Software and Systems Modeling
https://doi.org/10.1007/s10270-020-00835-0

SPECIAL SECTION PAPER

Coordinating large distributed relational process structures

Sebastian Steinau[1] · **Kevin Andrews**[1] · **Manfred Reichert**[1]

Received: 30 November 2019 / Revised: 2 September 2020 / Accepted: 5 October 2020

Abstract
Representing a business process as a collaboration of interacting processes has become feasible with the emergence of data-centric business process management paradigms. Usually, these interacting processes have relations and, thereby, form a complex relational process structure. The interactions of processes within this relational process structure need to be coordinated to arrive at a meaningful overall business goal. However, relational process structures may become arbitrarily large. With the use of cloud technology, they may additionally be distributed over multiple nodes, allowing for scalability. Coordination processes have been proposed to coordinate relational process structures, where processes may have one-to-many and many-to-many relations at run-time. This paper shows how multiple coordination processes can be used in a decentralized fashion to more efficiently coordinate large, distributed process structures. The main challenge of using multiple coordination processes is to effectively realize the coordination responsibility of each coordination process. Key components of the solution are the subsidiary principle and the hierarchy of the relational process structure. Finally, an implementation of the coordination process concept based on microservices was developed, which allows for fast and concurrent enactment of multiple, decentralized coordination processes in large, distributed process structures.

Keywords Process interactions · Relational process structure · Coordination process · Distributed process execution · BPM in the cloud

1 Introduction

Several approaches enabling business process management (BPM) advocate to represent business processes as collections of interacting, interdependent processes. Examples include the artifact-centric and object-aware approaches to BPM [23,26,32], where the collaboration of artifact/object lifecycle processes forms an entire business process. Fundamental challenges of these data-centric approaches are to determine which processes exist and how they relate to other processes, as well as the coordination of the resulting structure of interdependent processes. Recently, the *relational process structure* [39] and *coordination processes* [38] have been proposed to tackle these challenges. A relational process structure captures processes and their relations in a hierarchical construct, which is then used by a coordination process to specify and enforce *coordination constraints*. This allows the interactions of different processes to be guided toward a meaningful overall business process.

1.1 Problem statement

Fundamental challenges remain, as a relational process structure may become arbitrarily large, i.e., it may comprise dozens or hundreds of different types of processes. At run-time, hundreds or thousands of instances of these process types are created, as well as their interrelations, compounding the problem [30].

Existing approaches to coordinate such large process structures propose employing a single *central coordinator*

Communicated by Jens Gulden and Rainer Schmidt.

This work is part of the ZAFH Intralogistik, funded by the European Regional Development Fund and the Ministry of Science, Research and the Arts of Baden-Württemberg, Germany (F.No. 32-7545.24-17/3/1).

✉ Sebastian Steinau
Sebastian.Steinau@uni-ulm.de

Kevin Andrews
Kevin.Andrews@uni-ulm.de

Manfred Reichert
Manfred.Reichert@uni-ulm.de

[1] Institute of Databases and Information Systems, Ulm University, Building O27 Level 5, James-Franck-Ring, 89081 Ulm, Germany

Published online: 09 November 2020

(e.g., a master artifact [43]). The term *coordinator* is hereby intended as an umbrella term for any kind of process coordination model, independent of the used paradigm, e.g., activity-centric, or data-centric. It is also independent of the exact specification, e.g., choreography, coordination process, or Proclet [45]. In many scenarios, as will be shown in this paper, a single, central coordinator is unsuitable for a vast process structure. The coordinator has to incorporate all coordination requirements for all processes in its model. As a result, a central coordinator model can become overloaded, inflexible, costly to maintain, and difficult to understand. As another drawback, all distributed processes must communicate with the central coordinator, creating a huge communication overhead and, more importantly, a single point of failure. For example, in the automotive industry, cars may be highly customized, requiring varying constraints on the production, assembly, and testing of the parts of each car, thereby creating vast structures of interrelated processes [29].

Using multiple coordinators for coordinating one relational process structure is denoted as *decentralized process coordination*. Additionally, as process structures become larger, several independent substructures may emerge, each of them requiring an individual coordination, which a central coordinator may not be able to provide. If this is the case, decentralized process coordination is not only more convenient and performant, but also a fundamental prerequisite for the correct execution of the interrelated business processes.

Several variants exist on how to realize decentralized process coordination. As these variants are built on top of each other, they are denoted as *stages* [40]. Stage-0 Decentralized Coordination corresponds to central coordination and Stage-2 Decentralized Coordination to fully decentralized coordination. Stage-1 Decentralized Coordination can be summarily characterized as "many central coordinators" and therefore is located in between Stage-0 and Stage-2.

Another aspect is that multiple interacting processes are particularly suited to be employed in a distributed instead of a monolithic system. In consequence, some processes of a relational process structure may be located on one node of the distributed system, whereas other processes may be located on different nodes. As process structures may become very large and different substructures may be distributed across the nodes of a server cluster, it is beneficial to distribute and split up the coordination of processes as well. This is denoted as *distributed process coordination*.

Figure 1 shows a schematic overview of distribution and decentralization of coordinators and processes. Neither decentralization nor distribution of coordinators has been considered so far in other approaches to data-centric BPM [42]. A more detailed assessment of existing approaches is presented in Sect. 7.

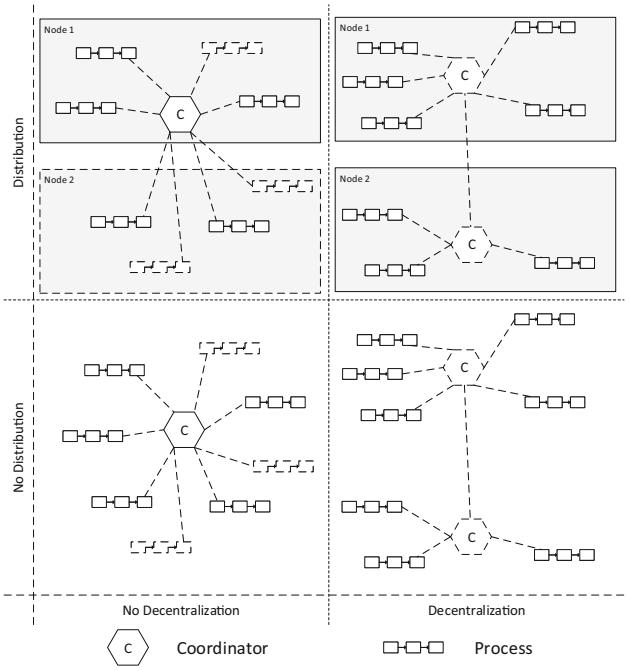

Fig. 1 Schematic decentralization and distribution of coordinators and processes

For both decentralized and distributed process coordination, the challenge of *coordination responsibility* needs to be solved, i.e., the question which coordinator is responsible for which processes. This involves deciding which stage of decentralization is necessary and the number of coordinators to be used. Moreover, a distribution of processes and coordinators needs to be taken into account.

1.2 Solution approach

The object-aware process management approach has introduced coordination processes to coordinate relational process structures [38]. While a coordination process can serve as a central coordinator, the concept itself is flexible enabling the use of *multiple coordination processes* to coordinate a relational process structure. Several coordination processes may be employed to coordinate different parts of the overall large relational process structure. For very large process structures, this avoids many of the disadvantages of centralized process coordination. Thus, the multiple coordination processes collaborate to achieve an overall coordination of the entire process structure.

Moreover, for object-aware process management, distributed process coordination is of particular importance, as the run-time engine of object-aware process management has a *hyperscale architecture* [2]. The term hyperscale denotes the ability of the process engine to effectively scale with additionally provisioned resources to provide more performance when computing demands are increasing. The run-time

engine is part of PHILharmonicFlows, the implementation of the object-aware approach. Having decentralized coordinators benefits distributed process coordination as well, as communication efforts between nodes may be reduced.

Coordination processes are particularly suited for a decentralized application by leveraging the hierarchical nature of the relational process structure. This allows implementing the *subsidiary principle*, where a coordination process only coordinates a subset of processes, defining its coordination responsibility, with the goal of avoiding overlap and redundancy between coordinators. The results are more flexible and smaller coordination models, a clear coordination responsibility of each coordination model, and superior maintainability. Furthermore, decentralization and distribution promises significant performance benefits for the coordination of interacting processes in context of an overall business process.

1.3 Contribution

This paper builds upon existing work of coordination processes [37–39] and contributes the decentralized and distributed application of coordination processes for object-aware business process management. The major contributions are as follows:

1. The paper presents the detailed stages of decentralized process coordination. The aim is to provide a conceptual framework for modeling decentralized processes and, subsequently, more performant process coordination.
2. The stages are the basis for a method for transforming existing central coordination into decentralized coordination. The method also enables designing decentralized coordination from scratch. Further, the method encompasses the use of coordination processes in distributed environments. The objective of this method is to define the coordination responsibility for all involved coordinators unambiguously.
3. A proof-of-concept prototype and a validation which shows that decentralized process coordination works in practice and achieves better performance compared to central process coordination. This is shown for distributed and non-distributed environments.

This paper extends a previous conference publication [40] in several ways. First, it is shown that the decentralization of coordination constraints over multiple coordination processes not only has conceptual benefits for modelers. The decentralization also enables significant performance increases in a hyperscale architecture [2]. A corresponding experiment with the goal of showcasing this performance advantage has been performed. Second, the performance benefits of distributing coordination processes across nodes of the hyperscale architecture are substantiated as well by appropriate experiments. Again, the goal of the experiment is to show that decentralized process coordination has performance advantages over central coordination. In summary, the paper provides a more elaborate validation of the benefits of the approach. Furthermore, an algorithm is sketched that may significantly improve the modeling of decentralized coordination processes. All results and concepts in this paper have been developed using the design science approach.

The remainder of the paper is organized as follows. Section 2 introduces the different stages of process decentralization and discusses distribution of processes across nodes. The challenges and benefits of decentralized and distributed process coordination are elaborated in Sect. 2 as well. In Sect. 3, background information on the relational process structure and the coordination processes is introduced. In Sect. 4, the key concepts of effectively using coordination processes in a large relational process structure are presented. In Sect. 5, decentralized process coordination is discussed and special emphasis is put on distributed coordination processes. Furthermore, an implementation of decentralized coordination processes is presented in Sect. 6, based on the hyperscale process engine of object-aware process management. Section 6 further presents performance measurements and benchmarks of centralized, decentralized and distributed process coordination. Section 7 discusses related work before Sect. 8 concludes the paper with a summary and an outlook.

2 Stages of process decentralization

The coordination of a multitude of different, interdependent processes is a complicated and challenging endeavor [30]. Processes and their relations have to be identified and, based on these connections, suitable *coordination constraints* need to be specified and enforced. A coordination constraint then denotes a dependency that exists between two or more processes [38]. A coordination constraint usually takes the form of a plain-text statement, e.g., “An application may only be created as long as the corresponding job offer is published,” though formal representations are possible as well. The different processes and their relations are summarized under the term *relational process structure*. Generally, approaches for coordinating process structures that consist of multiple process types advocate the use of a single entity with the purpose of coordinating all involved processes. This entity is called a *central coordinator*.

2.1 Stage-0 decentralized coordination

Central coordinators of any kind (e.g., a master artifact [43]) are capable of properly coordinating different processes. From the perspective of decentralization, a central coor-

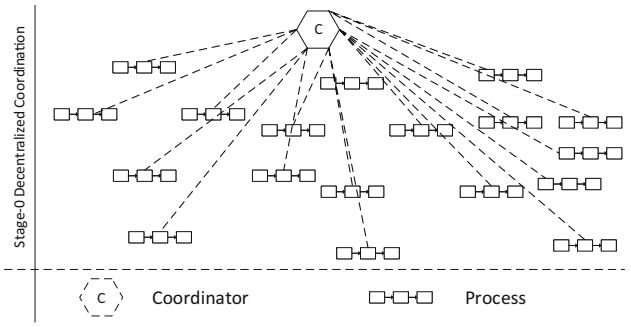

Fig. 2 Schematic view of Stage-0 decentralized coordination

dinator is denoted as *Stage-0 Decentralized Coordination*. Figure 2 shows a schematic view of Stage-0 Decentralized Coordination. The main disadvantage of central coordinators is poor scalability in regard to the process structure [35,36]. As the number of processes in a process structure grows, central coordinators must accommodate these additional processes in their coordination description. Moreover, additional coordination constraints must be incorporated into the coordination descriptions as well. Generally, this results in the central coordinator model becoming large and possibly overloaded. With increasing complexity, flexibility suffers, the central coordinator model becomes more difficult to adapt, and the understandability of the model is impaired as well. Furthermore, performance of the central coordinator may degrade due to the large number of processes and the resulting communication overhead. As a consequence, the central coordinator might become a bottleneck for the overall performance of the business process structure.

From a functional perspective, relying on one central coordinator for coordinating everything is neither the intuitive nor the most effective way of providing process coordination for large process structures.

In the following, the challenges and solution concepts are discussed alongside their illustrations in form of a running example. The solution concepts are by no means limited to the domain of the running example, but are generic and may be applied to other fields, such as logistics [9] and healthcare [11]. The running example represents a recruitment business process (cf. Example 1).

Example 1 *(Recruitment Business Process)* In the context of recruitment, applicants may apply for job offers. The overall process goal for a company is to determine who of the many applicants is best suited for the job. Applicants must write their application for a specific job offer and send it to the company. The company employees then evaluate each application by performing reviews. To reject an application or proceed with the application, a sufficient number of reviews need to be performed, e.g., the majority of reviews determines whether or not an application is rejected. If the majority of reviews are in favor of the application, the applicant is invited for one or more interviews, after which she may be hired or ultimately rejected. In the meantime, more applications may have been sent in, for which additional reviews are required, i.e., the evaluation of different applications may be handled concurrently, as well as the conduction of interviews.

Various interdependent process types can be identified in Example 1: *Job Offer*, *Application*, *Review*, and *Interview*. Each *Job Offer* is largely independent of other *Job Offers*, having its own set of applications and reviews. A single central coordinator is therefore tasked with coordinating all *Job Offers*, but each independently from others. The central coordinator must recognize and keep track of different executions states of processes and decision results made during the execution. It also must enforce the appropriate coordination constraints for the *Job Offers* and their connected processes, e.g., *Applications*. This constitutes an enormous complexity for the model of the central coordinator, especially concerning run-time. Moreover, the central coordinator acts as a single point of failure, as problems that might occur with any *Job Offer* may affect all other *Job Offers* as well.

2.2 Stage-1 Decentralized Coordination

As different *Job Offers* are conceptually independent from each other, a sensible solution would be to arrange that each *Job Offer* is coordinated individually together with its connected processes, e.g., *Applications* or *Reviews*. This means that there is one model of a coordinator that is instantiated multiple times at run-time, once for each *Job Offer*. This is denoted as *Stage-1 Decentralized Coordination.* Figure 3 shows a schematic view of Stage-1 Decentralized Coordination. This shift reduces model complexity, as the logic for distinguishing different *JobOffers* may be omitted due to the coordination happening on a *per-Job Offer*-basis, which in turn benefits understandability and maintainability of the coordinator models. The additional complexity of having to instantiate a model multiple times may generally be neglected, as instantiating a model multiple times is one of the core ideas of a process-oriented system. Another advantage is that this eliminates the single point of failure. If the coordination of one *Job Offer* fails for some reason, other *Job Offers* should remain unaffected. Stage-1 Decentralized Coordination is inherently supported by coordination processes (cf. [38]).

The distribution of coordinators has many advantages, while at the same time only small costs incur [2,7,36]. Adding more decentralized coordinators may still yield more benefits [35].

Example 2 *(Unsolicited Application)* Consider the recruitment scenario of an "unsolicited application," i.e. an applicant sends in an *Application* without a prior *Job Offer* from

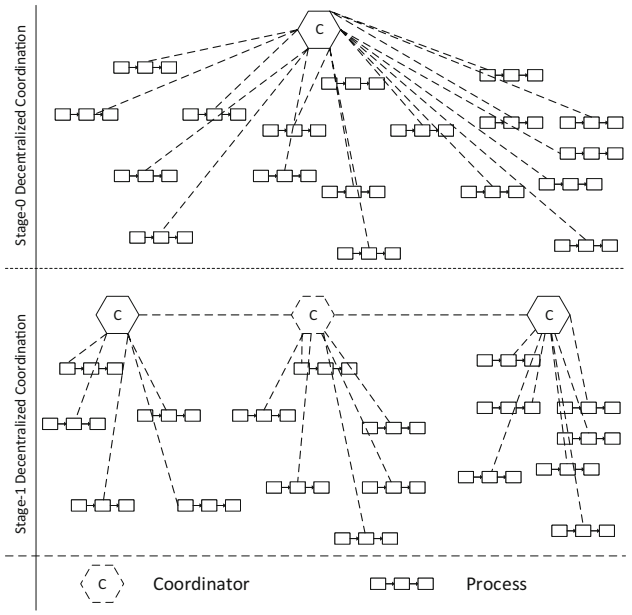

Fig. 3 Schematic view of Stage-1 Decentralized Coordination

the company. In case the unsolicited *Application* is accepted, a specific *Job Offer* will be created for the application.

As the coordinator that coordinates *Applications* with *Reviews* and *Interviews* is tied to a *Job Offer*, the unsolicited *Application* cannot be processed correctly without a link to a *Job Offer* in Stage-1 Decentralized Coordination. As the coordination constraints are modeled in the *Job Offer* coordinator, the unsolicited *Application* is not restricted by any coordination constraints. As a consequence, undesired outcomes might occur, such that an *Application* is accepted without any *Reviews* or that *Reviews* propose rejection but an *Interview* is created anyway.

Thus, it is reasonable to add another coordinator and transfer responsibilities to it from the *Job Offer* coordinator: The new coordinator coordinates *Applicati-ons* with *Interviews* and *Reviews*, and is tied to the respective *Application*. The existing *Job Offer* coordinator is subsequently only responsible for coordinating the *Job Offer* with its related *Applications*. As a result, an unsolicited *Application* may be handled correctly in addition to the usual recruitment procedure. This further reduces the complexity of the individual coordinator models.

2.3 Stage-2 Decentralized Coordination

Employing multiple coordinator models is denoted as *Stage-2 Decentralized Coordination*. Each coordinator is responsible for a different part of the process structure, i.e., different coordination responsibility. Stage-2 Decentralized Coordination encompasses Stage-1 naturally. Figure 4 shows a schematic view of Stage-2 Decentralized Coordination.

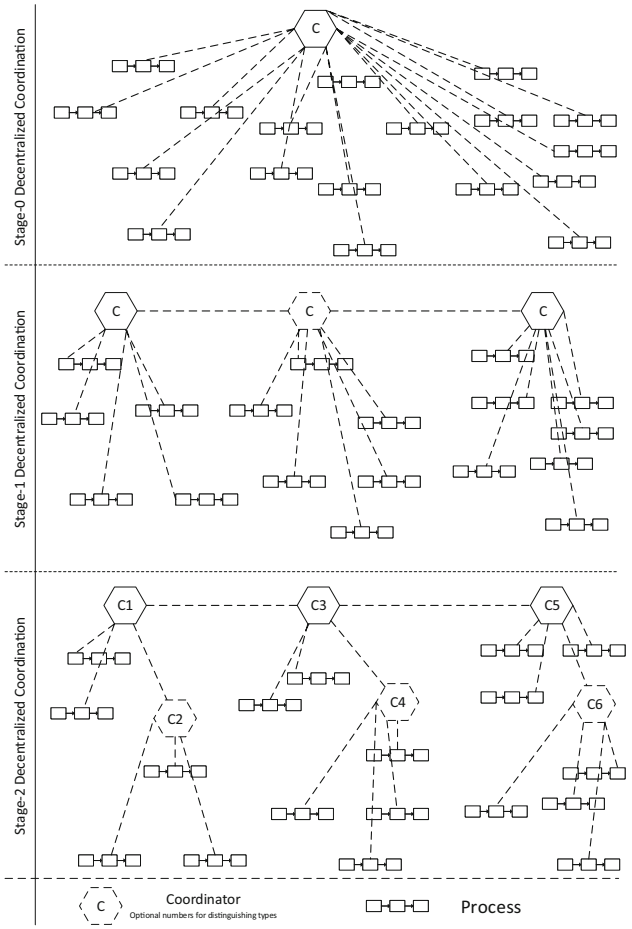

Fig. 4 Schematic view of stage-2 Decentralized Coordination

Of particular importance here is that Stage-2 Decentralized Coordination is not only advantageous with regard to smaller coordinator models, higher understandability, and performance. For some cases, such as the unsolicited application, Stage-2 Decentralized Coordination is downright necessary if the overall business process shall be executed correctly and the aforementioned advantages shall be retained. While a central coordinator is certainly capable of supporting an unsolicited application, the aforementioned advantages of decentralized coordination cannot be realized.

Stage-2 Decentralized Coordination is also advantageous in a distributed environment. Processes may run on different *nodes* in a distributed *cluster*, e.g., servers of different departments of the same company. The nodes and their communication paths are referred to as the *layout* of the cluster. As basic premise, communication within a node is performant and cheap, whereas communication between nodes is more costly. While the primary goal is the proper coordination of all involved processes, a secondary goal is to minimize communication between nodes due to its associated cost. A single central coordinator, running on one node, is forced to communicate with processes on other nodes.

By distributing coordinators among nodes, e.g., one coordinator for each node, communication between nodes can be minimized, resulting in more efficient and performant communication.

To realize the benefits from the use of decentralized coordinators in process structures, several issues need to be addressed. First, it must be determined how many coordinators are necessary for a given process structure, taking the layout of a potential cluster into account. Second, the processes that require coordination need to be assigned to a suitable coordinator, i.e., the *responsibility* of the coordinator needs to be defined. The responsibility includes that redundancies in the coordination constraints must be avoided. Processes should be assigned, if possible, only to one coordinator, i.e., the overlap between coordinators should be minimal. Otherwise, superfluous work would be performed, or communication costs cannot be reduced compared to the use of a single coordinator. Dividing the responsibility among several coordinators suitably and effectively is the primary challenge of decentralized coordinators. Table 1 gives a brief summary of the stages of coordination decentralization.

In summary, the decentralization of process coordination involves:

- Deciding the stage of decentralization
- Deciding the number of coordinators
- Determining the coordination responsibility of each coordinator, while
  - avoiding redundancy and overlap
  - taking the layout of the distributed cluster into account, if necessary

Coordination processes have been designed with a decentralized application to large process structures in mind and can therefore provide a solution to enable the discussed benefits. This paper contributes new applications of coordination processes for Stage-2 Decentralized Coordination of large process structures.

3 Background

The following section gives an overview over the context and basic conditions in which decentralization and distribution can be used and established with the method presented in this paper.

Object-aware process management is an comprehensive approach for managing data-centric processes [26]. The core of object-aware process management is presented as a meta-model in Fig. 5. Object-aware process management describes *business processes* in terms of *interacting processes*, e.g., object lifecycles, with the goal of providing better support for data and better flexibility. The business process only emerges through interactions between processes, and this requires *coordination* for guiding the business process toward a meaningful goal. Note that the meta-model only contains the concepts relevant for this paper.

The coordination approach of object-aware process management consists of three concepts: *relational process structures* capture and track process types and their relations. *Semantic relationships* use these relations for describing constraints for coordinating the interactions between processes. *Coordination processes* are used for concretely specifying semantic relationships and enforcing these constraints at runtime, relying on the information provided by the relational process structure. Moreover, for obtaining only the relevant information coordinating the processes, these processes are abstracted using a state-based view.

Coordination processes and their related concepts rely on a strict distinction between design-time and run-time entities. A design-time entity is designated as a *type* (formally superscript^T), whereas run-time entities are *instances* (formally ^{I}). For the sake of brevity, when referring to entities, e.g., processes, without a type or instance superscript or word member, this means that a statement applies to types as well as instances. By convention, instances are created by instantiating a type. The dot (.) represents the member access operator. The symbol $<:$ signifies the subtype relation, i.e., x is a subtype of y is written as $x <: y$. Also by convention, any set is denoted by a capital letter, whereas an element of the set is denoted with the same lowercase letter and vice versa. The concepts that constitute and support a coordination process are inextricably linked to each other, which necessitates mutual references and forward references in the formal definitions for completeness. The formal definitions mirror the implementation of the concepts and do not contain cyclic dependencies, but simply mutual references for navigating the resulting graph. Consequently, formal definitions may mention concepts and entities that will only be defined later in this section. Still, the introduction of concepts and entities follows a logical top-down manner despite the forward references. The intention is to keep this background section as concise as possible while still conveying the essential information. The (mutual) references are implicitly resolved using a globally unique identifier (GUID) for each entity. Furthermore, as this article is part of a larger body of work in context of the PHILharmonicFlows project, the formal definitions are kept consistent in every article.

For the purposes of this paper, a process (cf. Definition 1) is represented in an abstract, simplified manner, which is called a *state-based view* [37]. In a state-based view, each process model is partitioned into different states that are relevant for process coordination.

Table 1 Stages of process coordination decentralization

Stage	Description
Stage-0	One instance of one central coordinator model
Stage-1	Multiple instances of one coordinator model
Stage-2	Multiple instances of more than one coordinator model

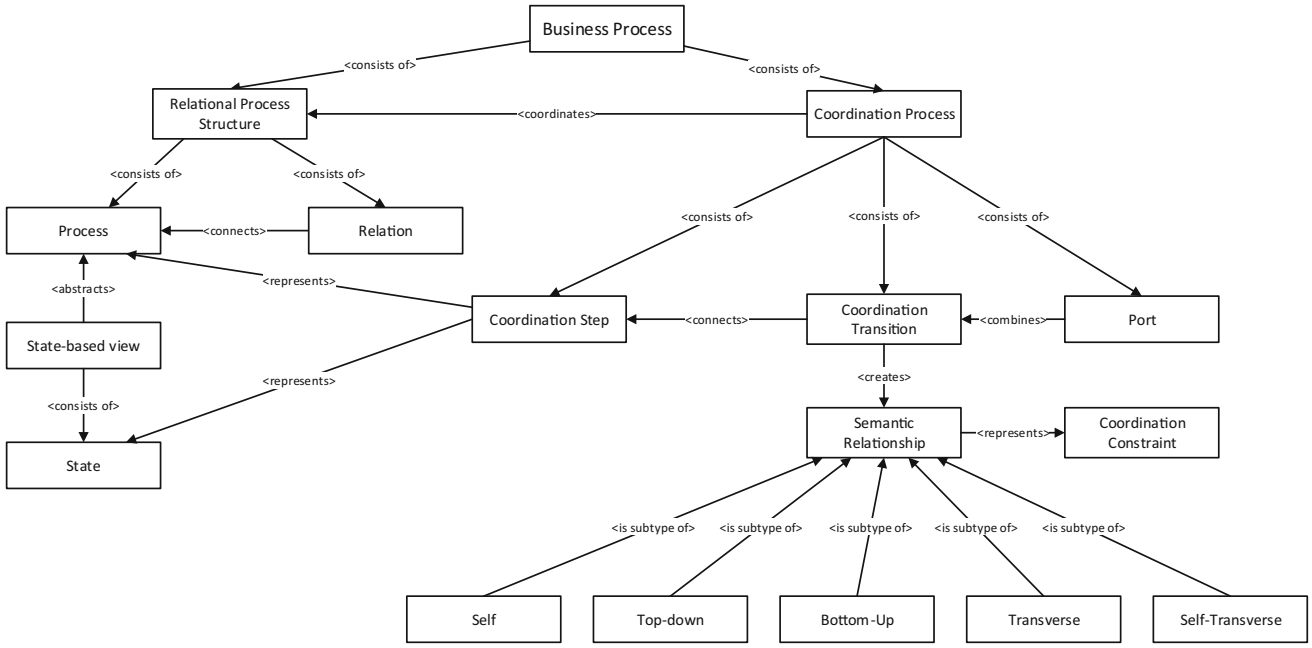

Fig. 5 Essential object-aware process management meta-model

Definition 1 **(Process Type)** A process type ω^T has the form $(d^T, n, \theta^T_{priv}, \theta^T)$ where

- d^T refers to a relational process structure to which this process type belongs (cf. Definition 6)
- n is a unique identifier (name) of the process type
- θ^T_{priv} is a process model specification not publicly visible
- θ^T is a state-based view mapped to θ^T_{priv}(cf. Definition 3)

Coordination processes originate in the object-aware business process management approach. While objects and their lifecycles have provided the initial motivation for coordination processes, the object and lifecycle model itself is not a prerequisite for coordination processes to work. Therefore, a generalized notion of process θ_{priv} is used that may represent, in principle, any kind of process model specification. For the purposes of coordination processes, the paradigm and modeling language in which processes are specified is unimportant. Consequently, a process θ_{priv} may be an object-aware process or a process that is specified using BPMN 2.0 [34]. Due to the arbitrary nature, no formal definition of θ_{priv} is possible. In every case, a state-based view θ provides an abstraction level over the actual process specification θ_{priv} [37] that a coordination process uses. Thereby, the process to be coordinated is partitioned into different states that *provide significant meaning for process coordination*. State-based views enable a coordination process to be *paradigm-agnostic*, i.e., processes from any paradigm or even different paradigms may be coordinated. This applies to both type and instance levels.

Definition 2 **(Process Instance)** A process instance ω^I has the form $(\omega^T, d^I, l, \theta^I_{priv}, \theta^I)$ where

- ω^T refers to the process type from which ω^I has been instantiated (cf. Definition 1)
- d^I refers to the relational process instance structure to which this object instance belongs (cf. Definition 7)
- l is the unique identifier (name) of the process instance. Default is $\omega^T.n$
- θ^I_{priv} is a process instance specification not publicly visible
- θ^I is a state-based view mapped to θ^I_{priv}(cf. Definition 3)

State-based views partition a process specification into distinct and non-overlapping states (cf. Definition 3). A state-based view θ is an abstraction over θ_{priv}, i.e., the actual process specification, mapping elements of θ_{priv} to states of the state-based view so that each element (e.g., an activity) belongs to exactly one state (cf. Fig. 6) [37]. States are used to indicate the progress of the underlying process θ_{priv}.

Definition 3 (State-based View) A state-based view θ has the form $(\omega, \Sigma, T, \Psi)$ where

- ω refers to the process to which this state-based view belongs (cf. Definitions 1 and 2)
- Σ is a set of states σ
- T is a set of transitions τ
- Ψ is a set of backward transition types ψ.

States σ are connected with directed edges τ denoting state transitions. At run-time, an *active state* σ_a of a process signifies its current execution status; the active state is determined by θ_{priv}, e.g., the currently executed activity is mapped to σ_a. Only one state σ may be active at a given point in time. As a consequence, branching state transitions categorically implement an exclusive choice semantics, i.e., states may be mutually exclusive regarding activation. Note that this does not prohibit parallel execution of activities, as parallelism may still occur within a state. As only one state may be active, in case of mutually exclusive states, non-active states are denoted as *skipped*. Furthermore, state-based views may include *backward transitions* ψ that allow re-activating a previous state σ, i.e., σ is a predecessor of the current active state σ_a. Figure 6 shows state-based views of the processes occurring in Example 1.

States and their transitions are, by default, the only entities that are publicly visible to an outside observer of a process. The state transitions τ^I and the active state σ_a^I are driven by θ_{priv}^I. Despite the simplistic specification, state-based views capture the essentials of a process in regard to process coordination. In addition, if desired, state-based views may introduce additional process properties, e.g., specific data attributes that may subsequently be used for process coordination.

Generally, processes may be interconnected by *relations*. A relation represents a connection between two processes, indicating one or more dependencies between them, i.e., multiple coordination constraints can be defined over the same relation. A relation type (cf. Definition 4) and relation instance (cf. Definition 5) are defined as follows:

Definition 4 (Relation Type) A relation type π^T represents a many-to-many relation between two processes and has the form $(\omega_{source}^T, \omega_{target}^T, m_{upper}, m_{lower}, n_{upper}, n_{lower})$ where

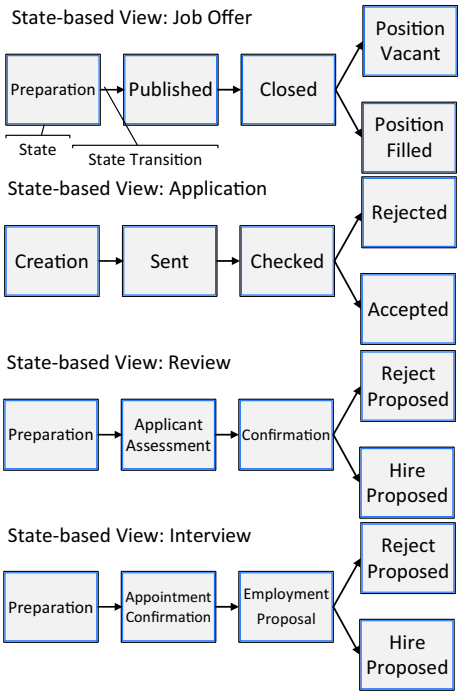

Fig. 6 State-based views of the processes in the recruitment example

- ω_{source}^T refers to the source process type (cf. Definition 1)
- ω_{target}^T refers to the target process type (cf. Definition 1)
- m_{upper} is an upper bound on the number of process instances ω_{target}^I with which ω_{source}^I may be related. Default: $m_{upper} = \infty$
- m_{lower} is a lower bound on the number of process instances ω_{target}^I with which ω_{source}^I may be related. Default: $m_{lower} = 0$
- n_{upper} is an upper bound on the number of process instances ω_{source}^I with which ω_{target}^I may be related. Default: $n_{upper} = \infty$
- n_{lower} is a lower bound on the number of process instances ω_{source}^I with which ω_{target}^I may be related. Default: $n_{lower} = 0$

Definition 5 (Relation Instance) A relation instance π^I has the form $(\pi^T, \omega_{source}^I, \omega_{target}^I)$ where

- π^T refers to the relation type from which π^I has been instantiated (cf. Definition 4)
- ω_{source}^I refers to the source process instance (cf. Definition 2)
- ω_{target}^I refers to the target process instance (cf. Definition 2)

Note that relation instances always have exactly one source and one target process instance, as one-to-many or

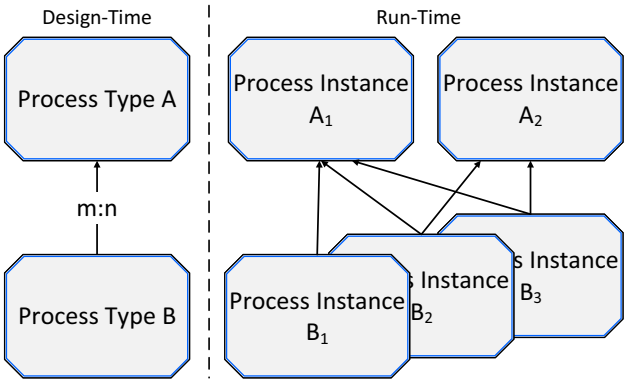

Fig. 7 Processes and relations at design- and run-time

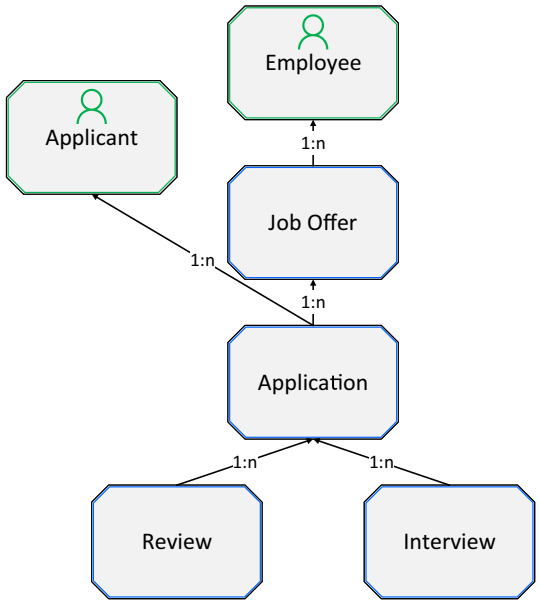

Fig. 8 Relational process type structure for the recruitment example

many-to-many relationships are comprised of multiple relation instances π^I (cf. Fig. 7). In particular, two processes may be related by a *transitive relation*, i.e., a path of relations exists connecting one process with another. Contrary to, for example, Entity-Relationship-Diagrams, relations are directed. This has various purposes, among them the definition of *semantic relationships* (cf. Sect. 3.2). For any process type or instance ω, two sets are maintained in regard to relations: Π_{in} is the set of incoming relation instances for a process instance ω^I , i.e., $\Pi_{in} = \{\pi \mid \pi.\omega_{target} = \omega^I\}$, and Π_{out}, which is defined analogously for outgoing relation instances. These sets allow realizing some efficiency optimizations in coordination process execution and are therefore mentioned for accuracy [39].

3.1 Relational process structures

Relational process structures provide a basis for the use of coordination processes. At design-time, a relational process type structure captures all processes and their relations (cf. Definition 6) [39]. Formally, a relational process type and instance structure (cf. Definition 7) are defined as follows:

Definition 6 (Rel. Process Type Structure) A relational process type structure d^T has the form (n, Ω^T, Π^T) where

- n is the name of the relational process type structure
- Ω^T is the set of process types ω^T (cf. Definition 1)
- Π^T is the set of relation types π^T (cf. Definition 4)

Definition 7 (Rel. Process Instance Structure) A relational process instance structure d^I has the form (d^T, Ω^I, Π^I) where

- d^T refers to the relational process type structure from which d^I has been instantiated
- Ω^I is the set of process instances ω^I(cf. Definition 2)
- Π^I is the set of relation instances π^I (cf. Definition 5)

Relation types π (and by extension, relation instances) that belong to relational process structure d only exist between processes in $d.\Omega$. Creating a new relation between two processes is referred to as *linking process instance*s. The new process instance and the new relation are then added to the respective sets of the relational process structure the other process instance belongs to.

At run-time, the purpose of the relational process instance structure is to track and capture every instantiation and deletion of processes and relations, enabling full process relation awareness [39]. Process instances may be added from time to time to an existing relational process instance structure, each creating a new relation between the process to add and a process instance that is already part of the relational process structure.

A coordination process can query the relational process instance structure to obtain up-to-date information about processes and their relations.

Example 3 *(Relational Type Structure)* Figure 8 shows the corresponding relational process type structure for the running example (cf. Example 1), showing various process types and their relations.

The process types *Applicant* and *Employee* are user process subtypes concerned with representing users, relevant for authorizations and permissions in object-aware process management. The formal notation $\omega_i \twoheadrightarrow \omega_j$ is used to signify a (transitive) directed relation from ω_i to ω_j. The directed relation between processes induce a hierarchy in a relational

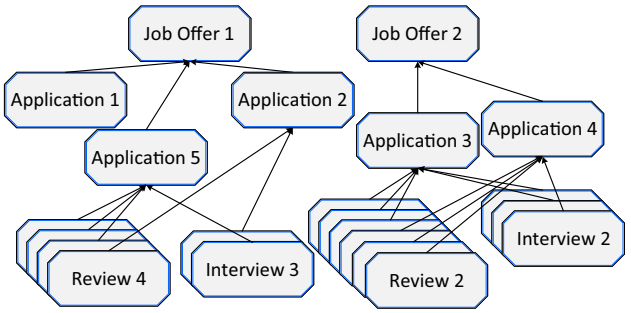

Fig. 9 Run-time relational process structure, tracking every process instance and relation (simplified view)

process structure. In this context, the terms *lower-* and *higher-level* become important. For illustration, *Job Offer* is denoted as a higher-level process in respect to process *Application*, as there is a directed relation from *Application* to *Job Offer* (cf. Fig. 8). *Job Offer* is higher-level to *Review* and *Interview*. Analogously, *Review* and *Interview* are lower-level processes in respect to process *Application*. This terminology applies to transitive relations as well. At run-time, a possible relational process instance structure d^I may look like as depicted in Fig. 9.

For the purpose of coordination processes, each process is required to know all its related processes, specifically its lower- and higher-level processes. In order to avoid computationally expensive queries every time lower- or higher-level processes are needed, the relational process structure maintains two sets per process ω: L_ω for all lower-level processes and H_ω for all higher-level processes. Process ω is part of these sets by definition, i.e., $\omega \in L_\omega$. Note that these sets exist at both design- and run-time. These sets are kept up to date as the process structure evolves, providing a crucial performance benefit to process coordination [39] at run-time.

Altogether, relational process structures allow a coordination approach to gain full knowledge over processes and their relations, thereby enabling fine-grained and comprehensive process coordination. Relational process structures represent one foundation for coordination processes.

3.2 Semantic relationships

Semantic relationships are means to specify *coordination constraints* at a high level of abstraction [37]. A coordination constraint is a formal or informal statement describing one or more conditions or dependencies that exist between processes. For example, the statement "An application may only be accepted if three or more reviews are positive" is a coordination constraint. *In essence, process coordination is tasked with formally capturing and enforcing coordination constraints.* Other coordination approaches, e.g., BPMN choreographies [34], choose messages to express the necessary interactions between the processes to be coordinated. However, due to complex process relationships and large amounts of process instances, defining messages in a procedural manner is cumbersome. This is especially true for larger relational process structures.

A coordination constraint must be expressed in terms of semantic relationships for its use in a coordination process. *A semantic relationship describes a recurring semantic pattern inherent in the coordination of processes in a one-to-many or many-to-many relationship* (cf. Table 2). As one example of a pattern, several process instances may depend on the execution of one other process instance. Semantic relationships thereby abstract over possibly multiple message exchange sequences and are inherently asynchronous. For a proper representation of coordination constraints, the combination of multiple different semantic relationships might become necessary. Moreover, a semantic relationship may only be established between processes if a (transitive) relation within the relational process structure, i.e., a dependency, exists between these processes. Figure 10 illustrates the types of semantic relationships between different processes. The *self-semantic relationship* is not depicted due to being trivial.

Semantic relationships are specified at design-time in context of a coordination process. Formally, a semantic relationship s^T is defined as follows:

Definition 8 A semantic relationship s^T has the form $(\acute{}, \lambda, \Sigma^T_{valid}, \omega^T_{ca})$ where

- $\acute{}$ is the identifier of the semantic relationship, $\acute{} \in \{top{-}down, bottom{-}up, transverse, self, self\text{ - }transverse\}$
- λ is an expression, configuring s^T in case of $\iota \in \{bottom{-}up, transverse, self{-}transverse\}$
- Σ^T_{valid} is a set of state types in case of $\iota \in \{top{-}down\}$
- ω^T_{ca} refers to the common ancestor in case of $\iota \in \{transverse, self{-}transverse\}$

Semantic relationships are always defined between two types of processes. Different semantic relationships, determined by the identifier $\acute{}$, signify different basic constraints (cf. Table 2). One of the outstanding features regarding semantic relationships is that the appropriate semantic relationship can be automatically inferred, helping a modeler of a coordination process. This is possible as the direction of the relations directly implies certain semantic relationships between process types [39]. This is exemplified in Example 4.

Example 4 (*Top-Down and Bottom-Up Semantic Relationships I*) Consider Fig. 8: A top-down semantic relationship can be established from *Job Offer to an Application*, as there is a relation from *Application* to *Job Offer*. *Additionally, a bottom-up semantic relationship can be established from*

Table 2 Overview over semantic relationships

Name	Description of the semantic relationship
Top-down	The execution of one or more lower-level processes depends on the execution status of one common higher-level process
Bottom-up	The execution of one higher-level process depends on the execution status of one or more lower-level processes of the same type
Transverse	The execution of one or more processes is dependent on the execution status of one or more processes of different type. Both types of processes have a common higher-level process
Self	The execution of a process depends upon the completion of a previous step of the same process
Self-transverse	The execution of a process depends on the execution process of other processes of the same type. All processes have a common higher-level process

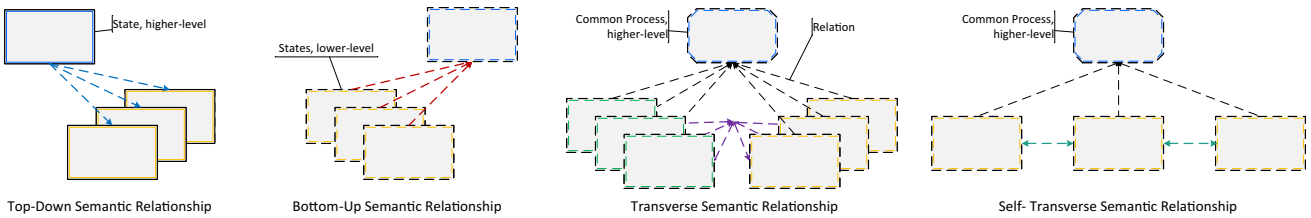

Fig. 10 Semantic relationships

Application to a *Job Offer*. *The direction of the connection and the direction of the relation determine directly the type of semantic relationship. Note also that one relation supports establishing multiple semantic relationships on top.*

The *execution status* referred to in Table 2 is represented by the state-based view of the process (cf. Sect. 3). At run-time, semantic relationships have a logical value to indicate whether or not they are satisfied; Boolean operators are required to express more complicated coordination logic involving more than one semantic relationship.

Semantic relationships feature either an expression in case of a bottom-up, transverse, or self-transverse semantic relationship [37]. Top-down semantic relationships feature a state set [37]. Self-semantic relationships cannot be configured and do not possess an expression or a state set (cf. Definition 8). Expressions and state sets may be addressed collectively by using the umbrella term *coordination condition*. A coordination condition modifies the basic semantics of the semantic relationship (cf. Table 2), which is needed to customize a semantic relationship to specifically represent a coordination constraint.

3.3 Coordination processes

Coordination processes are a generic concept for coordinating interdependent processes by expressing coordination constraints with the help of semantic relationships, which are then enforced at run-time [38]. The concept allows specifying sophisticated coordination constraints for vast structures of interrelated process instances with an expressive, high-level graphical notation using a minimum amount of modeling elements.

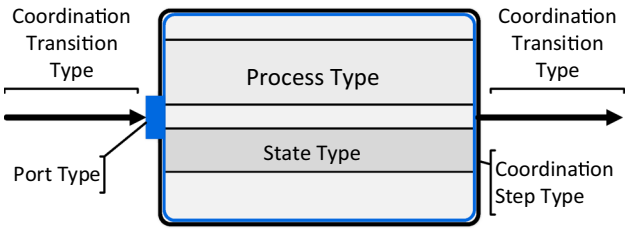

Fig. 11 Coordination process modeling elements

A *coordination process type* is a design-time entity and is represented as a directed, connected, acyclic graph that consists of *coordination step types*, *coordination transition types,* and *port types* (cf. Fig. 11). A formal definition for coordination process types is presented in Definition 9. Figure 12 shows the coordination process type for the processes of the running example, which ensures the correct enactment of the overall recruitment business process.

Definition 9 (Coordination Process Type) A coordination process type c^T has the form $(\omega^T_{coord}, B^T, \Delta^T, H^T)$ where

- ω^T_{coord} refers to the process type to which the coordination process type c^T belongs
- B^T is a set of coordination step types β^T (cf. Definition 10)
- Δ^T is a set of coordination transition types δ^T (cf. Definition 11)
- H^T is a set of port types η^T (cf. Definition 12)

Coordination steps are the vertices of the graph referring to a process type ω^T as well as to one of its states σ^T of its state-based view θ^T, e.g., *Job Offer* and state *Published*. For the sake of convenience, a coordination step β^T is addressed with referenced process type and state in the form of *ProcessType*:*State*, e.g., *Job Offer*:*Published*. A formal definition for coordination steps is presented in Definition 10.

Definition 10 (Coordination Step Type) A coordination step type β^T has the form $(c^T, \omega^T, \sigma^T, \Delta^T_{out}, H^T)$ where

- c^T refers to the coordination process type (cf. Definition 9)
- ω^T refers to a process type (cf. Definition 1)
- σ^T refers to a state type belonging to ω^T, i.e., $\sigma^T \in \omega^T.\theta^T.\Sigma^T$
- Δ^T_{out} is a set of outgoing coordination transition types δ^T (cf. Definition 11)
- H^T is a set of port types η^T (cf. Definition 12)

A *coordination transition* δ^T is a directed edge that connects a *source coordination step type* β^T_{src} with a *target coordination step type* β^T_{tar} (cf. Fig. 12 and Definition 11).

Definition 11 (Coordination Transition Type) A coordination transition type δ^T has the form $(\beta^T_{src}, \eta^T_{tar}, s^T)$ where

- β^T_{src} refers to the source coordination step type (cf. Definition 10)
- η^T_{tar} refers to the target port type (cf. Definition 12)
- s^T is a semantic relationship between $\beta^T_{src}.\omega^T$ and $\eta^T_{tar}.\beta^T.\omega^T$

More precisely, δ^T connects to one of multiple *ports* η^T_{tar} that are attached to β^T_{tar}. Definition 12 provides a formal definition of ports.

Definition 12 (Port Type) A port type η^T has the form (β^T, Δ^T_{in}) where:

- β^T refers to the coordination step type to which this port type belongs (cf. Definition 10)
- Δ^T_{in} refers to the set of all incoming coordination transitions δ^T (cf. Definition 11)

By creating a coordination transition between source step β^T_{src} and target step β^T_{tar}, a semantic relationship s^T is created as well. Conceptually, a semantic relationship is attached to a coordination transition. With the relations from the relational process structure and the definitions of semantic relationships (cf. Table 2), the identifier ´ can be automatically derived. The identifier ´ determines which semantic relationship is established between the process types referenced by the two coordination steps.

Example 5 (Top-down and bottom-up semantic relationships II) Consider Fig. 12: Connecting *Job Offer*:*Published* with *Application*:*Creation* constitutes a top-down relationship. The sequence in which the steps occur is important for determining the type of semantic relationship. By connecting *Application*:*Sent* with *Job Offer*:*Closed*, a bottom-up semantic relationship is established instead, as *Application* is a lower-level process type of *Job Offer*.

As coordination transitions represent coordination constraints with semantic relationships, coordination constraints depend on previous constraints for fulfillment. In Example 5, activating *Job Offer*:*Closed* requires at least one *Application* in state *Sent*, which in turn requires *Job Offer*:*Published to be activated. The coordination constraint between Job Offer*:*Closed and Application*:*Sent depends on the constraint between Job Offer*:*Published and Application*:*Creation*. Therefore, coordination process graphs must be acyclic, otherwise cyclic dependencies and, therefore, deadlocks are possible. Consequently, the acyclicity of coordination processes is not a restriction of expressivity, but a requirement for correctness.

Moreover, a coordination process is not required to coordinate all processes at every point in time. Depending on the coordination constraints, only the processes and states that are necessary for these constraints need to be modeled and are therefore subject to coordination. States and processes that do not occur in a coordination process model are not constrained in their execution by process coordination. Consequently, coordination process allow for a high degree of freedom in executing processes by only providing coordination when absolutely required.

Ports allow realizing different semantics for combining semantic relationships [39]. Connecting multiple coordination transitions to the same port corresponds to AND-semantics, i.e., all semantic relationships attached to the incoming transitions must be enabled for the port to become enabled as well. Enabling a port also enables the coordination

step, allowing the state of the coordination step to become active. Generally, at least one port of a coordination step must be enabled for the coordination step to become enabled as well. Consequently, connecting transitions to different ports of the same coordination step corresponds to OR-semantics.

A coordination process c corresponds to a directed, acyclic graph which possesses exactly one start coordination step $\beta_{start} \in c.B$ and a set of end coordination step types $B_{end} \subset c.B$. The notions of start and end coordination step apply equally to types and instances. A start coordination step has no ports η and, consequently, no incoming transitions δ, i.e., $\beta_{start}.H = \emptyset$. Analogously, an end coordination step β_{end} has no outgoing transitions, i.e., $\beta_{end}.\Delta_{out} = \emptyset$. Coordination process enactment begins at start step β_{start} and terminates when reaching an end step $\beta_{end} \in B_{end}$.

A coordination process is attached to a particular process type within the relational process structure. This process type is denoted as a *coordinating process type* ω^T_{coord}. Note that ω^T_{coord} is a short-hand notation for a process ω^T_i being a coordinating process type, i.e., $\exists c : c.\omega^T = \omega^T_i$, and does not signify one specific process. A coordinating process type is a process ω^T with an attached coordination process type c^T that functions as the coordinator. The notion of coordinating process applies as well to the run-time at the instance level, i.e., there may be one or more coordinating process instances. For the recruitment example (cf. Example 1), *Job Offer* is designated as the initial coordinating process type.

For illustrating the concepts of semantic relationships and coordination processes, the following Example 6 gives a rundown of the coordination process (cf. Fig. 12) of the recruitment example (cf. Example 1) and the *most important* coordination constraints that it represents. Encircled numbers ⓝ represent points of interest in Fig. 12.

Example 6 (Coordination Process Rundown) Any *Job Offer* process begins enactment in the start state *Preparation*, represented by the start coordination step type of the coordination process (cf. Fig. 12). The outgoing self-semantic relationship signifies the transition to state *Published* of the *Job Offer*. Then, Coordination Constraint 1 is represented using a top-down semantic relationship ①.

Coordination Constraint 1 *An application may only be created as long as the corresponding job offer is published*

Following coordination step type *Application*:*Creation*, again a self-semantic relationship allows an application to transition to state *Sent*. When in state *Sent*, *Reviews* may be created for the *Application* (cf. Coordination Constraint 2), a constraint that is represented again by a top-down semantic relationship ②. Multiple lower-level processes (*Reviews*) depend upon the execution status (state *Sent*) of one higher-level process (the *Application*) (cf. Table 2).

Coordination Constraint 2 *An application may only be reviewed once it has been sent to the company*

Moreover, at least one *Application* in state *Sent* allows a *Job Offer* to reach next state *Closed* (cf. Coordination Constraint 3). For representing this coordination constraint, a bottom-up semantic relationship is established between coordination step types *Application*:*Sent* and *Job Offer*:*Closed* ②. This is due to *Job Offer* being a higher-level process of *Application* (cf. Table 2).

Coordination Constraint 3 *A job offer may be closed once at least one application has been received*

Coordination Constraint 4 states when *Applications* may reach state *Rejected* or when *Interviews* may be created. Rejection is handled by a bottom-up semantic relationship between coordination step types *Review*:*Reject Proposed* and *Application*:*Rejected* ③. The precise semantics of the bottom-up semantic relationship are accomplished with an expression λ (cf. Definition 8).

Coordination Constraint 4 *An interview with the applicant may only be performed if at least three reviews or the simple majority of reviews are in favor of the applicant. Applications for which this is not the case must be rejected*

In case of favorable *Reviews*, a transverse semantic relationship is established between *Review*:*Invite Proposed* and *Interview*:*Preparation* ④. *Interviews* depend on *Reviews* in the context of a particular *Application* (cf. Table 2). The *Application* serves as the common ancestor ω_{ca} of the transverse semantic relationship (cf. Definition 8). The precise semantics of the transverse semantic relationship are again established with an expression λ (cf. Definition 8). In case of unfavorable reviews, the *Application* must be rejected. *Interview*:*Reject Proposed* is connected to a second port of coordination step *Application*:*Rejected* ⑤.

Two ports on the same coordination step constitutes OR-Semantics, as an *Application* may be rejected due to unfavorable *Reviews* or unfavorable *Interviews*. After *Interviews* have been created and conducted, another assessment of the applicant is accomplished. In case of favorable *Interviews*, the *Application* may be *Accepted* (cf. Coordination Constraint 5). Hence, a bottom-up semantic relationship is established between *Interview*:*Hire Proposed* and *Application*:*Accepted* ⑥.

Coordination Constraint 5 *At least one interview or a simple majority of interviews must be in favor of the applicant before the applicant can be accepted for the job offer*

In addition to the bottom-up semantic relationship representing Coordination Constraint 5, another coordination constraint affects the acceptance of an *Application* (cf. Coordination Constraint 6)

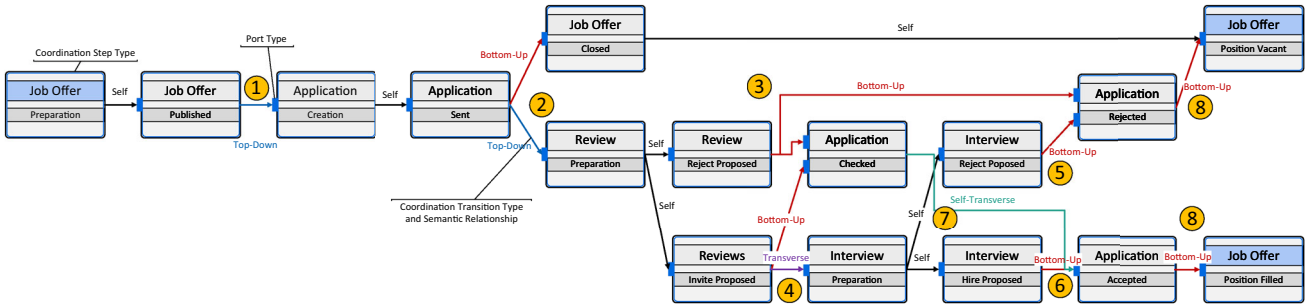

Fig. 12 Coordination process of the running example

Coordination Constraint 6 *Only one applicant may be accepted for a job offer.*

Here, *Applications* depend on other *Applications*; hence, a self-transverse semantic relationship is established ⑦ (cf. Table 2). The self-transverse semantic relationship permits only one *Application* to reach state *Accepted*, whereas subsequent *Applications* are blocked. The self-transverse semantic relationship connects to the same port of *Application*:*Accepted* as the previous bottom-up semantic relationship. This represents AND-semantics, as Coordination Constraints 5 and 6 need to be fulfilled simultaneously.

Finally, Coordination Constraint 7 determines under which conditions a *Job Offer*:*Closed* may terminate.

Coordination Constraint 7 *The job offer is successfully completed when an applicant has been found. If no suitable applicant is found, the job offer ends with status "Position vacant."*

The representation of Coordination Constraint 7 must be split into two semantic relationships. One bottom-up semantic relationship established between *Application*:*Rejected* and *Job Offer*:*Position Vacant* represents the case where no suitable applicant can be found. A second bottom-up semantic relationship between *Application*:*Accepted* and *Job Offer*:*Position Filled* represents the opposite case, i.e., a suitable applicant has been found ⑧.

Coordination constraints, as demonstrated in Example 6, can be found in any domain. These can be represented using semantic relationships and ports. Coordination processes already realize Stage-1 Decentralized Coordination by being able to be instantiated multiple times. In the following, concepts are presented to make coordination processes effectively realize Stage-2 Decentralized Coordination. These concepts address primarily the issue of coordination responsibility.

4 Decentralized process coordination

Coordination processes possess the technical capability to be employed in a decentralized fashion by design. However, additional concepts are required to effectively realize Stage-1 and Stage-2 Decentralized Coordination. In regard to establishing Stage-1 Decentralized Coordination, the baseline is as follows: A coordination process model c^T represents coordination constraints between process types in terms of (multiple) semantic relationships s^T. The process types ω^T to be coordinated and their relations π^T are captured in a relational process structure d^T. Furthermore, a coordination process c^T is always attached to a process type, which is then denoted as a *coordinating process type* ω^T_{coord}. The "coordinating process type" property meets the criteria for Stage-1 Decentralized Coordination. Consequently, coordination processes already represent Stage-1 Decentralized Coordination, as they are instantiated together with the coordinating process type ω^T_{coord}. A central coordinator (i.e., Stage-0 Decentralized Coordination) can be realized by instantiating a coordinating process type only once, with one coordinating process type per process structure, i.e., it holds $|[\omega^T | \exists c^T . \omega^T_{coord} = \omega^T, \omega^T \in d^T]| = 1$. In the following, it is shown how further decentralization can be achieved with coordination processes, i.e., how Stage-2 Decentralized Coordination can be realized.

In principle, any process type in a relational process structure d^T may become a coordinating process type ω^T_{coord}, i.e., there may be as many coordination processes $c^T_i \in C^T$ as there are process types $\omega^T_i \in \Omega^T$, $|C^T| \leq |\Omega^T|$. Whether this is actually a reasonable decentralization is an entirely different matter. As such, in principle, a relational process structure d^T may be coordinated using multiple coordination processes c^T, establishing a prerequisite for Stage-2 Decentralized Coordination. However, just creating multiple coordination processes does not result in a meaningful overall process coordination that leads toward the particular goal of the overall business process.

In general, when coordinators are decentralized, one of the primary challenges concerns *coordination responsibility*, i.e., deciding which coordination process c^T shall be responsible for which processes ω^T. If this remains arbitrarily defined, or not at all, detrimental consequences might occur. For example, a process modeler may specify coordina-

tion constraints in any coordination process. In consequence, a specific coordination process might not be an obvious choice to look for this particular coordination constraint later on. If a particular coordination constraint needs to be modified, potentially all coordination processes need to be searched for this particular coordination constraint. This problem becomes more pronounced as a process structure grows in size. Ultimately, this results in a process structure and business process, respectively, becoming unmaintainable and confusing to understand. While coordination responsibility is of general concern to any coordinator, this paper is particularly concerned with *coordination process responsibility*.

In particular, coordination processes may comprise coordination constraints for several processes, i.e., they enforce the same or different coordination constraints on the same set of processes. Consequently, it is crucial that multiple coordination processes do not model contradicting constraints, e.g., a combination of constraints stating exactly the opposite of another constraint. With decentralized coordinators, this challenge gains importance as coordinators are modeled individually, i.e., contradictions may not be spotted easily. Consequently, the relational process structure offers a way to address this challenge, i.e., avoiding the possibility for contradictions altogether by clearly defining the responsibility of each coordinator. Furthermore, this includes that there is little to no overlap between the individual responsibilities of each coordination process. A particular process type should clearly be the responsibility of one coordination process, not of multiple.

The fact that relations in a relational process structure are directed offers a fundamental building block for creating a solution for defining coordination process responsibilities. The directed relations imply that processes can be arranged hierarchically. This hierarchy is an integral part of how semantic relationships work, the cornerstone of the coordination process concept. Additionally, the hierarchy of a relational process structure offers advantages when using multiple coordination processes for coordinating the processes in a relational process structure.

4.1 Coordination process scope

For clearly defining responsibilities, the concept of *scope* of a coordination process is essential. A coordination process is attached to a coordinating process type, and its scope determines which other processes the coordination process is permitted to coordinate, i.e., its *responsibility*. The coordinating process can be easily identified from a coordination process model. By convention, the start and end steps of a coordination process must refer to the coordinating process type [39]. The hierarchy of the relational process structure provides an easy and intuitive solution for defining the scope.

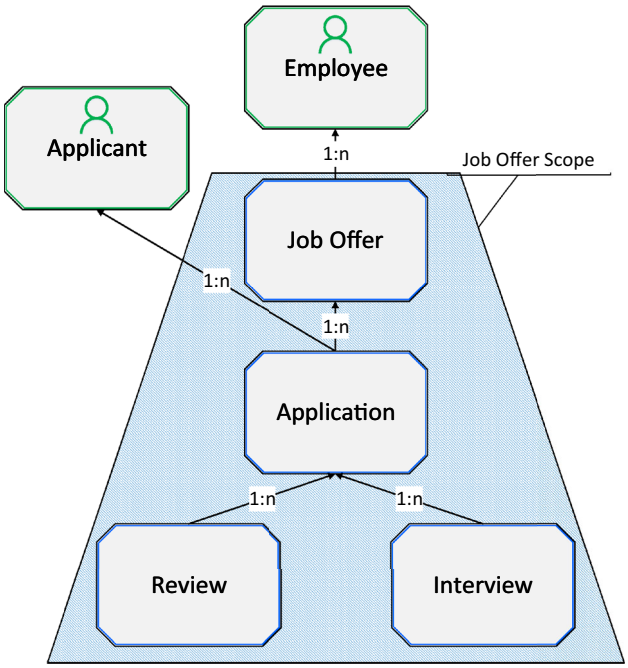

Fig. 13 Job offer coordination process scope

The scope of a coordination process is defined as all *lower-level process types* $L^T_{\omega^T_{coord}}$ of the coordinating process type ω^T_{coord}. Lower-level processes are all process types that have a (transitive) relation to one particular process type. For example, in the running example the coordinating process type ω^T_{coord} is *Job Offer*. For the sake of easier referencing, scopes of different coordination processes are distinguished by referring to the name of the coordinating process type ω^T_{coord}, e.g., the scope of *Application*.

Example 7 *(Scope)* Regarding the relational process structure from Fig. 8, *Review* and *Interview* are both lower-level processes of *Application*, which, in turn, are all lower-level processes of *Job Offer*. Figure 13 shows the scope of the *Job Offer* process type. Attaching a coordination process to the *Job Offer* consequently allows coordinating the entire relational process structure in Fig. 13, i.e., *Reviews*, *Interviews*, *Job Offers*, and *Applications*. This excludes processes that concern users, i.e., *Applicant* and *Employee*.

With the scope of a coordination process, it is achieved that the responsibility of a coordination process is not arbitrarily, but clearly defined. This provides a great advantage when modeling decentralized coordination processes, as arbitrary responsibilities of multiple coordinators create unnecessary redundancy as well as potentially contradicting constraints. Moreover, it would decrease the maintainability and understandability of the overall model.

While the scope defines the responsibility of a coordination process, in a relational process structure, the scopes of

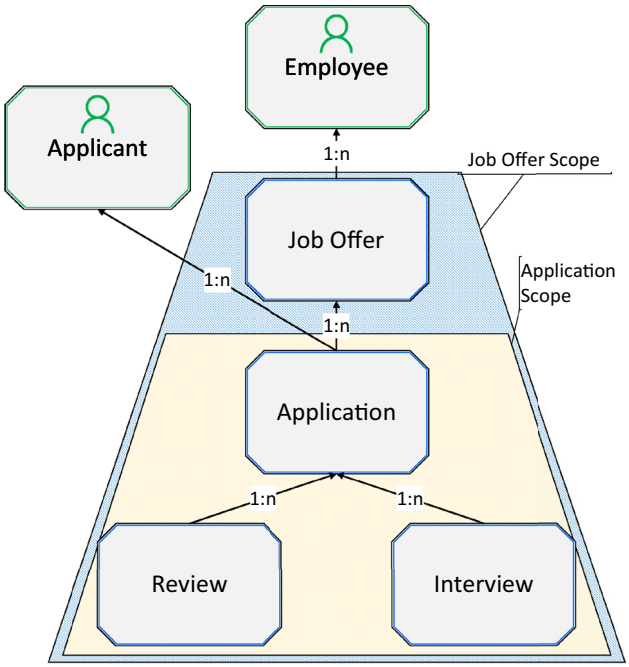

Fig. 14 Job offer and application coordination process scopes

multiple coordination processes may still overlap (cf. Example 8).

Example 8 *(Overlapping Scope I)* When a coordination process is attached to the top-level process in the hierarchy of the relational process structure, its scope overlaps with the scopes of coordination processes attached to lower-level processes (cf. Fig. 14). Consider the unsolicited application from Example 2. *Application* is a lower-level process type of *Job Offer* (cf. Fig. fig:Statespsbasedspsviewsspsof). An unsolicited application requires its own coordination process in absence of the coordination process from a *Job Offer*. However, a *Job Offer*, together with its associated coordination process, will be created in case the unsolicited application is accepted in the end.

As Example 8 and Fig. 14 show, the scope of one coordination process may be overlapping with scopes of other coordination processes. It can even be the case that the scope of one coordination process is fully contained within the scope of another coordination process. Overlapping scopes are synonymous with insufficiently defined coordination responsibility (cf. Example 9). Therefore, additional concepts are required to remedy the overlap between scopes of coordination processes.

Example 9 *(Overlapping Scope II)* The *Job Offer* coordination process has the process *Application* in scope. In fact, the scope of the *Application* is fully contained within the *Job Offer* scope. As such, the danger of modeling contradicting or redundant coordination constraints in the coordination processes of *Job Offer* and *Application* still exists.

4.2 Subsidiarity

The issue of overlapping scopes can be solved by defining which coordination constraints belong to a particular coordination process. As shown with Example 8, simply attaching a new coordination process to a process type would create overlapping scopes with other coordination processes. The coordination constraints required to coordinate process types present in both scopes would have to be replicated in the new coordination process. This would create unnecessary redundancy in the coordination constraints.

In addition to redundancy, contradicting constraints in multiple coordination processes may, in principle, inadvertently be specified. However, the hierarchy of the relational process structure allows for additional measures to remove overlap—the application of the *subsidiarity principle*. The Oxford dictionary defines subsidiarity as follows:

> **Subsidiarity** (noun)(in politics) the principle that a central authority should have a subsidiary function, performing only those tasks which cannot be performed at a more local level.[1]

Subsidiarity allows the scope of a particular process type $\omega^T_{coord,i}$ to extend only as far as another coordinating process type $\omega^T_{coord,i+x}$ downward in the process hierarchy. The hierarchy level i is counted from the top of the relational process structure hierarchy, where the top level processes belong to hierarchy level 0. The number $x > 0$ signifies an offset for the hierarchy level of the next coordination process in the hierarchy. Restricting the scopes of coordination processes in this manner achieves minimal overlap between scopes (cf. Example 10).

Example 10 *(Establishing Subsidiarity I)* In the running example, applying subsidiarity restricts the scope of the *Job Offer* process type to extend only as far the *Application*, and no longer involves *Review* or *Interview*, as illustrated in Fig. 15.

Note that the process type $\omega^T_{coord,i+x}$ still lies in both the scopes of the coordination processes $\omega^T_{coord,i}$ and $\omega^T_{coord,i+x}$. However, as coordination constraints always describe dependencies between two or more processes, a non-ambiguous assignment to a corresponding scope can be found regardless. Suppose there is a coordination constraint involving $\omega^T_{coord,i}$ and $\omega^T_{coord,i+x}$, the subsidiarity principle assigns the coordination constraint to the coordination process of $\omega^T_{coord,i}$. This is because process type $\omega^T_{coord,i}$ is not in the scope of $\omega^T_{coord,i+x}$. Suppose further there is a third process type further down the hierarchy, denoted ω^T_{i+y} with $y > x$. Then, a

[1] https://en.oxforddictionaries.com/definition/subsidiarity.

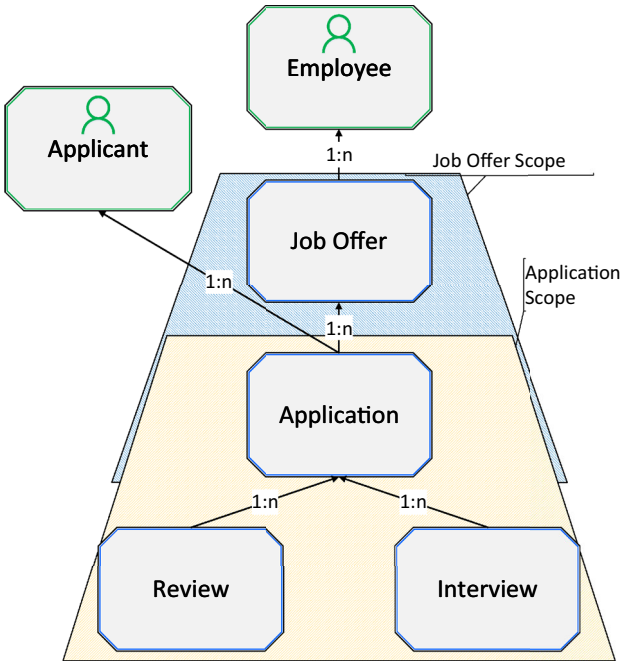

Fig. 15 Scopes adjusted based on the subsidiarity principle

coordination constraint involving $\omega^T_{coord,i+x}$ and ω^T_{i+y} would be assigned to the coordination process of $\omega^T_{coord,i+x}$, as both $\omega^T_{coord,i+x}$ and ω^T_{i+y} are in the scope of this coordination process. For the running example, subsidiarity is demonstrated in Example 11.

Example 11 *(Establishing Subsidiarity II)* The *Application* process type still lies in both scopes (cf. Fig. 15). For a coordination constraint (e.g., Coordination Constraint 1) that involves process types *Job Offer* and *Application*, the coordination constraint should belong to the *Job Offer* scope. Coordination constraints involving process types *Application* and *Review* consequently belong to the *Application* scope.

Transferring the subsidiarity principle to both coordination processes and the relational process structure, subsidiarity means that a coordination constraint should be modeled in the lowest coordination process whose scope comprises all process types involved in the constraint. So for a coordination constraint involving two process types ω^T_a and ω^T_b where a and b are hierarchy levels , this constraint should be assigned to a coordination process where its coordinating process type $\omega^T_{coord,i}$ fulfills the properties set out by Definition 13:

Definition 13 (Coordination Constraint Assignment) Given a coordination constraint involving process types ω^T_a and ω^T_b and the possible coordinating process types $\omega^T_{coord,i}$ with $i \in \{0, .., |\Omega|\}$, the coordination constraint should be assigned to $\omega^T_{coord,i}$if

1. $\omega^T_a \in L^T_{\omega^T_{coord,i}} \wedge \omega^T_b \in L^T_{\omega^T_{coord,i}}$: The processes ω^T_a and ω^T_b are related to $\omega^T_{coord,i}$ or identical to $\omega^T_{coord,i}$. This automatically implies they are within scope
2. $i \rightarrow max$: $\omega^T_{coord,i}$ is the coordinating process type furthest down in the hierarchy

For coordination constraints involving more than two process types, the properties for subsidiarity can be extended in straightforward way. For the recruitment business process, the application of the subsidiarity principle is shown in Example 12.

Example 12 *(Coordination Constraint Assignment)*

Regarding the unsolicited application (cf. Example 2), modeling any coordination constraints involving only *Application*, *Review*, and *Interview* in the *Job Offer* coordination process is a clear violation of subsidiarity.

By moving respective coordination constraints to the *Application* coordination process, subsidiarity is fulfilled. Only the coordination constraints for *Applica−tion* and *Job Offer* are kept in the *Job Offer* coordination process.

More precisely, consider Coordination Constraints 1-7 from Example 6.

- Coordination Constraints 1, 3, 6, and 7 involve only process types *Application* and *Job Offer*. Consequently, they should be modeled within the *Job Offer* scope.
- Coordination Constraints 2, 4, and 5 involve only process types *Review*, *Interview*, and *Application*. They therefore should be modeled within the *Application* scope.

Example 13 shows how Stage-2 Decentralized Coordination can be established.

Example 13 *(Established Stage-2 Decentralized Coordination)*

Considering the knowledge gained from Example 12, it becomes possible to re-model the *Job Offer* coordination process to account for decentralization and subsidiarity. This re-modeled coordination process is depicted in Fig. 16 and only contains coordination constraints pertaining to the subsidiarized *Job Offer* scope as outlined in Fig. 15. Consequently, the other coordination constraints involving *Review*, *Interview*, and *Application* process types have been moved to the *Application* coordination process (cf. Fig. 17).

The subsidiarity principle has been applied strictly, and the new coordination process fully meets its requirements. Moreover, together, both coordination processes of *Application* and *Job Offer* implement the same coordination constraints as before. As an added benefit, however, unsolicited applications may now be handled properly by the overall business process due to the dedicated coordination process of *Application*.

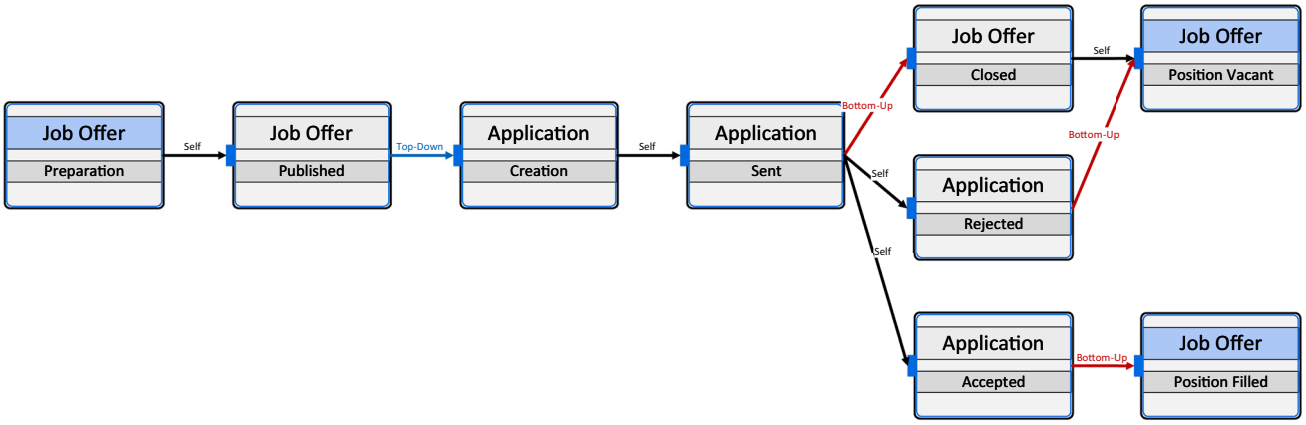

Fig. 16 Job offer coordination process

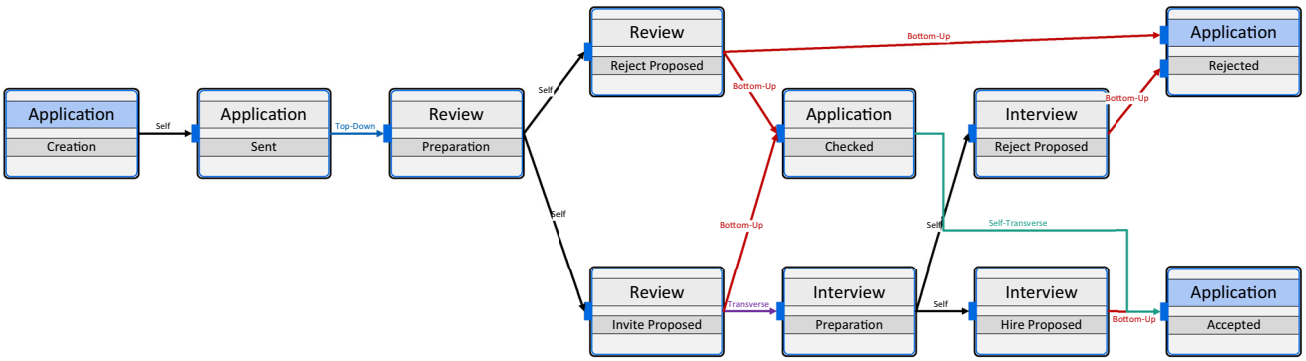

Fig. 17 Application Coordination Process

As depicted in Fig. 15, the *Application* process type is involved in both coordination processes. States *Application*:*Creation*, *Application*:*Sent*, *Application*:*Rejected*, and *Application*:*Accepted* are required to model coordination constraints for *Job Offers* as well as *Reviews* and *Interviews*.

As can be seen from Example 13, the overall number of elements in each coordination process has become noticeably smaller, reducing complexity significantly. Furthermore, due to the consequent application of the subsidiarity principle, there can be no conflict between coordination constraints. With the concepts of scope and subsidiarity, Stage-2 Decentralized Coordination can be achieved with coordination processes, unambiguously defining the coordination responsibility of each coordination process.

In summary, each coordination process is smaller, simpler, and more understandable in comparison with the coordination process depicted in Fig. 12. Altogether, the subsidiarity principle and scopes enable the decentralized coordination of small sections of a relational process structure with coordination processes, which, in turn, collaborate to provide coordination for the entire relational process structure. As such, Stage-2 Decentralized Coordination is fully viable.

4.3 Automatically establishing Stage-2 Decentralized Coordination

As an added value of the scope and subsidiarity concepts, the actual modeling of distributed coordination processes might benefit substantially from automation. Consider the recruitment example using Stage-1 Decentralized Coordination and the corresponding coordination process (cf. Fig. 12). The application of the subsidiarity principle results in the coordination processes depicted in Figs. 16 and 17. Establishing subsidiarity represents a pattern that, in principle, can be used to create an algorithm that automatically converts a model with Stage-1 Decentralized Coordination to a model with Stage-2 Decentralized Coordination. Essentially, the main idea of the algorithm consists of three steps. Note that these steps represent a sketch of an algorithm which ignores many special cases and details necessary for a fully viable algorithm. The intention of this sketch is to demonstrate feasibility of such an algorithm.

1. **Identification and Classification of Coordination Constraints.** Given a relational process structure with a Stage-1 coordination process model, the algorithm needs to identify the specific coordination constraints that may

be moved to a lower scope, as shown in Example 12. The Stage-1 coordination process model contains all the relevant constraints in the form of semantic relationships and coordination step types. The coordination constraints represented in this form are easily accessible for formal analysis of the involved process types. Due to the relational process structure, the process type involved in the coordination constraint that is highest up in the hierarchy can be easily identified. Each coordination constraint can consequently be classified by a process type using an appropriate classification algorithm. Thereby, different classes with a specific process type as class identifier are created. Based on these resulting classes additional coordination processes can be created. The class identifier process type becomes a new coordinating process type. The created coordination process models are initially empty, i.e., they do not contain coordination steps or coordination transitions. Moreover, this classification may be modified with parameters. For example, it should be possible to specify how many coordination processes should coordinate the relational process structure.

2. **Decentralization.** With the newly created empty coordination processes and the classified constraints, the constraints can be cut-and-pasted from the Stage-1 model to their respective Stage-2 coordination processes. The cut-and-paste fragments of the coordination processes, which represent one or more constraints, may be disconnected, i.e., no coordination transitions exist between them. The correct connections, however, can be deduced from the Stage-1 coordination process model. Moreover, the cut-and-paste of coordination constraints might leave gaps in the Stage-1 model. Both kinds of gaps may be filled by simply creating new coordination transitions between start and end connections of the coordination process fragments according to the original Stage-1 coordination process model. For example, this step of the algorithm can be shown by comparing coordination step *Application*:*Sent* connected with coordination steps *Application*:*Accepted* and *Application*: *Rejected* in Figs. 12 (Stage-1) and 16 (Stage-2). By cutting-and-pasting and creating new, additional coordination transitions, the modified Stage-1 coordination process model itself becomes a compliant Stage-2 Decentralized Coordination process model.
3. **Establishing Correctness.** The newly created Stage-2 coordination process models might not fully adhere to some of the correctness criteria outlined for coordination processes. For example, it is not guaranteed that start and end coordination steps reference the respective coordinating process type of the coordination process. In this case, proper start and end steps are added to the coordination process and connected to the existing start and end steps of the coordination process graph fragments.

In consequence, a Stage-1 decentralized business process model can be automatically transformed into a Stage-2 decentralized business process model. Note that this is only possible due to the inherent properties of relational process structures, semantic relationships, and coordination processes. Other approaches that support Stage-2 Decentralized Process Coordination might not be capable of automatically establishing Stage-2 Decentralized Coordination.

Incidentally, the way relational process structures and coordination processes support establishing Stage-2 Decentralized Coordination, it is possible to reverse Stage-2 Decentralized Coordination back to a model with Stage-1 Decentralized Coordination. Essentially, this means integrating the decentralized coordination process back into a single coordination process with the largest scope that is necessary. This can be used to test various configurations of decentralized coordination processes without having to model them individually and manually. The modeler may start with one central coordination process or multiple decentralized coordination processes. Using the algorithm and parametrization, the modeler can quickly adapt the coordination to various needs, depending on the goals or needs of the overall model.

Both establishing Stage-2 Decentralized Coordination and the reversal to Stage-1 Decentralized Coordination may be immensely beneficial for saving the process modeler significant effort when decentralizing or centralizing process coordination.

5 Distributed environments

Having decentralized coordination processes across different hierarchy levels yields significant benefits for the simplicity of the coordination process models. However, a factor that might significantly influence subsidiarity and Stage-2 Decentralized Coordination of a plethora of processes has not been discussed yet: The influence of distributed environments.

In settings where multiple processes collaborate to achieve a business goal, it is not unreasonable to assume that these processes may not all be executed on the same machine. Instead, processes may be executed on a multitude of different machines or servers, i.e., a distributed environment. With the advent of cloud computing, distributed applications are gaining even more momentum, as scalability is becoming an important issue [2,4].

The PHILharmonicFlows project has developed a hyper-scale process engine, called PHILharmonicFlows, that allows distributing processes horizontally across different computational nodes [2]. This enables superior performance in executing processes by leveraging the computing power of entire computational clusters. Consequently, the problem of executing relational process structures in a distributed environment is of considerable practical importance. Especially

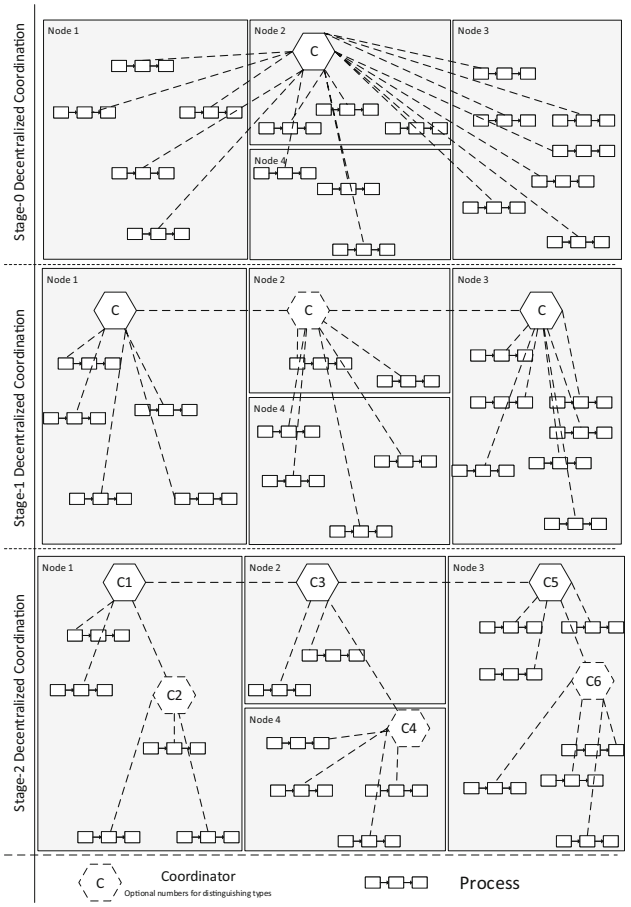

Fig. 18 Schematic view of the stages of decentralization in a distributed environment

regarding large process structures, coordination processes might become a bottleneck if the distributed environment is not properly taken into account

In detail, processes in a relational process structure may not all belong to the same (computational) node, e.g., a single server. In a distributed cluster, e.g., a cloud environment, there exist multiple nodes. Different processes may be assigned to different nodes. The distributed nodes may exist for different reasons, possible examples include different physical locations each being represented by a node, or simply multiple servers in the same company department or cluster. Figure 18 shows a schematic view how the different stages of decentralization and distribution fit together.

Consider the following example of a car manufacturing process, which exemplifies a distributed approach to a business process that produces a car (cf. Example 14).

Example 14 *(Car Manufacturing)* The automotive company "Generic Inc" assembles its best-selling model *GeneriCar* at its main facilities called *Plant A*. The constituting parts of *GeneriCar* are produced and assembled at various subsidiaries of Generic Inc. The *Substructure* of the car and its components are fabricated at *Plant B*, except for *Tires*, which are manufactured at *Plant T*. *Interior*, *Engine Compartment*, and *Body* are manufactured as well at different plants. The *Engine* itself, as a complex part, hast its own production facility called *Plant E*. All components of the *Car* are transported with trains and trucks between the different facilities. Each component undergoes a specific lifecycle process that describes production, testing, optional storage, delivery, and integration into the next higher-level component of the *Car*. As one of the first companies, Generic Inc rolled out a company-wide, *distributed* process-aware information system (PAIS) dedicated to the IT-support of enacting, coordinating, and monitoring the processes for each car component and their interdependencies. The system replicates the structure of the facilities, i.e., each plant has its own computational node in the company-wide PAIS.

Figure 19 shows a graphical overview of nodes, processes, and assignments of the processes to the nodes. The assignment of processes to nodes can have an impact on the optimal approach for reaching Stage-2 Decentralized Coordination. When choosing where to create decentralized coordination processes, the layout of clusters and nodes as well as the process assignment, must be taken into account.

In regard to process coordination in distributed environments, performance and scalability are the main challenges in addition to correct coordination. Specifically, communication between processes and, consequently, communication between nodes has an important impact on the overall performance of the distributed relational process structure. In general, *communication within a node is considered cheap, whereas communication between nodes is costly in terms of time and performance*. This holds regardless of any specific metrics, and communication between nodes should therefore be reduced to a minimum. Costly extra-node communication is showcased by Example 15.

Example 15 *(Extra-node communication I)* Consider node *Plant D* in Fig. 19 tasked with assembling the *Engine Compartment*. Processes for the *Cooling System*, *Battery*, and *Alternator* run together with the *Engine Compartment* on the same *Plant D* node. These processes are coordinated using a coordination process with *Engine Compartment* as the coordinating process type. As all these processes run on the same *Plant D* node, the coordination of the processes is performant as communication stays entirely within the node. However, the *Engine* is also part of the *Engine Compartment* assembly, but is produced at a different plant called *Plant E*. So the processes that comprise the *EngineCompartment*, i.e., *Engine*, *Piston*, *Spark Plug*, *Oil*, and *Crankshaft*, are also coordinated by the *Engine Compartment* coordination process, but are located on a different node. As such, the communication required to coordinate the processes on *Plant E* must cross node boundaries, as the coordination process is located on the *Plant D*

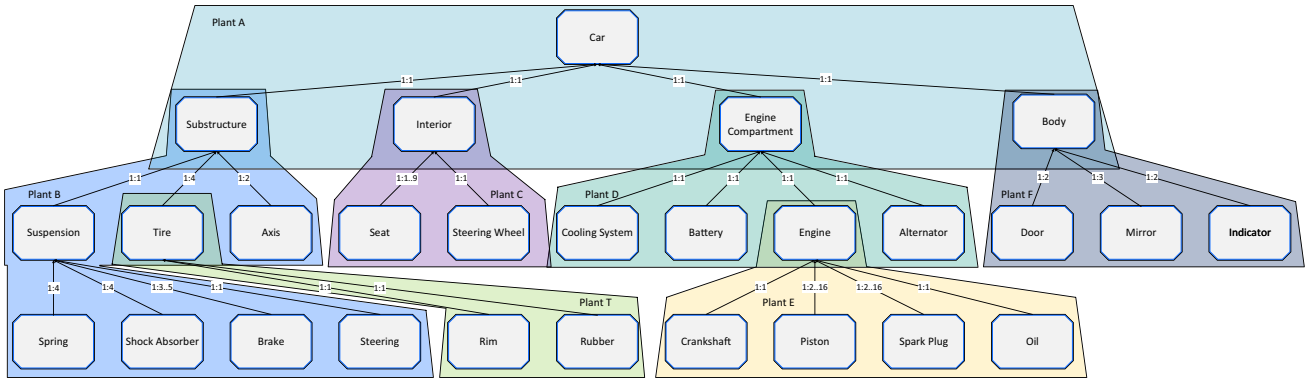

Fig. 19 Car manufacturing relational process structure

node. As there are five processes on the *Plant E* node, the communication overhead is severe due to extra-node communication cost.

Obviously, communication between nodes cannot be totally avoided, as processes need to be coordinated across nodes. Coordination processes, however, allow minimizing the communication between nodes significantly. By attaching coordination processes to process types where the scope encompasses the entire node, the communication is kept within a node, as exemplified by Example 16. Note that further coordination processes within a node are still possible by using additional coordination process types for Stage-2 Decentralization within the node itself (cf. Fig. 18).

Example 16 *(Extra-node communication II)* The communication overhead in Example 15 can be significantly reduced by creating a new coordination process with *Engine* as the coordinating process type. The previous extra-node communication required between *Piston*, *Spark Plug*, *Oil*, and *Crankshaft* and the *Engine Compartment* coordination process is internalized, i.e., communication solely occurs within node *Plant E* due to use of the *Engine* coordination process. *Piston*, *Spark Plug*, *Oil*, and *Crankshaft*, which are all located on the *Plant E* node, now exclusively communicate with the *Engine* coordination process. The *Engine Compartment* coordination process only coordinates with the *Engine* process externally.

As shown with Examples 15 and 16, distributed environments may have a huge influence on the appropriate decentralization of process coordination. Taking the layout of clusters and nodes into account when modeling multiple coordination processes increases the overall benefit offered by a decentralized approach, allowing for optimal process coordination.

Altogether, coordination processes allow for the decentralized coordination of large process structures. The relational process structure hierarchy, scope, and subsidiarity principle provide clear responsibilities for each coordination process, facilitating modeling and reducing modeling errors. In particular, the coordination approach no longer contains a single point of failure. By using multiple coordination processes for the same large process structure, the individual coordination process models become smaller and simpler, resulting in greater understandability and maintainability of the models. As shown, these advantages also translate well to a distributed cluster, where a coordination process can be used for each node, significantly reducing communication overhead and, therefore, increasing the performance of executing the processes.

6 Technical implementation

Coordination processes have originated in the object-aware process management paradigm [26]. The concepts of object-aware process management, i.e., objects, lifecycles, relations, and coordination processes, have been implemented in the PHILharmonicFlows prototype. A lifecycle process of an object conforms to the definition of process type (cf. Definition 1).

The initial prototypical implementation of PHILharmonicFlows as an object-aware process management system was developed from 2008 to 2012. It involved functional design- and run-time of the basic concepts of object-ware process management, though many advanced features could not be realized due to the technology available at the time. In 2015, a fully new implementation internally named "Proteus" was started, leveraging the emerging concepts of *microservices*. Microservices allow for a scalable and performant execution of object-aware processes. However, this also rendered almost all of the existing codebase of the previous prototype obsolete. The new paradigm of microservices also required to re-think and adapt many concepts of object-aware process management, accounting for the new requirements imposed by this fundamental change. Concepts such as object lifecycles [41], the relational pro-

cess structure [39], and coordination processes [38] were extended and adapted, with a major focus on the run-time. In particular, the operational semantics required a substantial overhaul. The re-implementation of PHILharmonicFlows also paved the way for the development of advanced features such as ad hoc changes for object aware-process management [3] and a hyper-scalable run-time environment [2] and enabled the concepts Decentralization and Distribution of process coordination, as presented in this article. This new and improved microservice-based implementation continues using the branding PHILharmonicFlows.

Both distribution and decentralization are inherently relevant to the object-aware process management paradigm in particular, but also to other data-centric approaches to BPM [42]. As business processes emerge from potentially large relational process structures containing interacting objects with lifecycles, decentralization of process coordination yields significant benefits. PHILharmonicFlows, the implementation of the object-aware paradigm, comprises a distributed process engine based on microservices, and may be deployed to a cloud-based cluster. As such, multiple coordination processes have significant benefits as well. The intention is to show that, in addition to the functional benefits of having multiple coordination processes, it also yields significant performance improvements, even when there is no distribution over multiple nodes.

6.1 Actors and microservices

With PHILharmonicFlows, much effort has been put into development to create a scalable process management system that supports a large number of concurrently running processes [2]. Object-aware process management is uniquely suited for this, as its conceptual elements, e.g., objects and their lifecycle processes, can be represented as individual *actors*. Actors in the *actor model theory* are the basic building blocks for concurrent computation [1]. Actor model theory serves as a theoretical foundation for implementing concurrent and distributed systems. Object-aware process management, with objects, lifecycles, and coordination processes corresponding to individual actors and the requirement for concurrent and scalable process execution, fits the basic notions of actor model theory flawlessly. Therefore, it was logical to design the new implementation of PHILharmonicFlows around actors and microservices.

In essence, an actor is an independent entity that consists of a message queue and a store for arbitrary data. An actor may receive messages from other actors or from external sources and processes them using data contained in the message and data from its store. An actor may only work on exactly one task at a time, i.e., conceptually it runs on one single computational thread. An actor servicing a message may only work on this single message, whereas all other messages are put in the queue until the current message will have been serviced. An *actor system* is realized by having multiple actors of different types that express different functionality. In such a system, the actors then may run concurrently and in parallel. Because of the single computational thread and the message queuing, most of the concurrency problems regarding persistence and computation, e.g., race conditions and dirty reads/writes, are not present in an actor system. Moreover, actors may communicate asynchronously.

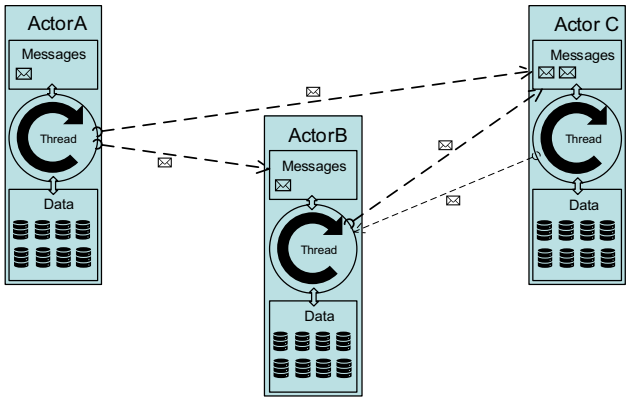

Fig. 20 Schematic actors and their communication

PHILharmonicFlows is realized as an actor system. Each object instance, together with its lifecycle process and its attributes, is implemented as one actor. Coordination processes are actors as well, but have a different actor type. Figure 20 shows a schematic view of actors and their communication. In particular, an actor may involve other actors when servicing a request, as required data or functionality may be located with other actors. Note that in Fig. 20, Actor A is servicing an external request, depicted by the message in its message queue and the outgoing communication from its thread.

Each actor in the PHILharmonicFlows system is realized as a *microservice* using Microsoft's *Azure Service Fabric Framework*[2]. Azure Service Fabric combines microservices with the actor paradigm, and therefore constitutes an ideal technical framework for building PHILharmonicFlows. The overall architecture of the PHILharmonicFlows system can be seen in Fig. 21. As has been demonstrated in [2], PHILharmonicFlows can scale horizontally very well, i.e., across distributed machines or a cloud.

Microservices are capable of running concurrently or in parallel by definition. As each process is implemented using a microservice, logically the concurrent execution of process instances is guaranteed by the PHILharmonicFlows implementation.

Still, the implementation must enable the asynchronous interactions between processes. Any object lifecycle process

[2] https://docs.microsoft.com/en-us/azure/service-fabric/.

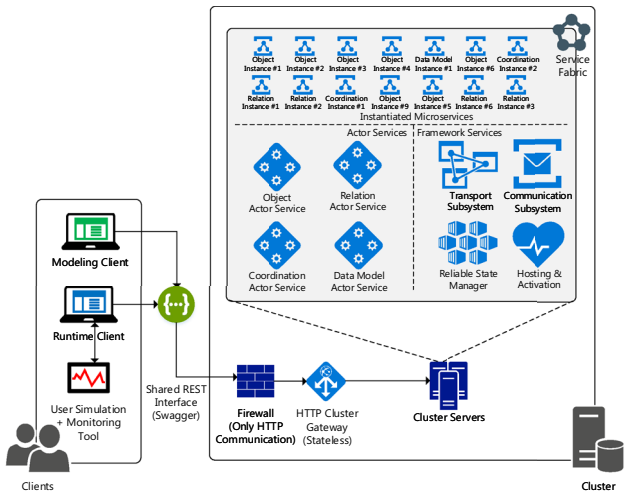

Fig. 21 PHILharmonicFlows architecture

is, in principle, independent from any other object lifecycle process, there is no coordination of object lifecycle processes, apart from coordination processes. Conceptually, semantic relationships enable the asynchronous execution of the coordinated processes. Semantic relationships are represented by actor data, and several communication exchanges between the actors to represent their functionality at a fundamental level. Their implementation uses the message stores and message exchange capability of the actors. Therefore, semantic relationships constitute abstractions over multiple, conditional series of messages between actors. As actors are inherently capable of asynchronous communication, the implementation of semantic relationships enables asynchronous process interactions as well. Therefore, true asynchronous execution of interdependent lifecycle processes is enabled by PHILharmonicFlows.

In summary, PHILharmonicFlows is capable of executing a multitude of processes concurrently and in parallel by using the Azure Service Fabric Framework to implement the concepts. Asynchronous communication is enabled by the underlying actors.

6.2 Decentralized coordination process performance on a single node

In order to prove that coordination processes have performance advantages when they are decentralized, an experiment was set up.

Experimental goal

It must be demonstrated that the execution time of the processes in a process structure, which is coordinated by multiple decentralized coordination processes, is (significantly) lower or equal than the same process executions using one single coordination process.

Experiment basics

For enabling the quantitative measurements for decentralized coordination processes, two PHILharmonicFlows models were defined:

1. the recruitment business process described in the running example (cf. Example 1) and
2. an insurance claim business process (cf. Fig. 22).

These models have been defined in three variants, which are used for comparison:

- Central coordination: One single coordination process
- Decentralized coordination: Two coordination processes
- No coordination: No coordination processes

In terms of object types, the insurance claim model is slightly larger than the recruitment model. All processes and coordination processes are located on the same node, i.e., there is no distribution across nodes for this experiment.

Furthermore, for each model, an execution sequence was defined that resembles a fairly standard and sufficiently complex execution of the business processes. An execution sequence defines a series of *actions*, describing at which points process instances are created or deleted, or when they change their state. In detail, an execution sequence action is created using one of the following functions (cf. Table 3) and supplying it with concrete parameter values.

Function $InstantiateProcess(\omega^T)$ creates a new process instance, given a process type ω^T. The function $LinkInstances(!_1^I, !_2^I)$ takes two process instances ω_i^I, $i = 1, 2$ as arguments and creates a relation between them, provided that a respective relation type exists in the model. Function $ChangeAttributeValue(\omega^I, \phi^T, v)$ writes value v to attribute instance ϕ^I of process instance ω^I. Attribute instances can be uniquely identified by their type ϕ^T, given the process instance ω^I. In case ϕ^I already has a value, the value is overwritten with v. Finally, function $CommitTransition(!^I, \phi^T)$ causes a state change, i.e., after completion of the function the target of the transition becomes the active state σ_a^I.

In PHILharmonicFlows, it is possible to execute processes using only these four main functions (cf. Table 3). This is enabled by the data-driven lifecycle processes in PHILharmonicFlows, on which the models are based. The details of lifecycle process execution have been described in [41]. Based on these functions, an execution sequence is designed that realizes a full business process execution involving multiple process instances.

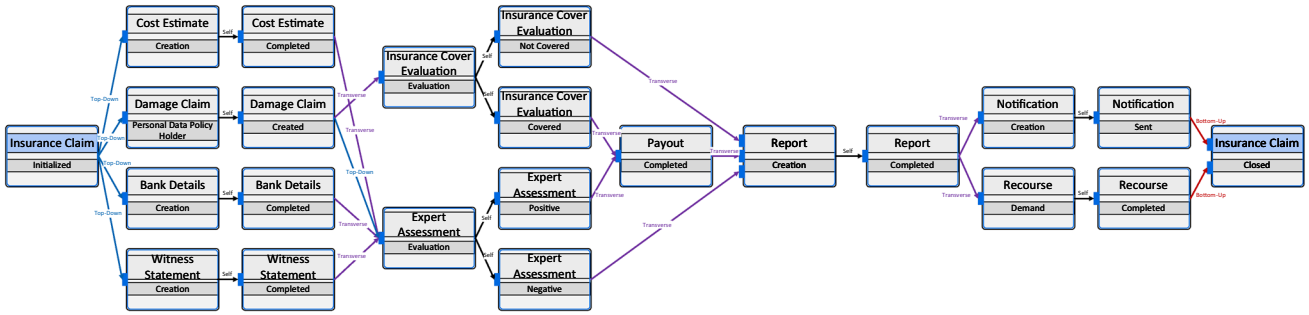

Fig. 22 Central coordination process of the insurance claim model

Table 3 Execution sequence operations

Function	Description
$InstantiateProcess(\omega^T)$	Creates a new instance of process type ω^T
$LinkInstances(!^I_1, !^I_2)$	Creates a new relation instance between process instances ω^I_1 and ω^I_2.
$ChangeAttributeValue(\omega^I, \phi^T, v)$	Writes value v of an attribute instance ϕ^I that has type ϕ^T of process ω^I
$CommitTransition(!^I, \phi^T)$	Commits transition τ^I that has type τ^T of process ω^I. Implies a state change of ω^I

The execution sequence describes how the instances of both models are executed, i.e., any instance of the model, regardless of the configuration of potential coordination processes, performs the same actions in the same order. The execution sequence is designed to not violate any coordination constraints in order to achieve identical results even when there is no coordination process involved. Otherwise, in one case, an action may be blocked by a coordination process. In case of a missing coordination process, the same action would not be blocked and create different results and therefore bias in the performance measurements. A full description of both models, together with the detailed execution sequences and their descriptions, as well as the results of all benchmarks, has been made available.[3]

Measurement setup

As PHILharmonicFlows supports parallel and concurrent process execution as enabled by the actor microservices, performance measurements follow the *guidelines for measuring the performance of parallel computing systems*, as described in [2]. This experiment reuses the exact methodology from [2] and is therefore not replicated in detail here for the sake of brevity. In short, the general idea is to dynamically determine the number of runs n needed to achieve a given confidence interval CI for the measured value, for a given confidence percentage $1-\alpha$.

The value measured is the execution time t^{1c}_{exec} of the execution sequence with the processes being coordinated by a central coordination process. Execution time t^{2c}_{exec} measures the same for decentralized coordination processes. For establishing a baseline, t^{nc}_{exec} denotes the time for enacting the execution sequence without any coordination process. t^{1c}_{exec}, t^{2c}_{exec}, and t^{nc}_{exec} are the summation of the execution time of each individual action in the execution sequence. All measurements in context of a scenario are denoted as a benchmark.

All benchmarks have been run on a Lenovo T470p notebook. It features a Intel(R) Core(TM) i7-7700HQ 4 Core/8 Thread CPU running at base clock 2.80GHz in stock configuration. The CPU was neither overclocked, undervolted, nor locked to a specific frequency. The laptop further has 16 GB RAM DDR3-2400 and an SSD. Software-wise, it runs Windows 10 Pro x64 v1903, Visual Studio Enterprise in the most up-to-date version (as of October 15th, 2019), and the debug-compiled, up-to-date PHILharmonicFlows software. The benchmarks were performed with the laptop plugged in, using best performance mode of Windows 10.

Results, observations, and interpretation

The overall execution times are reported in Table 4, giving an overall impression of the performance of the PHILharmonicFlows process engine. All execution times are provided in the form of standard intervals $[lower, upper]$, where time has the format $[ss:fff]$. Three Scenarios #1-3 have been run, two times using the recruitment business process, one time the insurance model containing the coordination process in Fig. 22. The confidence $1-\alpha$ of the respective t_{exec} confidence interval is given as a percentage rounded to two decimal places.

[3] The data can be found at https://bit.ly/2DvFZvk.

Table 4 Performance measurements of two process models

#	1	2	3
Model	Recruitment	Recruitment	Insurance
Instances	32	10	20
Actions	289	82	168
$CI\ t^{nc}_{exec}$	[01:613, 01:761]	[00:502, 00:530]	[01:069, 01:140]
n^{nc}	6	6	6
$1-\alpha$	96.88	96.88	96.88
$CI\ t^{2c}_{exec}$	[02:291, 02:421]	[00:597, 00:609]	[01:202, 01:268]
n^{2c}	6	6	6
$1-\alpha$	96.88	96.88	96.88
$CI\ t^{1c}_{exec}$	[02:484, 02:596]	[00:911, 00:978]	[01:256, 01:284]
n^{1c}	6	6	6
$1-\alpha$	96.88	96.88	96.88

- Scenario #1 is the recruitment business process, where 5 *Applications* are submitted and reviewed, each having 3-5 *Reviews*.
- Scenario #2 is a cut-down version of Scenario #1, where only one *Application* is submitted and subsequently accepted to fill the position.
- Scenario #3 is an insurance business process, comprising one instance of each process type.

Regarding the setup of the measurements, the execution sequences have been designed to resemble what can be considered fairly standard process executions. Within the constraint of being fairly standard, the sequences still try to prolong process execution, i.e., maximizing t_{exec}. Whenever branches may be chosen during decisions, the execution sequence chooses the longer path. Furthermore, the execution sequences take no advantage of the parallelism possible with the PHILharmonicFlows engine. Each sequence simulates a single user, executing each action sequentially. Note that this does not prevent the PHILharmonicFlows engine from using some parallel execution, i.e., its inherent parallelism. Still, the execution sequences constitute a worst case as far as the concurrent execution of processes is concerned.

Moreover, the models used for the measurements exhibit very high degrees in the amount of coordination required. Especially the recruitment example shows very tight coordination. The model has 4 processes with 5 states each, and of these 20 states in total, 4 are not subject to coordination by a coordination process (cf. Figs. 6 and 12). As such, the model almost maximizes the amount of coordination, leading to almost another worst case for the total execution time. The insurance example is less tightly coordinated. In light of these detrimental conditions, a maximum execution time of less than 3 seconds for Scenario #1 is satisfactory.

As can be seen in Table 4, running the same execution sequence of Scenario #1 with a central coordination process takes roughly $1s$ more compared to running without any coordination process. For all scenarios, the variant with decentralized coordination processes is slightly ahead of the central coordination variant using one coordination process. For Scenarios #1 and #2, the difference is $\sim 300ms$, whereas in Scenario #3 both intervals are approximately identical, with decentralized coordination being ahead by $\sim 20ms$ over one central coordination process.

Given the number of process instances (32) and number of actions (289) for Scenario #1 and sequential execution, total execution time manages to remain significantly below 3 seconds (highest interval $CI\ t^{1c}_{exec}$= [02:484, 02:596]. Scenarios #2 and #3 have less actions and consequently achieve better total execution times (highest interval $CI\ t^{1c}_{exec}$=[00:911, 00:978] for Scenario #2 and $CI\ t^{1c}_{exec}$=[01:256, 01:284] for Scenario #3. Note also that the execution sequences produce consistent results, as it takes only the minimum amount of runs (6) to obtain the necessary confidence level of $\geq 95\ \%$.

Table 4 shows that decentralization has clear benefits over central coordination. This is most likely due to the decentralized coordination processes allowing for better utilization of parallelism in the PHILharmonicFlows engine. Note that the usage of parallelism is not entirely prevented by the sequential application of individual actions. The sequential application only prevents more parallel execution. The decentralization allows splitting the workload across multiple coordination processes, creating a significant reduction of overall execution time. Without decentralization, all updates are done on the same coordination process. While the size of the effects varies with the specific model and execution sequence (cf. Scenarios #1 and #3 in Table 4), it is noticeable in all Scenarios shown in Table 4.

Table 5 displays various metrics related to the benchmarks of Scenarios #1 - #3. The data are obtained from the last (sixth) run of Scenario #1 of each individual benchmark run, as the total execution time is guaranteed to be within the interval bounds and, therefore, is representative. As the execution times are based on a single run, there is no variance and the interval notation of Table 4 is not needed.

Table 5 is partitioned by the functions presented in Table 3, permitting to draw some conclusions on where the performance benefit of decentralized process coordination comes from. Comparing the values for function $LinkInstances(!^I_1, !^I_2)$, there is a significant reduction in execution time comparing decentralized and central coordination processes. This supports the previous assumptions that better utilization of parallelism is responsible for the overall performance increase. Whenever a new process instance is linked to a relational process structure, coordination pro-

Table 5 Statistical data for the last run of Scenario #1 (Recruitment)

		$InstantiateProcess(\omega^T)$	$ChangeAttributeValue(\omega^I, \phi^T, v)$	$CommitTransition(\omega^I, \tau^T)$	$LinkInstances(\omega_1^I, \omega_2^I)$
Total Time	0CP	00:378	00:323	00:223	00:738
	2CP	00:591	00:351	00:313	01:120
	1CP	00:495	00:358	00:271	01:423
Average	0CP	00:012	00:003	00:003	00:024
	2CP	00:018	00:003	00:004	00:036
	1CP	00:014	00:003	00:004	00:046
Median	0CP	00:011	00:002	00:003	00:023
	2CP	00:013	00:002	00:004	00:038
	1CP	00:013	00:003	00:003	00:046
Minimum	0CP	00:010	00:001	00:002	00:021
	2CP	00:011	00:001	00:001	00:020
	1CP	00:010	00:001	00:002	00:019
Maximum	0CP	00:021	00:009	00:007	00:030
	2CP	00:047	00:027	00:014	00:046
	1CP	00:026	00:036	00:010	00:074

cesses must be notified of the change. Then, the coordination process must perform an update by reevaluating affected coordination constraints. This can be done in parallel for two coordination processes, and as each coordination process comprises less constraints compared to a central coordination process, the update can be performed faster.

The speedup of $LinkInstances(!_1^I, !_2^I)$ is, however, counteracted by $InstantiateProcess(\omega^T)$. Decentralized coordination requires the instantiation of multiple coordination processes, which results in increasing instantiation times for coordinating process types. The function $ChangeAttributeValue(\omega^I, \phi^T, v)$ is not affected much by having multiple coordination processes; the values for both variants are roughly the same. The function $CommitTransition(!^I, \phi^T)$, however, is slower with decentralized process coordination than with central process coordination, though only by $40ms$. This can be explained with the coordination constraints that are shared between decentralized coordination processes, e.g., coordination step *Application*:*Sent* is present in both decentralized coordination processes. As such, whenever a change affects state *Sent* of an *Application* instance, a slight overhead occurs from having to communicate with two coordination processes.

The absolute differences between no coordination, central coordination, and decentralized coordination are largely unimportant. These values are highly dependent on the business process, as well as the specific execution sequence. However, as the execution sequence constitutes a worst case regarding parallel execution, the performance measurements allow drawing the conclusion that decentralized coordination is generally faster than central coordination, i.e., a qualitative ranking regarding performance can be made. Furthermore, Table 5 indicates where these performance advantages are realized.

In essence, the performance advantage of decentralized process coordination is due to the inherent parallelism of the PHILharmonicFlows engine. The engine is capable to distribute workloads across different coordination processes in variant with decentralized coordination, resulting in an overall speedup compared to central coordination. Note that this occurs even when the execution sequence is a worst case regarding parallelism. Therefore, allowing more parallel execution of processes generally shifts the advantage further toward decentralized coordination.

Limitations

However, note that the experiments only show some cases that should give a reasonable estimate of the performance of decentral vs. central process coordination. Due to the amount of possible combinations of models, execution sequences, and the number of ways to organize coordination processes, it is impossible to guarantee favorable performance for decentralized coordination in every case. In consequence, the experiment has a limited generalizability. The experiment uses a single representation of decentralized coordination with only two coordination processes. Other setups with more or different coordination processes might yield dif-

ferent results. Generally, however, as various factors affect performance differently, it is certainly possible to create or encounter exceptions where decentralized process coordination performs poorly or even worse than central coordination. For the given execution sequence representing a fairly standard case, decentralized process coordination performs adequately.

Due to the various possibilities decentralization in this experiment can be organized, the quantitative, relative differences between central and decentralized coordination have limits regarding expressiveness. However, the experiment represents the worst case regarding parallelism. On a qualitative level, decentralization performs almost always better due to the better usage of parallelism.

Summary

Performance benefits of decentralized process coordination *on a single node* could be demonstrated. At the very least, the results show that decentralized process coordination does not perform worse than central process coordination for the given benchmarks. While the significance of the quantitative differences is limited, qualitatively a clear advantage is present. This is due to increased parallelism of decentralized coordination, compared to central coordination in the PHILharmonicFlows Engine. Moreover, the following experiment will demonstrate the performance of decentralized process coordination in a distributed environment, where processes are located on multiple nodes.

6.3 Decentralized coordination process performance on multiple nodes

In order to prove that coordination processes have performance advantages if they are decentralized *and distributed*, an experiment with a distributed cluster was set up.

Experiment goal

For this experiment, the goal is to show the performance advantage of decentralized coordination by reducing extra-node communication in a distributed environment.

Experiment setup

In the basic setup, processes are distributed across multiple nodes on a computing cluster. In the central variant, one coordination process is located on one node, coordinating every process on every node. This requires significant extra-node communication. In the decentralized variant, each node is coordinated by its own coordination process located on the same node, which should require less extra-node communication, resulting in a better overall performance.

Concretely, two scenarios CM1CP and CM7CP are defined. Both scenarios constitute edge cases regarding decentralization and are based on the car manufacturing model (cf. Example 14). This example exemplifies a sufficiently large relational process structure for multiple configurations of virtual clusters and coordination processes. Scenario CM1CP presumes one coordination process with coordinating process type *Car* and a virtual cluster as depicted in Fig. 19.

- Scenario CM1CP represents the edge case of fully central process coordination.
- Scenario CM7CP retains the same virtual cluster (cf. Fig. 19), but is coordinated by seven coordination processes. Each coordination process is located in another node of the cluster, and the coordinating process type is highest in the hierarchy of the relational process structure, but still belongs to the respective node (cf. Fig. 19). Therefore, coordinating process types for Scenario CM7CP are *Car*, *Substructure*, *Interior*, *Engine Compartment*, *Body*, *Tire*, and *Engine*. Scenario CM7CP realizes fully decentralized coordination in a distributed cluster.

For the sake of comparison, the recruitment business process example (cf. Example 1) is evaluated as well. Here, distribution is defined as follows: *Job Offers* are located on node A, whereas *Applications*, *Reviews*, and *Interviews* are assigned to node B.

- Scenario R1CP has one coordination process, and
- Scenario R2CP has 2 coordination processes.

Both scenarios reuse the execution sequence defined in Scenario #1 (cf. Sect. 6.2). Table 6 gives an overview of the amount of intra- and extra-node communication of all scenarios in regard to their respective execution sequences.

Measurement setup

The goal is to determine the amount of communication occurring between processes and coordination processes and whether this is extra-node or intra-node communication. For this reason, PHILharmonicFlows logs each communication unit between processes and coordination processes. Note that a communication unit in this context is not equal to a single message exchange as, for example, known from BPMN 2.0 choreographies [34]. This is due to the fact that PHILharmonicFlows uses semantic relationships, which are defined on a higher level of abstraction and, therefore, may comprise multiple message exchanges. Instead, as objects with lifecycle processes and coordination processes are implemented as

Table 6 Intra- and extra-node communication for different scenarios

Id	Model	Number CP	Intra-node	Extra-node
CM1CP	Car Manufacturing	1	27	765
CM7CP	Car Manufacturing	7	799	81
R1CP	Recruitment	1	70	810
R2CP	Recruitment	2	705	175

actors, individual actor method invocations are logged. Communication between actors occurs by one actor invoking an *actor method* on another actor. Accordingly, actor method invocations are the communication unit used for these benchmarks.

For each communication unit, i.e., actor method call, the source and target are logged, which are identified by their ID. Source and target are either a process instance or a coordination process instance. Moreover, the name of the actor method is logged as well. For both source and target, the corresponding object types are determined and added to the log.

The aforementioned log setup describes a qualitative benchmark. For the purposes of this benchmark, qualitative logs are sufficient to arrive at dependable results. From the logs, it is possible to calculate intra- and extra-node communication by superimposing a virtual cluster with multiple nodes on the actual single-node cluster. In other words, each process and coordination process running on the single-node cluster is assigned a node where it virtually resides. This simulates a multi-node cluster, with the drawback that actual time measurements are meaningless, as it is actually still the same node of the cluster. However, as source and target of the message exchange are logged, it can be determined whether the resulting communication was intra-node or extra-node on the virtual cluster. For intra-node and extra-node communication, different virtual costs may be assigned. In consequence, the measurement results are no longer quantitative, but only allow for a qualitative assessment of the benefits.

The virtual cost assignment to extra- and intra-node communication is done under the assumption that external communication takes longer. One of the advantages of this qualitative approach is that costs can be varied, i.e., large and small differences between intra-node and extra-node communication can easily be realized. This allows quickly simulating different settings that may closely resemble realistic settings. Moreover, as the multi-node cluster is also virtual, its layout can be changed easily. Changing the layout allows assessing a wide variety of cluster layouts and coordination process setups for their costs associated with extra-node communication.

Table 7 Performance estimates based on different cost ratios for the car manufacturing model

Ratio	Total cost CM7CP	Total cost CM1CP	Difference
1.5	920	1174	+27.6%
2	961	1557	+62.0%
3	1042	2322	+122.8%
5	1204	3852	+219.9%
10	1609	7677	+377.1%

Results, observations, and interpretation

As can be seen in Table 6, the scenarios significantly differ in the amount of intra-node and extra-node communication. This can be entirely expected, as the respective scenarios are on opposite sides of the decentralization spectrum. However, it is yet unclear how much this difference might affect the performance of the overall system. In other words, how much does extra-node communication cost in terms of performance.

Though we cannot measure the impact directly, instead several estimates based on different, fictitious performance numbers can be given. For this purpose, intra-node communication is assigned a fixed performance cost of 1. For extra-node communication, a ratio is defined by how much slower extra-node communication is compared to intra-node communication, i.e., the cost ratio in terms of performance. The total cost in terms of performance for each scenario is calculated using the formula

$$\$totalCost = \lfloor \$intra{-}node + \$extra{-}node * ratio \rfloor$$

where $\$intra{-}node$ and $\$extra{-}node$ correspond to the values shown in Table 6. The total cost is rounded down to the nearest Integer value. Table 7 reports on performance cost estimates based on different ratios between intra- and extra-node communication.

Obviously, the centrally coordinated variant *CM1CP* of the car manufacturing model displays a large increase in performance cost with increasing ratio, as it generates substantially more extra-node communication. At the same time, the decentralized variant shows only a moderate increase with increasing ratios. According to these numbers, the decentralization and distribution of coordination process across nodes becomes more beneficial with increasing cost for extra-node

communication. However, for all these different ratios, it is unknown which of them represents a realistic value for the cost ratio. In the following, it is assessed which ratio is more likely to be encountered in the real world.

Estimating realistic cost ratios

Concerning intra-node communication, real values from communication within datacenters can be taken as a basis. Various websites[4] give the round trip time within a datacenter as 0.5 ms. For the sake of caution, a quadrupling of this value results in 2 ms for average intra-node communication. Datacenters are highly optimized for network latency and are likely not the primary location where PHILharmonicFlows might be used. In general, the more conservative estimate of 2 ms should be more accurate. Note that a communication unit also comprises a round trip, as the actor method invocation brings back a result. Therefore, intra-node communication and extra-node communication in PHILharmonicFlows are comparable to datacenter communication.

As for extra-node communication, a reasonable estimate is highly dependent on the distance between each node. For the car manufacturing example, nodes represent physical construction facilities, which are likely located in different countries or, more precisely, cities in different countries. An estimate of 500 km between nodes is likely to be on the low end of the possible spectrum of distances between cities in different countries. For obtaining a real-life time value for a message round trip for a 500 km distance, pings between major cities are a good source. The website wondernetwork.com[5] maintains regular pings between their servers in various locations all over the world. 500 km roughly corresponds to the distance between Paris, France and Frankfurt am Main, Germany (477.79 km), and the website reports an average ping of 10 ms. Note that the ping utility[6] measures round trip time, making these values comparable.

Judging from these values, a ratio of 5 ($= 10\ \text{ms}/2\ \text{ms}$) is likely to be a realistic value in regard to the underlying distance. This is a very cautious estimate, and real-life ratios may be higher. According to Table 7, with a ratio of 5, car manufacturing with central coordination takes roughly triple the time of the car manufacturing using decentralized coordination. In consequence, the results show a clear incentive for using decentralized process coordination in distributed environments. There is a clear performance benefit for decentralized process coordination on multiple nodes.

[4] https://gist.github.com/jboner/2841832, https://people.eecs.berkeley.edu/~rcs/research/interac-tive_latency.html.

[5] https://wondernetwork.com/pings.

[6] https://en.wikipedia.org/wiki/Ping_(networking_utility).

Limitations

For determining the benefits of multiple coordination processes, the experiment only describes a qualitative analysis of communication patterns. Instead of logging the time taken to perform a communication unit, it is logged how many individual units of communication have occurred in total, without logging execution time. A quantitative analysis is not possible as the necessary infrastructure to perform such a benchmark is not at our disposal. Furthermore, even if the infrastructure was available, it remains extremely difficult to *reliably* measure accurate execution times of message exchanges in a multi-node setup. Therefore, this experiment cannot give conclusive quantitative results.

The qualitative analysis further relies on some data values that cannot be obtained by direct measurement in the PHILharmonicFlows engine, e.g., the time for intra-node and extra-node communication. The logical argument undertaken to arrive at concrete data values was made conservatively, with the intent of erring on the side of caution. Additionally, the initial data obtained for the logical argument might be unreliable or unrepresentative, e.g., data center ping times. Other sources for the same data might report different values. The results that have been obtained in this experiment therefore have a large margin of error, but most likely in favor of the approach.

Finally, the underlying assumption that extra-node communication is slower than intra-node communication might prove to be false, however, this can be deemed rather unlikely in general. The comparison distance of 500 km for extra-node communication is arbitrary and the results vary if the distance is lowered or increased.

Summary

In summary, the performance measurements emphasize the benefit of decentralized process coordination when using multiple coordination process. Especially in distributed environments, the use of multiple coordination processes provides clear advantages in performance. This performance advantage is complemented by the functional benefits which include smaller models, clear coordination responsibility, and better maintainability.

7 Related work

Most approaches to business process management that rely on interacting processes are closely related to data-centric process management paradigms [42]. As they primarily rely on process interactions, they may therefore also be classified as interaction-centric approaches in addition to being data-centric.

Proclets are small, lightweight processes that focus on interactions between processes [44,45]. As one of the first approaches, proclets abandoned monolithic process models in favor of multiple, interacting processes. Proclets are defined using the well-known formalism of Petri nets. The approach also recognized that instances of proclets may need to communicate with more than one other proclet. Therefore, the proclet approach supports one-to-many interactions between proclets. For this purpose, the used Petri net formalism is extended with ports, enabling communication with other proclets. Ports are fully integrated into the Petri net formalism, supporting the formal analysis techniques known from standard Petri nets. The communication between proclets goes over channels that connect to ports on other proclets.

The actual communication between proclets over a channel is realized by performatives, a special form of message. A major advantage of the proclet approach is the support for the full range of formal analysis techniques enabled by Petri nets. Proclets have the potential to form large interconnected structures of different proclet types. However, interactions between proclets are coordinated by the individual proclets themselves, using performatives. As Proclets coordinate with other proclets individually in a one-to-many fashion, no concept for separate coordinators exists. Furthermore, no specific research regarding decentralization of individual proclets or coordinators is known.

Artifact-centric process management [33] describes business processes as interacting *artifacts*. The behavior of artifacts is expressed in lifecycles. Central to this approach is the *artifact*, which holds all process-relevant information in an *information model*. An artifact lifecycle is specified using the Guard-Stage-Milestone (GSM) meta-model [22,23]. In general, an artifact may interact with other artifacts. However, GSM does not provide dedicated coordination mechanisms or explicit artifact relations, contrary to the object-aware approach presented in this paper. Instead, GSM incorporates an arbitrary information model as well as an expression framework with which artifact interactions may be specified. While this, in theory, allows expressing any concept or constraint, in practice many of the capabilities of artifact-centric process management hinge on the power of the expression framework. As a drawback, expressions might become very complex and must be supported by a rule engine to realize the full potential of artifact-centric process management. In principle, the concepts of the relational process structure and the semantic relationships may be recreated in GSM with complicated expressions to realize at least the basic functionality of coordination processes. While this is not impossible, it requires great effort on part of the modeler to achieve the same functionality as object-aware process management provides out-of-the-box.

Artifact-centric process management has been prototypically implemented in the BizArtifact demo tool[7], whose predecessors include Barcelona [20] and Siena [12]. Due to the complexity of an artifact-centric business process, model verification [6,8] constitutes an important aspect of artifact-centric process management. Moreover, several variants of artifact-centric process management exist in regard to coordination.

Artifact-centric hubs [21] constitute one of the first ideas to allow collaboration using artifacts. However, the interactions take place between process participants, not among the artifacts themselves. The basic idea is that participants use artifacts to interact with each other, where an artifact is similar to a bulletin board. [27] reused these ideas that lead to the creation of artifact-centric hubs, but instead used these ideas for introducing an approach enabling artifact choreographies. Process participants, called agents, use artifacts and execute them. Artifacts are assigned to a specific location. By knowing where artifacts are located and who is using them, a choreography between these agents can be automatically generated. While both [21] and [27] provide approaches for managing interactions, the interacting parties are not the artifacts themselves. Instead, choreographies between participants are created, a stepping stone to artifact-based cross-organizational business process. Consequently, both approaches are not directly comparable to coordination processes. Moreover, using artifacts in a large-scale setting has not been investigated by artifact-centric process management. As the artifact-centric hubs are supposed to be centralized, no decentralization or distribution concepts have been investigated.

In contrast, [43] presents an approach for providing declarative choreographies for artifact-centric processes where artifacts and not participants are the interacting entities. The artifacts in this approach use a type-instance schema as well. Declarative choreographies recognize the need for explicitly knowing the relations between artifacts and their multiplicity. Consequently, one-to-many relationships and many-to-many relationships are supported by a concept called a correlation graph. The artifact instances are coordinated using messages, which are exchanged based on the constraints of the declarative choreography. The constraints are specified by using expressions, where the expressions require greater expressiveness than the expressions used for semantic relationships. In turn, this makes expressions for artifacts more complicated in comparison. Similar to [21], decentralized or distributed coordination of artifacts has not been a specific concern, though decentralization and distribution are of interest for the approach in principle.

[18] investigates many-to-many interactions between processes. The need for supporting many-to-many relationships

[7] https://sourceforge.net/projects/bizartifact/.

when dealing with interactions between artifacts is recognized. Artifact lifecycles are specified by using Petri nets, specifically proclets, instead of GSM, with the intention of using the formal properties of Petri nets to verify an entire artifact-centric business process [17]. Many-to-many interactions between processes are fully incorporated into the Petri net descriptions of these processes [44,45]. The interactions between different Petri net-based processes are expressed in terms of correlation and cardinality constraints, and full operational semantics are provided. This form of description is accessible for formal reasoning and verification. As opposed to coordination processes in object-aware process management, which aim at high-level abstractions for different concepts by using specialized notations, [17,18] aim at notational simplicity by restricting themselves to few syntactical concepts, i.e., Petri nets only. Similar to Proclets, dedicated coordinators do not exist.

The coordination of large process structures with a focus on the engineering domain is considered in [30,31]. The COREPRO approach explicitly considers process relations with one-to-many process relations and dynamic changes at run-time, but transitive relations are not covered. In comparison with COREPRO, semantic relationships are similar to external state transitions of a lifecycle coordination model. However, the external state transitions do not take the semantics of the respective process interaction into account. While COREPRO considers large-scale application of its specific modeling concepts, no specific investigations into decentralized coordination have taken place.

In principle, artifacts and proclets can be used in a distributed environment. Conceptually, no coordinator, similar to the coordination process concept, is present in these approaches. Consequently, no research toward decentralized process coordination may be found in the literature. Furthermore, distribution of proclets or artifacts has not been investigated as well.

As PHILharmonicFlows comprises an execution engine capable of supporting decentralized processes and coordination processes, it is necessary to assess other process engines. [36] shows that most decentralization efforts in BPM are achieved based on process engines, but not individual processes or coordinators. In this regard, PHILharmonicFlows and its hyperscale process engine are far ahead. Other types of decentralization take place on the task level, where individual tasks of a single process instance are executed on different nodes.

[15,16] and [10] distribute workloads of business processes between a client-side engine and a cloud-based engine, taking into consideration that users might not want to store their business data in the cloud. The approaches suggest to primarily run compute-intensive workloads on the cloud-based engine and transfer business data only when necessary. [15] further presents a method for decomposing the process model into two complementary process models: one for the client engine and one for the cloud engine.

[24] deals with very large workflow engines and presents concepts to provide high availability of the engine. The counteract failing workflows, a backup strategy is presented. The backup allows resuming a particular process instance at any point in their execution.

[7] presents concepts for the executing processes in a distributed environment. The paper presumes a cluster with different nodes, and each node has a workflow engine server. Workflows may be transferred between nodes to achieve optimal performance, though individual servers may become overloaded. The approach replicates workflow engine server within nodes of the cluster to achieve more performance by evenly distributing computing load within the node.

8 Summary and outlook

With coordination processes, the conceptual, technical and methodological capabilities exist to successfully implement decentralized process coordination for large process structures. The concepts of scope, hierarchy of the relational process structure, and the principle of subsidiarity decrease complexity and, thus, make the entire approach feasible. On the benefits side, large-scale coordination of large process structures becomes feasible, while at the same time the complexity and size of individual coordination process models are reduced compared to a central coordinator. As has been shown, this also applies to distributed relational process structures. However, in a different sense multiple coordination processes are more complex than a central coordinator. Again, subsidiarity and hierarchy are central to manage this complexity, enabling designers to model the coordination of large process structures. Furthermore, it has been shown that decentralization of processes and process coordination has performance advantages compared to central process coordination. Through state-based views, the concepts presented in this paper can, in principle, be transferred to other paradigms, given these paradigms can conform to the concept of a relational process structure.

While coordination processes can already deal with a vast number of coordination problems, there are still several areas left for improvement. One challenge concerns the monitoring of a business process which is constituted by interacting, interdependent processes. Coordination processes may be used to gain valuable insights into the overall progress of the business process, as coordination processes may be used to aggregate status information from the coordinated processes. This is especially challenging when decentralized and distributed coordination processes are involved, but offers promising perspectives as well. Decentralized process coordination, in principle, allows for the decentralized monitoring

of related process instances, enabling a more fine-grained view of the progress.

Currently, PHILharmonicFlows offers some practically oriented verification for single coordination processes. Expanding this verification to include decentralized coordination processes is immensely beneficial to a designer. Regarding theoretical concepts such as controllability and realizability, significant contributions have been made for choreographies of activity-centric processes [13,19,25,27,28] and artifact-centric processes [5,6,8,14]. As coordination processes are based on semantic relationships, it is unclear how these results translate to coordination processes and in particular to decentralized coordination processes. A thorough investigation into the applicability of these results to coordination processes is the subject of future work.

Funding Open Access funding enabled and organized by Projekt DEAL.

References

1. Agha, G., Hewitt, C.: Concurrent programming using actors: exploiting large-scale parallelism. In: Readings in Distributed Artificial Intelligence, Morgan Kaufmann, pp. 398–407 (1988)
2. Andrews, K., Steinau, S., Reichert, M.: Engineering a highly scalable object-aware process management engine using distributed microservices. In: 26th Int'l Conference on Cooperative Information Systems (CoopIS), Springer, LNCS, Vol. 11229, pp. 80–97 (2018)
3. Andrews, K., Steinau, S., Reichert, M.: Enabling runtime flexibility in data-centric and data-driven process execution engines. Inf. Syst. (2019). https://doi.org/10.1016/j.is.2019.101447
4. Baeyens, T.: BPM in the Cloud. In: 11th Int'l Conference on Business Process Management (BPM), Springer, LNCS, Vol. 8094, pp. 10–16 (2013)
5. Bagheri Hariri, B., Calvanese, D., de Giacomo, G., de Masellis, R., Felli, P.: Foundations of relational artifacts verification. In: 9th Int'l Conference Business Process Management (BPM), Springer, LNCS, Vol. 6896, pp. 379–395 (2011)
6. Bagheri Hariri, B., Calvanese, D., Montali, M., Santoso, A., Solomakhin, D.: Verification of semantically-enhanced artifact systems. In: 11th Int'l Conference on Service-Oriented Computing (ICSOC), Springer, LNCS, Vol. 8274, pp. 600–607 (2013)
7. Bauer, T., Reichert, M., Dadam, P.: Intra-subnet load balancing in distributed workflow management systems. Int. J. Cooper. Inf. Syst. **12**(3), 295–323 (2003)
8. Belardinelli, F., Lomuscio, A., Patrizi, F.: Verification of GSM-based artifact-centric systems through finite abstraction. In: 10th Int'l Conference on Service-Oriented Computing (ICSOC), Springer, LNCS, Vol. 7636, pp. 17–31 (2012)
9. Bonini, M., Urru, A., Steinau, S., Ceylan, S., Lutz, M., Schuhmacher, J., Andrews, K., Halfar, H., Kunaschk, S., Haque, A., Nair, V., Rollenhagen, M., Shaik, N., Reichert, M., Bartneck, N., Schlegel, C., Hummel, V., Echelmeyer, W.: Automation of intralogistic processes through flexibilisation: a method for the flexible configuration and evaluation of systems of systems. In: 15th Int'l Conference on Informatics in Control, Automation and Robotics—Vol. 1: ICINCO, SciTePress, pp. 380–388 (2018)
10. Ceri, S., Grefen, P., Sanchez, G.: WIDE-a distributed architecture for workflow management. In: 7th Int'l Workshop on Research Issues in Data Engineering. High Performance Database Management for Large-Scale Applications, pp. 76–79 (1997)
11. Chiao, C.M., Künzle, V., Reichert, M.: Object-aware process support in healthcare information systems: requirements, conceptual framework and examples. Int. J. Adv. Life Sci. **5**(1 & 2), 11–26 (2013)
12. Cohn, D., Dhoolia, P., Heath, FT., Pinel, F., Vergo, J.: Siena: from powerpoint to web app in 5 minutes. In: 6th Int'l Conference on Service-Oriented Computing (ICSOC), Springer, LNCS, Vol. 5364, pp. 722–723 (2008)
13. Decker, G., Weske, M.: Interaction-centric modeling of process choreographies. Inf. Syst. **36**(2), 292–312 (2011)
14. Deutsch, A., Li, Y., Vianu, V.: Verification of Hierarchical Artifact Systems. ArXiv e-prints (2016)
15. Duipmans, E.F., Pires, L.F., Santos, Luiz O Bonino da Silva.: Towards a BPM cloud architecture with data and activity distribution. In: IEEE 16th International Enterprise Distributed Object Computing Conference Workshops, pp. 165–171 (2012)
16. Duipmans, E.F., Ferreira Pires, L., Bonino da Silva Santos, L.O.: A transformation-based approach to business process management in the cloud. J. Grid Comput. **12**(2), 191–219 (2014)
17. Fahland, D.: Describing Behavior of Processes with Many-to-Many Interactions. Application and Theory of Petri Nets and Concurrency, Springer, LNCS, Vol. 11522, pp. 3–24 (2019)
18. Fahland, D., de Leoni, M., van Dongen, BF., van der Aalst, W.M.P.: Many-to-many: some observations on interactions in artifact choreographies. In: 3rd Central-European Workshop on Services and their Composition (ZEUS), CEUR-WS.org, CEUR Workshop Proceedings, pp. 9–15 (2011)
19. Güdemann, M., Poizat, P., Salaün. G., Dumont, A.: VerChor: a framework for verifying choreographies. In: Fundamental Approaches to Software Engineering, Springer, LNCS, Vol. 7793, pp. 226–230 (2013)
20. Heath, F.T., Boaz, D., Gupta, M., Vaculín, R., Sun, Y., Hull, R., Limonad, L.: Barcelona: a design and runtime environment for declarative artifact-centric BPM. In: 11th Int'l Conference on Service-Oriented Computing (ICSOC), Springer, LNCS, vol 8274, pp. 705–709 (2013)
21. Hull, R., Narendra, N.C., Nigam, A.: Facilitating workflow interoperation using artifact-centric hubs. In: 7th Int'l Conference on Service-Oriented Computing (ICSOC), Springer, LNCS, Vol. 5900, pp. 1–18 (2009)
22. Hull, R., Damaggio, E., de Masellis, R., Fournier, F., Gupta, M., Heath, FT., Hobson, S., Linehan, M., Maradugu, S., Nigam, A., Sukaviriya, PN., Vaculín, R.: Business artifacts with guard-stage-milestone lifecycles: managing artifact interactions with conditions and events. In: 5th ACM Int'l Conference on Distributed Event-based System (DEBS), ACM, pp. 51–62 (2011)
23. Hull, R., Damaggio, E., Fournier, F., Gupta, M., Heath, FT., Hobson, S., Linehan, M., Maradugu, S., Nigam, A., Sukaviriya. PN., Vaculín, R.: Introducing the guard-stage-milestone approach for specifying business entity lifecycles. In: 7th Int'l Workshop on Web

Services and Formal Methods (WS-FM) 2010, Springer, LNCS, Vol. 6531, pp. 1–24 (2011)
24. Kamath, M., Alonso, G., Günthör, R., Mohan, C.: Providing high availability in very large workflow management systems. Advances in Database Technology. Springer, LNCS, Vol. 1057, pp. 425–442 (1996)
25. Knuplesch, D., Reichert, M., Pryss, R., Fdhila, W., Rinderle-Ma, S.: Ensuring compliance of distributed and collaborative workflows. In: 9th IEEE Int'l Conference on Collaborative Computing: Networking, Applications and Worksharing, pp. 133–142 (2013)
26. Künzle, V., Weber, B., Reichert, M.: Object-aware business processes: fundamental requirements and their support in existing approaches. Int. J. Inf. Syst. Model. Des. **2**(2), 19–46 (2011)
27. Lohmann, N., Wolf, K.: Artifact-centric choreographies. In: 8th Int'l Conference on Service-Oriented Computing (ICSOC), Springer, LNCS, Vol. 6470, pp. 32–46 (2010)
28. Lohmann, N., Wolf, K.: Decidability results for choreography realization. In: 9th Int'l Conference on Service-Oriented Computing (ICSOC), Springer, LNCS, Vol. 7084, pp. 92–107 (2011)
29. Müller, D., Herbst, J., Hammori, M., Reichert, M.: IT Support for release management processes in the automotive industry. In: 4th Int'l Conference on Business Process Management (BPM), Springer, LNCS, Vol. 4102, pp. 368–377 (2006)
30. Müller, D., Reichert, M., Herbst, J.: Data-driven Modeling and Coordination of Large Process Structures. In: 15th Int'l Conference on Cooperative Information Systems (CoopIS). Springer, LNCS, Vol. 4803, pp. 131–149 (2007)
31. Müller, D., Reichert, M., Herbst, J.: A new paradigm for the enactment and dynamic adaptation of data-driven process structures. In: 20th Int'l Conference on Advanced Information Systems Engineering (CAiSE), Springer, LNCS, Vol. 5074, pp. 48–63 (2008)
32. Nandi, P., Kumaran, S.: Adaptive business objects—a new component model for business integration. In: 7th Int'l Conference on Enterprise Information Systems (ICEIS), pp. 179–188 (2005)
33. Nigam, A., Caswell, N.S.: Business artifacts: an approach to operational specification. IBM Syst. J. **42**(3), 428–445 (2003)
34. Object Management Group: Business Process Model and Notation (BPMN). Version **2**, (2011)
35. Rahman, M., Ranjan, R., Buyya, R.: Cooperative and decentralized workflow scheduling in global grids. Future Gener. Comput. Syst. **26**(5), 753–768 (2010)
36. Schulte, S., Janiesch, C., Venugopal, S., Weber, I., Hoenisch, P.: Elastic business process management: state of the art and open challenges for BPM in the cloud. Future Gener. Comput. Syst. **46**, 36–50 (2015)
37. Steinau, S., Künzle, V., Andrews, K., Reichert, M.: Coordinating business processes using semantic relationships. In: 19th IEEE Conference on Business Informatics (CBI), IEEE Computer Society Press, pp. 33–43 (2017)
38. Steinau, S., Andrews, K., Reichert, M.: Modeling process interactions with coordination processes. In: 26th Int'l Conference on Cooperative Information Systems (CoopIS). Springer, LNCS, Vol. 11229, pp. 21–39 (2018)
39. Steinau, S., Andrews, K., Reichert, M.: The relational process structure. In: 30th Int'l Conference on Advanced Information Systems Engineering (CAiSE). Springer, LNCS, pp. 53–67 (2018)
40. Steinau, S., Andrews, K., Reichert, M.: Coordinating large distributed process structures. In: 20th Int'l Working Conference on Business Process Modeling, Development, and Support (BPMDS). Springer, LNBIP, Vol. 352, pp. 19–34 (2019)
41. Steinau, S., Andrews, K., Reichert, M.: Executing lifecycle processes in object-aware process management. Data-Driven Process Discovery and Analysis. Springer, LNBIP Vol. 340, pp. 25–44 (2019)
42. Steinau, S., Marrella, A., Andrews, K., Leotta, F., Mecella, M., Reichert, M.: DALEC: a framework for the systematic evaluation of data-centric approaches to process management software. Softw. Syst. Model. **18**(4), 2679–2716 (2019)
43. Sun, Y., Xu, W., Su, J.: Declarative choreographies for artifacts. In: 10th Int'l Conference on Service-Oriented Computing (ICSOC). Springer, LNCS, Vol. 7636, pp. 420–434 (2012)
44. van der Aalst, W.M.P., Barthelmess, P., Ellis, C.A., Wainer, J.: Workflow modeling using proclets. In: 7th Int'l Conference on Cooperative Information Systems (CoopIS). Springer, pp. 198–209 (2000)
45. van der Aalst, W.M.P., Barthelmess, P., Ellis, C.A., Wainer, J.: Proclets: a framework for lightweight interacting workflow processes. Int. J. Cooper. Inf. Syst. **10**(04), 443–481 (2001)

Sebastian Steinau is a Ph.D. student at the Institute of Databases and Information Systems at Ulm University. His research interests include data-centric process management systems and the coordination of business processes. He is part of a project in which PHILharmonicFlows, a new data-centric process management system, is being developed. An essential part of the overall PHILharmonicFlows concept is the coordination of the interactions of multiple objects and their lifecycles.

Kevin Andrews is a Ph.D. student at the Institute of Databases and Information Systems of Ulm University. He works on the PHILharmonicFlows project, with the goal of creating a data-centric business process management system. His research interests include data-centric process management systems, in particular applying the concepts of variability and ad hoc changes to data-centric processes.

Manfred Reichert is a full professor at Ulm University, where he is director of the Institute of Databases and Information Systems. His research interests include business process management, information systems, and mobile services. Manfred was PC Co-chair of the BPM'08, CoopIS'11 and EDOC'13 conferences. Furthermore, he served as General Chair of the BPM'09 and EDOC'14 conferences as well as the BPM'15 workshops. Recently, he has co-authored a Springer book on process flexibility and obtained the BPM Test of Time Award at the BPM 2013 conference.