if there exist at least two independent and concurrent processes, each with a separate thread of execution. The processes need to communicate and synchronize with each other in some form in order to achieve their own goal or a common goal through collaboration or competition. They are either coordinated by some external coordinator or coordinate themselves autonomously. In either case, a set of rules must be defined that describe the interaction among the processes including access to shared resources, group decisions, task allocation, etc. A dynamic change of rules enables advanced features like adaptivity.

The concept of patterns and pattern languages stems from the field of architecture^{5,6} and was adapted by the “Gang-of-Four”⁷ (GoF) for the design of software architectures, describing architectural and design patterns. A pattern is the description of experiences how to solve a special situation related to software development. The description introduced by GoF basically is informal and comprises a name for the pattern, its classification, purpose, synonyms, motivation, applicability, structure, involved stakeholders, interaction of concerned classes, consequences, implementation hints, example codes, example applications and references to other patterns. This way, so-called *pattern languages* can be built up. The major purpose of patterns is to communicate knowledge and to make it reusable for other software developers in order to ease their development tasks and let them profit from the experiences of others. A pattern cannot be invented, but has its constitution in its proving in many scenarios and real use cases. The emphasis of GoF patterns is on solid programming practices based on object-oriented principles.

In “Patterns for Concurrent and Networked Objects”⁸ patterns for service access and configuration, event handling, synchronization and concurrency are collected. These patterns rely on sets of structural elements that are common in distributed and parallel systems and can be parameterized and composed towards more complex patterns. Coordination-related patterns build on these concepts. They share characteristics with message exchange patterns (MEPs), enterprise application integration (EAI), workflow (WF) and cloud computing patterns.

2.1. *Message exchange patterns*

“A message exchange pattern specifies in a reusable manner the ability of a service to receive and/or send messages. It describes the set of exchanged messages in terms of their order and multiplicity, i.e. whether a message is sent to or received from a single node or whether a message is sent to or received from multiple instances of a node”.⁹ MEPs formalize basic situations where processes communicate by means of exchanging messages. They define templates for push- or pull-based messaging on a basis whether the communication is one-way only and in which direction (e.g. input-only or output-only); or if it is two-way (i.e. request/response) and whether outgoing or incoming messages are optional.

MEPs categorize all forms of direct port-to-port communication. Patterns on a higher abstraction level like enterprise application integration, workflow or coordination patterns build on top of MEPs. The structural elements of MEPs are therefore necessary but not sufficient for the description of coordination patterns.

2.2. *Enterprise application integration patterns*

EAI patterns¹⁰ represent a collection of 65 patterns* that build on message-oriented middleware. The abstraction layer they provide for developers is on the level of how to send one or more messages between senders and receivers along channels. All patterns are directed and route messages from the senders to the receivers. They are grouped along six categories (see Table 1).

EAI patterns are directed and describe how to integrate systems by routing messages through a network of collaborating applications. As such there is overlap between some EAI and coordination patterns. As an example, let us take the case of subscribers to a topic queue. This pattern can be considered as an EAI as well as a coordination pattern. From the point of view of EAI, a topic queue serves to disseminate information among subscribing applications. From the point of view of coordination, a topic queue might serve as a mechanism to synchronize the state of collaborating applications. Coordination patterns will add additional aspects like timing, content categorization or life cycle control to the abstraction.

* www.enterpriseintegrationpatterns.com.

Table 1. Abstraction level of EAI patterns.

Messaging	Construct a message (representing a command, document, event, etc.) with regard to its intent, form and content, including issues like message correlation, expiration and sequence numbers.
Messaging routing	Route a message from the right sender to the right receiver and by using one or more message channels. Examples: pipe-and-filter, based on content, splitting the message, aggregating messages, using a broker, etc.
Messaging transformation	Modify the content of a message, e.g. translate it, wrap it into an envelope, normalize it and filter it.
Messaging endpoints	Produce a message and put it into a channel (for senders), and to get a message from the channel and consume it (for receivers).
Messaging channels	Messages are transported via the channels, e.g. point-to-point, guaranteed delivery and publish/subscribe.
System management	Manage the system, deal with errors, performance problems, change requests, etc.

2.3. Workflow patterns

WF patterns¹¹ address patterns for the modeling and development of workflow applications. Workflow languages like BPEL4WS, BPML, XLANG, WSFL and WSCI emerged from the need to communicate and describe workflows between participants in an implementation-independent manner. The workflow patterns catalog is a systematic collection of structural elements, termed patterns. It started with patterns for control flow only; later workflow resource patterns, workflow data patterns and exception handling patterns were added¹² (see Table 2). The Workflow

Table 2. Abstraction level of WF patterns.

Control	<ul style="list-style-type: none"> • Elementary aspects of process branches (sequence, parallel split, synchronization, exclusive choice and simple merge). • Advanced branching and synchronization of process branches (e.g. more sophisticated possibilities to split, join and merge branches, to select (choice) branches or to merge and split process threads). • Multiple threads of execution in the process model. • Decision on which branch to take dependent on a state in the system. • Canceling of activities. • Repetitive behavior (cycle, loop and recursion). • How to terminate a workflow? • Trigger external signals in order to start other tasks.
Resource	Create work items, push them into the workflow system, manage their life cycle, control their visibility, and allow them to process multiple items simultaneously or request further resources.
Data	Define and use data, to pass and transfer data among interacting components of a process (push- or pull-oriented); and to cope with data- and event-based controls.
Exception handling	Assess the level of exceptions support by existing systems and languages.

Patterns Initiative[†] started with 20 patterns in 2003 and in 2009 reached 126 patterns.¹³ For our focus to model the interdependencies in collaborative information systems, control patterns are of most interest and summarized in more detail.

A pattern can be modeled with any modeling technique and does not imply the usage of a particular programming language.¹² They are rather informal and do not aim towards a formal ground model.¹³ However, this approach fully suffices for the intent to help users to (a) understand which patterns they need to realize a business process and (b) select the right workflow system that best supports these patterns.

2.4. Cloud computing patterns

Cloud computing patterns¹⁴ describe architectural patterns and a pattern language for developing applications in one or more clouds. The patterns are mined from data provided by different enterprises. They support users in understanding which services their applications require and then to derive which cloud and architecture fit. Table 3 gives a short overview of the pattern catalog.

With the help of this knowledge, developers shall get a better understanding to decide for the right cloud provider, whether the application will profit from transferring it into a cloud, and how to deploy it in the cloud. Cloud computing patterns reside at a very abstract level. They are highly specialized for their respective domain. Concerning communication and coordination among participants, they refer to EAI patterns and thus provide a similar abstraction layer as EAI patterns.

2.5. Summary of MEP, EAI, WF and cloud patterns

MEPs describe communication use cases, EAI patterns support integration use cases, WF patterns aim at the comparison of workflow languages and cloud computing patterns target the development of applications in the cloud. All assume an asynchronous, message-oriented communication style. MEPs, EAI and WF patterns

Table 3. Abstraction level of cloud computing patterns.

Cloud computing fundamentals	Application workloads (static, elastic, periodic, once-in-a-lifetime, unpredictable and continuously changing), cloud service (infrastructure, platform and software as a service) and deployment (public, private, community or hybrid cloud) models.
Cloud offerings	Impacts and dependencies of offered environments for processing (e.g. map/reduce), storage and communication (based on virtual networking or message-oriented middleware, cf. EAI patterns) on properties like availability, consistency, etc.
Cloud application architectures	How to build cloud applications, and to integrate clouds?
Cloud application management	Elasticity, resiliences, updates and automatized management.
Composite cloud applications	Example applications in native and hybrid clouds.

[†]www.workflowpatterns.com.

follow static assumptions, a defined context, and are characterized by a centralized control, which makes them not suitable for coordination. All of these patterns have less focus on a “ground model”.

Since coordination mechanisms rely on messaging, integration and workflows, there is a considerable overlap of these patterns with coordination-related patterns. Coordination patterns reuse the structural elements of these patterns with an added emphasis on concurrency, timing and collaborative aspects.

2.6. Coordination patterns

We define *coordination patterns* as a new category of patterns that describe the generics of communication and synchronization and separate it from the application logic. They are characterized by concurrency, distribution, dynamics and multi-party interactions.

As pointed out above, coordination patterns overlap with MEPs, EAI and WF patterns. Their main emphasis is on concurrency, timing and dynamic adaptation/self-organization. Aspects of message routing, splitting and joining of concurrent branches, etc., which are addressed by MEPs, EAI and WF patterns, can be seen rather at the communication level, whereas patterns like load balancing, load clustering, replication, migration, consensus finding, self-organization, failover strategies, collaboration, cooperation, competition and negotiation are examples at the coordination level.

In the following we will only refer to a formal reference implementation of each pattern (*generic pattern solution*), introduce important characteristics needed by a coordination model, argue that a ground model is necessary for the specification of the pattern solution and introduce the Peer Model for it. The full description of coordination patterns in the sense of GoF that is the basis for a pattern language, and a comprehensive and systematic coordination pattern catalog is out of the scope of this paper and will be part of our future work.

3. Requirements on a Coordination Model

As discussed above, pattern descriptions are usually informal and/or quite abstract. A precise specification of pattern solutions by means of a suitable coordination model is needed to bridge the gap between design and implementation. The model must be formally grounded and provide domain-specific modeling abstractions for coordination. In addition it shall be usable, provide a graphical notation and offer efficient runtime support as well as development tools.

A coordination model must therefore provide explicit concepts for the following coordination features:

- *Concurrency control*: It refers to the ability to specify the allowed concurrency in the system, including the explicit constraining of it where necessary. This includes constraints like ordering and priorities of processes, concurrent number of processes, interlocking, etc.

- *Separation of concerns*: The application logic must be clearly separated from the coordination logic and from the communication layer. In Reo¹⁵ this is called *exogenous coordination*. It is a necessary condition for extracting the coordination part and reusing it in other applications. Application logic includes both application data and services.
- *Flow correlation*: Modeling support for concurrent business processes whereby services and data remain clearly separated while each business process progresses through the network of collaborating applications. Services and data are automatically correlated by a built-in mechanism.
- *Time*: Modeling of timing aspects like scheduling, duration of computations and communications and the life cycle management of resources must be provided.
- *Pattern composition*: It requires means for the parametrization, configuration and composition of patterns in order to build up more complex patterns.
- *Error handling*: It must be possible to cope with failures.
- *Dynamic changes*: Possibilities to dynamically change the model. This implies the provision of a meta model.

The accommodation of a ground model requires a single concise way of expressing coordination policies: We call this mechanism the *coordination principle* whose expressiveness allows constructing the necessary structural elements (like ordering, priorities, choices, conditions, etc.) underlying coordination patterns.

Relevant criteria for the acceptance of a coordination model are:

- *Usability*: The usability of the model will play an important factor for its applicability and usage by the developer community. It shall be understandable (even for people from the application domain) and suitable for complex applications (larger problems do not induce cumbersome and unreadable specifications).
- *Graphical notation*: This refers to the provision of an intuitive graphical notation (enabling a modeling tool).
- *Runtime support*: Models are designed in a platform-independent way; however, they should be efficiently executable on different target platforms. This requires implementations for enterprise platforms as well as for embedded systems — as, for example, required by the mentioned industrial application scenarios. Interoperability among these platforms is important. Relevant is also the support of a remoting abstraction (including naming, modeling of the distribution of components and their communication), deployment aspects and the control of process instances.
- *Tool chain*: Availability of development tools like modeler, code generator, simulator, verification tool, etc.

4. Peer Model

The Peer Model is a coordination model that is based on asynchronously communicating peers whose coordination behavior is explicitly modeled. It targets all

criteria presented in Sec. 3. The participants in the Peer Model¹⁶ are called *peers*. Each peer has a certain role like, for example, fabricating a product, routing of information, translating information, collecting/processing sensory data, generating tasks, placing orders, etc. Peers may be distributed and act concurrently. They interact by exchanging *entries*.

The behavior of a peer is triggered by sending the peer an entry into its incoming mailbox called the *peer input container* (PIC). It is in a peer's own responsibility to read entries from a mailbox, to remove entries from a mailbox, to process entries, and to write entries into a mailbox. The outgoing mailbox of a peer is termed *peer output container* (POC). Note that the peer may access all mailboxes in its scope: These are the own ones, as well as those of direct subpeers (see below). Also a subpeer that produces a result, i.e. writes one or more entries into its outbox, can this way trigger the peer to act.

The strategy with which a peer reads (and removes) entries from the mailbox, whether and when it does so, which *service* it carries out to process the received entries and where it places results is reflected by a single coordination principle. The application-specific functionality of services is out of the scope of the model: Application logic is "injected" into a peer via corresponding service and data definitions. In other words, the Peer Model models solely the coordination logic. The transport system is represented by the Peer Model's artifacts termed *wirings*. The coordination logic is realized as *concurrent*, *timed* and *distributed flows*. The application logic is considered a black box and wrapped into services that are called by wirings. The model is independent of the underlying communication mechanism and platform.

Its design is influenced by the tuple-space-based paradigm,^{17,18} which abstracts the communication layer and provides a decoupling of the communication partners with regard to time, space and reference.¹⁹ Further foundations of the Peer Model are the Actor Model²⁰ (concerning asynchronous messaging between concurrent processes), Petri Nets²¹ (PN) (whose transitions inspired the modeling of concurrency by means of wirings) and abstract state machines²² (with regard to modeling wirings with guards and actions). The following subsections explain these concepts, introduce an exception handling mechanism, system peers for remoting and dynamic model changes and a graphical notation.

4.1. Concepts of the ground model

The Peer Model constitutes the single basic mechanism with which all structural elements of coordination patterns are constructed. Its ground concepts are: *entry*, *container*, *peer*, *wiring*, *link* and *service*. The Peer Model strictly separates coordination logic (modeled with wirings) and communication layer (represented by the mailboxes) from application logic (specified as services) (see Fig. 1).

4.1.1. Entry

An entry represents any kind of information passed around in the network and shared between concurrent processes like a message, an event, a task, a request, data

Fig. 1. Example peer: Ground concepts and separation of concerns.

and documents. Entries consist of system- and application-defined coordination properties. A property has a name and a value. System properties use reserved names and have well-defined semantics in the model.[‡] Major system properties are: **type** (the entry's coordination type), **ttl** (time-to-live, determining the entry's life cycle time), **tts** (time-to-start, specifying when an entry becomes operational), **dest** (destination, i.e. the name of a peer to which the entry shall be sent) and **fid** (flow identifier).

The value of the **data** property encapsulates the application-specific data contained in an entry. These data have an application-specific data type. This concept is comparable to the envelope in messaging-oriented systems: The message header is related to the system and application coordination properties of an entry, whereas the message body is related to the value of **data**. The **data** type differs from the **type** of the entry.

The Peer Model uses all coordination properties (system and application) for modeling of coordination. The possibility to define arbitrary application properties, as needed for a particular use case, generalizes message headers. The concept of coordination properties makes the model extensible.

[‡]The **type** property is mandatory. If not explicitly specified, default values are assumed for the other system properties.

4.1.2. Container

Containers represent mailboxes. They are based on space containers of the tuple-space-based middleware termed extensible virtual shared memory (XVSM).¹⁸ Many extensions and improvements of the original tuple space model have been proposed in the literature.^{23–27} XVSM[§] (see Fig. 2) is the definition of an extended tuple space model that takes into consideration many of these ideas in a modular reference architecture.^{28–30} Its main features extending tuple spaces are:

- *Distributed, peer-to-peer architecture*: Each participant possesses its own embedded space that is accessible by other participants.
- *Structuring of the space into containers*: XVSM structures the space into sub-spaces called containers. A container possesses a name by which it is addressable in the network. Therefore, no explicit method to connect to a space is required. Each space operation is self-contained in addressing the container that it wants to access. For a container one can define a boundary size referring to the maximum number of entries it can hold. Write operations on the container block or raise an exception if there is not sufficient space in the container. The API provides the classic JavaSpaces³¹ operations for container access: **write**, **read** and **take**. Additional methods are **create**, **lookup** and **destroy** containers. All operations support transactions and timeouts. Timeouts exist also for entire transactions.
- *Extension of coordination expressiveness*: The coordination law³² of a container is not limited to template matching, but supports many system coordinators like FIFO, LIFO, any, random, label, key, vector, template matching, etc. for container access, as well as the possibility to add application-defined coordinators.

Fig. 2. Distributed XVSM spaces with secure containers.

[§] www.xvsm.org.

Some coordinators require supplementary information when an entry is written to the container like key, label, index, priority, etc. The selection of entries uses *selectors* like key, template, FIFO, random, etc. Also, XVSM supports a general query mechanism for the selection of entries comparable to SQL without joins.²⁹ The concept of coordinators clearly separates application data that reflect the business logic from coordination data that are needed for the collaboration and coordination of the communication parties.

- *Bulk operations*: XVSM foresees a count specification for bulk operations. The count is a number (exact amount), range, **ALL** (all available entries, possibly 0) or **NONE** (no entry must exist that matches). If the count is not fulfilled, the operation fails or blocks (depending on its timeout). A **write** operation may write multiple entries in one single step.
- *Aspect orientation*: The space functionality is extensible through *aspects* (comparable to reactions in LIME³³), which are application-defined functions that are injected before or after a certain space operation. This way, any kind of extension profile (e.g. for security, replication, lookup, loading data from a database, etc.) can be plugged into the middleware.
- *Interoperable communication protocol*: The reference architecture defines an open protocol that allows applications written in different programming languages and running on heterogeneous platforms to communicate via the shared space.
- *Security*: A recent extension is a role-based access control model. The Peer Model security³⁴ relies on the XVSM security model³⁵ for the space containers.

XVSM generalizes the concept of tuple spaces and queues — it realizes indirect communication with a medium that is neither restricted to FIFO queues nor to template matching. Due to this generality it subsumes queueing, repository, blackboard, client/server, peer-to-peer, publish/subscribe and event-based communication.

4.1.3. Peer

A peer has a unique name by which it is identified. It stands for any kind of processing unit with a well-defined behavior like a service provider, consumer, sensor, machine in a factory, robot, agent, repository, etc. It reacts on entries, processes them and produces result entries. Each peer possesses two space containers that store entries, termed *peer input container* and *peer output container*, respectively. Containers allow concurrent access to shared entries and provide decoupling of the communication partners. The behavior of the peer is specified by means of *wirings* and *subpeers*. Subpeers encapsulate behavior that is only needed by the peer to which they belong to and thus is not visible from outside.

4.1.4. Wiring

Wirings are the active parts as they model concurrent processes. They react on the arrival of entries, trigger the right services to process the entries and finally

transport the result entries either to a container of the peer or to the PIC of a *subpeer*. They may throw away entries if they are not needed any more. The runtime instance of a wiring has its own execution thread and is termed *wiring instance*. Many wiring instances may run in parallel, controlled by the wiring system coordination property `max_threads`. After an instance terminates, it restarts again (controlled by the system coordination property `repeat_count`).

Wirings are comparable to behaviors in the Actor Model²⁰ and to transitions in Petri Nets,²¹ albeit with a slightly different semantics. A wiring has *guard* and *action* links and an internal container, without transactions and without blocking behavior, to temporarily store entries retrieved by guards or produced by services called by the wiring. Guards transport entries from a peer container to the internal wiring container; actions transport entries in the other direction. Only entries with compatible `fid` are considered by guards: All entries with defined `fid` must have the same `fid`. An exception are entries transported on a guard whose link property `flow` is turned off. The first entry with defined `fid` (if any) that is transported by a guard determines the `fid` of the current wiring instance. If there is no such entry, the `fid` remains undefined. The `fid` of newly created entries is undefined, if not explicitly set.

In the specified order, the wiring first executes the guard links, then its service(s) and finally the action links. Figure 1 shows an example peer that uses N wirings $\text{Wiring}_1, \dots, \text{Wiring}_N$. Wiring_1 has two guard and two action links, whereas Wiring_N has one guard and two action links.

4.1.5. Link

A link comprises a source container, a target container, an operation, a query, a sequence of assignments (of local wiring variable and/or entry properties) and link system coordination properties. If an entry property is denoted in an assignment, this refers to the first selected entry selected by the query. The link may use application or system variables. The latter are automatically set by the framework. The scope of application variables is the wiring instance. The scope of system variables depends on what they stand for, e.g. there are global variables denoting the current peer, the actually read amount of entries on a link, etc.

Link operations are `take`, `read` and `create` (to create entries). `Take` and `read` relate to the source container of the link. All selected or created entries are automatically written into the target container with a `write` operation. The query consists of: (i) A type specification that refers to the coordination type of the entry (on a link only entries of the same type can be transported). (ii) A count specification (given in square brackets), saying how many entries must be selected by the operation (*cf.* count specification in XVSM). The default value is 1. The count must be fulfilled. (iii) A selection specification (*cf.* XVSM query mechanism).

A link has either the role of *guard* or *action*. Guard links transport entries from a peer container or a subpeer's POC to the wiring's internal container, and

action links from the wiring's internal container to a peer container or a subpeer's PIC. All links of a wiring are numbered. The wiring executes them in the specified order in one transaction. If a mandatory link execution fails, depending on the link coordination property `rollback_on_failure` either this wiring instance execution fails and a rollback takes place or the link conceptually "blocks," meaning that locks set by link operations on containers are held and the wiring instance is waked up to continue when necessary conditions to proceed are fulfilled. The maximum number of concurrent wiring instances is configurable by the system wiring property `max_threads`; `mandatory` is a link system property that states whether the execution of a link must succeed or is optional.

In addition, there exists a special guard (marked with `*`) termed *init guard* which is exactly once fulfilled, namely when the peer is started. Only one wiring may use an init guard. Analogously, also shutdown guards are supported.

Note that all Peer Model elements (entry, container, wiring, link, peer and service) possess system properties. For example, the action link property `dest` means that all entries transported by this link are wrapped into a "cargo" entry with `dest` set to `dest` of the link; it is written into the PIC of the local I/O peer (see below) that takes care of the asynchronous delivery.

4.1.6. Service

In between guard and action execution, wirings may call *services*. Services wrap application functions which are not modeled by the Peer Model. Services receive as input all entries that are currently in the wiring's internal container. All results they produce are emitted as entries into the wiring's internal container.

4.2. Exception handling

Exceptions occur if, for example, a `t1` expires, a service fails, the I/O peer cannot deliver an entry, etc. For example, if the `t1` of an entry expires, the original entry is converted into an exception entry that has the coordination type `exception`. The original entry is wrapped into the `exception` entry. The destination location of the `exception` entry may be explicitly specified with the system property `t1_exc_dest`. If it is not set, the exception is ignored and does not cause the creation of an `exception` entry.

Entries of type `exception` are treated like any other entry in the Peer Model. They are transported by wirings that use `exception` as coordination type in the link's query. An `exception` entry also possesses a `t1` as well as other system properties; by default if an `exception` entry expires, its exception is ignored if not configured otherwise.

4.3. System peers

System peers are modeled by means of the already described mechanisms. They are a configurable constituent of a Peer Model runtime system.

4.3.1. I/O peer

The I/O Peer Models the asynchronous communication between peers. It takes entries from its PIC (which must have their **dest** coordination property set). If **dest** denotes a local peer, the entry is written to this peer's PIC within an auto-committing transaction; otherwise the I/O peer calls a configurable "send" service to send to the remote site's I/O peer that employs a corresponding receive service. The remote I/O peer delivers the received entry to the right destination peer's PIC. If the entry is the cargo entry then the entries it contains are unwrapped and written into the PIC of the destination peer.

4.3.2. Meta model peer

The Peer Model possesses a meta model that is modeled by means of space containers.³⁴ A meta model peer accepts entries that represent commands to change the meta model, e.g. add/delete a peer/wiring/pattern, etc. Predefined entry types that the meta model peer accepts are **add** and **delete**. The details about the added or deleted components are given by means of system properties of these entries.

4.4. Graphical notation

Figure 3 shows the graphical representation of an example peer modeling a client issuing tasks to be performed by workers. Section 5 integrates this client peer in a load balancing example. The graphical notation abstracts a peer as two gray boxes — standing for the PIC and POC, respectively, a box that holds the peer's name and a box in the middle, where the behavior of the peer is modeled with wirings and subpeers. Guard and action links are represented by arrows. Above the arrow the operation on the source container is given (**read** or **take**) or **create**, followed by the coordination type of the entries to be selected (or created), a count specification (default is 1) in "[]" brackets and a selector (*cf.* XVSM selector) in "[[]]" brackets; next, the setting of entry properties and/or local wiring variables is denoted in "< >" brackets. Below the arrow, the link properties are modeled. Operation and type are mandatory, the other specifications are optional.

The example client peer uses application entries of type **start**, **ctrl**, **answer** and **task**. It possesses four wirings termed **InitWiring**, **RequestWiring**, **TreatAnswersWiring** and **ExceptionWiring** whose instances run concurrently. The maximal number of instances is not explicitly bounded in the example; the determination of the optimal amount is left to the runtime system.

Variables syntactically start with "\$\$" (system variables) or with "\$" (application variables). The former ones are written in upper case letters, the latter ones start with a lower case letter. For example, \$\$THIS_PEER denotes the name of the current local peer, and \$\$CNT the number of entries selected by the current link. Configurable pattern parameters start with "\$" and are written in upper case letters.

Fig. 3. Client peer.

- The InitWiring is marked by the init guard "*" and fires exactly once when the client peer is initialized. It creates \$MAX_FLOWS start entries and writes them into the PIC. \$MAX_FLOWS is a configurable parameter of this peer and its meaning is how many flows (composed of an arbitrary number of tasks) a client may start concurrently. The start entries have no flow id, this means they are compatible with any other flow.
- RequestWiring has one guard (G1) and two actions (A1 and A2). G1 waits until one entry of type start exists in the client peer's PIC, takes this entry out of the PIC and writes it into the wiring's internal container. Then a configurable service (denoted by the parameter \$GENERATE_TASKS) is called that generates one or more task entries and writes them to the internal wiring container. A1 takes all task entries from the wiring's internal container. It sets four properties on each of them: the application property client, which holds the name of the requesting client peer, is set to the name of the local peer so that the answer can be sent back; ttl is set to the configuration parameter \$TASK_TTL that defines how long a client is willing to wait for an answer — if it expires, the entry turns into an exception; fid is set to a new flow identification number using "fid()"; ttl_exc_dest is set to this peer — this property controls explicitly to which

peer an **exception** entry caused by **t1** expiration is written. A1 transports all tasks that were generated by the service to the load manager: It sets two local wiring variables: The newly generated **fid** is stored in the local variable **\$fid** and the number of read entries (**\$\$CNT**) is stored in the local variable **\$n**. The link property **dest** causes all tasks transported along this link to be wrapped into a “cargo” entry with **dest** set to the load manager peer denoted by the configurable parameter **\$LOAD_MANAGER**. If the load manager peer is local, the I/O peer unwraps the cargo entry and directly writes the task entries contained in it into the load manager peer’s PIC. Otherwise, it asynchronously delivers the cargo entry to the I/O peer located at the site of the remote load manager peer. The remote I/O peer then unwraps the cargo entry and writes the task entries contained in it into the remote load manager peer’s PIC. A2 creates a **ctrl** entry, sets its **fid** to the flow to which the tasks belong to, sets the application property **ntasks** to the number of tasks of this flow and writes the entry into the PIC.

- **TreatAnswersWiring** takes the **ctrl** entry from the peer’s PIC (G1) and stores its **ntasks** property in the local variable **\$n**. This variable is used in G2 as query count. Here the flow correlation comes into place: Only answers that belong to the same flow are taken. Finally in A1 the wiring plays back one new start entry into its PIC so that the client can issue one more flow.
- **ExceptionWiring** treats the case that a task entry expired and became an exception that was propagated back to the client’s PIC. The exception has the same **fid** as the entry that expired. It removes the **ctrl** entry belonging to the same flow via G2 and creates a new start entry (A1). All other **exception** and answer entries are eventually cleaned up when their **t1** expires.

4.5. Pattern composition

Pattern composition³⁶ is based on the Peer Model concepts: Any collection of these concepts may form a pattern that is parameterized via the variables it uses.

The graphical notation of a pattern is to group the pattern elements by means of a dotted line. An example is given in Fig. 4, where a wiring is defined as a pattern. It uses six configurable parameters: **\$PIC** and **\$POC** (to be configured with the peer’s PIC and POC, respectively, to which this wiring shall be added), **\$SELEC-TOR** (selector part of the query), **\$PROCESS_TASK** (a service), **\$ANSWER_TTL** (life cycle time of the answer) and **\$MAX_WORKERS** (controlling the maximum number of allowed concurrent instances of this wiring).

Fig. 4. Worker pattern.

5. Proof-of-Concept

As a proof-of-concept for a recurring coordination pattern, the load balancing coordination pattern is selected. The load balancing problem is well known from literature and we have identified and described it in previous work to demonstrate the usefulness of different swarm-based algorithms.³⁷ The load balancing solution served as a testbed to compare the different algorithms. The pattern assumes active workers that autonomously fetch tasks if they are idle — selecting them according to their priority, time of arrival or fetching those that match their skills.

In this paper, a Peer Model-based specification of the pattern is presented. Relying on this pattern, generic coordination components can be developed as solutions to be applied in real industrial use cases.

5.1. Example: Modeling of a load balancing pattern solution

A generic pattern for balancing of load³⁸ in a distributed network without a central coordinator assumes autonomous workers with different capabilities: The participant roles of the load balancing pattern are clients, workers and agents for allocation and routing. The basic abstraction is that clients generate tasks and send them to a load manager. On each node there exists one load manager peer and many client peers. The load manager maintains workers of varying number according to the actual load, and optionally also the allocation and routing agents. The workers actively and autonomously fetch tasks from a shared load “space,” process them and push the answers back to the requesting client. The agents take care of the allowed number of workers on a node, and that the load is fairly distributed to the nodes in the network. They react adaptively to the current load situation. The pattern is able to cope with worker failures. If also node failures shall be masked without losing already issued tasks, composition with a replication pattern is necessary. In this paper, the original pattern is refined in that it may also adjust the number of workers on a node, if necessary.

Two policies termed *transfer policy* and *location policy* drive the load balancing pattern. The transfer policy determines whether a node is underloaded (“UL”), ok-loaded (“OK”) or overloaded (“OL”). It may also tune the local node by adding or removing workers.[¶] In the UL case it gets work from another node and/or withdraws workers, in the OL case it gives away work to another node in the network and/or spawns further local workers. Whether one or both of underloadedness and overloadedness triggers the transfer of work is configurable. The location policy assumes that at any point the information about a currently best partner is available with whom to exchange tasks. This either is statically configured or dynamically accomplished by routing agents. The location policy finds the best partners in the network with whom a node shall exchange load. This varies dynamically and therefore this policy is applied continuously. The allocation agents have the responsibility to take

[¶]This functionality could be delegated to another service.

care of the transfer policy, and the routing agents of the location policy algorithm. Routing agents that collaborate must implement the same location policy. Routing agents with different algorithms can be deployed. Depending on the topology configuration, for example in a subnetwork a round robin algorithm may be used, in another one a genetic algorithm and between the subnetworks a swarm intelligence algorithm. Both policies are configurable. We have benchmarked an XVSM-based implementation of the pattern with bioinspired location policy algorithms (ants and bees) as well as with classical ones³⁷ with different configurations and distributed topologies.

In the following subsections, all needed pattern solutions are modeled with the Peer Model and finally composed towards the generic load balancing pattern solution. Note that in all subsequent patterns PIC and POC are configurable via the respective parameters \$PIC and \$POC.

5.1.1. *Client peer pattern*

The client peer pattern has already been introduced in Fig. 3.

5.1.2. *Worker pattern*

The worker pattern (see Fig. 4) consists of one wiring termed WorkerWiring. It models a worker that in G1 waits for the next task, takes it from PIC using a configurable query selector (parameter \$SELECTOR) that, for example, selects the next task in FIFO order, or by a query on application coordination properties that match with the role and capability of this worker, remembers which client has issued the task and also its `fid`, processes the task by means of a configurable service (parameter \$PROCESS_TASK) and in A1 sends the answer within the same flow and with `ttl` set to the configurable parameter \$ANSWER-TTL back to the client.

The number of concurrent instances of WorkerWiring is bounded by the wiring system coordination property `max_threads` which is set to the configurable parameter value \$MAX_WORKERS. The pattern is a local node pattern. Through composition with other patterns (allocation and routing) it becomes operable in a distributed setting (shown below).

5.1.3. *Allocation pattern*

The allocation pattern consists of one basic and two optional patterns.

The *basic allocation pattern* (see Fig. 5) comprises one link and one wiring:

- The first element is a single link that is intended to extend the InitWiring of the peer into which the pattern is composed. The action number is configurable via \$A. It creates a status entry with the application property load set to OK.
- The second element is the AllocationWiring which continuously (in the start time interval configured by the parameter value \$ALLOCATION_TTS of the system

Fig. 5. Basic allocation pattern.

property **tts** on its first guard link G1) checks the load status of its peer: It takes the status entry (G1) and writes the updated one back into the PIC (A1). For updating its application property **load** it involves a configurable service that executes the transfer policy (configured by the parameter **\$TRANSFER_POLICY**) on this node. The wiring passes the information about all locally available tasks (retrieved by G2) and the current status to the service. The transfer policy computes and sets the load on the status entry and emits it into the internal wiring container. In addition it may generate entries that request to add or delete workers (A2 and A3) at the local meta model peer. The counter specification **ALL** accepts 0, 1, 2, ..., of these request entries.

The basic allocation can be refined by adding pull and/or push patterns: The *pull pattern* (see Fig. 6) has two wirings. It assumes that a best partner for exchanging of tasks is always stored in an entry of type **partner** in its PIC. As explained above, this entry is either statically configured or dynamically renewed by the below routing pattern.

- PullWiring continuously reads the local status (the pull interval is configured with **\$PULL_TTS** on G1) and if it is **UL**, it retrieves a currently suited partner

Fig. 6. Pull pattern.

Fig. 7. Push pattern.

whose load status is “OL” (G2). The partner selection consists of a query that first selects all OL partners and then (using the “|” operator of the XVSM query language which “pipes” the result of the first selector into the second one) randomly takes one of them (using the XVSM container coordinator **RANDOM**). A1 creates and sends a pull request entry to this partner and sets its **ttl** to the configurable parameter **\$PULL_TTL**.

- **TreatPullWiring** treats pull requests (G1) obtained from other agents. It removes a local task (G3) and sends it to the requester of the pull request (A1). In order to prevent oscillation, it checks in G2 the current status to be OL.

The *push pattern* (see Fig. 7) has one wiring:

- **PushWiring** checks the current load status to be OL (G1), retrieves a best partner (G2) in any order (which is the default coordination law), takes one task from the local load space represented by PIC (G3) and directly pushes it to the partner peer (A1). The interval in which the **PushWiring** becomes active is controlled by the **tts** on its G1, which is set to the configurable parameter **\$PUSH_TTS**.

5.1.4. Routing peer pattern

Figure 8 shows a very abstract routing peer pattern that assumes an unstructured peer-to-peer overlay network.³⁹ It has one link and three wirings:

- The action link denoted by \$A must be configured to be part of the **InitWiring** of the respective peer. It creates one **routingInfo** entry holding information about the overlay topology structure for the routing algorithm as well as other knowledge to be shared among the agents.
- **RoutingWiring** figures out the right partner to exchange work with. It waits for a start entry (G1) which is the signal when to start the routing algorithm. Then it takes the routing information (G2), takes all **routingMessages** (G3) if there are any, reads the local load status (G4), takes all partner information (G5), executes the location policy (which is a configurable service — see parameter **\$LOCATION_POLICY**) that updates partner(s) and **routingInfo** and in addition optionally generates **routingMessages** entries, stores back the updated routing information (A1) and the updated partner information (A2) into its PIC and sends out **routingMessages** (with treated flag set to **false**) if the service generated

Fig. 8. Unstructured peer-to-peer routing pattern.

any (A3). The service sets the **dest** property appropriately on all emitted routingMessage entries, based on the routing information it got; this causes A3 to deliver them to the I/O peer instead of the POC. Note that the exchange of routingMessages entries among the agents is necessary so that they can build up shared knowledge about the network (e.g. in swarm-based ant algorithms they will send pheromones³⁷).

- ReactOnMessageWiring creates a start entry (A1) to trigger the RoutingWiring, if a routingMessage is received (G1) that was not yet treated by this wiring. It updates the application property treated of the routingMessage to **true** (A2).
- ScheduleWiring creates a start entry (A1) to continuously trigger the RoutingWiring in a defined time interval, configured with \$ROUTING_TTS.

5.1.5. Composition and configuration

The solution can now be constructed by composing the client, worker, allocation and routing pattern solutions (see Fig. 9) by employing all subpatterns of the allocation pattern. The values for the configuration parameters of the patterns are not shown in the example. Any numbers of clients and worker patterns (e.g. with different query selectors) may be configured. Also, several routing patterns with different algorithms can be configured simultaneously.

Fig. 9. Composition of a load balancing pattern.

6. Related Coordination Models

In this section, models are discussed with respect to their suitability for the modeling of coordination patterns and obtaining generic coordination components as pattern solutions.

6.1. Colored and timed Petri Nets

Petri Nets²¹ are a well-funded, token-driven model for concurrent systems. They are very generally applicable, but provide modeling on a rather low level. Colored Petri Nets⁴⁰ combine Petri Nets with the Standard ML language and introduce colors for tokens to model their data value. Colored Petri Nets includes a timing concept. They support modules to build hierarchical structures. Parametrization is possible with help of global variables. Timed Petri Nets⁴¹ introduce the possibility to model the duration of transitions and the life cycle of tokens.

However, Petri Nets have the problem that there is no clear separation of application and coordination logics. Therefore coordination generics cannot be extracted in the form of patterns. Errors must be handled explicitly without further modeling support. Correlation of flows is not foreseen. The model is static and does not support any domain-specific abstractions that could hide details. Albeit a graphical notation is provided and modeling tools exist, designs become unreadable if the problem size increases.^{16,42} The tradeoff is that the static nature of Petri Nets enables system verification and corresponding tools are supported for verification and simulation. Runtime systems for different platforms are not provided.

6.2. Reo

Reo¹⁵ is a data-driven model for the composition of concurrent software components and web services, providing *exogenous coordination*. This implies a separation of communication and application logics.

Channels are the most basic form of connectors and there are primitives for synchronous and asynchronous communication, FIFO order and filtering. Communication is carried out via input and output operations on channels. Complex connectors can be composed of simple connectors. Moreover, connectors can be reused and parameterized. Business logic is realized within the composed software components, while the primitives used for building the connectors form the coordination logic.

Reo is supported by a graphical notation. It is powerful but too verbose for creating more complex coordination patterns. Basically, connectors represent self-contained components. Even small changes in the requirements could lead to far-reaching modifications and may have crucial side effects, which are rather difficult to comprehend compared to other coordination models. Neither flow correlation nor error handling is considered in the model. Dynamic topology changes are possible and there exist also extensions for the treatment of time^{43,44} based on timed automata. Pattern-based data selection on channels can be seen as a basic form of coordination principle.

Reo supports verification, but there is no runtime support.

6.3. Actor Model

The Actor Model^{20,45} is a mathematical model for concurrency. Actors are the basic building blocks and communicate via asynchronous messages. In response to a received message, an actor can create further actors, send messages to itself or other actors and dynamically change its behavior by specifying how to handle the next message. Actors form hierarchies, complex tasks are split up and delegated to child actors. Thus, reuse and modularity is supported. Parametrization depends on the concrete realization of the Actor Model.

However, separation of concerns is not explicitly given; it depends on the discipline of the developer who must explicitly take care for a layered design. Developers are also responsible to take care of configuration and extension mechanisms, but this often leads to complex actors, where different functionalities are mixed.

An extension to factor in timing has been proposed by introducing deadlines for the reception of job fulfillment messages.⁴⁶

Many frameworks that are based on the Actor Model's concepts exist for different platforms like Java (Akka)⁴⁷ and .Net (Orleans).⁴⁸ These frameworks offer the control of process instances (aka actors) and a well-defined communication abstraction via asynchronous messages and FIFO queues. But the existing frameworks do not connect directly to the formal Actor Model, which precludes the benefit of formal verification. The Actor Model lacks coordination principles for the selection of messages as well as flow correlation and error handling. However, most frameworks support features of this kind, albeit they are not defined in the model.

6.4. Business process modeling notation

Business Process Modeling Notation (BPMN) is a notation that provides language constructs that can be considered structural elements of coordination (e.g. choice, sequence and parallel execution; message reception and sending; waiting times; flow correlation; error handling, etc.). They can be parameterized and composed towards more complex structures. However, they do not lend themselves for the construction of patterns due to the lack of an explicit templating mechanism. This means that every new coordination situation must be modeled individually. The BPMN does not allow to factor out the coordination mechanism of a whole process. A graphical modeling notation is provided and models provide a high usability.

Many frameworks that implement this language constructs exist; however, they assume a centralized instance to control the workflow. BPMN does not provide a ground model.¹³ Dynamic adaptation of business processes is not possible. BPMN-based models focus on the sequence of activities that make up a single business process. This is in contrast to coordination which puts the interdependency between many processes in focus.¹

6.5. Peer Model

The Peer Model is a coordination model that fulfills all criteria (see Sec. 3):

- Concurrency control is modeled by the number of wirings in a peer, the coordinators of the space container that control the links' access to entries and the wiring system property `max_threads` that models the maximum number of allowed threads for wiring instances.
- Separation of concerns is explained in Fig. 1. Application logic is wrapped into services; it is exchangeable by configuration. Coordination and communication concerns can be extracted and reused in other use cases, within different topologies and with other parameters. The system coordination property `data` of entries encapsulates the application-specific data and treats them as a black box.
- Flow correlation is implicit in that a wiring only treats entries that are flow compatible. For example, in the client example (see Fig. 3), the `TreatAnswersWiring` takes only answer entries whose `fid` is compatible with the `fid` of the `ctrl` entry and with each other; analogously for the `ExceptionWiring` of the client.
- Time is modeled with the system coordination properties `tts` and `ttl`. These are supported for all ground concepts of the Peer Model.
- Pattern composition (see Sec. 4.5) was demonstrated by the example of load balancing pattern solution, which is composed out of several other patterns (see Fig. 9).
- Error handling uses a special entry type named `exception` to signal an error. The Peer Model runtime generates an `exception` entry in case an error occurs. For example, in the running load balancing example, on each task that a client

of the example shown in Fig. 3 generates, A1 of RequestWiring sets its `ttl` and `ttl_exc_dest`. If a task expires somewhere during its flow, an `exception` entry is therefore written back to the client's PIC. In the worker pattern (see Fig. 4, WorkerWiring (A1)) the answer has a `ttl`, and in the pull pattern (see Figure 6, PullWiring (A1)) the pull request has a `ttl`.

- Dynamic changes are possible, because the Peer Model possesses a meta model. The access to the meta model is accomplished by a system peer termed meta model peer (see Sec. 4.3). This feature is used in the example basic allocation pattern (see Fig. 5, AllocationWiring (A2 and A3)) to add or delete workers on demand.
- The usability of the Peer Model has been evaluated by means of a .NET framework termed "PeerSpace.net" that implements a lean version of the Peer Model,⁴⁹ and by using the API Concepts Framework measurement method.⁵⁰ The result was: "the PeerSpace.net framework is better suited to implement complex distributed systems than the SOA-focused WCF framework, however, even in case of SOA patterns like request/multiple-response the PeerSpace could keep up and even surpass the WCF framework's API usability".⁴⁹ We plan to carry out further usability studies in the future work.
- There is a single coordination principle, namely the query mechanism of the space-based XVSM containers. This mechanism is used on all guard and actions links. It controls the way how entries are selected and also the concurrency of wirings (e.g. a FIFO coordinator will constrain concurrency, whereas an "any" coordinator will enable it). More sophisticated concurrency control coordinators are currently under investigation. In the worker pattern example (see Fig. 4), a configurable query selector is used to select the next task that a worker wants to process.
- A graphical notation is supported that was introduced in Sec. 4.4 and used in the examples.
- Prototypes of runtime systems of the Peer Model are currently provided for different platforms: embedded C,⁵¹ .Net,⁴⁹ Go and Java. The Peer Model is independent on the runtime system and platform, as well as on the distribution and topology.⁵¹ It provides an abstraction of remotng by means of the system coordination property `dest` and the I/O system peer. This property is used, for example, in the worker pattern example (see Fig. 4) to direct the answer back to the right client. The system coordination property `max_threads` allows — beyond the above described mechanisms — modeling of the restrictions for process instances in the runtime.

7. Coordination Patterns in Some Industrial Projects

Experience with several industrial projects is summarized that demonstrates that it is possible to identify a lot of recurring coordination patterns beyond the load balancing pattern in cooperative information systems. The identified coordination

Fig. 10. Use cases from different traffic management domains. (a) Notification of level crossing about approaching train. (b) Infrastructure to vehicle communication. (c) Ground-to-ground communication of air traffic data. (d) Near-time tracking and monitoring of trucks.

patterns can also be found in many other domains. The claim is not yet a systematic coordination pattern catalog; this is subject to future work, but to indicate the level of abstraction that a coordination pattern shall support. First the use case is described and then the coordination patterns that were identified.

7.1. Railway

In the railway use case (see Fig. 10(a)) trains are detected by means of sensors and this information must be reliably and timely forwarded to the next level crossing where the road users are informed about the approaching train. The information must be transported by means of low-cost smart nodes along the tracks that communicate via radio, instead of using expensive copper cables.

Coordination patterns: *Reliable forwarding*⁵² of information in a peer-to-peer manner without assuming a central coordinator and under the assumption that single nodes may fail.

7.2. Road/cars

The roads/cars use case (see Fig. 10(b)) copes with the development of a scalable and dependable vehicle-to-infrastructure telematics system. A telecommunication control center publishes traffic information (e.g. traffic congestion, speed limit, construction warning, accident and ghost driver) that must be forwarded in near-time

to the on-board unit in the cars concerned by this information. The road infrastructure foresees distributed road-side units that communicate via WiFi with the cars passing by. The connection time of a car passing by is limited to only a few seconds. The information must be reliably routed and replicated in the peer-to-peer network of road-side units to guarantee that despite of interruptions the car gets the information. Also, cars sense data and feed them back to the infrastructure so that they may in turn contribute to further traffic messages.

Coordination patterns: *Reliable forwarding*⁵³ and *replicating*⁵⁴ traffic information in the peer-to-peer network. *Load balancing*⁵⁵ of data and processing tasks among the road-side units.

7.3. Air

This use case (see Fig. 10(c)) deals with the development of an information sharing network for the air traffic management domain.⁵⁶ Different heterogeneous sources (radar, weather stations, etc.) must be integrated. For example, the system must guarantee that the central flight controller and ground control receive timely weather and radar data from at least three stations each. These data are routed across many countries. There is no centralized control.

Coordination patterns: *Reliable forwarding* and *replication* of data. *Automatic failover* if nodes are not available, by using alternative routes.

7.4. Road/trucks

See the use case in Fig. 10(d): Fleets of trucks carrying dangerous goods (oil and gas) are equipped with a mobile box that in regular intervals sends the position data as well as other logistics information about the truck to a computing center, run by a service provider. The provider offers the vehicle fleet operator interfaces to monitor its truck in near-time and to pose queries. This causes heavy loads at the provider's site and a parallelization and load balancing of the software running there must be guaranteed, as well as resistance against failures.

Coordination patterns: *Load balancing* and *failover* in the backend, i.e. the private cloud of the provider.

7.5. Summary

The load balancing pattern (see Sec. 5) occurred in two of the presented use cases, namely in road/cars and in road/trucks. In both use cases the same pattern solution could be applied, only the services were implemented differently, and other configuration parameters were used. Forwarding, replication and failover are other coordination patterns that occurred often in industrial use cases.

8. Conclusion

A new approach to configure and compose cooperative information systems from reusable, proven, open, standardized, interoperable *coordination components* was presented as a step towards the vision to ease the development by means of prefabricated software building blocks. These components can be used cross-domain and on different platforms. As a proof-of-concept, a load balancing pattern has been shown that we have observed in different real-life use cases.

The approach is based on the idea to separate all coordination concerns from the application logic. Coordination is independent from transport mechanisms and platforms. The recurring generics in coordination are identified and modeled in the form of reusable patterns. For the modeling a suitable coordination model is needed. The Peer Model integrates concepts of space-based computing, Petri Nets, actors and staged event-driven architectures. It consists of five basic concepts termed peer, entry, container, wiring and service. Coordination patterns modeled with the Peer Model can be composed and configured towards more expressive patterns. The Peer Model offers a ground model for the specification of pattern solutions and in the future work we plan to investigate the simulation, modeling and verification tools for it. Code generation from the domain-specific language of the Peer Model to different languages and platforms is possible. This allows bridging the gap between design and implementation.

The advantage of the pattern-based approach is that the reuse principle leads to faster and less risky developments. Future work will also comprise a catalog of coordination patterns.

Acknowledgments

Many thanks to Stefan Craß, Geri Joskowicz, the editors and the anonymous reviewers for their feedback on this paper, and Herbert Pohlai for proofreading it.

References

1. T. W. Malone and K. Crowston, The interdisciplinary study of coordination, *ACM Comput. Surv.* **26**(1) (1994) 87–119.
2. A. Ranganathan and R. H. Campbell, What is the complexity of a distributed computing system? *Complexity* **12**(6) (2007) 37–45.
3. J. A. McDermid, Complexity: Concept, causes and control, in *Proc. Sixth IEEE Int. Conf. Engineering of Complex Computer Systems (ICECCS)* (2000), pp. 2–9.
4. B. Colwell, Complexity in design, *Computer* **38**(10) (2005) 10–12.
5. C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King and S. Angel, *A Pattern Language: Towns, Buildings, Construction* (Oxford University Press, 1978).
6. C. Kühn and M. Herzog, Modeling the representation of architectural design cases, *Autom. Construct.* **2**(1) (1993) 1–10.
7. E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software* (Addison-Wesley, 1995).

8. D. C. Schmidt, M. Stal, H. Rohnert and F. Buschmann, *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects* (Wiley, 2000).
9. J. Nitzsche, T. van Lessen and F. Leymann, WSDL 2.0 message exchange patterns: Limitations and opportunities, in *Proc. Third Int. Conf. Internet and Web Applications and Services (ICIW)* (IEEE, 2008), pp. 168–173.
10. G. Hohpe and B. Woolf, *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions* (Addison-Wesley, 2003).
11. W. van der Aalst, A. ter Hofstede, B. Kiepuszewski and A. Barros, Workflow patterns, *Distrib. Parallel Databases* **14**(1) (2003) 5–51.
12. W. van der Aalst and A. ter Hofstede, Workflow patterns put into context, *Softw. Syst. Model.* **11**(3) (2012) 319–323.
13. E. Börger, Approaches to modeling business processes: A critical analysis of BPMN, workflow patterns and YAWL, *Softw. Syst. Model.* **11**(3) (2012) 305–318.
14. C. Fehling, F. Leymann, R. Retter, W. Schupeck and P. Arbitter, *Cloud Computing Patterns — Fundamentals to Design, Build, and Manage Cloud Applications* (Springer, 2014).
15. F. Arbab, Reo: A channel-based coordination model for component composition, *Math. Struct. Comput. Sci.* **14**(3) (2004) 329–366.
16. E. Kühn, S. Craß, G. Joskowicz, A. Marek and T. Scheller, Peer-based programming model for coordination patterns, in *Proc. 15th Int. Conf. Coordination Models and Languages (COORDINATION)* (Springer, 2013), pp. 121–135.
17. D. Gelernter, Generative communication in Linda, *ACM Trans. Program. Lang. Syst.* **7**(1) (1985) 80–112.
18. 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 *Proc. 8th Int. Conf. Autonomous Agents and Multiagent Systems (AAMAS)* (IFAAMAS, 2009), pp. 625–632.
19. P. Th. Eugster, P. Felber, R. Guerraoui and A.-M. Kermarrec, The many faces of publish/subscribe, *ACM Comput. Surv.* **35**(2) (2003) 114–131.
20. G. A. Agha, *ACTORS: A Model of Concurrent Computation in distributed systems* (MIT Press, 1990).
21. C. A. Petri, Kommunikation mit Automaten, Ph.D. thesis, Technische Hochschule Darmstadt (1962).
22. E. Börger and J. K. Huggins, Abstract state machines 1988–1998: Commented ASM bibliography, arXiv:cs/9811014 [cs.SE].
23. N. Busi, R. Gorrieri and G. Zavattaro, Comparing three semantics for Linda-like languages, *Theor. Comput. Sci.* **240**(1) (2000) 49–90.
24. E. Denti, A. Natali and A. Omicini, Programmable coordination media, in *Proc. Second Int. Conf. Coordination Languages and Models (COORDINATION)* (Springer, 1997), pp. 274–288.
25. A. L. Murphy, G. P. Picco and G.-C. Roman, LIME: A coordination model and middleware supporting mobility of hosts and agents, *ACM Trans. Softw. Eng. Methodol.* **15**(3) (2006) 279–328.
26. A. Omicini and F. Zambonelli, TuCSoN: A coordination model for mobile information agents, in *Proc. 1st Workshop Innovative Internet Information Systems* (1998), pp. 177–187.
27. A. I. T. Rowstron and A. Wood, Solving the Linda multiple RD problem, in *Proc. First Int. Conf. Coordination Languages and Models (COORDINATION)* (Springer, 1996), pp. 357–367.

28. E. Kühn, J. Riemer, R. Mordinyi and L. Lechner, Integration of XVSM spaces with the web to meet the challenging interaction demands in pervasive scenarios, *Ubiquit. Comput. Commun. J.* **3** (2008) 1–12.
29. S. Craß, E. Kühn and G. Salzer, Algebraic foundation of a data model for an extensible space-based collaboration protocol, in *Proc. 2009 Int. Database Engineering and Applications Symp. (IDEAS)* (ACM, 2009), pp. 301–306.
30. S. Craß, A formal model of the extensible virtual shared memory (XVSM) and its implementation in Haskell — Design and specification, Master’s thesis, TU Vienna (2010).
31. E. Freeman, K. Arnold and S. Hupfer, *JavaSpaces Principles, Patterns, and Practice*, 1st edn. (Addison-Wesley, 1999).
32. P. Ciancarini, Coordination models and languages as software integrators, *ACM Comput. Surv.* **28** (1996) 300–302.
33. G. P. Picco, A. L. Murphy and G.-C. Roman, LIME: Linda meets mobility, in *Proc. 1999 Int. Conf. Software Engineering (ICSE)* (ACM, 1999), pp. 368–377.
34. S. Craß, G. Joskowicz and E. Kühn, A decentralized access control model for dynamic collaboration of autonomous peers, in *Proc. 11th EAI Int. Conf. Security and Privacy in Communication Networks (SecureComm)* (Springer, 2015), pp. 519–537.
35. S. Craß, T. Dönz, G. Joskowicz, E. Kühn and A. Marek, Securing a space-based service architecture with coordination-driven access control, *J. Wirel. Mob. Netw. Ubiquit. Comput. Dependable Appl.* **4**(1) (2013) 76–97.
36. E. Kühn, S. Craß and G. Schermann, Extending a peer-based coordination model with composable design patterns, in *Proc. 23rd Euromicro Int. Conf. Parallel, Distributed, and Network-Based Processing (PDP)* (IEEE, 2015), pp. 53–61.
37. V. Šešum-Čavić and E. Kühn, Self-organized load balancing through swarm intelligence, in *Next Generation Data Technologies for Collective Computational Intelligence*, Studies in Computational Intelligence, eds. N. Bessis and F. Xhafa (Springer, 2011), pp. 195–224.
38. E. Kühn and V. Šešum-Čavić, A Space-based generic pattern for self-initiative load balancing agents, in *Proc. 10th Int. Workshop Engineering Societies in the Agents World (ESAW)*, Lecture Notes in Computer Science, Vol. 5881 (Springer, 2009), pp. 17–32.
39. S. Androutsellis-Theotokis and D. Spinellis, A survey of peer-to-peer content distribution technologies, *ACM Comput. Surv.* **36**(4) (2004) 335–371.
40. L. M. Kristensen, S. Christensen and K. Jensen, The practitioner’s guide to coloured Petri Nets, *Int. J. Softw. Tools Technol. Transf.* **2**(2) (1998) 98–132.
41. F. D. J. Bowden, A brief survey and synthesis of the roles of time in Petri Nets, *Math. Comput. Model.* **31**(10–12) (2000) 55–68.
42. E. Börger, Modeling distributed algorithms by abstract state machines compared to Petri Nets, in *Proc. 5th Int. Conf. Abstract State Machines, Alloy, B, TLA, VDM, and Z (ABZ)* (Springer, 2016), pp. 3–34.
43. F. Arbab, C. Baier, F. de Boer and J. Rutten, Models and temporal logical specifications for timed component connectors, *Softw. Syst. Model.* **6**(1) (2007) 59–82.
44. N. Kokash, C. Krause and E. De Vink, Time and data-aware analysis of graphical service models in Reo, in *Proc. 8th IEEE Int. Conf. Software Engineering and Formal Methods* (2010), pp. 125–134.
45. C. Hewitt, P. Bishop and R. Steiger, A universal modular ACTOR formalism for artificial intelligence, in *Proc. 3rd Int. Joint Conf. Artificial Intelligence (IJCAI)* (Morgan Kaufmann, 1973), pp. 235–245.

46. M. M. Jaghoori, F. S. de Boer, T. Chothia and M. Sirjani, Schedulability of asynchronous real time concurrent objects, *J. Log. Algebr. Program.* **78**(5) (2009) 402–416.
47. Lightbend, Inc., *Akka Java Documentation*, Release 2.4.7, edn. (Lightbend, 2016).
48. P. Bernstein, S. Bykov, A. Geller, G. Kliot and J. Thelin, Orleans: Distributed virtual actors for programmability and scalability, Technical Report MSR-TR-2014-41, Microsoft (2014).
49. D. Rauch, PeerSpace.net: Implementing and evaluating the Peer Model with focus on API usability, Master’s thesis, TU Wien (2014).
50. T. Scheller and E. Kühn, Automated measurement of API usability: The API concepts framework, *Inf. Softw. Technol.* **61** (2015) 145–162.
51. E. Kühn, S. Craß and T. Hamböck, Approaching coordination in distributed embedded applications with the Peer Model DSL, in *Proc. 40th Euromicro Conf. Software Engineering and Advanced Applications (SEAA)* (IEEE, 2014), pp. 64–68.
52. E. Kühn, S. Craß, G. Joskowicz and M. Novak, Flexible modeling of policy-driven upstream notification strategies, in *Proc. 29th Annu. ACM Symp. Applied Computing (SAC)* (2014), pp. 1352–1354.
53. S. Bessler, A. Fischer, E. Kühn, R. Mordinyi and S. Tomic, Using tuple-spaces to manage the storage and dissemination of spatial-temporal content, *J. Comput. Syst. Sci.* **77**(2) (2011) 322–331.
54. S. Craß, J. Hirsch, E. Kühn and V. Sesum-Cavic, An adaptive and flexible replication mechanism for space-based computing, in *Proc. 8th Int. Joint Conf. Software and Technologies (ICSOFIT)* (SciTePress, 2013), pp. 599–606.
55. S. Craß, E. Kühn, S. Bessler and T. Paulin, A generic load balancing framework for cooperative ITS applications, in *Proc. 3rd Int. Conf. Connected Vehicles and Expo (ICCVE)* (IEEE, 2014), pp. 385–390.
56. R. Mordinyi, T. Moser, E. Kühn, S. Biffl and A. Mikula, Foundations for a model-driven integration of business services in a safety-critical application domain, in *Proc. 35th Euromicro Conf. Software Engineering and Advanced Applications (SEAA)* (IEEE, 2009), pp. 267–274.