Chapter 1
Coordination Mechanisms in Complex Software Systems

Richard Mordinyi and Eva Kühn

Abstract. Software developers have to deal with software systems which are usually composed of distributed, heterogeneous and interacting application components representing higher-level business goals. Although the message-passing paradigm is a common concept allowing application components to interact with each other in an asynchronous manner, the technology is not entirely suitable for complex coordination requirements since the processing and state of coordination have to be handled explicitly by the application component. Data-driven frameworks support the coordination of application components, but have a limited number of coordination policies requiring from the software developer to implement coordination functionality that is not directly supported by the coordination framework. We illustrate the Space-Based Computing (SBC) paradigm aiming to support and facilitate software developers efficiently in their efforts to control complexity regarding concerns of interaction in software systems. Major results of the evaluation in this context are improved coordination efficiency accompanied with reduced complexity within application components.

1 Introduction

Complex systems are systems [1, 2] whose properties are not fully explained by an understanding of their single component parts. Complex systems (e.g., financial markets, bacteria life cycles) usually consist of a large number of mutually interacting, dynamically interwoven, and indeterminably dis- and reappearing component parts. The understanding often is that the complexity of a system emerges by interaction of a (large) number of component parts, but cannot be explained by looking at the parts alone. Software systems, especially software-intensive systems [3], can be interpreted as complex systems as well, because they usually interact with other software, systems, devices, sensors and people. Over time these systems become more distributed, heterogeneous, decentralized and interdependent, and are operating more often in dynamic and frequently unpredictable

Richard Mordinyi · Eva Kühn
Vienna University of Technology, Institute of Computer Languages,
Space-Based Computing Group, Argentinierstrasse 8, 1040 Vienna, Austria
e-mail: `{richard.mordinyi,eva.kuehn}@tuwien.ac.at`

environments. Therefore, software developers have to deal with issues like heterogeneity and varying size of components, variety of protocols for interaction with internal and external components, or with a number of potential incidents, like crashed or unreachable components in distributed environments. In the course of developing distributed software systems, software developers cannot avoid coping with these complexity issues. Today's software systems typically consist of mainly distributed application components representing higher-level business goals and a middleware technology abstracting the complexity concerns related to network and distribution. However, software developers still have to deal with the interaction of application components.

The message-passing paradigm is a common concept allowing application components to communicate with each other. The message-oriented middleware [4, 5], prominent representative is the Enterprise Service Bus (ESB) [6], provides synchronous and asynchronous message-passing properties and promises to interconnect application components in a loosely coupled manner. Since message-oriented middleware is only capable of transmitting and transforming messages between application components, it lacks support for complex interaction requirements which involve the participation of several application components for decision making, like in the telecommunication domain [7]. The software developers have no other choice but to take into account both, the application logic representing the business goal and additional coordination logic needed to fulfill the specific coordination requirement. Such logic for instance may contain implementation matters related to synchronization problems. Furthermore, it may include logic for the management and supervision of the latest state of the coordination process itself otherwise the application may get "lost" and the common business goal cannot be reached. Additional management is needed in case the coordinating component crashes and after recovery the failed application component still wants to be part of the running coordinating process [8]. These additional issues introduce potential sources of error, decrease the efficiency of the system, and increase the cognitive complexity [9, 10] of the application component. However, the responsibility of the software developer should focus on the application's business goals and not on concerns related to distribution or coordination.

A framework that has been explicitly designed for coordination purposes is the so called tuple space, based on the Linda coordination model of David Gelernter [11]. It is a data-centered, blackboard based, architectural style that describes the usage of a logically shared memory, the tuple space, by means of simple operations as interaction mechanisms. The approach promotes a clear separation between the computation model, used to express the computational requirements of an algorithm, and the coordination model, used to express the communication and synchronization requirements. The state of coordination is not embedded in the coordinating process itself but in the space [12]. The state of the coordination information on the blackboard determines the way of execution of the process. By means of this coordination model the application may entirely focus on business goals since the model "gives application builders the advantage of ignoring some of the harder aspects of multi-client synchronization, such as tracking names (and addresses) of all active clients, communication line status, and conversation status" [13]. The Linda coordination model uses template matching with random, non-deterministic tuple access to coordinate processes (see Sect. 0) supporting one coordination model only that can be considered a restriction limiting the benefit of using such a communication abstraction. Therefore, with respect to more complex coordination requirements the software developer still needs to implement coordination functionality not directly supported by the coordination framework within application components. Consequently, this increases the complexity of the application component, decreases performance due to additional implementation logic, and leads once again to an unclear separation between computation model and coordination model.

Taking into account the previously mentioned issues regarding interaction in software systems, we illustrate the so called Space-Based Computing (SBC) paradigm [14] supporting and facilitating software developers efficiently in their efforts to control complexity concerns in software systems. SBC extends and strengthens the clear separation between computation and coordination logic by allowing the selection and injection of scenario specific coordination models. From the application's point of view the SBC paradigm is comparable to the blackboard architectural style, orientated on the Linda coordination language. In contrast to traditional Linda coordination frameworks, we define the SBC paradigm to extend the Linda coordination model by introducing exchangeable mechanisms for structuring data in the space using special ordering characteristics and reducing dependencies between application components and coordination models. SBC explicitly embeds sophisticated coordination capabilities in the architectural style, and thus makes the style itself dynamic with respect to the scenario's coordination problem statement. This means that SBC is capable of abstracting coordination requirements and changes from the application. Since coordination requires and thus inherently consists of communication, consequently the abstraction of coordination also means that SBC abstracts communication requirements as well.

We evaluate the Space-Based Computing (SBC) paradigm using an industrial use case from an assembly workshop of a production automation system. The evaluation will demonstrate SBC's coordination and recovery capabilities focusing on aspects like feasibility, effort, robustness, performance, and complexity. Major results of the evaluation are higher coordination efficiency accompanied with minimized complexity within application components.

The remainder of this chapter is structured as follows: Section 2 summarizes related work on coordination models and platforms. Section 3 describes the industrial use case while Section 4 concentrates on the conceptual details of the Space-based Computing paradigm. Section 5 presents the evaluation whereas Section 6 discusses the advantages and limitations of SBC. Finally, Section 7 concludes the chapter and provides further research issues for future work.

2 Related Work

Since significant characteristics of complex systems refer to the interaction between components of complex systems, coordination between these components is an important issue to be investigated. This section summarizes related work with emphasis on coordination theory by giving a definition of coordination, describing coordination models, and presenting technologies built for supporting coordination.

2.1 Coordination Theory

Coordination [15] is the additional organizing activity (like information processing) that is needed in case multiple actors pursue the same goal, that a single actor would not perform. In a more general perspective [16], coordination refers to "*the act working together harmoniously*". However, it can be derived that coordination itself consists of different components, like actors performing some activities which are directed to a goal. Therefore, the definition implies that activities are not independent and thus coordination can be seen as "*the act of managing interdependencies between activities performed to achieve a goal*". Later, Malone and Crowston [17], the founders of interdisciplinary science of coordination theory, describe their definition in a refined form just as "*managing dependencies between activities*". It has to be pointed out that coordination makes only sense if tasks are interdependent. If there are no interdependencies, there is nothing to coordinate either. Given the unavoidable existence of dependencies, a detailed characterization of different sorts of dependencies is given in [17, 18].

2.2 Coordination Models

A coordination model [19] is either a formal or a conceptual framework to model the space of interaction. A formal framework expresses notations and rules for the formal characterization of coordinated systems, as used in the frameworks listed in [20] and [21]. A conceptual framework is required by software developers to manage inter-component interactions, since it provides abstraction mechanisms. In general, the emphasis is on the expressiveness of the abstraction mechanism of the coordination model, and on its effectiveness helping software developers in managing interactions.

From a functionality point of view distributed systems are typically divided into the following three concerns:

- Computational logic (i.e. business logic) performs calculations representing the main intention of the system (i.e. business specific goals)
- Communication responsible for sending and receiving data between components to be further processed.
- Coordination or dependency management responsible to execute tasks in a way where no dependencies are violated and the common coordination goal is achievable.

Sancese et. al. [22] argue that a clear separation of the three parts leads to a reduction of complexity of the entire systems also enabling a reliable and more stable implementation. The process of coordination follows a certain coordination model for which Ciancarini [19] defines a generic coordination model as a triple of

{E, M, L}. In the model, {E} stands for either physical or logical entities to be coordinated. These can be software processes, threads, services, agents, or even human beings interacting with computer-based systems. {M} represents the coordination media (i.e. communication channels) serving as a connector between the entities and enables communication, which is a mandatory prerequisite for coordination [18, 23]. Such coordination media may be message-passing systems, pipes, tuple spaces [11] etc. {L} specifies the coordination laws between the entities defining how the interdependencies have to be resolved and therefore, semantically define the coordination mechanisms. According to [12], existing variations of coordination models and languages can be mainly divided into two categories: control-driven (or task- or process-oriented) or data-driven coordination models, as described in the following sections.

2.2.1 Control-Driven Coordination

In control-driven coordination models [12] processes are treated as black boxes and any data manipulated within the process is of no concern to other system processes. Processes communicate with other processes by means of well defined interfaces, but it is entirely up to the process when communication takes place. In case processes communicate, they send out control messages or events with the aim of letting other interested processes know their interest, in which state they are, or informing them of any state changes.

From a stylistically perspective, in the control-driven coordination model it is easy to separate the processes into two components, namely purely computational ones and purely coordination ones. The reason is that "*the state of the computation at any moment in time is defined in terms of only the coordinated patterns that the processes involved in some computation adhere to*" [12] and that the actual values of the data being manipulated by the processes are almost never involved enabling a coordination component written in a high-level language. Usually, a coordinator process is employed for executing the coordination code. The computations are regarded as black boxes with clearly defined input and output interfaces which are plugged into the coordination code, i.e. they are executed when the program reaches a certain part of the coordination code. In which way (e.g., RPC [24], RMI [24], messaging [5, 6], publish/subscribe [25-28]) events are transmitted to the consumers is up to the middleware technology used in the given context. Examples for control-driven coordination languages include WS-BPEL [29], Manifold [30], CoLaS [31], or ORC [32].

2.2.2 Data-Driven Coordination

In contrast to control-driven coordination models, the main characteristic of the data-driven coordination model is the fact that "*the state of the computation at any moment in time is defined in terms of both the values of the data being received or sent and the actual configuration of the coordinated components*" [12]. This means that a coordinated process is responsible for both examining and manipulating data as well as for coordinating either itself and/or other processes by invoking the coordination mechanism each language provides. A data-driven coordination language typically offers some coordination primitives which are mixed within the computational code implying that processes cannot easily be distinguished as either coordination or computational processes.

Carriero and Gelernter define in [33] that "*a coordination model is the glue that binds separate activities into an ensemble*". They express the need for a clear separation between the specification of the communication entities of a system and the specification of their interactions or dependencies; i.e. a clear separation between the computation model, used to express the computational requirements of an algorithm, and the coordination model, used to express the communication and synchronization requirements. They explain that these two aspects of a system's construction may either be embodied in a single language or, as they prefer, in two separate, specialized languages. Such a coordination language is e.g., the Linda coordination model (see Sect. 0). In this data-driven coordination model, processes exchange information by adding and retrieving data from a so called shared dataspace.

2.3 Linda Coordination Frameworks

The Linda coordination model [11] was developed in the mid-1980's by David Gelernter at Yale University. It describes the usage of a logically shared memory, called tuple space, together with a handful of operations (*out, in, rd, eval*) as a communication mechanism for parallel and distributed processes. In principal, the tuple space is a bag containing tuples with non-deterministic *rd* and *in* operation access. A tuple is built-up of ordered fields containing a value and its type, where unassigned fields are not permitted, e.g. a tuple with the three fields <"index", 24, 75> contains "index" of type string and 24 resp. 75 of type integer.

The defined operations allow placing tuples into the space (*out*) and querying tuples from the space (*rd* and *in*). The difference between *rd* and *in* is that *rd* only returns a copy of the tuple, whereas *in* also removes it from the tuple space. Both operations return a single tuple and will block until a matching tuple is available in the tuple space. There are also non-blocking versions of the *rd* and *in* operation, called *rdp* and *inp*, which return an indication of failure instead of blocking, when no matching tuple is found [34]. The *eval* operation is like the *out* operation, but the tuple space initiates a single or several threads and performs calculations on the tuple to be written. The result of these calculations is a tuple that is written into the space after completed evaluation and that can then be queried by other processes.

The Linda model requires the specification of a tuple as an argument for both query operations and thus supports associative queries, similar to query by example [35]. In such a case, the tuple is called template that allows the usage of a wildcard as a field's value. A wildcard declares only the type of the sought field, but not its value, e.g. the operation *rd("index" ?x, ?y)* returns a tuple, matching the size, the type of the fields and the string "index". A tuple containing wildcards is called an anti-tuple. If a tuple is found, which matches the anti-tuple, the wildcards are replaced by the value of the corresponding fields. The non-deterministic *rd* and *in* operation semantics comes from the fact that in case of several matching tuples a random one is chosen.

Implementations that support the exact tuple matching of the Linda coordination model are: Blossom [36], JavaSpaces [37], LIME [38, 39], MARS [40, 41], and TuCSon [42-44]. Although both MARS and TuCSoN enable the modification of the operations' semantics by adding so called reactions, they cannot influence the way how tuples are queried. JavaSpaces adds subtype matching to the exact tuple matching mechanism to query objects from the space.

The drawback of exact tuple matching is that all collaborating processes must be aware of the tuple's signature they use for information exchange. Hence, there are several tuple space implementations that offer additional queries mechanisms, such as TSpaces [13, 45, 46], XMLSpaces.Net [47, 48] and eLinda [34, 49-51]. TSpaces offers the possibility to query tuples by named fields or by specifying only the field's index and a value or wildcard. Furthermore, TSpaces allows the definition of custom queries by introducing the concept of factories and handlers. Both TSpaces and XMLSpaces.Net support the use of XML-documents in tuple fields and therefore enable the use of several XML query languages such as XQL or XPath. In addition, XMLSpaces.Net uses an XML-document like structuring for its space, which allows the utilization of sophisticated XML queries on the space. eLinda enables the usage of more flexible queries, via its Programmable Matching Engine (PME), such as maximum or range queries. Beside these queries the PME also provides aggregated operations that allow the summary or aggregation of information from a number of tuples, returning the result as a single tuple. The PME allows, like TSpaces with its concept of custom factories and handlers, the simple definition of custom matchers [49].

The Linda coordination model exhibits the problem that access to local tuples is tied to the built-in associative mechanisms of tuple spaces. This implies that any non-directly supported coordination policy, like automatically reading several tuples, has to be charged upon the coordinating processes. This means that processes have to be made aware of the coordination laws increasing the complexity of the application design and so breaking the separation between coordination and business issues. The LuCe framework (stands for Logic Tuple Centres [52-54] and is further development of MARS and TuCSoN) introduces the concept of tuple centres as an extended tuple space, which can work as a *programmable coordination medium*. Beside normal tuples, information about the behavior of the centre is stored in the so called *specification tuples*. The main difference between a tuple space and a tuple centre is that the former supports only Linda coordination while the latter can be programmed to bridge between different representations of information shared by coordinated processes to provide new coordination mechanisms. Such mechanisms are realized by reactions allowing the extension of effects from the execution of communication operations as needed. Reactions map a logical operation onto one or more system operations. Furthermore, the results of an operation can be made visible to the coordinating processes as a single transition.

LuCe extends the Linda coordination model by a dynamic coordination behavior realized by means of reactions. This allows LuCe to satisfy complex coordination requirements, like handling of ordered tuples. Reactions are limited to Linda primitives only. Therefore, they are only capable of handling coordination requirements which do not need the integration of other components for interaction than tuple spaces themselves. Beside the fact that reactions cannot perform blocking operations, to the best knowledge they introduce accidental complexity into the coordination framework due to missing structuring and separation of concern mechanisms. For instance, aggregation and ordering logic has to be implemented in one reaction. Furthermore, in case tuples need to be sorted according to a specific requirement, they have to be extended with additional information representing the current position of the tuple. This implies that every operation performed on the space has to be adapted to the new structure of tuples decreasing the overall performance of the system.

3 Use Case Description

The SAW (Simulation of Assembly Workshop) research project [8] investigates coordination requirements and recovery capabilities of software agents representing functional machines in an assembly workshop. The overall goal is to increase the efficiency of the assembly workshop. This is achieved in two different ways, as described in the following.

The scenario from the production automation domain (Fig. 1) consists of several different software agents each being responsible for the machine it represents. Such an agent may be:

- a **pallet agent** (PA) representing the transportation of a production part and knowing the next machine to be reached by the real pallet,
- a **crossing agent** (CA) routing pallets towards the right direction according to a routing table,
- a **conveyor belt agent** (CBA) transporting pallets, with optionally speed control, from one crossing agent to another,
- a **machine agent** (MA) controlling robots of a docking station for e.g., painting or assembling product parts,
- a **strategy agent** (SA) which, based on the current usage rate of the production system, knows where to delegate pallets, so that by taking some business requirements, like order situation, into consideration, a product is created in an efficient way, or
- a **facility agent** (FA) which specifies the point in time when machines have to be turned off for inspection.

Fig. 1 shows a software simulator for a production system, in the concrete case for an assembly workshop. Such manufacturing systems are very complex and distributed. The usage of a digital simulator instead of a miniature hardware model has a lot of advantages like, low operating costs, the easy reconfiguration and parallel testing.

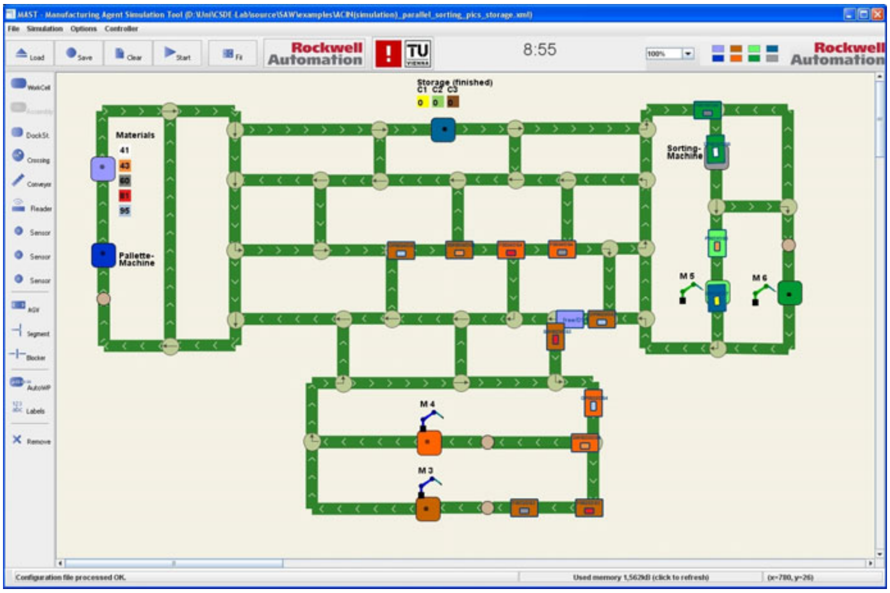


Fig. 1 View of a simulated Production Automation System[1] [55-57]

Multi-agent system (MAS) [58] is an accepted paradigm in safety-critical systems, like the production automation. A major challenge in production automation is the need to become more flexible. The requirement is to react quickly to changing business and market needs by efficiently switching to new production strategies and thus supporting the production of new market relevant products. However, the overall behavior of the many elements in a production automation system with distributed control can get hard to predict as these elements may interact in complex ways (e.g., timing of fault-tolerant transport system and machines) [59]. Therefore, an issue in this context refers to the implementation of agents with reduced complexity of their implementation by e.g., minimizing the communication effort to be managed by the agent.

An approach towards fast reactions may be the prioritization [60, 61] of pallets. Some special parts of the product with higher priority have to be favored by the agents rather than pallets with lower priority. This approach may help to a) produce a small number of products quickly, or b) to phase out products as soon as possible in order to free resources for brand new products to be assembled. Therefore, the aspect of priority has to be considered between all neighboring CAs and all CBAs connecting them. In the described scenario a CA has to check first, whether there is a pallet with high priority on one of the transporting conveyor belts. If this is the case that particular CBA may speed up its transportation speed as well as the CA may force the other conveyor belts to stop. This may happen by e.g. either not handling any pallets coming from them and so forcing those CBAs to stop, or by requesting the other CBAs to halt. So, the high priority pallet is

[1] Thanks to Rockwell Automation for the provision of the simulator.

routed earlier than the other pallets, and it overtakes other pallets which may occupy machines needed by the prioritized pallet based on its production tree.

4 Space-Based Computing

Similar to the Linda coordination model, SBC[2] is mainly a data-driven coordination model, but can be used in a control-driven way as well (see Sect. 0). As shown in Fig. 2, application components running on different physical nodes coordinate each other by means of writing, reading, and removing shared structured entries from a logically central space entity.

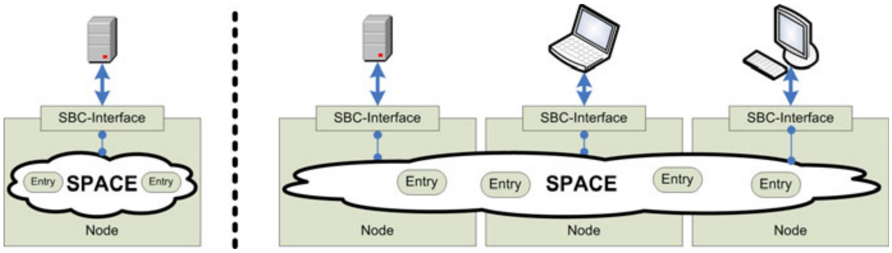


Fig. 2 High-level view of the Space-Based Computing Paradigm

An implementation of the SBC paradigm can be deployed on a physical central server, or on several multiple nodes. In the latter case, internal mechanisms have to make sure, that the shared data structures on the participating nodes are synchronized by taking into account use case specific requirements.

The following paragraphs summarize the XVSM[2] reference architecture based on the latest MozartSpaces[2] implementation in detail. Fig. 3 illustrates a general overview of the XVSM reference architecture divided into an application part (left side) and a space part (right side).

Container-Engine: As in Linda, in SBC application components coordinate each other by means of placing and retrieving data into/from a shared “space”. In XVSM data is stored in so called containers that can be interpreted as a bag containing data entries. In XVSM multiple containers may exist at the same time and the number of containers defines the XVSM space. The responsibility of the container-engine layer is the creation and destruction of containers. In its basic form a container is similar to a tuple space - a collection of entries. The main difference to a tuple space is that a container

- extends the original Linda API with a *destroy* method
- introduces so called coordinators enabling a structuring of the space
- may be bounded to a maximum number of entries

[2] SBC has been realized in the eXtensible Virtual Shared Memory (XVSM) reference architecture which has been implemented among others in Java (mozartspaces.org) and .Net (xcoordination.com)

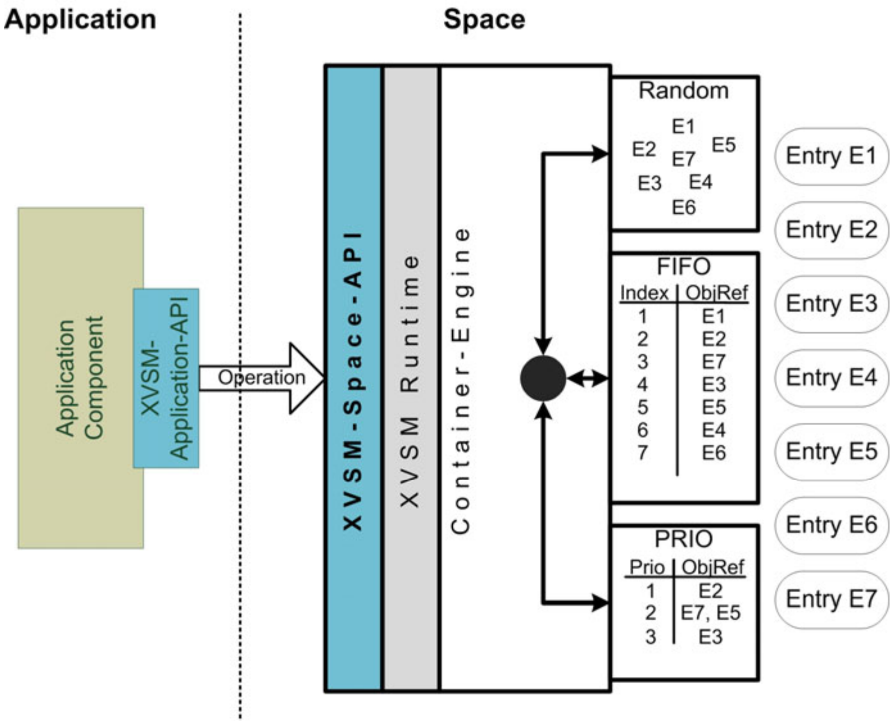


Fig. 3 XVSM architecture with a container hosting a random-, a FIFO-, and a PRIO coordinator structuring 7 entries [55]

Container-API: As in Linda (*out, in, rd*), a container's interface provides a simple API for *reading, taking*, and *writing* entries, but extends the original Linda API with a *destroy* operation. Similar to a take operation, a *destroy* operation removes an entry from the container. Although a *destroy* operation could be mapped onto a *take* operation where the result is omitted, it is still necessary to induct this kind of operation that does not return an operation value. The reason is that this way a lot of data traffic is avoided since the removed data does not need to be transferred back to the initiator of the operation.

The *destroy* operation is also helpful especially in the case of bulk operations [62]. Containers support bulk operations, so that it is possible to insert multiple entries into a container resp. to retrieve/remove multiple entries out of it within one operation.

While Linda makes an explicit distinction between blocking (*rd, in*) and non-blocking (*rdp, inp*) primitives, XVSM primitives are restricted to the mentioned four basic operations. Whether an operation blocks depends on the coordination policy a coordinator represents.

Coordinator: A container possesses one or multiple coordinators. Coordinators implement and are the programmable part of a container. They are responsible for managing certain views on the entries in the container. The aim of a coordinator is to represent a coordination policy. Each coordinator has its own internal data structures which help it perform its task. If the business coordination context and requirements are known beforehand, the coordinator can be implemented in an efficient way with respect to its policy. A coordination policy is represented in the implementation of each coordinator. This implies that the semantics of two coordinators may be the same, but they may be implemented in different ways; each of them taking into account different business specific requirements. A coordinator has an optimized view on the stored entries by taking into account scenario specific coordination requirements. Fig. 3 shows three exemplary coordinators (Random, FIFO, and PRIO Coordinator) referencing seven entries (E1-E7). The Random Coordinator contains all existing entries in the container and returns/removes an arbitrary entry in case of *read/take, destroy* operations. The FIFO Coordinator imitates a queue. It stores in the lowest index the entry that has been in the container for the longest time and in the highest index the entry that has been added last. The PRIO Coordinator groups references only to specific entries according to a priority defined by the software developer.

In general, whenever an operation is performed on a container, the parameters of the operation are collected in a so called selector. Every coordinator has its specific selector which can be interpreted as the coordinator's logical interface for the performed operation. Comparing the relation between selector and coordinator with OOP concepts, the selector is the interface and the coordinator the actual implementation of that interface. In case of *read*, *take*, and *destroy* operations the selector contains parameters (like a counter for the exact number of entries to be retrieved) for querying the view on the managed entries. In case of a *write* operation parameters influencing the coordinator in updating its view are required.

In case a container hosts several coordinators, operations may define multiple selectors as well. The number of specified selectors depends on the business coordination requirements and is not bound to the number of coordinators in the container. If more than one selector is used in querying operations, the outcome of the execution of the first selector will be used as input to the second and so on [63, 64]. The sequence of selectors in read operations is non-commutative *AND* concatenated (i.e. filter style). This means that it makes a crucial difference if 10 entries are selected from a FIFO Coordinator and then a template matching is performed or if the template matching is done first and then the FIFO Coordinator tries to return ten entries.

Before explaining how operations are executed two classes of coordinators have to be introduced. The software developer may declare a coordinator at the time of its creation to be either obligatory or optional. An obligatory coordinator must be called for every write operation on the container, so that a coordinator always has a complete view of all entries. An optional coordinator, however, only manages entries if it is explicitly addressed in the write operation, while other entries in the container remain invisible. The FIFO Coordinator can be used as an obligatory one since it does not need any additional parameters.

In the following the execution of the operations in the container-engine is explained in general. The given explanation does not consider the semantics of XVSM operation transactions or operation timeout. Those aspects are described in [65] in detail.

- **Write:** the *write* operation is executed on all optional coordinators for which parameters have been specified. Afterwards, the *write* operation is executed on all remaining obligatory coordinators even if the operation cannot provide parameters for those coordinators. When a *write* operation has to be blocked depends on the semantics of the coordinator. A semantic may be that an operation has to block if for instance, a Key Coordinator already has a key in its view that the *write* operation of a new entry uses too.
- **Read:** the container-engine iterates over the specified selectors of the operation and queries the corresponding coordinators. In case multiple selectors are specified the result set of the first queried coordinator is the set the next coordinator has to use to execute its query. A *read* operation has to be blocked in case the query cannot be satisfied.
- **Take:** the operation is executed the same way as a *read* operation whereas the result set of the last coordinator defines the set of entries which have to be removed from the container. Therefore, before returning the result set to the initiator of the operation the container-engine asks all coordinators which store a reference on the entries of that result set to remove the entry from their views. Similar to a *read* operation, a *take* operation has to be blocked in case the query cannot be satisfied.
- **Destroy:** the operation is executed like a *take* operation without returning the final result set to the initiator of the operation. Similar, a *destroy* operation has to be blocked in case the query cannot be satisfied.

The **XVSM Runtime** is a layer that is responsible for executing the basic operations by concurrent runtime threads. Operations executed in the container-engine are called requests in the XVSM Runtime layer. Beside the operation itself requests contain context specific meta-information (e.g., timeout, location of the receiver of the request result). Analogously to Linda primitives that block if a tuple does not match a specific template, the XVSM Runtime is also responsible for managing the blocking semantics of operations. The difference to the Linda coordination model is that software developers can alter the semantics of the initiated request. This is achieved by so called aspects (see below) that are treated by the Runtime as well.

Containers are Internet addressable using an URI of the addressing scheme "*xvsm://namespace/ContainerName*", like "*xvsm://host.mydomain.com:1234 /CName*". Every XVSM Runtime hosts several different transportation profiles responsible for accepting requests and sending responses over the physical network. Transportation profiles implement mechanisms for transporting data between nodes. The protocol type "*xvsm*" makes the usage of transportation profiles transparent to the application component. This means that as a transportation medium for accessing that particular container one of the transportation profiles is used without impact on the application component. The application component may specify the properties of transportation (e.g., reliability).

Aspects: The XVSM Runtime layer realizes aspect-oriented Programming (AOP) [66] by registering so called aspects [67] at different points, i.e. before the operation accesses the container-engine or when the operation returns from the container-engine. Aspects are executed on the node where the container is located and are triggered by operations either on a specific container (i.e. container aspect) or on operations related to the entire set of containers (i.e. space aspect). The join points of AOP are called interception points (IPoints). Interception points on container operations are referred to as local IPoints, whereas interception points on space operations are called global IPoints. IPoints are located before or after the execution of an operation, indicating two categories: pre and post. Local pre- and post-IPoints exist for read, take, destroy, write, local aspect appending, and local aspect removing. The following global pre- and post-IPoints exist: transaction creation, transaction commit, transaction rollback, container creation, container destruction, aspect add, and aspect delete. In case multiple aspects are installed on the same container, they are executed in the order they were added. Adding and removing aspects can be performed at any time during runtime.

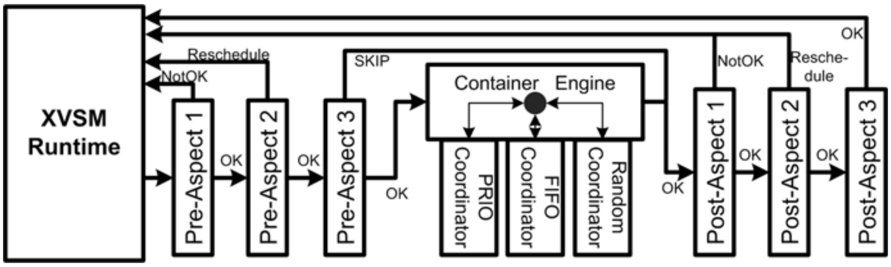


Fig. 4 Data- and control-flow in a container with three installed pre- and post-aspects [67]

Fig. 4 shows a container with three local pre and three post aspects and their various return values. The XVSM Runtime layer accepts incoming requests and passes them immediately to the first pre-aspect of the targeted container. The request passed to and analyzed by the aspect contains the parameters of the operation, like entries, transaction, selectors, operation timeout, and the aspect's context. The called aspect contains functionality that can either verify or log the current operation, or initiate external operations to other containers or third-party services. Aspects can be used to realize security (authorization and authentication) [68], the implementation of highly customizable notification mechanisms (see below), or the manipulation of already stored incoming or outgoing entries.

The central part of a container is the implementation of the container's business logic, i.e. the storage of the entries and the management of coordinators. A request is successful if it passed all pre-aspects, the container-engine, and all post-aspects without any errors. However, an aspect may return several values by which the execution of the request can be manipulated. The following return values are supported:

- OK: The execution of the current aspect has been finished and the execution of the next aspect or of the operation on the container proceeds.
- NotOK: The execution of the request is stopped and the transaction is rolled back. This can be used by e.g. a security aspect denying an operation if the user does not have adequate access rights.
- SKIP: This return value is only supported for pre-aspects and triggers the execution of the first post-aspect. This means that neither any other pre-aspects nor the operation on the container is executed.
- Reschedule: The execution of the request is stopped and will be rescheduled at a later time. This can be used to delay the execution of a request until an external event occurs.

Depending on the result of the last post-aspect the result of the request is either returned to the initiator of the request, or the request is rolled back.

The **XVSM-Application API** extends the XVSM-Space API with a notify method. It is a programming language specific implementation which communicates with the XVSM-Space API. The exchange of requests between the two APIs is performed in an asynchronous way. Fig. 5 shows the general structure of processing a notification in XVSM. In contrast to a specific notification mechanism in e.g., JavaSpaces, the introduced notification approach is flexible, thus can be adapted to business specific needs. In the example there is a container "X" and an application component 3 that wants to be notified whenever container X is accessed. When that application component invokes the notify method, XVSM Runtime registers an aspect (e.g., a so called notification aspect) on container X and creates a so called notification container. The notification aspect intercepts the processing of the operation on container X and writes data into the notification container. When the operation is intercepted (pre or post) information (e.g., a copy of the executed operation or use case specific information about the operation) is written into the notification container, depending on the scenario. A notification container is an "ordinary" container that is therefore capable of hosting additional pre and post-aspects for e.g., aggregation of entries.

Beside the notification aspect and the notification container, XVSM Runtime performs *take* operations on the notification container and specifies a virtual answer container where the result for that *take* operation has to be placed. The virtual answer container is addressed like an ordinary container but is bound to a call back method of the application component specified at the time of creating the notification. Therefore, whenever an entry is written into the virtual answer container the application component receives that entry. This way an application is notified about events on container X.

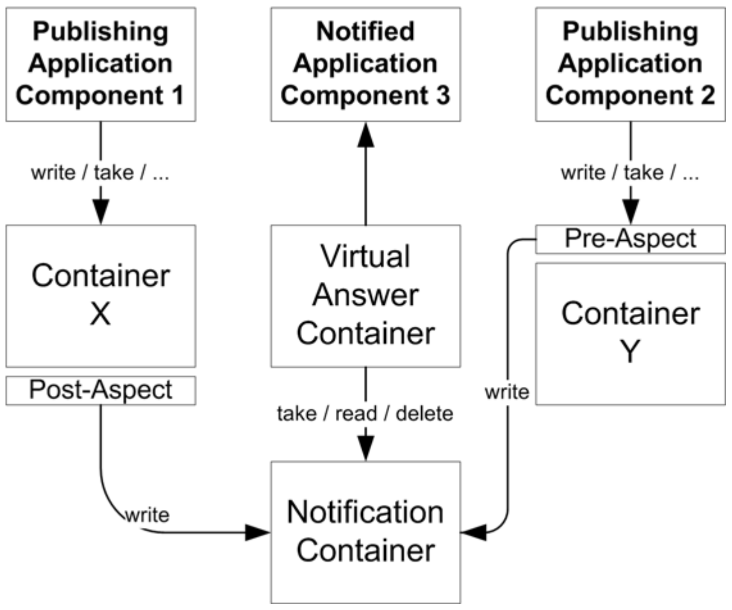


Fig. 5 General structure of an XVSM Notification [69]

As it can be seen, the introduced notification mechanism builds on already described XVSM architectural concepts. This allows software developers to create domain and application specific notification mechanisms which exactly meet given requirements. In Fig. 5 application component 2 wants to be notified in case entries were written. Since that component is not always online, notifications are temporarily and transparently stored in container Y [26].

The described mechanism shows several points where tuning of notification is possible. For instance, the notification aspect can be placed either before or after the execution of the operation on the container. If the aspect is installed as a pre-aspect, the application component cannot be sure whether the operation was really successfully executed on the container or had to be aborted due to errors. Furthermore, if the operation has to be blocked the application component is notified every time that operation comes to execution. The notification aspect can be registered for any XVSM operation (*read, write* …) and therefore a notification cannot only be created when an entry is written. It is also possible to create notifications which notify a user when entries are *read, taken* or *deleted*. Furthermore, Fig. 5 does not define where the shown containers are placed physically. It is possible that the containers are on the same node or on different ones. The latter one enables the creation of durable subscriptions [26] by placing the notification container on a node which is always reachable. The notification events are collected in the notification container whether or not the client is reachable. When the subscribing application component is online again, the XVSM Runtime fetches the notifications from the notification container which contains new entries written and optionally aggregated during its absence and pushes them via the specified call back method to the application component.

5 Evaluation

A major challenge in production automation is the need to be flexible in order to support a fast and efficient reaction to changing business and market needs. An approach towards fast reactions may be the prioritization of pallets. As mentioned before, some special parts of the product with higher priority have to be favored by the agents to pallets with lower priority. Simplified, the scenario can be summarized as the following: entries have to be ordered by means of the sequence of writing and grouped according to the priority of the entry written. Then, the task is to remove the entry first written from the non-empty group with the highest priority. Additionally, a conveyor belt has only a limited amount of space available depending on the length of the conveyor. In the following the proposed SBC based architecture is compared with architectures based on JMS messaging middleware or the Linda tuple space.

5.1 Java Message Service

For communication between agents in the production automation system, JMS [4] queues are appropriate. With respect to the described statement, Fig. 6 depicts how queues would realize the coordination problem with three different priority categories whereas 1 is the highest priority. In contrast, Fig. 7 shows the realization with an XVSM space container containing a PRIO-FIFO Coordinator. The PRIO-FIFO Coordinator stores messages in a FIFO order grouped according to their priority. Additionally, both figures show the sequence to write an entry and to take the next entry with the highest priority from the FIFO perspective.

In case of queues there are two possible implementations. In the first variant there is one queue for each priority. In the second variant a single queue hosts all messages (i.e. entries) whereas parameters in the message header define its priority for which so called selectors allow querying.

In the first solution, when an agent (Agent A1) wants to place an entry into a queue it looks up its priority. Based on the entry's priority the send operation (operations 1, 2, or 3) of the proper queue is executed. This implies that the application component has to manage three different queue connections. However, before placing the entry into the queue the agent has to retrieve its size. If the number of stored messages is greater than the maximum of permitted ones, then the sender has to look for alternative routing paths. On the receiver side, the agent (Agent A2) has two options of how to receive an entry (operations 4, 5, or 6). Either it polls queues starting with the queue with the highest priority, or it is notified by JMS in case an entry has been written into one of the queues. If it polls, then the agent accesses the queue with the highest priority (Q-Priority 1, operation 4) first. If it is empty then it accesses the queue with the second highest priority (operation 5), and so on. Once a queue has been found that is not empty it removes the entry from the queue and processes it. If the agent is notified then messages are pushed to the subscribed agents. However, in this case the concepts of a queue have to be changed from QueueSession and QueueReceiver to e.g., TopicSubscriber and MessageConsumer triggering an update of the agent's implementation logic. The difference between the two approaches is mainly concerned with the question of who controls an agent. If the agent is notified then it has to process the pushed entry immediately. If the agent polls a queue it can act more autonomously since it can specify when to access a queue and according to which strategy (e.g., configuration of polling rate).

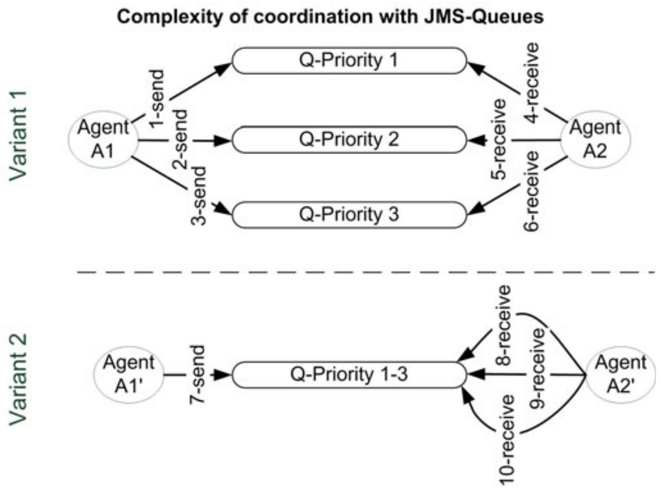
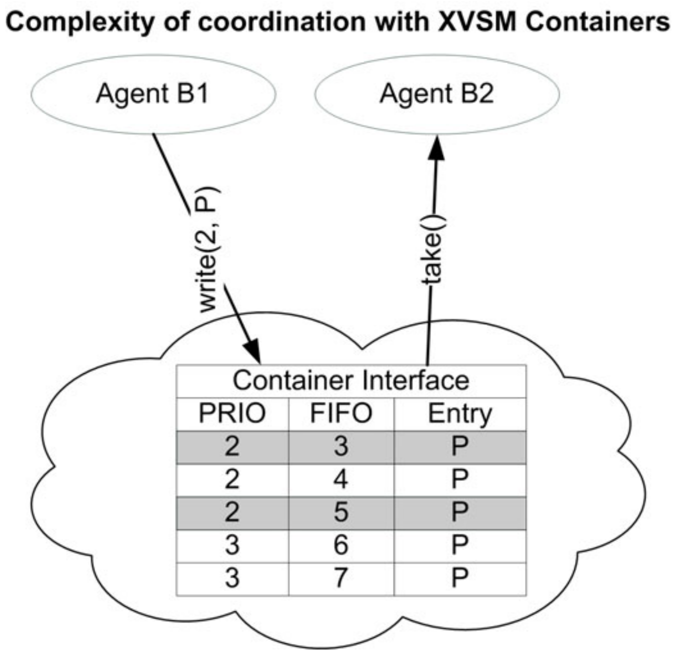


Fig. 6 Prioritized JMS queues

In the second implementation, agents (Agent A1' and A2') access a single queue. The difference to the first implementation is the usage of selectors specifying the priority of entries to be accessed. This means that instead of three different connections to a queue, three different selectors have to be used appropriately.

In the proposed SBC architecture (see Fig. 7), the usage of a "PRIO-FIFO" coordinator allows the software developer to specify the coordination policy transparent to the agents. A write operation needs a priority parameter and the entry. How entries are stored in the coordinator is up to the software developer and of no concern to the application component (coordination category). Since the coordination policy is represented in the coordinator the agent's take operation already reflects its semantics regarding priority restrictions. This means that the take operation does not need any parameters as the coordinator already knows that the entry with the highest possible priority has to be returned.

The migration from a take operation to a notification of written entries does not imply any change of concepts. The application component just executes a notify operation where it specifies the callback method. As described in the previous section, aspects make sure that consuming notifications are pushed to the application component. In contrast to the three queues, aspects can also help sort notifications according to the concurrently written entries' priorities before delivering them to the application component.



PRIO	FIFO	Entry
2	3	P
2	4	P
2	5	P
3	6	P
3	7	P

Fig. 7 Container with PRIO-FIFO coordinator (P..payload)

5.2 Linda Tuple Space

Fig. 8 depicts how the Linda tuple space approach would realize the coordination problem. Additionally, the diagrams show the sequence to write an entry and to take the next entry with the highest priority from the FIFO perspective.

For the implementation of a queue in Linda two additional tuples have to be placed into the tuple space. One tuple that represents the first index (i.e. beginning) of the queue (in-token) and one that represents the last index (i.e. end) of the queue (out-token). Therefore, each tuple in the space has to follow a specific structure. Either it is an index tuple containing information about its index type (in-token or out-token), the priority of the queue representing, and the actual value of the index, or it is a message type consisting of its type (i.e. message) and its index in the queue. Whenever a tuple is placed into the queue the last index tuple has to be taken out, the new tuple and an updated index tuple (i.e. index is increased by one) written into the space. Whenever the first tuple needs to be read, the first index tuple has to be found, its index read, and according to this information the tuple retrieved. Whenever the first tuple needs to be taken out, the first index tuple has to be found, its index read, the message based on this index taken out the space, and an updated index tuple (i.e. index is increased by one) written into the space. If no message can be retrieved then it implies that the current queue is empty. Therefore, the process has to be repeated until a message has been found with a lower priority.

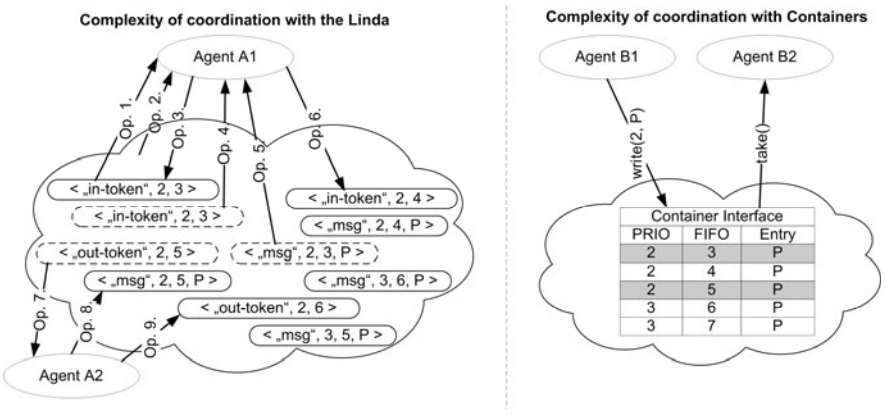


Fig. 8 Prioritized queue realized with the traditional Linda approach

Listing 1 shows how to retrieve an entry based on Fig. 8 as an example setting for stored entries in queues. It can be seen, that while the XVSM approach (see Fig. 7) needs a single API operation to write or to retrieve an entry from the space, the Linda tuple space approach requires three API operations: one to remove the index tuple, one to remove/write the message, and one to write back the index tuple. This is because the realization of a prioritized queue requires the agent taking over a part of the coordination problem.

Listing 1. Retrieving a FIFO sorted entry with Linda

Nr.	Operation

1.	`//retrieve index of first message with highest priority 1`
	`index = in("in-token", 1, ?int)`

2.	`//retrieve message from index with highest priority 1`
	`message = inp("msg", 1, index, ?P )`

3.	`// write back retrieved index tuple`
	`out("in-token", 1, index)`

4.	`//retrieve index of first message with new priority 2`
	`index = in("in-token", 2, ?int)`

5.	`//retrieve message from index with new priority 2`
	`message = inp("msg", 2, index, ?P )`

6.	`//write back new index tuple of new priority 2`
	`out("in-token", 2, index+1)`

Measured times required to retrieve the next entry, with highest priority, from a prioritized queue are shown in Table 1. A benchmark has been set up, which compares the performance of a JavaSpaces (as a Linda tuple space implementation), and a PRIO-FIFO coordinator. The benchmark demonstrates that a PRIO-FIFO coordinator is both able to retrieve entries faster than a coordinator with Linda pattern matching techniques and behaves retrieves entries in a constant access time, as expected from a FIFO queue.

Table 1 Time in ms to retrieve a single entry using different coordinators

Entries	Linda	PRIO-FIFO
10000	5,24	0,20
20000	15,15	0,20
30000	47,93	0,21
40000	58,66	0,20
50000	70,10	0,21

In order to run the benchmark the container was first filled with a specific amount of entries (10000, 20000, 30000, 40000 and 50000 entries). After that a take operation was issued, and the time needed to get the entry measured. The results of the benchmarks clearly show that the PRIO-FIFO coordinator is always the fastest. The results also show that the PRIO-FIFO coordinator offers constant access time, thus perfectly representing the coordination requirements within a single operation call. The benchmarks were run three times on a single node using an Intel Core2Duo T9500 with 4GB RAM to calculate the average access time.

6 Discussion

Besides XVSM the LuCe coordination framework, described in Sect. 0, offers the possibility to enrich the semantics of coordination operations. XVSM achieves this property by means of changeable coordinators. LuCe relies on the usage of so called reactions. Such reactions are hidden from the application and triggered transparently to the application whenever an entry is written or read. Reactions are also changeable and capable of simulating any kind of coordination policies.

However, LuCe just maps a single logical operation onto one or more system operations. This means that one operation in the application is mapped onto several Linda operations in the system. Therefore, the complexity of coordination policies has been just moved from the application (and consequently from the application developer) to the coordination middleware, thus to the software developer of that platform. In contrast, XVSM also moves the complexity of coordination from the application to the coordination middleware, but allows the usage of language specific primitives (i.e. the semantics of a FIFO coordination can be mapped on e.g., a java.list) which shift complexity further away from the software developer to the compiler of that language.

For example if LuCe had to support coordination models with ordering requirements, every tuple in the space had to be additionally wrapped into a tuple managed by reactions. This extra tuple stores meta-information, like the position in the queue, of each written tuple. Consequently, every incoming operation has to be adapted according to the new structure of the tuples, which decreases performance. In the MozartSpaces implementation of XVSM Java specific functions/libraries are used to organize entries in a queue resulting in a single and efficient operation.

Based on the fact that reactions in LuCe are implemented by means of Linda primitives, they cannot access other resources but the tuple space. Aspects in XVSM are written in higher-level languages and allow therefore the integration of other technologies, like web services or databases, into the coordination process. Furthermore, reactions cannot execute blocking operations. The limitation may arise due to the missing separation between reactions responsible for coordination and reactions responsible for e.g., tuple aggregation. Reactions must be non-blocking since strategies for synchronizations of reactions had to be implemented which would significantly decrease the performance of the platform.

Discussing similarities and differences of XVSM and control-driven coordination models like JMS then it can concluded that the FIFO coordination model represents the characteristics and behavior of messaging. However, in JMS the used interface for representing the FIFO coordination model is almost strongly coupled to underlying queuing technologies. This implies that in case of JMS the coordination of processes is not only limited to FIFO capabilities but also to the predefined middleware technology. On the other hand, XVMS's interface specifies only the way of coordination. Its interface is therefore capable of abstracting heterogeneous middleware technologies [70]. It allows injecting aspects e.g., used to coordinate services provision of a group rather than only of a single receiver. Additionally, aspects help manage different integration strategies depending on the used middleware technology. Adding the possibility to intercept communication methods in the XVSM platform minimizes the complexity of implementation. Compared to traditional integration solution XVSM abstracts any kind of middleware technologies. While in traditional solutions specific connectors between each used combination of different middleware technologies need to be implemented, the XVSM requires only the binding to the interface of the middleware adapter only. Although the approach of a common interface is not sophisticated, the benefit of it is a common interface with different transmission semantics. The semantic of the method, e.g. reliable or secure communication, depends on the capability of the middleware that is represented by that interface.

7 Conclusion

Today's software systems can be seen as complex systems in the sense that they usually interact with other software, systems, devices, sensors and people over distributed, heterogeneous, decentralized and interdependent environments while operated more often in dynamic and frequently unpredictable circumstances. Therefore, software developers have to deal with issues like heterogeneity and varying size of components, variety of protocols for interaction with internal and external components. Those software systems typically consist of mainly distributed application components representing higher-level business goals and a middleware technology usually representing an architectural style and abstracting the complexity concerns related to network and distribution.

The message-passing paradigm is a common concept allowing application components to interact with each other. But even asynchronous message-oriented middleware technologies are not suitable for complex coordination requirements since the processing and state of coordination have to be handled explicitly by the application component, thus increasing its complexity. Data-driven frameworks, like tuple spaces, support the coordination of application components, but have a limited number of coordination policies. Therefore, with respect to more complex coordination requirements application components still need to implement coordination functionality that is not directly supported by the coordination framework. Control-driven coordination models suit best in scenarios with point-to-point or 1:N communication requirements. Data-driven coordination models on the other hand are effective when several processes need to be synchronized to reach a common goal. The evaluation of the Simulation of Assembly Workshop (SAW) project shows that Space-Based Computing (SBC) is capable of representing both coordination models. The paradigm allows software developers to build applications being suitable for both coordination models and to switch between the models requiring small changes (regarding operation parameters) in the implementation of coordinating processes.

In the SBC paradigm coordination requirements are reflected in so called coordinators which explicitly distinguish between coordination data and payload. The evaluation of benchmark results shows that this distinction improves the efficiency of coordination significantly. This is due to the fact that a coordinator can be implemented efficiently by taking into account scenario specific context and coordination requirements.

With respect to complexity management the provided SBC concept of coordinators in containers moves the complexity of coordination requirement away from application components to a central point in the SBC coordination framework. The complexity of a coordination issue is concentrated at one point enabling a clear separation between business logic and coordination logic again. Process models comparing the number of processing steps needed to realize a coordination requirement show that by moving the complexity into the coordinator coordination requirements can be reduced to a single operation call on a container. Additionally, since coordination inherently consists of communication, aspects of communication can be abstracted as well by reducing the number of operations to a minimum.

Remaining future work refers to research topics such as the improvement of evaluation strategies for complexity measurement, investigation of scenarios with high-frequently changing conditions both of infrastructure and application requirements and capabilities, and wide-scale benchmarks of the proposed reference architecture with respect to scalability. Additionally, the proposed SBC paradigm will be further investigated in several research projects. In the research project

SecureSpace [68] the main issue is to develop a software platform for the secure communication and collaboration of autonomous participants across enterprise boundaries in the Internet and to prove its usability by means of industrial applications from the security domain. Moreover, in the research project AgiLog [71] the aspect of mobility is investigated in the context of SBC. Industrial scenarios from the logistics domain are used to evaluate the strengths and limitations of SBC with respect to development, configuration, and deployment of distributed applications running on mobile, embedded devices.

Acknowledgments. The work is funded by the Austrian Government under the program FIT-IT (Forschung, Innovation und Technologie für Informationstechnologien), project 825750 Secure Space - A Secure Space for Collaborative Security Services.

References

[1] Solomon, S., Shir, E.: Complexity; a science at 30. Europhysics News 34(2), 54–57 (2003)
[2] Cilliers, P.: Complexity and Postmodernism: Understanding Complex Systems. Routledge, London (1998)
[3] Broy, M.: The 'Grand Challenge' in Informatics: Engineering Software-Intensive Systems. Computer 39(10), 72–80 (2006)
[4] Monson-Haefel, R., Chappell, D.: Java Message Service, p. 220. O'Reilly & Associates, Inc., Sebastopol (2000)
[5] Hohpe, G., Woolf, B.: Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions. Addison-Wesley Longman Publishing Co., Inc., Boston (2003)
[6] Chappell, D.: Enterprise Service Bus. O'Reilly Media, Inc., Sebastopol (2004)
[7] Karhinen, A., Kuusela, J., Tallgren, T.: An architectural style decoupling coordination, computation and data. In: Proceedings of Third IEEE International Conference on Engineering of Complex Computer Systems (1997)
[8] Kühn, E., Mordinyi, R., Lang, M., Selimovic, A.: Towards Zero-Delay Recovery of Agents in Production Automation Systems. In: IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology (IAT 2009), vol. 2, pp. 307–310 (2009)
[9] Auprasert, B., Limpiyakorn, Y.: Structuring Cognitive Information for Software Complexity Measurement. In: Proceedings of the 2009 WRI World Congress on Computer Science and Information Engineering, vol. 07. IEEE Computer Society, Los Alamitos (2009)
[10] McDermid, J.A.: Complexity: Concept, Causes and Control. In: 6th IEEE international Conference on Complex Computer Systems. IEEE Computer Society, Los Alamitos (2000)
[11] Gelernter, D.: Generative communication in Linda. ACM Trans. Program. Lang. Syst. 7(1), 80–112 (1985)
[12] Papadopoulos, G.A., Arbab, F.: Coordination Models and Languages. In: Advances in Computers (1998)

[13] Lehman, T.J., Cozzi, A., Xiong, Y., Gottschalk, J., Vasudevan, V., Landis, S., Davis, P., Khavar, B., Bowman, P.: Hitting the distributed computing sweet spot with TSpaces. Comput. Netw. 35(4), 457–472 (2001)
[14] Mordinyi, R.: Managing Complex and Dynamic Software Systems with Space-Based Computing. Phd Thesis, Vienna University of Technology (2010)
[15] Malone, T.W.: What is coordination theory? MIT Sloan School of Management, Cambridge (1988)
[16] Malone, T.W., Crowston, K.: What is coordination theory and how can it help design cooperative work systems? In: CSCW 1990: Proceedings of the 1990 ACM Conference on Computer-Supported Cooperative Work. ACM, New York (1990)
[17] Malone, T.W., Crowston, K.: The interdisciplinary study of coordination. ACM Comput. Surv. 26(1), 87–119 (1994)
[18] Weigand, H., van der Poll, F., de Moor, A.: Coordination through Communication. In: Proc. of the 8th International Working Conference on the Language-Action Perspective on Communication Modelling (LAP 2003), pp. 1–2 (2003)
[19] Ciancarini, P.: Coordination models and languages as software integrators. ACM Comput. Surv. 28(2), 300–302 (1996)
[20] Ciancarini, P., Jensen, K., Yankelevich, D.: On the operational semantics of a coordination language. In: Object-Based Models and Languages for Concurrent Systems, pp. 77–106 (1995)
[21] Zavattaro, G.: Coordination Models and Languages: Semantics and Expressiveness. Phd Thesis, Department of Computer Science, University of Bologna (2000)
[22] Sancese, S., Ciancarini, P., Messina, A.: Message Passing vs. Tuple Space Coordination in an Aerodynamics Application. In: Malyshkin, V.E. (ed.) PaCT 1999. LNCS, vol. 1662. Springer, Heidelberg (1999)
[23] Franklin, S.: Coordination without Communication. Inst. For Intelligent Systems, Univ. of Memphis (2008)
[24] Tanenbaum, A.S., Steen, M.v.: Distributed Systems: Principles and Paradigms, 2nd edn. Prentice-Hall, Inc., Englewood Cliffs (2006)
[25] Triantafillou, P., Aekaterinidis, I.: Content-based publish-subscribe over structured P2P networks. In: International Conference on Distributed Event-Based Systems (2004)
[26] Eugster, P.T., Felber, P.A., Guerraoui, R., Kermarrec, A.M.: The many faces of publish/subscribe. ACM Comput. Surv. 35(2), 114–131 (2003)
[27] Cugola, G., Di Nitto, E., Fuggetta, A.: The JEDI Event-Based Infrastructure and Its Application to the Development of the OPSS WFMS. IEEE Trans. Softw. Eng. 27(9), 827–850 (2001)
[28] Huang, Y., Garcia-Molina, H.: Publish/subscribe in a mobile environment. Wirel. Netw. 10(6), 643–652 (2004)
[29] Web services business process execution language version 2.0. OASIS Committee Specification 2007 (2007), `http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html`
[30] Arbab, F., Herman, I., Spilling, P.: Manifold: Concepts and Implementation. In: Proceedings of the Second Joint International Conference on Vector and Parallel Processing: Parallel Processing. Springer, Heidelberg (1992)
[31] Cruz, J.C., Ducasse, S.: A Group Based Approach for Coordinating Active Objects. In: Ciancarini, P., Wolf, A.L. (eds.) COORDINATION 1999. LNCS, vol. 1594, pp. 355–370. Springer, Heidelberg (1999)

[32] Jayadev, M.: Computation Orchestration - A basis for wide-area computing. In: Engineering Theories of Software Intensive Systems, pp. 285–330 (2005)

[33] Gelernter, D., Carriero, N.: Coordination languages and their significance. Commun. ACM 35(2), 96 (1992)

[34] Wells, G.C.: Coordination Languages: Back to the Future with Linda. In: Proceedings of the Second International Workshop on Coordination and Adaption Techniques for Software Entities (WCAT 2005), pp. 87–98 (2005)

[35] Zloof, M.M.: Query-by-example: the invocation and definition of tables and forms. In: Proceedings of the 1st International Conference on Very Large Data Bases. ACM, Framingham (1975)

[36] van der Goot, R., Schaeffer, J., Wilson, G.V.: Safer Tuple Spaces. In: Garlan, D., Le Métayer, D. (eds.) COORDINATION 1997. LNCS, vol. 1282. Springer, Heidelberg (1997)

[37] Freeman, E., Arnold, K., Hupfer, S.: JavaSpaces Principles, Patterns, and Practice. Addison-Wesley Longman Ltd., Essex (1999)

[38] Murphy, A.L., Picco, G.P., Roman, G.C.: LIME: A coordination model and middleware supporting mobility of hosts and agents. ACM Trans. Softw. Eng. Methodol. 15(3), 279–328 (2006)

[39] Picco, G.P., Murphy, A.L., Roman, G.C.: LIME: Linda meets mobility. In: ICSE 1999: Proceedings of the 21st International Conference on Software Engineering. IEEE Computer Society Press, Los Alamitos (1999)

[40] Cabri, G., Leonardi, L., Zambonelli, F.: MARS: a programmable coordination architecture for mobile agents. IEEE Internet Computing 4(4), 26–35 (2000)

[41] Cabri, G., Leonardi, L., Zambonelli, F.: Mobile Agent Coordination for Distributed Network Management. Journal of Network and Systems Management 9(4), 435–456 (2001)

[42] Cremonini, M., Omicini, A., Zambonelli, F.: Coordination and Access Control in Open Distributed Agent Systems: The TuCSoN Approach (2000)

[43] Omicini, A., Ricci, A.: MAS Organization within a Coordination Infrastructure: Experiments in TuCSoN (2004)

[44] Omicini, A., Zambonelli, F.: Coordination for Internet Application Development. Autonomous Agents and Multi-Agent Systems 2(3), 251–269 (1999)

[45] Wyckoff, P., McLaughry, S.W., Lehman, T.J., Ford, D.A.: T spaces. IBM Systems Journal 37(3), 454–474 (1998)

[46] Lehman, T.J., McLaughry, S.W., Wycko, P.: T-Spaces: The Next Wave. In: Hawaii Intl. Conf. on System Sciences (HICSS-32) (1999)

[47] Tolksdorf, R., Glaubitz, D.: Coordinating Web-Based Systems with Documents in XMLSpaces. In: CooplS '01: Proceedings of the 9th International Conference on Cooperative Information Systems. Springer, London (2001)

[48] Tolksdorf, R., Liebsch, F., Nguyen, D.M.: XMLSpaces.NET: An Extensible Tuplespace as XML Middleware. Report B 03-08, Free University Berlin (2003), `ftp://ftp.inf.fu-berlin.de/pub/reports/tr-b-0308.pdf`; Open Research Questions in SOA 5-25 and Loose Coupling in Service Oriented Architectures (2004)

[49] Wells, G., Chalmers, A., Clayton, P.: Extending the matching facilities of linda. In: Arbab, F., Talcott, C. (eds.) COORDINATION 2002. LNCS, vol. 2315, p. 380. Springer, Heidelberg (2002)

[50] Wells, G.C.: A Programmable Matching Engine for Application Develoment in Linda. Phd Thesis, University of Bristol (2001)

[51] Wells, G.C.: New and improved: Linda in Java. Sci. Comput. Program. 59(1-2), 82–96 (2006)

[52] Denti, E., Natali, A., Omicini, A.: Programmable Coordination Media. In: Garlan, D., Le Métayer, D. (eds.) COORDINATION 1997. LNCS, vol. 1282. Springer, Heidelberg (1997)

[53] Denti, E., Omicini, A.: An architecture for tuple-based coordination of multi-agent systems. Softw. Pract. Exper. 29(12), 1103–1121 (1999)

[54] Denti, E., Omicini, A., Toschi, V.: Coordination Technology for the Development of Multi-Agent Systems on the Web. In: Proceedings of the 6th AI*IA Congress of the Italian Association for Artificial Intelligence (AI*IA 1999), pp. 29–38 (1999)

[55] Kühn, E., Mordinyi, R., Keszthelyi, L., Schreiber, C.: Introducing the concept of customizable structured spaces for agent coordination in the production automation domain. In: Proceedings of the 8th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2009), International Foundation for Autonomous Agents and Multiagent Systems, Richland (2009)

[56] Vrba, P.: MAST: manufacturing agent simulation tool (2003)

[57] Vrba, R., Marik, V., Merdan, M.: Physical Deployment of Agent-based Industrial Control Solutions: MAST Story. In: IEEE International Conference on Distributed Human-Machine Systems (2008)

[58] Wooldridge, M.: An Introduction to MultiAgent Systems. John Wiley & Sons, Inc., New York (2009)

[59] Lüder, A., Peschke, J., Sauter, T., Deter, S., Diep, D.: Distributed intelligence for plant automation based on multi-agent systems: the PABADIS approach. Production Planning and Control 15, 201–212 (2004)

[60] Kemppainen, K.: Priority scheduling revisited - dominant rules, open protocols and integrated order management. Phd Thesis, Acta Universitatis oeconomicae Helsingiensis. A. (2005)

[61] Rajendran, C., Holthaus, O.: A comparative study of dispatching rules in dynamic flowshops and jobshops. European Journal of Operational Research 116(1), 156–170 (1999)

[62] Hirmer, S., Kaiser, H., Merzky, A., Hutanu, A., Allen, G.: Generic support for bulk operations in grid applications. In: Proceedings of the 4th International Workshop on Middleware for Grid Computing. ACM, Melbourne (2006)

[63] Kühn, E., Mordinyi, R., Schreiber, C.: An Extensible Space-based Coordination Approach for Modeling Complex Patterns in Large Systems. In: 3rd International Symposium on Leveraging Applications of Formal Methods, Verification and Validation, Special Track on Formal Methods for Analysing and Verifying Very Large Systems (2008)

[64] Craß, S., Kühn, E., Salzert, G.: Algebraic foundation of a data model for an extensible space-based collaboration protocol. In: Proceedings of the 2009 International Database Engineering &#38; Applications Symposium (IDEAS 2009). ACM, New York (2009)

[65] Crass, S.: A Formal Model of the Extensible Virtual Shared Memory (XVSM) and its Implementation in Haskell. Institute of Computer Languages, Vienna University of Technology (2010)

[66] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., Irwin, J.: Aspect-oriented programming (1997)

[67] Kühn, E., Mordinyi, R., Keszthelyi, L., Schreiber, C., Bessler, S., Tomic, S.: Aspect-oriented Space Containers for Efficient Publish/Subscribe Scenarios in Intelligent Transportation Systems. In: Proceedings of the 11th International Symposium on Distributed Objects, Middleware, and Applications, DOA 2009 (2009)
[68] Secure Space - A Secure Space for Collaborative Security Services (2010), `http://tinyurl.com/34ymays` (cited)
[69] Kühn, E., Mordinyi, R., Schreiber, C.: Configurable Notifications for Event-based Systems, Vienna University of Technology (2008), TechRep. at `http://tinyurl.com/oht888`
[70] Mordinyi, R., Moser, T., Kuhn, E., Biffl, S., Mikula, A.: Foundations for a Model-Driven Integration of Business Services in a Safety-Critical Application Domain. In: Euromicro Conference on Software Engineering and Advanced Applications, pp. 267–274 (2009)
[71] Agile-Logistics. Komplexitätsreduzierende Middleware - Technologien für Agile Logistik (2010), `http://tinyurl.com/2u8qovc`

Glossary

AOP	Aspect-oriented Programming
API	Application Programming Interface
ESB	Enterprise Service Bus
JMS	Java Message Service
MAS	Multi-agent System
SAW	Simulation of an Assembly Workshop
SBC	Space-based Computing
XVSM	eXtensible Virtual Shared Memory