# A feature-based taxonomy of coordination approaches

Tim Kräuter[1], Julien Deantoni[2] Adrian Rutle[1], Harald König[3,1], and Yngve Lamo[1]

[1] Western Norway University of Applied Sciences, Bergen, Norway
tkra@hvl.no, aru@hvl.no, yla@hvl.no
[2] University Cote d'Azur, Sophia Antipolis, France
julien.deantoni@univ-cotedazur.fr
[3] University of Applied Sciences, FHDW, Hanover, Germany
harald.koenig@fhdw.de

**Abstract.** Complex domains necessitate the utilization of multiple interacting software systems. Various categories of coordination approaches, such as coordination languages, co-simulation approaches, architecture description languages, and coordination frameworks, have been proposed to ensure seamless integration of these systems. In this paper, we present a comprehensive feature-based taxonomy of these categories, which have previously only been studied in isolation. We apply our taxonomy to 17 distinct approaches, uncovering standard and unique features across the categories and allowing us to provide overarching insights. Furthermore, we employ clustering analysis to explore the similarities among these approaches, revealing the closeness coordination categories.

**Keywords:** Co-simulation · Coordination language · ADL · Coordination framework · Taxonomy · Feature model

## 1 Introduction

Complex domains require multiple interacting software systems to meet their demands. Coordination approaches are necessary to ensure the effective integration of these systems. The study of coordination approaches appears to be fragmented, with disparate investigations conducted on different abstraction levels, pursuing varied goals, and utilizing diverse terminology, resulting in different categories of approaches, namely *coordination languages* ([47]), *Co-simulation approaches* ([22]), *architecture description languages* ([14]), and *coordination frameworks* ([29,57]). To the best of our knowledge, evaluations of coordination approaches have been limited to their respective communities. This isolation hinders a comprehensive understanding of coordinating software systems, leading to the development of the same features independently with limited reuse.

First, we introduce the different categories of coordination approaches to paint a picture of the different communities (section 2) and discuss our methodology in section 3. Our main contribution is a comprehensive taxonomy, represented as a *feature model* [27] to categorize coordination approaches within a unified framework ((section 4)). The taxonomy allows the comparison of coordination approaches across the different categories. Moreover, we apply the taxonomy to evaluate 17 distinct approaches spanning the aforementioned categories (section 5) and publish our findings in a public dataset [55]. Finally, we discuss our findings in section 6 before concluding in section 7, i.e., we identify the typical features for each category, provide general insights, and cluster the approaches regarding their similarity.

## 2 Categories of coordination approaches

Figure 1 gives an overview of the different categories of coordination approaches. The categories of coordination approaches broadly operate on different abstraction levels, namely the *execution*, *model*, and *language* levels. We will explain each abstraction level and the accompanying coordination approaches from the bottom up. A detailed explanation of each category is given in the following subsections. In general, the right abstraction level to address coordination depends on the specific use case and goals at hand, i.e., approaches on each abstraction level have their advantages and disadvantages.

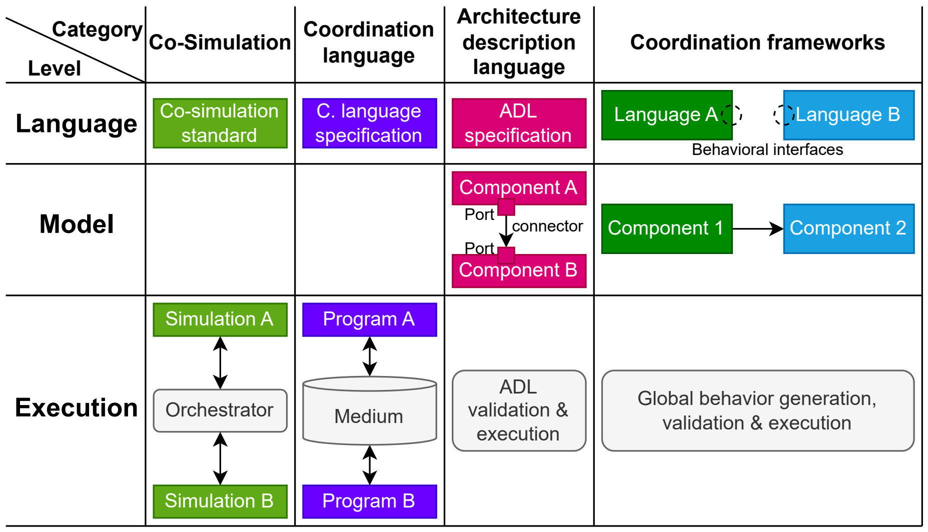

**Fig. 1.** Overview of coordination approaches

**Execution level:** Approaches on the *execution level* aim to deal with coordination by interfacing directly with executable binaries or source code. We identified two distinct categories on this level: co-simulation approaches and coordination languages. Such approaches are overviewed in subsection 2.1 and subsection 2.2.

**Model level:** *Architecture Description Languages (ADLs)* operate on the *model* level, i.e., each *component* is given by a behavioral model, for example, a process algebra. An overview of ADLs is given in subsection 2.3.

**Language level:** *Coordination frameworks* operate on the language level since they allow coordination of models conforming to *different* behavioral languages. They are outlined in subsection 2.4.

## 2.1 Co-simulation approaches

*Co-simulation approaches* involve coordinating timely information exchanges between multiple simulations running concurrently. In this configuration, a simulation is the execution of a simulation unit that consumes inputs and produces outputs. A simulation unit is typically specified as a *black box*, i.e., its behavior is opaque and accessible only through its simulation interface. Using such interfaces, the co-simulation must ensure the temporal and causal relationships between each simulation. This is usually achieved by defining an *orchestrator* [22], which dictates the progression of simulated time and transfers data between simulations. Co-simulation approaches are categorized as *Discrete Event* (DE), *Continuous Time* (CT), or *Hybrid* if DE and CT aspects are mixed [22].

Currently, there are two primary standards for co-simulation: the Functional Mock-up Interface (FMI) [41] and the High Level Architecture (HLA) [16]. The FMI standard is better suited for CT co-simulation, while the HLA standard is better suited for DE co-simulation [22,31].

To bridge CT and DE co-simulations, MECSYCO [10,11] uses a DE-based orchestrator but achieves hybrid co-simulation by wrapping CT simulations specified using FMI in the Discrete Event System specification (DEVS). Wrapping CT simulations using DE simulations or vice versa is a common strategy to achieve Hybrid co-simulation [22].

## 2.2 Coordination languages

Many different styles of coordination languages exist, which achieve coordination between systems differently. In this context, a system is an executable program of varying size. Still, we also use the term *system* to describe components in an ADL or behavioral models in coordination frameworks.

Coordination languages such as Linda [12] provide a *set of operations* to enrich a host programming language with coordination capabilities. The operations in Linda are used to read from and write to a virtual shared memory, which serves as a *global* communication medium. Such coordination languages are often called *tuple-based* because the shared memory contains sequences of elements, i.e., tuples. Tuple-based coordination languages such as Linda are surveyed in [51,42,44].

Other coordination languages such as Manifold [4,48] and REO [6] do not rely on a global communication medium but rather use *channels* to connect systems locally. Each system then uses I/O operations to interact with connected channels without knowing who has sent or will receive data through the channel. These coordination languages share many ideas with Architecture Description Languages (ADLs) discussed in subsection 2.3.

Furthermore, coordination languages might also use communication models such as the actor or reactor [32] model. For example, the Lingua Franca [32,33] coordination language is described using the reactor model, i.e., one defines coordination by defining how each system reacts to incoming events. A reaction to an event in one system might trigger reactions in connected systems. To encode reactions, the user can choose an existing programming language.

## 2.3 Architecture description languages

Architecture Description Languages (ADLs) aim to describe the structure of systems, allowing developers to focus on high-level system components and their connections rather than lines of source code [14,39,40]. Many different ADLs have been proposed in the academic literature and by the industry [39,58]. Nevertheless, clearly defining ADLs is challenging due to overlap with general-purpose modeling languages [14]. This distinction is deeply studied in [39].

The three buildings blocks of ADLs are defined as (1) *components*, (2) *connectors*, (3) *architectural configuration* [39,40]. A *component* is a unit of computation or data repository [39]. Components vary in size, ranging from representing individual services to entire systems. In this paper, we use the more general term *system* when we are not in the immediate context of ADLs.

*Connectors* serve as architectural elements to model interactions between components and the regulations that oversee those interactions [39]. A difference to components is that connectors must not be implemented as distinct entities such as message brokers but can also represent shared variables or links between applications realized by client-server protocols [39].

*Architectural configuration*, also known as topology, represents the structural arrangement of components and connectors in a connected graph, defining the overall architecture [39]. This structure determines if the combined semantics satisfy the desired behavior. Verification relies heavily on specifying components and connectors. For instance, one common application ensures the architectural configuration is free from deadlock and starvation.

The first ADLs were developed in the 90s and use process algebras such as Communicating Sequential Processes (CSP), Calculus of Communicating Systems (CCS), and $\pi$-calculus [45]. For example, Wright uses CSP [2] while Darwin uses $\pi$-calculus [34]. More recent ADLs, such as MontiArc [24], use automata to define the behavior of components.

However, to the best of our knowledge, no ADL supports heterogeneous components such as the coordination frameworks discussed in the next section [39]. Furthermore, despite the creation of numerous ADLs in the literature, they are not mainstream, i.e., not often used by practitioners in the industry [14,58,46,45,38].

### 2.4 Coordination frameworks

We introduce the term *coordination framework* to describe the systematic establishment of coordination at the language level through the utilization of patterns, followed by an automatic derivation of the coordination on models that adhere to the languages. (see Figure 1). Coordination frameworks primarily concentrate on coordinating models that adhere to *different* behavioral languages. They employ the concept of *language behavioral interfaces* to define the *synchronization points* that can be used during the coordination of models. This is valuable because each part of the system might be defined using different behavioral languages best suited to its nature. Additionally, reifying coordination allows for 1) the inspection, sharing, and review of proposed coordination patterns and 2) automatically providing the coordination for each model conforming to that language.

For example, in [29], the authors define a standard language behavioral interface that identifies state-changing elements used across different behavioral languages. In this paper, we will refer to our approach outlined in [29] as the *Behavioral Coordination Language* (BCoorLang) to enhance clarity in communication. Using this uniform interface in BCoorLang, one can define coordination between systems using heterogeneous behavioral models. Concretely, one can define that a state change in one system should be synchronized with a state change in a different system by identifying the corresponding state-changing elements, i.e., synchronization points in each model. The semantics of BCoorLang are given by rewriting system specifications, such as graph- or term-rewriting, which can then be executed by corresponding tools such as Groove [50] and Maude [36] for simulation and verification purposes.

## 3 Methodology

We created an initial version of our taxonomy and then continuously revised it by consulting the literature in two steps until it reached a stable state, which is described in the next section.

First, we conducted a *meta-analysis* (tertiary study) on surveys (secondary studies) within each coordination category. We found the following secondary studies for coordination languages ([47,23,51]), ADLs ([14,39,26,45,35]), and co-simulation approaches ([22,54,25]). We have not found surveys on coordination frameworks since we introduced the term in this publication. We used Google Scholar to locate these papers, utilizing the keywords “survey”, “feature model”, “taxonomy”, and for each category, “coordination language”, “architecture description language”, or “co-simulation” accordingly. We then collected the first five pages of papers and filtered them by title, abstract, and content to arrive at the previously listed papers. Our exact search queries are given in [55].

Second, we investigated individual approaches in each category to construct our taxonomy. Due to the immense number of approaches and limited time, we focus on the 17 approaches listed in section 5. Our selection of these approaches is based on a mix of different criteria: distribution across categories, expert knowledge of the authors regarding relevance, and heterogeneity inside categories. For example, we chose heterogeneous coordination languages, namely Linda, Reo, and Lingua Franca, which employ different coordination models, namely tuple spaces, channels, and reactors.

## 4 Taxonomy

We represent our taxonomy as a *feature model* [27]. We aim to comprehensively understand the coordination landscape and compare coordination approaches from different categories. To achieve this goal, we first identify the coordination *foundation*: coordination mechanism, syntax, and semantics. Second, we investigate the coordination approaches *goal* such as simulation, execution, and formal verification. Third, we study the approaches' *properties*, such as system transparency, domain, execution, system languages, and specification.

The top level of the feature model is shown in Figure 2. A coordination approach has the three previously discussed features: *Foundation*, *Goal*, and *Properties*, which we depict and describe in detail in the following sections.

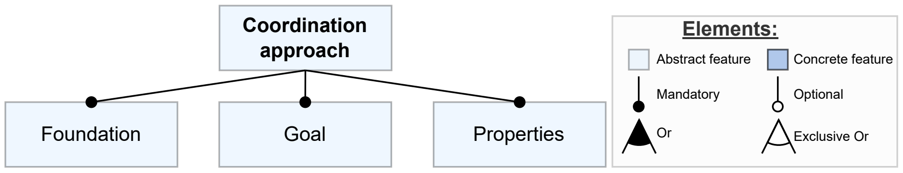

**Fig. 2.** Feature model overview

### 4.1 Foundation

Figure 3 shows the **Foundation** feature, which consists of **Syntax**, **Semantics**, and **Mechanism**.

**Syntax** The coordination syntax of the elements on which to apply coordination can either be *explicit* or *implicit*. An implicit syntax is usually based on the names of elements. For example, one could synchronize the firing of transitions from multiple state machines if they have identical names.

Explicit syntax is either *query based* or *structural*. For example, in BCOoL, one can write queries to define which elements should be coordinated [56,57].

A structural definition of coordination uses *dedicated elements* or is *architecture based*. For example, interactions in BCoorLang are specified by picking dedicated elements that should be coordinated, while coordination in ADLs depends on the architectural configuration [39], i.e., how ports of systems are connected.

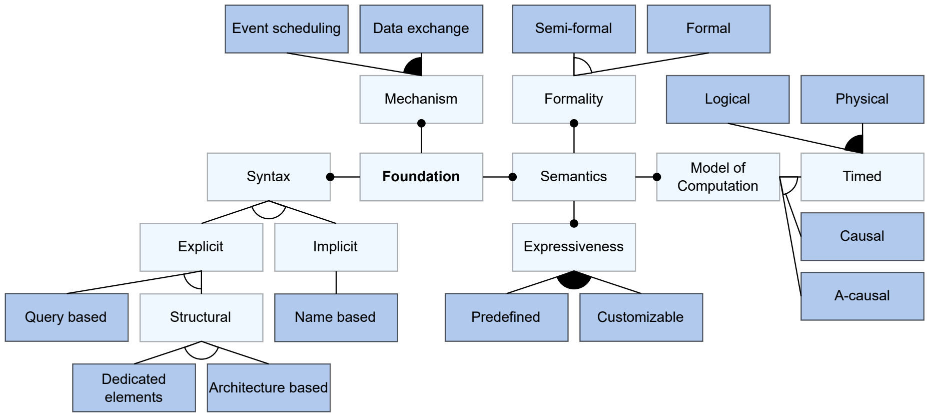

**Fig. 3.** Foundation feature

**Mechanism** The main mechanism driving coordination can be *event scheduling*, *data exchange*, or a mix of both. In the literature, these mechanisms are sometimes called control/event-driven or data-driven, respectively [47,56].

First, *event scheduling* synchronizes events or imposes a specific event order, i.e., order of state changes in different systems. Usually, event scheduling is used when systems are given by behavioral models since there might not be a clear concept of an event in a programming language. For example, in BCOoL [57], one can define relationships between events, such as *happens before* or *synchronize*.

*Data-exchange* means systems communicate by exchanging data. For example, in the ADL MontiArc [24], systems send data from the output to the input port. Data can also be exposed as variables, which can be coupled with variables in other systems, for example, to be equal. However, data exchange might also happen between systems during the synchronization of events, which is why both coordination mechanisms can be supported simultaneously.

**Semantics** The semantics feature consists of *expressiveness*, *model of computation*, and *formality*. The expressiveness of coordination can be *predefined*, i.e., a user has a fixed set of coordination mechanisms, for example, a fixed set of connectors with predefined behavior. Expressiveness is *customizable* if one can define the semantics of coordination, typically by using a dedicated language. For example, in BCOoL, one can employ the Clock Constraints Specification Language (CCSL) [3] to define new coordination operators besides the predefined ones [56,57].

Coordination approaches utilize different *models of computation* (MoC). A MoC is a set of rules that governs the timed, possibly concurrent execution and coordination of systems [49]. We categorize MoCs along three types: *timed*, *causal*, *a-causal*. A *timed* MoC can use *physical* or *logical* time. The categorization of *causal* and *a-causal* MoCs is also present in the co-simulation field [22]. A causal MoC means that there is a notion of inputs and outputs in the events and data exchanged between system parts. An output can cause reactions, i.e., state changes in the input of the other systems. For example, a system based on a state machine raises an event, which causes a reaction in a different system.

In an a-causal MoC, there is no notion of inputs or outputs of data but rather of equality (in the mathematical sense). Two data a-causally coordinated must be equal at any time. This is, for instance, typical in differential equations.

*Logical* time is a concept that provides a way to reason about the partial order of events (and data exchanges) without needing to rely on a physical clock. It allows for the specification of causal and a-causal relationships between events and helps ensure the consistency and correctness of loosely coupled systems [20,17]. For instance, one can specify that an action is performed every ten startups of the system, regardless of how frequently the startup physically occurs.

In contrast, *physical* time describes time passing in the physical world. It is also referred to as *wall-clock* time [22]. For example, one can define that an event should happen 50 milliseconds after another. In Lingua Franca [32], logical time "chases" physical time, i.e., logical time tries to match the physical time provided by the execution platform.

*Formality* of a coordination approach is either *formal* or *semi-formal*. Semi-formal approaches are directly implemented in a general-purpose programming language, often with a guiding formalism in mind but without strict adherence to that formalism.

Formal coordination approaches are implemented in a *formal language*, which can then be run on an execution engine implemented for that formalism. For example, the coordination framework BCoorLang [29] uses graph or term-rewriting as formal languages to coordinate heterogeneous systems in a centralized manner. It relies on the graph-transformation tool Groove [50] or the term-rewriting tool Maude for execution.

### 4.2 Goal

We identified three different *goals* of coordination approaches: **simulation**, **execution**, and **formal verification**, see Figure 4. A coordination approach can have one or multiple goals, which we will now describe in detail.

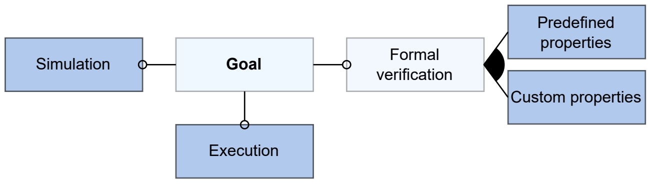

**Fig. 4.** Goal feature

**Simulation** The most common goal of coordination approaches is a coordinated simulation of multiple systems or behavioral models representing an entire system. A coordinated simulation is useful because the composition of multiple systems can lead to unexpected behavior, often called *emergent behavior* [18]. Emergent behavior can then be uncovered during simulation before deploying and using the system in a production environment. Co-simulation approaches such as DACCOSIM [21,15] or MECSYCO (Multi-agent Environment for ComplexSYstem CO-simulation) [10,11] are examples of approaches that have coordinated simulation as their goal. Simulation is also a goal of coordination frameworks such as BCoorLang [29] and BCOol [57]. Even recent ADLs such as MontiArc [24] provide simulation capabilities.

**Execution** In contrast to only simulation, some coordination approaches aim to provide a coordination infrastructure for system execution upon deployment. For example, Linda provides a virtual shared memory with read and write operations to coordinate systems.

Furthermore, the recently developed polyglot coordination language *Lingua Franca* provides determinism guarantees during distributed execution [33]. Thus, it eliminates typical coordination concerns faced during coordinating concurrent *execution.*

**Formal verification** Another goal of coordination approaches, especially for safety-critical systems, is formal verification of the coordinated system.

We encountered coordination approaches that validate *predefined properties*, while others support *custom properties.* For example, the ADL Wright [2] can check *deadlock freedom* of system interactions. The coordination framework for behavioral consistency [29] supports custom properties written in temporal logic such as Linear Temporal Logic (LTL) and Computational Tree Logic (CTL).

## 4.3 Properties

We identified several reoccurring properties when analyzing coordination approaches. Figure 5 depicts the resulting *properties* feature, which we will now explain in detail.

**Domain** Coordination approaches operate in different domains like the categorization of co-simulation approaches described in [22,31]. First, if a coordination approach is part of the *Discrete Event* (DE) domain, it allows discrete but not continuous state changes [22]. Typically, coordination frameworks and ADLs operate in the DE domain since variables change their values discontinuously during execution, i.e., immediately change between states.

Second, coordination approaches in the *Continuous Time* (CT) domain support systems to have a state that evolves continuously over time. Often, systems are given by differential equations, which induce continuous variable changes, such as physical systems involving springs and dampeners, see [22].

Third, the *hybrid* domain is a mix of the DE and CT domains that occur, for example, when simulating cyber-physical systems. In a hybrid domain, systems should coordinate where some change their state discretely (DE domain) while others evolve continuously over time (CT domain). Two strategies deal with hybrid systems: either adapt the systems from the CT to the DE domain or vice versa [22].

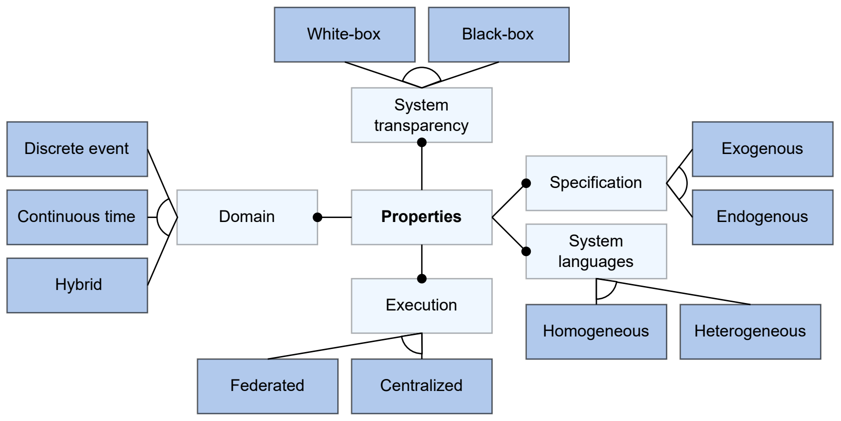

**Fig. 5.** Properties feature

**System transparency** Some coordination approaches require constituent systems to be entirely transparent (*white-box*), while others only require a fixed interface (*black-box*). Usually, black-box approaches are needed to protect *intellectual property* (IP) when different companies want to work together without sharing all the details concerning their systems.

For example, DACCOSIM only requires each system to conform to the FMI specification for Co-Simulation.

**Specification** We distinguish between *endogenous* and *exogenous* specification of coordination, as described for coordination languages in [5]. For example, when using the coordination language Linda [12], one uses coordination-specific operations such as `out` and `in` inside system programs to write to and read from the shared tuple space, respectively. Thus, Linda is *endogenous* since one must modify the behavior of the systems to add coordination.

On the other hand, *exogenous* coordination approaches keep coordination aspects outside of systems. However, systems must still be created with coordination in mind to expose possible coordination points. For example, BCOol [57] uses language behavioral interfaces to define events, which can then be used to specify *exogenous* coordination rules outside the systems. Endogenous coordination approaches tend to mix computation and coordination [5].

**System languages** Coordination approaches typically only support *homogeneous* systems. For example, coordination languages, such as Lingua Franca [32], currently only support coordination if each system is given in the same programming language.

However, coordination frameworks allow *heterogeneous* systems, i.e., systems specified in different behavioral modeling languages. For example, BCoorLang [29] supports all modeling languages that one can describe using rewriting semantics.

**Execution** A coordination approach is realized using either a *centralized* or *federated* execution. In a *centralized* execution, all systems are executed by one engine, which enforces coordination. For example, BCoorLang composes the behavior of all system specifications into a global behavior specification, which is then executed by a graph transformation or term-rewriting tool.

On the other hand, in a *federated execution*, systems run independently and only coordinate using a shared infrastructure through predefined connections. Federated execution can still happen on the same physical machine, i.e., it does not mean that systems must be *distributed* across different physical machines.

For example, the coordination language Lingua Franca [32] supports federated execution on one machine by providing a runtime infrastructure. DACCOSIM [21] goes one step further and allows a *distributed* federated execution.

## 5 Application of the feature model

Table 1 shows the application of our feature model to four different coordination approaches. Each approach comes from a different category: co-simulation (*DACCOSIM* [21,15]), coordination language (*Linda* [12,13]), ADL (*MontiArc* [24]), and coordination framework (*BCoorLang* [28,29]).

**Table 1.** Approach classification

| Approach / Feature | DACCOSIM | Linda | MontiArc | BCoorLang |
|---|---|---|---|---|
| **Foundation** | | | | |
| Syntax | Architecture based | Name based | Architecture based | Dedicated elements |
| Mechanism | Data-Exchange | Data-Exchange | Data-Exchange | Event-Scheduling |
| Expressiveness | Predefined | Predefined | Predefined | Predefined |
| MoC | Causal | Causal | Logical, Causal | Logical |
| Formality | Semi-formal | Semi-formal | Semi-formal | Formal |
| **Goal** | | | | |
| Simulation | + | - | + | + |
| Execution | - | + | - | - |
| Formal verification | - | - | - | Custom properties |
| **Properties** | | | | |
| Domain | Hybrid | Discrete event | Discrete event | Discrete event |
| Comp. transparency | Black-box | White-box | White-box | White-box |
| Specification | Exogenous | Endogenous | Endogenous | Exogenous |
| Comp. languages | Homogeneous | Homogeneous | Homogeneous | Heterogeneous |
| Execution | Federated | Federated | Centralized | Centralized |

In addition, we classified the following approaches: Ptolemy [18,49], Wright [2,1], CommUnity [19,43], Metropolis [7], UMoC++ [37], Lingua Franca [32,33], Reo [6], BIP [9,8], MECSYCO [11,10], CoSim20 [31], BCOoL [57,56], Manifold [4,48], and ForSyDe [52,53]. Due to space constraints, we cannot show all classifications, but they are available in the artifacts [55].

In Figure 6, we compare BCoorLang and MontiArc with Linda on the left and DACCOSIM on the right. The figure shows Venn diagrams comparing the features of each approach, where each part states which features are contained. In addition, the numbers state the amount of contained features in each area.

For example, when looking at the Venn diagram on the left in Figure 6, the 1 in the circle part at the bottom describes that MontiArc has one *original* feature, while the 3 in the intersection of all approaches in the middle declares that they all have three features in *common*, see Table 1.

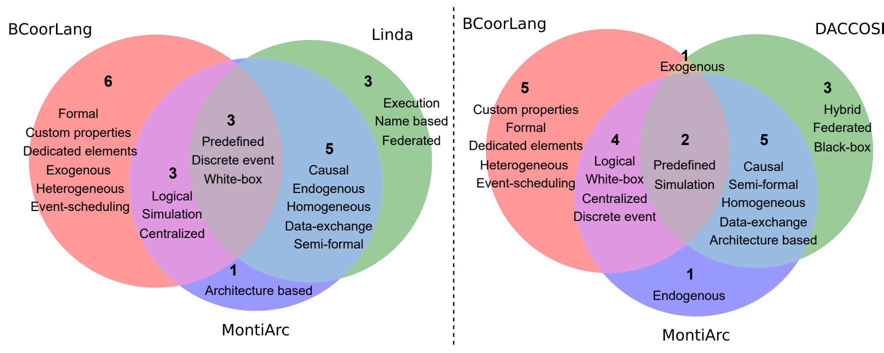

**Fig. 6.** Feature comparison: BCoorLang, MontiArc, and Linda/DACCOSIM

The Venn diagram on the left in Figure 6 shows that Linda has three original features compared to BCoorLang and MontiArc: execution as a goal, a name-based syntax, and a federated execution. Furthermore, one can see that coordination frameworks, i.e., BCoorLang (the diagram is similar for BCOol [57,56]) greatly differ from ADLs and coordination languages. Coordination frameworks have multiple original features, such as supporting heterogeneous system languages and formal verification using custom properties.

Similarly, on the right in Figure 6, BCoorLang and MontiArc are compared with the co-simulation approach DACCOSIM. One can see that BCoorLang and DACCOSIM do not have many features in common, probably due to their different goals as a coordination approach. Co-simulation approaches like DACCOSIM have original features, such as a federated execution, hybrid domain, and black-box system transparency.

Venn diagrams give a good approximation of the similarity between three specific coordination approaches. However, they do not tell the whole story since each feature is given equal weight, which is not always adequate. For example, a difference in the *domain* feature is more profound than most other differences. We stick to this simple weighting throughout this paper since the importance of features can be subjective. However, one can adapt the analysis using the provided data and scripts [55].

## 6 Findings

This section describes the findings of applying our taxonomy to the different coordination approaches. We start by summarizing the common features of each coordination category before stating the general insights we have attained. In addition, we cluster the approaches based on their feature similarity and discuss the results.

### 6.1 Common features

**ADLs** primarily have *simulation* as their goal, while a small subset allows predefined properties to be verified. As expected, the coordination syntax of all ADLs is architecture-based, using components with ports and connectors between them. All ADLs operate in the discrete event domain. Furthermore, components in ADLs are white-box and homogeneous, and coordination specification is usually exogenous, i.e., part of the connectors. An exception is MontiArc, where ports can be marked as *synchronized* only allowing synchronous interactions through a port; that is, the coordination specification is endogenous and partly present inside components.

**Coordination languages** can have simulation and execution as their goal while operating in the discrete event domain. Furthermore, coordination syntax is also most likely architecture-based, i.e., the idea of subsystems, ports, and connectors/channels between them is also used in most coordination languages. In addition, the systems in coordination languages are white-box and homogeneous but have both endogenous and exogenous coordination specifications. Notably, some coordination languages offer a federated execution not supported by ADLs.

As the name suggests, all **co-simulation** approaches have merely simulation as their goal. Like ADLs and coordination languages, coordination syntax is architecture-based. However, compared to the other categories, they operate in the hybrid domain, facilitating data exchange between black-box systems. The systems can be homogeneous or heterogeneous by wrapping them in specific formalisms such as DEVS in the case of MECSYCO. Typically, execution is federated, while some approaches, such as DACCOSIM, even allow for a distributed execution.

**Coordination frameworks** have simulation and formal verification as their goal. Coordination syntax is diverse but usually applies event scheduling in the discrete event domain. Furthermore, coordination specification is exogenous to support heterogeneous systems, which are white-box. In contrast to coordination languages and co-simulation approaches, execution is centralized. In addition, some coordination frameworks, such as BCoorLang, facilitate formal verification of custom properties, which is a rare feature.

### 6.2 General insights

**Architecture-based coordination syntax is widely used.** Remarkably, coordination approaches from all categories use concepts such as components, ports, and connectors to express coordination. The precise coordination details may differ across various approaches, but there seems to be a broad consensus on the importance of accurately representing the system architecture for efficient coordination.

**Formal verification is rarely supported.** Most of the studied coordination approaches have simulation as their goal, while formal verification is uncommon. This could be attributed to the problem of formally representing heterogeneous systems engaged in coordination and the rapid growth of state spaces when multiple systems are involved.

**Coordination frameworks do not support Data-Exchange.** All coordination frameworks besides Ptolemy support *event-scheduling* but not *data-exchange* as a coordination mechanism. This difficulty likely arises due to heterogeneous systems, making data exchange challenging. Nevertheless, data exchange is omnipotent in real-world systems, serving as the primary coordination mechanism utilized in co-simulation methods and coordination languages. Therefore, coordination frameworks must be able to facilitate data exchange to be effectively utilized in real-world scenarios.

### 6.3 Feature clustering analysis

We further analyze the classification of the approaches mentioned in section 5 to determine how they compare across and inside the four categories. We clustered the approaches to see similarities and differences by just looking at the feature data of each approach without considering to which category they belong. The data and scripts to reproduce our analysis are available in [55]. We apply a standard clustering algorithm using Jaccard distance to measure the similarity of the feature sets. Jaccard distance is the one-complement of Jaccard similarity for two sets, defined as the ratio of their intersection to their union [30]. It is a widely used and reasonable dissimilarity measure for sets [30].

Figure 7 shows a scatter plot with approximate positions of each approach derived solely from the distances between them. It provides a bird's-eye view of the similarity or dissimilarity between all the studied coordination approaches—this is in contrast to the previous Venn diagrams, which only compare three approaches. Furthermore, it highlights the clustering results by coloring each data point. We selected clustering parameters to reduce the number of unclustered approaches while ensuring that we do not consolidate everything into just one cluster.

The clustering algorithm finds three clusters, including all but one approach. **Cluster 1** contains only the coordination frameworks BCoorLang and BCOol, while **cluster 2** consists of a mix of coordination languages and ADLs, such as Linda and Lingua Franca, as well as Wright and MontiArc, respectively. **Cluster 3** includes three co-simulation approaches, but no cluster contains Ptolemy.

We interpret cluster 1 as representing the coordination framework category. Similarly, cluster 3 is the co-simulation cluster, showing that according to our taxonomy, co-simulation approaches are similar and have original features not found in the other categories. Cluster 2 contains coordination languages and

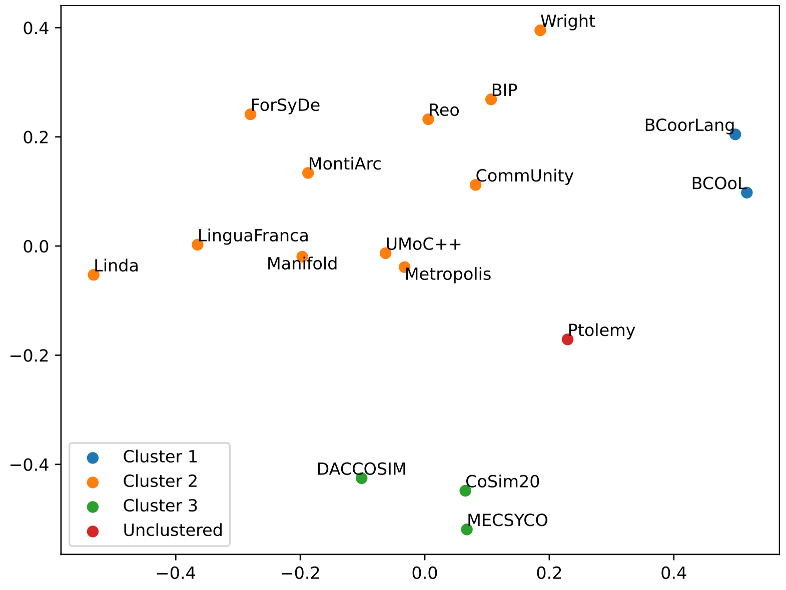

**Fig. 7.** Feature distance and clustering of approaches

ADLs. Even if the maximum distance between approaches is reduced to be considered part of the same cluster (clustering parameter), one cannot separate the approaches into two categories. Thus, we conclude that coordination languages and ADLs have more similarities in features compared to, for instance, co-simulation approaches or coordination frameworks.

As the only approach, Ptolemy remains unclustered. Although we would classify Ptolemy as a coordination framework, it includes features from all categories and is not part of any cluster.

## 7 Conclusion

The main contribution of this paper is a taxonomy of coordination approaches represented as a feature model. This taxonomy facilitates the categorization and comparison of approaches across previously isolated categories. Consequently, we provide a comprehensive overview of coordination in the literature and the state of the art.

Furthermore, we apply our taxonomy to 17 coordination approaches and publish the results as open data [55]. Analyzing the resulting data, we identify characteristic and unique features in each category. As overarching insights, we find that architecture-based coordination is widely used, formal verification is rarely supported, and coordination frameworks do not support data exchange.

In addition, we cluster the approaches regarding their similarity. The clustering shows that ADLs and coordination languages are very similar from the perspective of our taxonomy, while the other categories stay separate.

## References

1. Allen, R.: A Formal Approach to Software Architecture. Ph.D. thesis, Carnegie Mellon, School of Computer Science (Jan 1997)
2. Allen, R., Garlan, D.: A formal basis for architectural connection. ACM Transactions on Software Engineering and Methodology **6**(3), 213–249 (Jul 1997). `https://doi.org/10.1145/258077.258078`
3. André, C.: Syntax and semantics of the clock constraint specification language (CCSL). Technical Report RR-6925, INRIA (2009)
4. Arbab, F., Herman, I., Spilling, P.: An overview of manifold and its implementation. Concurrency: Practice and Experience **5**(1), 23–70 (Feb 1993). `https://doi.org/10.1002/cpe.4330050103`
5. Arbab, F.: What do you mean, coordination? Bulletin of the Dutch Association for Theoretical Computer Science pp. 11–22 (1998)
6. Arbab, F.: Reo: A channel-based coordination model for component composition. Mathematical Structures in Computer Science **14**(3), 329–366 (Jun 2004). `https://doi.org/10.1017/S0960129504004153`
7. Balarin, F., Watanabe, Y., Hsieh, H., Lavagno, L., Passerone, C., Sangiovanni-Vincentelli, A.: Metropolis: An integrated electronic system design environment. Computer **36**(4), 45–52 (Apr 2003). `https://doi.org/10.1109/MC.2003.1193228`
8. Basu, A., Bensalem, B., Bozga, M., Combaz, J., Jaber, M., Nguyen, T.H., Sifakis, J.: Rigorous Component-Based System Design Using the BIP Framework. IEEE Software **28**(3), 41–48 (May 2011). `https://doi.org/10.1109/MS.2011.27`
9. Bliudze, S., Sifakis, J.: The Algebra of Connectors—Structuring Interaction in BIP. IEEE Transactions on Computers **57**(10), 1315–1330 (Oct 2008). `https://doi.org/10.1109/TC.2008.26`
10. Camus, B., Galtier, V., Caujolle, M.: Hybrid co-simulation of FMUs using DEV&DESS in MECSYCO. In: 2016 Symposium on Theory of Modeling and Simulation (TMS-DEVS). pp. 1–8 (2016). `https://doi.org/10.23919/TMS.2016.7918814`
11. Camus, B., Paris, T., Vaubourg, J., Presse, Y., Bourjot, C., Ciarletta, L., Chevrier, V.: Co-simulation of cyber-physical systems using a DEVS wrapping strategy in the MECSYCO middleware. SIMULATION **94**(12), 1099–1127 (Dec 2018). `https://doi.org/10.1177/0037549717749014`
12. Carriero, N., Gelernter, D.: Linda in context. Communications of The Acm **32**(4), 444–458 (Apr 1989). `https://doi.org/10.1145/63334.63337`
13. Carriero, N.J., Gelernter, D., Mattson, T.G., Sherman, A.H.: The Linda alternative to message-passing systems. Parallel Computing **20**(4), 633–655 (Apr 1994). `https://doi.org/10.1016/0167-8191(94)90032-9`
14. Clements, P.: A survey of architecture description languages. In: Proceedings of the 8th International Workshop on Software Specification and Design. pp. 16–25. IEEE Comput. Soc. Press, Schloss Velen, Germany (1996). `https://doi.org/10.1109/IWSSD.1996.501143`
15. Dad, C., Tavella, J.P., Vialle, S.: Synthesis and feedback on the distribution and parallelization of FMI-CS-based co-simulations with the DACCOSIM platform. Parallel Computing **106**, 102802 (Sep 2021). `https://doi.org/10.1016/j.parco.2021.102802`
16. Dahmann, J.: High level architecture for simulation. In: Proceedings First International Workshop on Distributed Interactive Simulation and Real Time Applications. pp. 9–14 (1997). `https://doi.org/10.1109/IDSRTA.1997.568652`

17. Deantoni, J., Mallet, F.: Timesquare: Treat your models with logical time. In: Furia, C.A., Nanz, S. (eds.) Objects, Models, Components, Patterns. pp. 34–41. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
18. Eker, J., Janneck, J., Lee, E., Jie Liu, Xiaojun Liu, Ludvig, J., Neuendorffer, S., Sachs, S., Yuhong Xiong: Taming heterogeneity - the Ptolemy approach. Proceedings of the IEEE **91**(1), 127–144 (Jan 2003). https://doi.org/10.1109/JPROC.2002.805829
19. Fiadeiro, J.L., Lopes, A.: Semantics of architectural connectors. In: Goos, G., Hartmanis, J., Van Leeuwen, J., Bidoit, M., Dauchet, M. (eds.) TAPSOFT '97: Theory and Practice of Software Development, vol. 1214, pp. 503–519. Springer Berlin Heidelberg, Berlin, Heidelberg (1997). https://doi.org/10.1007/BFb0030622
20. Fidge, C.: Logical time in distributed computing systems. Computer **24**(8), 28–33 (1991). https://doi.org/10.1109/2.84874
21. Galtier, V., Vialle, S., Dad, C., Tavella, J.P., Lam-Yee-Mui, J.P., Plessis, G.: FMI-Based distributed multi-simulation with DACCOSIM. In: Proceedings of the Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium. pp. 39–46. DEVS '15, Society for Computer Simulation International, San Diego, CA, USA (2015)
22. Gomes, C., Thule, C., Broman, D., Larsen, P.G., Vangheluwe, H.: Co-Simulation: A Survey. ACM Computing Surveys **51**(3), 1–33 (May 2019). https://doi.org/10.1145/3179993
23. Goos, G., Hartmanis, J., Van Leeuwen, J. (eds.): Coordination Models and Languages, vol. 2039, pp. 33–52. Springer Berlin Heidelberg, Berlin, Heidelberg (2001). https://doi.org/10.1007/3-540-44933-7_3
24. Haber, A., Ringert, J.O., Rumpe, B.: MontiArc - Architectural Modeling of Interactive Distributed and Cyber-Physical Systems (2014). https://doi.org/10.48550/ARXIV.1409.6578
25. Hafner, I., Popper, N.: An Overview of the State of the Art in Co-Simulation and Related Methods. SNE Simulation Notes Europe **31**(4), 185–200 (Dec 2021). https://doi.org/10.11128/sne.31.on.10582
26. Hussain, S.: Investigating Architecture Description Languages (ADLs) A Systematic Literature Review (2013)
27. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature-Oriented Domain Analysis (FODA) Feasibility Study:. Tech. rep., Defense Technical Information Center, Fort Belvoir, VA (Nov 1990). https://doi.org/10.21236/ADA235785
28. Krauter, T.: Towards behavioral consistency in heterogeneous modeling scenarios. In: 2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C). pp. 666–671. IEEE, Fukuoka, Japan (Oct 2021). https://doi.org/10.1109/MODELS-C53483.2021.00107
29. Kräuter, T., König, H., Rutle, A., Lamo, Y., Stünkel, P.: Behavioral consistency in multi-modeling. The Journal of Object Technology **22**(2), 2:1 (2023). https://doi.org/10.5381/jot.2023.22.2.a9
30. Levandowsky, M., Winter, D.: Distance between Sets. Nature **234**(5323), 34–35 (Nov 1971). https://doi.org/10.1038/234034a0
31. Liboni, G.: Complex Systems Co-Simulation with the CoSim20 Framework : For Efficient and Accurate Distributed Co-Simulations. Theses, Université Côte d'Azur (Apr 2021)
32. Lohstroh, M.: Reactors: A Deterministic Model of Concurrent Computation for Reactive Systems. Ph.D. thesis, EECS Department, University of California, Berkeley (Dec 2020)

33. Lohstroh, M., Menard, C., Bateni, S., Lee, E.A.: Toward a Lingua Franca for Deterministic Concurrent Systems. ACM Transactions on Embedded Computing Systems **20**(4), 1–27 (Jul 2021). https://doi.org/10.1145/3448128
34. Magee, J., Dulay, N., Eisenbach, S., Kramer, J.: Specifying distributed software architectures. In: Goos, G., Hartmanis, J., Van Leeuwen, J., Schäfer, W., Botella, P. (eds.) Software Engineering — ESEC '95, vol. 989, pp. 137–153. Springer Berlin Heidelberg, Berlin, Heidelberg (1995). https://doi.org/10.1007/3-540-60406-5_12
35. Malavolta, I., Lago, P., Muccini, H., Pelliccione, P., Tang, A.: What Industry Needs from Architectural Languages: A Survey. IEEE Transactions on Software Engineering **39**(6), 869–891 (Jun 2013). https://doi.org/10.1109/TSE.2012.74
36. Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, Carolyn Talcott: All About Maude - A High-Performance Logical Framework, Lecture Notes in Computer Science, vol. 4350. Springer Berlin Heidelberg, Berlin, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71999-1
37. Mathaikutty, D.A., Patel, H.D., Shukla, S.K., Jantsch, A.: UMoC++: A C++-Based Multi-MoC Modeling Environment. In: Vachoux, A. (ed.) Applications of Specification and Design Languages for SoCs, pp. 115–130. Springer Netherlands, Dordrecht (2006). https://doi.org/10.1007/978-1-4020-4998-9_7
38. Medvidovic, N.: Moving Architectural Description from Under the Technology Lamppost. In: 32nd EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO'06). pp. 2–3. IEEE, Cavtat, Dubrovnik, Croatia (2006). https://doi.org/10.1109/EUROMICRO.2006.47
39. Medvidovic, N., Taylor, R.: A classification and comparison framework for software architecture description languages. IEEE Transactions on Software Engineering **26**(1), 70–93 (Jan/2000). https://doi.org/10.1109/32.825767
40. Medvidovic, N., Taylor, R.N.: A framework for classifying and comparing architecture description languages. In: Goos, G., Hartmanis, J., van Leeuwen, J., Jazayeri, M., Schauer, H. (eds.) Software Engineering — ESEC/FSE'97, vol. 1301, pp. 60–76. Springer Berlin Heidelberg, Berlin, Heidelberg (1997). https://doi.org/10.1007/3-540-63531-9_7
41. Modelisar: Functional Mock-up Interface Specification. https://fmi-standard.org/docs/3.0.1/ (Jul 2023)
42. Nixon, L.J.B., Simperl, E., Krummenacher, R., Martin-Recuerda, F.: Tuplespace-based computing for the Semantic Web: A survey of the state-of-the-art. The Knowledge Engineering Review **23**(2), 181–212 (Jun 2008). https://doi.org/10.1017/S0269888907001221
43. Oliveira, C., Wermelinger, M.: The CommUnity Workbench. Science of Computer Programming **69**(1-3), 46–55 (Dec 2007). https://doi.org/10.1016/j.scico.2006.09.005
44. Omicini, A., Viroli, M.: Coordination models and languages: From parallel computing to self-organisation. The Knowledge Engineering Review **26**(1), 53–59 (Feb 2011). https://doi.org/10.1017/S026988891000041X
45. Ozkaya, M., Kloukinas, C.: Are We There Yet? Analyzing Architecture Description Languages for Formal Analysis, Usability, and Realizability. In: 2013 39th Euromicro Conference on Software Engineering and Advanced Applications. pp. 177–184. IEEE, Santander, Spain (Sep 2013). https://doi.org/10.1109/SEAA.2013.34
46. Pandey, R.K.: Architectural description languages (ADLs) vs UML: A review. ACM SIGSOFT Software Engineering Notes **35**(3), 1–5 (May 2010). https://doi.org/10.1145/1764810.1764828

47. Papadopoulos, G.A., Arbab, F.: Coordination Models and Languages. In: Advances in Computers, vol. 46, pp. 329–400. Elsevier (1998). https://doi.org/10.1016/S0065-2458(08)60208-9
48. Papadopoulos, G.A., Arbab, F.: Modelling activities in information systems using the coordination language MANIFOLD. In: Proceedings of the 1998 ACM Symposium on Applied Computing - SAC '98. pp. 185–193. ACM Press, Atlanta, Georgia, United States (1998). https://doi.org/10.1145/330560.330667
49. Ptolemaeus, C. (ed.): System Design, Modeling, and Simulation: Using Ptolemy II. UC Berkeley EECS Dept, Berkeley, Calif, 1. ed., version 1.02 edn. (2014)
50. Rensink, A.: The GROOVE Simulator: A Tool for State Space Generation. In: Kanade, T., Kittler, J., Kleinberg, J.M., Mattern, F., Mitchell, J.C., Naor, M., Nierstrasz, O., Pandu Rangan, C., Steffen, B., Sudan, M., Terzopoulos, D., Tygar, D., Vardi, M.Y., Weikum, G., Pfaltz, J.L., Nagl, M., Böhlen, B. (eds.) Applications of Graph Transformations with Industrial Relevance, vol. 3062, pp. 479–485. Springer Berlin Heidelberg, Berlin, Heidelberg (2004). https://doi.org/10.1007/978-3-540-25959-6_40
51. Rossi, D., Cabri, G., Denti, E.: Tuple-based Technologies for Coordination. In: Omicini, A., Zambonelli, F., Klusch, M., Tolksdorf, R. (eds.) Coordination of Internet Agents, pp. 83–109. Springer Berlin Heidelberg, Berlin, Heidelberg (2001). https://doi.org/10.1007/978-3-662-04401-8_4
52. Sander, I., Jantsch, A.: System Modeling and Transformational Design Refinement in ForSyDe. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **23**(1), 17–32 (Jan 2004). https://doi.org/10.1109/TCAD.2003.819898
53. Sander, I., Jantsch, A., Attarzadeh-Niaki, S.H.: ForSyDe: System Design Using a Functional Language and Models of Computation. In: Ha, S., Teich, J. (eds.) Handbook of Hardware/Software Codesign, pp. 1–42. Springer Netherlands, Dordrecht (2016). https://doi.org/10.1007/978-94-017-7358-4_5-1
54. Schweiger, G., Gomes, C., Engel, G., Hafner, I., Schoeggl, J., Posch, A., Nouidui, T.: An empirical survey on co-simulation: Promising standards, challenges and research needs. Simulation Modelling Practice and Theory **95**, 148–163 (Sep 2019). https://doi.org/10.1016/j.simpat.2019.05.001
55. Tim Kräuter: Artifacts Coordination. https://github.com/timKraeuter/Coordination-2024 (Feb 2024)
56. Vara Larsen, M.: BCOol : The Behavioral Coordination Operator Language. Ph.D. thesis, Université Nice Sophia Antipolis (Apr 2016)
57. Vara Larsen, M.E., Deantoni, J., Combemale, B., Mallet, F.: A Behavioral Coordination Operator Language (BCOoL). In: 2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS). pp. 186–195. IEEE, Ottawa, ON, Canada (Sep 2015). https://doi.org/10.1109/MODELS.2015.7338249
58. Woods, E., Hilliard, R.: Architecture Description Languages in Practice Session Report. In: 5th Working IEEE/IFIP Conference on Software Architecture (WICSA'05). pp. 243–246. IEEE, Pittsburgh, PA, USA (2005). https://doi.org/10.1109/WICSA.2005.15