Different Glasses to Look into the Three Cs: Component, Connector, Coordination

Farhad Arbab[1], Marco Autili[2], Paola Inverardi[2], and Massimo Tivoli[2](✉)

[1] Centrum Wiskunde & Informatica,
Science Park 123, 1098 XG Amsterdam, The Netherlands
Farhad.Arbab@cwi.nl

[2] Department of Information Engineering, Computer Science and Mathematics,
University of L'Aquila, L'Aquila, Italy
{marco.autili,paola.inverardi,massimo.tivoli}@univaq.it

Abstract. Component, connector, and coordination have been key concepts exploited in different communities to manage the complexity of concurrent and distributed system development. In this paper, we discuss three approaches within three different classes: composition in software architectures, coordination models, and programming abstractions for concurrency. These classes encompass different perspectives and solutions to face crucial challenges in developing concurrent and distributed systems. The approaches are discussed with respect to some characteristics of interest for the above classes: compositionality, incrementality, scalability, compositional reasoning, reusability, and evolution.

Keywords: Software components · Connectors · Software architectures · Coordination models · Programming abstractions

1 Introduction

Since late 70's, the development of concurrent and distributed systems has been receiving much attention from the research community [72,77]. Later, since 90's, component, connector, and coordination have been key concepts exploited in different communities to manage the complexity of concurrent and distributed systems development [2,4,9,17,18,24,32,37,43,47,49,52,75,78,79,83,84,87]. Process calculi and algebras laid the theoretical foundation for concurrency. The concept of *coordination* was introduced to offer software developers programming language constructs and models at a level of abstraction higher than the primitives offered by the parsimony of process algebras [3,39]. The concept of *connectors* emerged in software architectures as a useful construct to facilitate communication among independently developed *components*.

In this paper, we discuss three different approaches within three different classes: composition in software architectures, coordination models, and programming abstractions for concurrency. These classes encompass different perspectives and solutions to face crucial challenges in developing concurrent and

© Springer Nature Switzerland AG 2019
M. Boreale et al. (Eds.): De Nicola-Festschrift, LNCS 11665, pp. 191–216, 2019.
https://doi.org/10.1007/978-3-030-21485-2_12

distributed systems. Strictly concerning the purpose of this festschrift paper, we limit our discussion to previous work by De Nicola et al. and previous work by the authors. Specifically, we focus on programming abstractions for concurrent and autonomic systems [84]; exogenous coordination [5]; and the architectural synthesis of software coordinators for the distributed composition of heterogeneous components [18].

Since these approaches are radically different in nature, it is not possible, neither would it make sense, to make a point-to-point comparison or to force a description of them by adopting a uniform writing strategy at the same level of detail. However, this paper makes an effort to discuss the three approaches with respect to the same set of characteristics of interest for the above classes: compositionality, incrementality, scalability, compositional reasoning, reusability, and evolution.

This paper is organized as follows. Section 2 preludes our discussion by collocating the three works above within the state of the art. Section 3 provides a concise, yet complete, description of the three works, which are then characterized in Sect. 5 with respect to the six dimensions defined in Sect. 4. Section 6 concludes the paper.

2 Praeludium

In this section, we give a brief account of the three approaches that we consider followed by a brief discussion on related approaches.

The first approach by De Nicola et al. [84] is based on the definition of programming abstractions for concurrent and autonomic systems. The starting point of the work is represented by the notions of autonomic components (ACs) and autonomic-component ensembles (ACEs). The defined programming abstractions permit to model their evolution and their interactions. The authors define SCEL (Software Component Ensemble Language), a kernel language for programming the behavior of ACs and the formation of ACEs, and for controlling the interaction among different ACs. These abstractions permit describing autonomic systems in terms of behaviors, knowledge, and aggregations by complying with specific policies.

The second approach essentially concerns the work conducted by Arbab as described in [5], among other references. The author emphasizes the separation between computation and coordination defining a data-flow paradigm. Arbab defines the notion of *Abstract Behavior Type*s (ABTs) as a higher-level alternative to ADT (*Abstract Data Type*) and proposes it as a proper foundation model for both components and their composition. An ABT defines an abstract behavior as a relation among a set of *timed-data-streams*, without specifying any detail about the operations that may be used to implement such behavior or the data types it may manipulate for its realization. ABTs allows for loosely coupling and exogenous coordination that are considered as two essential properties for components and their composition. ABTs serve as the primary formal semantic model for the coordination language Reo.

The third approach that we consider is described in the work conducted by Inverardi et al. [18]. The authors provide a complete formalization of an automated synthesis method for the distributed composition and coordination of software services. The method takes as input a specification of the global collaboration that the involved services must realize. This specification is given in the form of a state machine. The methods automatically generates a set of *Coordination Delegates* (CDs). CDs are additional software entities with respect to the participant services, and are synthesized in order to proxify and control their interaction. When interposed among the services, the synthesized entities enforce the collaboration prescribed by the specification. The synthesized CDs are proved to be correct by construction, meaning that the resulting distributed system realizes the specification. The synthesis method is able to deal with heterogeneous services that communicate synchronously and/or asynchronously. CDs are able to handle asynchrony through 1-bounded FIFO queues.

As already introduced, beyond the above mentioned approaches, there are many other approaches in the literature that should be considered (see [85] for an early comprehensive survey). For instance, in [2], the authors define a control-flow event-based paradigm for both computation and coordination. The WRIGHT architecture description language [1] is used as a specialized notation for architectural specification. As underlining formalism, the authors embed in WRIGHT an approach based on process algebra. In fact, in [2], CSP [86] (*Communicating Sequential Processes*) is used by the authors in order to provide an operational formalization of the separation between computation and coordination.

A family of process calculi called "*Kell calculus*" is presented in [33,34,87,90]. It has been intended as a basis for studying distributed (and ubiquitous) component-based programming. Essentially, the Kell calculus is an *high-order* extension of π-Calculus. Its aim is to support the modeling of different forms of process *mobility* (e.g., logical and physical mobility). This is done by considering the possibility to directly transmit *processes* as messages and not only channels (used by processes in order to communicate) as it is in π-Calculus.

A further approach concerns the work described in [36]. The authors propose an algebraic formalization of the structure of the interactions enabled by connectors in component-based system implemented in the BIP framework [26,89]. It is a control-flow paradigm based on active/inactive communication ports of components.

The work described in [79] presents a modeling approach based upon the Bigraphical Reactive Systems framework developed by Milner, which consists of a bigraph together with a collection of bigraphical rewrite rules. Analogously to Kell calculus, this approach introduces mobility, locality and dynamism.

Further approaches are described in [37,47,75] and, as pure algebraic modeling approaches, they are theoretically very powerful although, they result to be hard to be used by practitioners.

Beyond the notational/algebraic/mathematical class of works on connector/component modeling, another interesting class of works that should be

considered concerns *quantitative* approaches, e.g., [9,24]. These approaches are quantitative in the sense that they are able to express, and reason about, characteristics such as the probability of an event occurring, the elapse of time, performance, QoS, etc. In particular, in [9], the work described in [5] is extended in order to take into account QoS attributes of both computation and coordination, e.g., *shortest time for data transmission*, *allocated memory cost for data transmission*, and *reliability* represented by the probability of successful transmission. Furthermore, this work defines a semantic model for connectors different from ABTs, i.e., it is an operational model based on a QoS extension of *constraint automata* [24] called *Quantitative Constraint Automata*. In spirit, this model is a variant of a *labelled transition-system* model. Other extensions of Reo are also based on the constraint-automata semantics, and allow two forms of probability distributions, continuous-time (with no nondeterminism) and discrete-time (with nondeterminism) [25].

3 Looking into the Three Cs

Within the "*three Cs sphere*", the three approaches considered in this paper represent three possible ways of dealing with component-based system development, component connection and coordination. For different purposes and at different levels, these approaches target the complexity of concurrent and distributed system development, and address crucial challenges to be faced when developing component-based systems, possibly reusing existing third-party components, connecting them and coordinating their interaction.

This section provides a concise, yet complete, description of the three approaches. For each of them, we first provide an overview by summarizing notions and aspects borrowed from the corresponding original work; then, we discuss the approach with respect to the best-fitting "**C**".

3.1 Software Component Ensemble Language

The aims of the work in [84] is to provide language designers with appropriate programming abstractions and constructs to deal with the development of concurrent and autonomic systems, adaptation with respect to possibly unforeseen changes of the working environment, evolving requirements, emergent behaviors resulting from complex interactions. The work in [84] is based on the two fundamentals notions of *Autonomic Components* (ACs) and *Autonomic-Component Ensembles* (ACEs), and defines programming abstractions to model their evolutions and their interactions. The authors define the *Software Component Ensemble Language* (SCEL) that is a programming language for programming the behavior of ACs and the formation of ACEs, and for controlling the interaction among different ACs. These abstractions permit describing autonomic systems in terms of *Behaviors*, *Knowledge*, and *Aggregations* by complying with specific *Policies*.

Overview of the SCEL's Design Principles – ACs and ACEs serve to structure systems into independent and distributed building blocks that interact and adapt.

ACs are entities with dedicated knowledge units and resources; awareness is guaranteed by providing them with information about their state and behavior via their knowledge repositories. These repositories also can be used to store and retrieve information about ACs' working environments, and thus can be exploited to adapt their behavior to the perceived changes. Each AC is equipped with an *interface*, consisting of a collection of *attributes*, describing the component's features such as identity, functionalities, spatial coordinates, group memberships, trust level, response time, and so on.

Attributes are used by the ACs to dynamically organize themselves into ACEs. Indeed, one of the main novelties of SCEL is the way groups of partners are selected for interaction and thus how ensembles are formed. Individual ACs can single out communication partners by using their identities, but partners can also be selected by taking advantage of the attributes exposed in the interfaces. Predicates over such attributes are used to specify the targets of communication actions, thus permitting a sort of *attribute-based* communication. In this way, the formation rule of ACEs is endogenous to ACs: members of an ensemble are connected by the interdependency relations defined through predicates. An ACE is therefore not a rigid fixed network but rather a highly flexible structure where ACs' linkages are dynamically established.

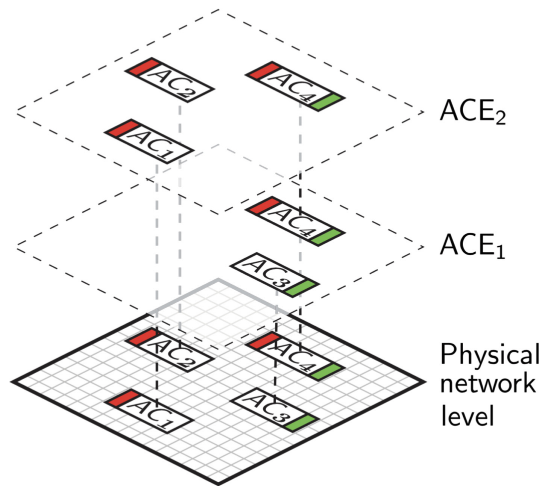

Fig. 1. Autonomic component ensembles

A typical scenario that gives rise to ACEs is reported in Fig. 1. It suggests that ACEs can be thought of as logical layers (built on top of the physical ACs' network) that identify dynamic ubnetworks of ACs by exploiting specific

attributes; in the picture, these are the different colours associated to individual ACs.

The main linguistic abstractions that SCEL provides developer with for programming the evolution and the interactions of ACs and the architecture of ACEs are listed as follows.

- *Behaviors* describe how computations may progress and are modeled as processes executing actions, in the style of process calculi.
- *Knowledge* repositories provide the high-level primitives to manage pieces of information coming from different sources. Each knowledge repository is equipped with operations for adding, retrieving, and withdrawing knowledge items.
- *Aggregations* describe how different entities are brought together to form ACs and to construct the software architecture of ACEs. Composition and interaction are implemented by exploiting the attributes exposed in ACs' interfaces.
- *Policies* control and adapt the actions of the different ACs for guaranteeing accomplishment of specific tasks or satisfaction of specific properties.

By accessing and manipulating their own knowledge repository or the repositories of other ACs, components acquire information about their status (self-awareness) and their environment (context awareness) and can perform self-adaptation, initiate self- healing actions to deal with system malfunctions, or install self-optimizing behaviors. All these self-* properties, as well as self-configuration, can be naturally expressed by exploiting SCEL's higher-order features, namely, the capability to store/retrieve (the code of) processes in/from the knowledge repositories and to dynamically trigger execution of new processes. Moreover, by implementing appropriate security policies (e.g., limiting information flow or external actions), components can set up self-protection mechanisms against different threats, such as unauthorised access or denial-of-service attacks.

Discussion – More on the **C**omponent side, the work by De Nicola et al. addresses the challenges to develop large systems composed of a massive numbers of components, featuring complex interactions among components, as well as with humans and other systems. These complex systems are often referred to as ensembles. The complexity of the ensembles is due to their large dimension and their need to adapt to the changes of the working environment and to the evolving requirements. Self-* abilities are thus desirable to make this kind of systems *autonomic*, hence capable to self-manage by continuously monitoring their behavior and context, and by selecting corrective actions if needed.

Due to such an inherent complexity, today's engineering methods and tools do not scale well, and new engineering techniques are needed to address the challenges of developing, integrating, and deploying them. As the blending of different concepts that have emerged in different fields of computer science and engineering, the work by De Nicola et al. proposes programming abstractions specific to autonomic system development, and reconcile them under a single and uniform formal semantics. Main advances brought by De Nicola et al. are (i) ability to deal with heterogenous systems and different application domains;

(ii) flexibility and suitability to support adaptive context-aware activities in pervasive and mobile computing scenarios together with transparent monitoring; (iii) strict relation with component-based design, which has been indicated as a key approach for adaptive software design; (iv) flexible and expressive forms of communication and adaptation that are adequate to deal with highly dynamic ensembles; (v) strict relation with context-oriented programming, which has been advocated to program autonomic systems.

3.2 Reo Connectors

Reo [4,6,7] is a dataflow-inspired language for incremental construction of complex connectors by composing simpler ones, with a graphical as well as a textual syntax [45]. Every Reo connector encapsulates a concrete interaction protocol. In contrast to traditional models of concurrency, where actions or processes constitute the basic building blocks, Reo espouses and advocates an *interaction-centric* model of concurrency, where the only first class primitive is *interaction*.

Reo views components in a concurrent system as black boxes, each of which has an interface consisting of a set of ports. A *port* is a uni-directional means of communication through which a component exchanges with its environment by performing blocking I/O operations *get* and *put*. Because a component has access to only its own ports and Reo offers no other means of inter-process communication, components cannot communicate with each other directly. Instead, a separate construct, a *connector*, connects to the ports of various components and mediates the flow of data amongst them. Every connector imposes the interaction protocols that it encapsulates upon the communication of the components, exogenously (from the outside of the components, which remain oblivious to the interaction protocol that engages them).

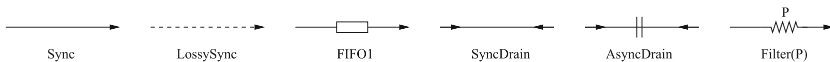

Fig. 2. A typical set of Reo channels

Overview of Reo – A complex connector in Reo is constructed as a graph whose edges comprise of primitive binary connectors, called *channels*, and whose vertices consist of particular synchronous dataflow components, called *nodes*.

A channel is a medium of communication that consists of two ends and a constraint on the dataflows observed at those ends. There are two types of channel ends: *source* and *sink*. A source channel end accepts data into its channel, and a sink channel end dispenses data out of its channel. Every channel (type) specifies its own particular behavior as constraints on the flow of data through its ends.

Although all channels used in Reo are user-defined and users can indeed define channels with any complex behavior (expressible in the semantic model)

that they wish, a very small set of channels, each with very simple behavior, suffices to construct useful Reo connectors with significantly complex behavior. Figure 2 shows a common set of primitive channels often used to build Reo connectors.

A `Sync` channel has a source and a sink end and no buffer. It accepts a data item through its source end iff it can simultaneously (i.e., atomically) dispense it through its sink.

A `LossySync` channel is similar to a synchronous channel except that it always accepts all data items through its source end. This channel transfers a data item if it is possible for the channel to dispense the data item through its sink end; otherwise the channel loses the data item. Thus, the context of (un)availability of a ready consumer at its sink end determines the (context-sensitive) behavior a `LossySync` channel.

A `FIFO1` channel represents an asynchronous channel with a buffer of capacity 1: it can contain at most one data item. When its buffer is empty, a `FIFO1` channel blocks I/O operations on its sink, because it has no data to dispense. It dispenses a data item and allows an I/O operation at its sink to succeed, only when its buffer is full, after which its buffer becomes empty. When its buffer is full, a `FIFO1` channel blocks I/O operations on its source, because it has no more capacity to accept the incoming data. It accepts a data item and allows an I/O operation at its source to succeed, only when its buffer is empty, after which its buffer becomes full.

More exotic channels are also permitted in Reo, for instance, synchronous and asynchronous *drains*. Each of these channels has two source ends and no sink end. No data value can be obtained from a drain channel because it has no sink end. Consequently, all data accepted by a drain channel are lost. `SyncDrain` is a synchronous drain that can accept a data item through one of its ends iff a data item is also available for it to simultaneously accept through its other end as well. `AsyncDrain` is an asynchronous drain that accepts data items through its source ends and loses them exclusively one at a time, but never simultaneously.

For a *filter channel*, or `Filter(P)`, its pattern $\mathtt{P} \subseteq \mathit{Data}$ specifies the type of data items that can be transmitted through the channel. This channel accepts a value $d \in \mathtt{P}$ through its source end iff it can simultaneously dispense d through its sink end, exactly as if it were a `Sync` channel; it always accepts all data items $d \notin \mathtt{P}$ through its source end and loses them immediately.

A Reo node is a logical place where channel ends coincide and coordinate their dataflows as prescribed by its *node type*. Figure 3 shows the three possible node types in Reo. A node is either *source*, *sink*, or *mixed*, depending on whether all channel ends that coincide on that node are source ends, sink ends, or a combination of the two. Reo fixes the semantics of (i.e., the constraints on the dataflow through) Reo nodes, as described below.

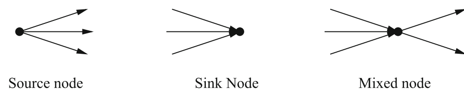

Fig. 3. Reo nodes

The source and sink nodes of a connector are collectively called its *boundary nodes*. Boundary nodes define the interface of a connector. Components attach their ports to the boundary nodes of a connector and interact anonymously with each other through the interface of the connector. Attaching a component to a (source or sink) node of a connector consists of the identification of one of the (respectively, output or input) ports of the component with that node. The blocking I/O operations performed by components on their own local ports, triggers dataflow through their attached connector nodes.

A component can write data items to a source node that it is attached to. The write operation succeeds only if all (source) channel ends coincident on the node accept the data item, in which case the data item is transparently written to every source end coincident on the node. A source node, thus, acts as a synchronous replicator. A component can obtain data items, by an input operation, from a sink node that it is attached to. A take operation succeeds only if at least one of the (sink) channel ends coincident on the node offers a suitable data item; if more than one coincident channel end offers suitable data items, one is selected nondeterministically. A sink node, thus, acts as a nondeterministic merger. A mixed node nondeterministically selects and takes a suitable data item offered by one of its coincident sink channel ends and replicates it into all of its coincident source channel ends. Note that a component cannot attach to, take from, or write to mixed nodes.

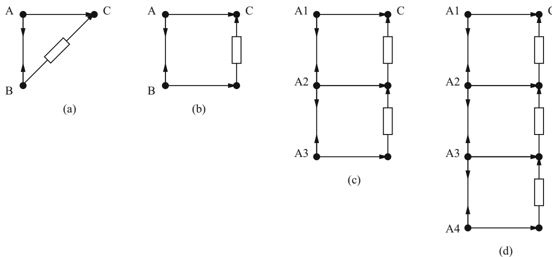

Fig. 4. Reo circuits for alternators

The connector shown in Fig. 4(a) is an *alternator* that imposes an ordering on the flow of the data from its input nodes A and B to its output node C. The `SyncDrain` enforces that data flow through A and B only synchronously (i.e., atomically). The empty buffer of the the `FIFO1` channel together with the `SyncDrain` guarantee that the data item obtained from A is delivered to C while the data item obtained from B is stored in the `FIFO1` buffer. After this, the buffer of the `FIFO1` is full and data cannot flow in through either A or B, but C can dispense the data stored in the `FIFO1` buffer, which makes it empty again. Thus, subsequent take operations at C obtain the data items written to $A, B, A, B, ...$, etc.

The connector in Fig. 4(b) has an extra `Sync` channel between node B and the `FIFO1` channel, compared to the one in Fig. 4(a). It is trivial to see that these two connectors have the exact same behavior. However, the structure of the connector in Fig. 4(b) allows us to generalize its alternating behavior to any number of producers, simply by replicating it and "juxtaposing" the top and the bottom `Sync` channels of the resulting copies, as seen in Fig. 4(c) and Fig. 4(d).

The connector in Fig. 4(d) is obtained by replicating the one in Fig. 4(b) 3 times. Following the reasoning for the connector in Fig. 4(c), it is easy to see that the connector in Fig. 4(d) delivers the data items obtained from $A1$, $A2$, $A3$,and $A4$ through C, in that order.

Semantics. Reo allows arbitrary user-defined channels as primitives; arbitrary mix of synchrony and asynchrony; and relational constraints between input and output. This makes Reo more expressive than, e.g., dataflow models, Kahn networks, synchronous languages, stream processing languages, workflow models, and Petri nets. On the other hand, it makes the semantics of Reo quite non-trivial. Various models have been developed to capture (various aspects of) the semantics of Reo, each to serve some specific purposes [55]. In this paper, we briefly describe only the ABTs [5], which constitute the primary formal semantics of Reo.

Formally, an ABT is a relation on a set of *time-data-streams*. ABTs yield an expressive compositional semantics for Reo where coinduction is the main definition and proof principle to reason about properties involving both data and time streams [14].

Discussion – Building upon earlier work on classical dataflow [15,44,56,57], synchronous languages [30,31,40,48], and Ptolemy [38,74], interaction-centric concurrency allows treatment of protocols as concrete objects of discourse. Besides Reo, more recent work, such as BIP [16,27], multiparty session types [51], Scribble [50,91], and Pabble [81,82] represent other examples of interaction-centric models that to various degrees of expressiveness and generality. Allowing arbitrary user-defined primitive building blocks and arbitrary mix of synchronous and asynchronous communication in its compositions, Reo relaxes restrictions and limitations implicit in most other above-mentioned models. See, for instance, [46] for an in-depth comparison of BIP and Reo.

The examples in Fig. 4 demonstrate how more complex connectors can be constructed by incremental composition of simpler ones. They also show how (1) arbitrary mix of synchrony and asynchrony, (2) preservation of synchrony through composition, which results in (3) propagation synchrony and exclusion though composition make Reo an expressive language. Several model checking tools are available for verification of Reo connectors [20–23,35,58–61,67–69].

A high-level language like Reo that supports this form of protocol specification offers clear software engineering advantages (e.g., programmability, maintainability, verbatim-reusability, verifiability, scalability, etc.). The results of on-going work on compiling Reo connectors suggest that smart optimizing compilers for high-level protocol languages can generate executable code with better performance than hand-crafted code produced by programmers written in contemporary general-purpose languages with constructs of traditional models of concurrency [54]. For some protocols, existing Reo compilers already generate code that can compete with code written by a competent programmer [53].

Reo has been used for composition of Web services [12,64,73], modeling and analysis of long-running transactions in service-oriented systems [66],

coordination of multi-agent systems [8], performance analysis of coordinated compositions [9,10,13,80], modeling of business processes and verification of their compliance [11,65,88], and modeling of coordination in biological systems [42].

3.3 Coordination Delegates

The work in [18] is based on the notion of *Coordination Delegate* and aims to formalize an automated synthesis method for the distributed composition and coordination of software services or, more in general, of software components. Following a modular and reuse-based approach, the business functionality of the system is assumed to be implemented by a set of software services, possibly black-box since provided by third parties. The system to be realized – out of the set of considered services – is specified as a global collaboration that the involved services have to realize by interacting via either synchronous or asynchronous message passing. This specification is given in the form of a state machine. Starting from this specification, and accounting for the specification of the interaction protocol performed by the involved services, the synthesis method is able to automatically generates a set of Coordination Delegates (CDs). CDs are additional software entities with respect to the participant services, and are synthesized in order to proxify and control their interaction. When interposed among the services by following the rules of a well-defined architectural style, the synthesized entities enforce the collaboration prescribed by the system specification. The synthesized CDs are correct by construction, meaning that the resulting distributed system realizes the specification. CDs are able to handle asynchrony through 1-bounded FIFO queues.

Overview of the Synthesis Method – Figure 5 shows an overview of the method for the automatic synthesis of CDs that, when interposed among the participant services, control those interactions that need coordination in order to enforce the realizability [28,29] of the specified global collaboration.

The method is organized into four steps that are performed in the following order: Projection, Selection, Synthesis, and Concretization.

1: Projection – It takes as input the system specification given in terms of a state machine where transitions model possibly simultaneous message exchanges among participants. As such, the system specification describes the way participants perform their interactions from a global perspective defining the (partial) order in which the message exchanges occur. Each single message exchange involves two participants: the sender and the receiver of the message. The specification abstracts from the way participants communicate to exchange messages, e.g., synchronous communication versus asynchronous one.

Out of the specification, Projection generates a behavioral model for each participant. This model is a state machine where transitions model (sets of possibly simultaneous) actions sending or receiving message, or actions internal to the participant that are not observable from outside. Message send and receive are, instead, observable actions. Simultaneous actions serve to deal with parallel

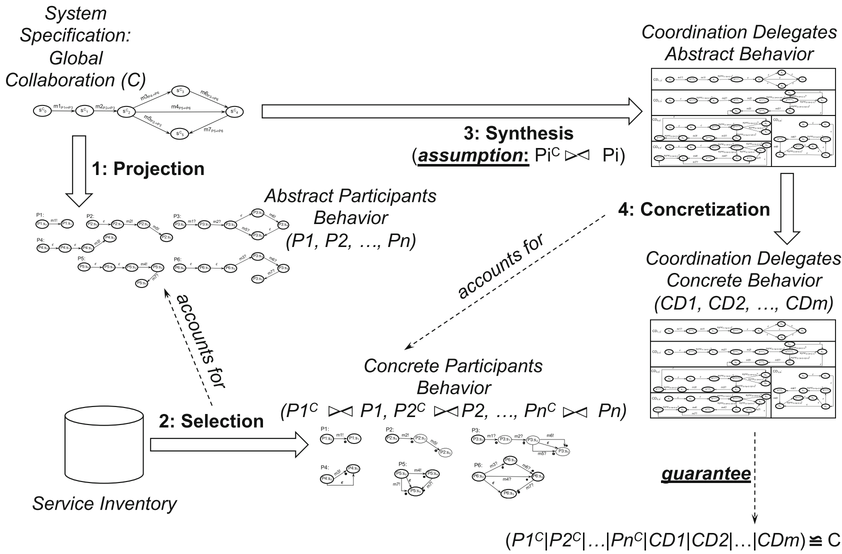

Fig. 5. Synthesis method overview

flows specified in the global collaboration and, hence, simultaneous executions. A projection represents the participant expected behavior according to the flows of message exchanges specified by the collaboration. Being derived from the system specification, also this model abstracts from the type of the send and receive actions (synchronous or asynchronous). We call this model Abstract Participant Behavior.

2: Selection – We recall that our approach is reuse-oriented, meaning that it allows to enforce system realizability in contexts in which the system is not implemented from scratch but it is realized by reusing, as much as possible, third-party services published in a Service Inventory. Services are selected from the inventory to play the roles of the abstract participants in the system specification. This calls for exogenous coordination of the selected concrete participants since, in general, we cannot access the participant code or change it.

A concrete service in the inventory comes with a behavioral specification of its interaction protocol. We call this model Concrete Participant Behavior. It is a state machine where transitions model (sets of possibly simultaneous) actions sending or receiving message, or internal actions. Similarly to the choreography specification, it can also specifies parallel flows that are joined afterwards. Differently from the Abstract Participant Behavior, for each transition, its type is specified: synchronous, asynchronous, or internal. That is, our approach does not impose constraints on the way concrete participants communicate, hence dealing with hybrid participants that can support both synchrony and asynchrony. For instance, a concrete participant could be a SOAP Web Service whose

WSDL[1] interface defines both Request/Response (synchronous interaction) and One-way operations (asynchronous interaction). In order to exchange messages asynchronously, concrete participants make use of bounded message queues. Our approach does not impose constraints on the size of the participants queues.

In order to select concrete participants that can suitably play the roles of abstract participants, our approach exploits a notion of behavioral refinement in order to automatically check whether the behavior of a concrete participant Pi^C is a refinement ($\bowtie$ in the figure) of the behavior of an abstract participant Pi. In the best case, for each abstract participant, a suitable concrete participant is found in the inventory. Otherwise, it might be the case that the set of selected participants covers a subset of the abstract participants in the specification. In this case, the abstract behavior of the remaining participants can support code generation activities to implement the missing concrete participants from scratch. Furthermore, the newly implemented concrete participant can be published in the inventory for possible future reuse.

An important consideration here is that, even in the case of limited reuse of third-party participants, our approach realizes separation of concern between the pure business logic implemented locally to each participant and the coordination logic needed for the realization of the global collaboration specified for the system. This logic is automatically generated as a set of CDs (Synthesis step). Keeping the needed coordination logic separated from the business one saves developers from writing code that goes beyond the development of the pure business logic internal to single participants. This allows developers to realize the specified system, without requiring any specific attention to what concerns coordination aspects. This aspect permits practitioners to develop the specified distributed system according to their daily development practices.

3 and 4: Synthesis and Concretization – The Synthesis step takes as input the system specification and automatically generates a set of CD Abstract Behavior models. Similarly to the Abstract Participant Behavior, each of them is a state machine where transitions model (sets of possibly simultaneous) actions sending or receiving message, or internal actions. These actions are related to the *standard* communication performed to achieve the choreography business logic. Differently from the Abstract Participant Behavior, there are also transitions modeling the synchronous exchange of coordination/synchronization messages. These actions model additional communication required to realize the coordination logic that is needed to enforce the realizability of the specified global collaboration. Standard communication takes place between a CD and the participant it controls and supervises, or directly among participants in case coordination is not required. When needed, additional communication messages are exchanged among the involved CD.

The Synthesis step is performed after a set of suitable concrete participants is obtained. Since the CD Abstract Behavior is generated out of the system specification, it abstracts from the way the supervised participants communicate (synchronously or asynchronously). This information will be added by the

[1] www.w3.org/TR/wsdl.

Concretization step that enriches the CD Abstract Behavior to achieve the so called CD Concrete Behavior.

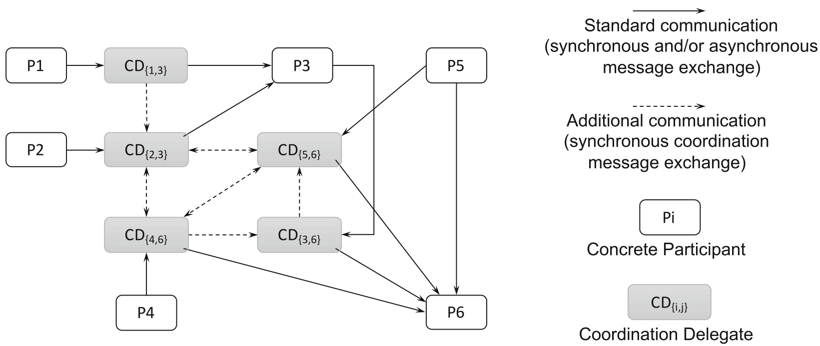

Fig. 6. Collaboration-based architectural style (a sample instance of)

For the set of synthesized CDs, correctness by construction means that when they are composed with the selected participants, the behavior of the resulting system realizes the specified global collaboration. That is, the generated CDs enforce by construction the realizability of the specified collaboration. Leveraging results on choreography realizability and its decidability from the work in [28,29], to correctly deal with asynchrony, the concrete CDs (Concretization step) in the controlled system make use of 1-bounded message queues. According to a predefined architectural style, CDs are interposed only among the participants needing coordination. Figure 6 shows an instance of the architectural style underlying our synthesis method.

CDs perform coordination (i.e., additional communication in the figure) of the participants interaction (i.e., standard communication in the figure) in a way that the resulting collaboration realizes the specified system. According to the type of actions performed by the concrete participants, standard communication can be synchronous or asynchronous. Additional communication is always synchronous. It is worth to note that CDs coordinate the interaction among the participants only when it is strictly needed for realizability enforcement purposes. That is, some participants are left free to communicate directly on those interactions that do not prevent the realizability of the specified global collaboration. Furthermore, depending on the specified collaboration, CDs do not necessarily require to be connected one to each other.

Discussion – Last but not least, on the **C**oordination side, the work by Inverardi et al. targets the development of reuse-based concurrent and distributed systems, from specification to composition and coordination code synthesis. The approach finds its most effective application in the distributed computing environment offered by the current Internet, which is increasingly populated by a virtually

infinite number of software services that can be opportunistically composed to realize more complex and powerful distributed applications.

According to John Musser, founder of the ProgrammableWeb[2], the production of application programming interfaces (APIs) growths exponentially and some companies are accounting for billions of dollars in revenue per year via API links to their services. The evolution of today Internet is expected to lead to an ultra large number of available services, hence increasing their number from 104 services on 2007 to billions of services in the near future. This situation radically changes the way software will be produced. Modern service-oriented systems will be more and more often built by reusing and assembling existing pieces of software, exposed through their APIs. Thus, the ability to automatically *compose* and *coordinate* these pieces of software enables the productive construction of innovative and revolutionary everyday-life scenarios within smart cities and related software ecosystems [76].

Most of the existing approaches to software composition are heavily based on central coordination. A centralized approach composes multiple components into a larger application, in which one component centrally coordinates the whole system interaction. The approach by Inverardi et al. permits to describe the interactions among the different system parties from a global perspective. It permits to model a peer-to-peer communication by defining a multiparty protocol that, when put in place by the cooperating parties, allows reaching the overall goal in a fully distributed way. In this sense, it differs significantly from a centralized approach, where all participants (but one) play the passive role of serving requests. Future software systems will be increasingly composed of active entities that, communicating peer-to-peer, proactively make decisions and autonomously perform tasks according to their own imminent needs and the emergent global collaboration. Each involved party knows exactly when to execute its operations and with whom to interact. The system execution becomes a collaborative effort focusing on the exchange of messages among several business participants to reach a common global goal. Thus, (i) the ability to reuse, compose and coordinate existing pieces of software are all basic ingredients to achieve this vision; (ii) automated supported is needed to realize correct-by-construction coordination logic.

4 Characteristics of Interest

The approaches presented in previous sections will be characterized in next section by using the six characteristics of interest defined in the following.

- **Compositionality:** this characteristic concerns the ability to compose a system in a hierarchical way out of simpler components/sub-systems and, roughly speaking, it does not matter the way we conduct this hierarchical construction, the result is always equivalent. This means that the system

[2] https://www.programmableweb.com.

construction process is based on a composition operator '$*$' that is *associative*, i.e., for all x, y, z then $x*(y*z) \equiv (x*y)*z$. Compositionality is crucial for system analysis purposes since it may improve the efficiency of the analysis.

- **Incrementality:** incrementality is implied by compositionality but the former does not imply the latter. This characteristic concerns hierarchical system construction. However, differently from compositionality, the associativity property is not required. Icrementality is another crucial aspect for system design purposes since it promotes reuse. It is implied by the existence of a composition operator that hides the internal details of the composition and exposes its observational (external) behavior.
- **Scalability:** also scalability is implied by compositionality and it refers to the ability for a composition to scale to systems with an increasing number of components (i.e., systems of systems).
- **Compositional reasoning:** this characteristic is related to compositionality but not necessarily. It refers to the ability to infer properties held by the *whole* by locally checking properties held by its *constituents*. This characteristic promotes efficient system analysis by performing local checks instead of a global one, hence facing complexity issues in some cases.
- **Reusability:** this characteristic concerns the reuse degree of components/ sub-systems. A sub-system can be: (i) reusable in any context (i.e., it is context-free), (ii) parameterized with respect to an abstract characterization of a set of contexts and, hence, reusable only in some contexts (i.e., it is partially context-free), or (iii) not reusable at all (i.e., it is not context-free) since it is tailored to a specific context.
- **Evolution:** this characteristic refers to the ability to express and deal with dynamicity and reconfiguration, two aspects that promote system evolution. It is related to programming constructs useful to model specific forms of system evolution.

Note that the characteristics of interest above must be considered to be general in nature and, as such, in the following are inflected in different ways and interpreted according to the purposes of the three approaches.

5 Matching the Characteristics of Interest

In this section, we characterize the approaches described in Sect. 3 with respect to the characteristics of interest introduced in Sect. 4. The results of the characterization are summarized in the tables below, and discussed just after in the following subsections.

We make use of "Yes", "No", and "Limited" to rank at a glance how the considered approaches match the characteristics of interest. Obviously, "Yes" and "No" are used to indicate that an approach enjoys or does not enjoy at all, respectively, the ability/property associated to the indicated characteristic. "Limited" is used to indicate either limited or constrained (i.e., if some assumptions hold) support for the indicated characteristic (Tables 1 and 2).

Table 1. Matching the characteristics of interest (Part 1)

Approach	Compositionality	Incrementality	Scalability
SCEL	Yes	Yes	Yes
Reo connectors	Yes	Yes	Yes
Coordination delegates	Yes	Yes	Yes

Table 2. Matching the characteristics of interest (Part 2)

Approach	Compositional reasoning	Reusability	Evolution
SCEL	Either Yes or Limited (to reachability properties)	Yes	Yes
Reo connectors	Yes	Yes	Yes
Coordination delegates	Yes	Limited (under refinement)	Limited (under variation points specification)

5.1 Software Component Ensemble Language Characterization

The main benefits of SCEL can be summarized as follows with respect to the characteristics of interest introduced in Sect. 4.

- ***Compositionality:*** as formalized in [84], SCEL builds systems by composing in parallel subsystems/components in a process algebra style. The parallel composition operator is both commutative and associative, hence directly achieving incrementality and compositionality.
- ***Incrementality:*** it is directly implied by compositionality.
- ***Scalability:*** systems programmed in SCEL are able to self-manage by continuously monitoring their behavior and their working environment and by selecting the actions to perform for best dealing with the current status of affairs. The self-* properties supported by SCEL allows developer to overcome typical scalability issues of ensembles, by improving their development, integration and deployment.
- ***Compositional reasoning:*** compositional reasoning is not explicitly discussed in [84]. However, it is shown that SCEL supports the verification of reachability properties such as checking the probability of reaching a configuration where a given predicate on collected data is satisfied within a given deadline. The fact that a SCEL system is built by means of an associative composition operator and its constituents are well-understood, independent and distributed suggests that compositional reasoning might be supported at least for such reachability properties.

- ***Reusability:*** SCEL supports different forms of reusability. It defines abstractions to program behaviors (ACs) and aggregations (ACEs) and its syntax is parametric with respect to knowledge and policies. Thus, reusability of ACs and ACEs with respect to different approaches to knowledge handling and policies specification is supported. Similarly to what is done by the object-oriented paradigm, SCEL components are exposed through their interface that allows developers to control the access to their internal knowledge, policies and processes. Thus, another form of reusability that is directly supported concerns the one achievable through subtyping. Furthermore, SCEL provides high-order features to store/retrieve the code of processes in/from the knowledge repositories and to dynamically trigger the execution of new processes.
- ***Evolution:*** as briefly discussed above, SCEL components are self-aware and context-aware and enjoy a number of self-* properties, e.g., they are capable to perform self-adaptation and self-reconfiguration. Thus, dynamic evolution is completely supported by the programming abstractions provided by the language.

5.2 Reo Connectors Characterization

For what concerns Reo, main benefits are as follows:

- ***Compositionality:*** Reo connectors are fully compositional. Starting with a set of (user-defined) primitive binary connectors—i.e., channels—Reo's composition rules, manifested as nodes, hierarchically construct more complex connectors. Examples in Figs. 3 and 4 demonstrate this property. Composition in Reo is associative.
- ***Incrementality:*** This property is impled by Reo's compositionality.
- ***Scalability:*** This property is impled by Reo's compositionality. Figure 4 serves as an example that demonstrates scalability.
- ***Compositional reasoning:*** Hiding the internal nodes of a connector, e.g., the exclusive router in Fig. 4(a), simplifies its semantics to the behavior of the connectors that is externally observable through its boundary nodes which comprise its interface, e.g., $\mathit{XRout}(\langle \alpha, a\rangle; \langle \beta, b\rangle, \langle \gamma, c\rangle)$, above. Once this behavior is verified, using this simplified semantcs avoids the need to repeat in-situ re-verification of its internal details whenever the connector is used as a sub-connector in a construction.
- ***Reusability:*** Figures 3 to 4 demonstrate verbatim reusability of Reo connectors.
- ***Evolution:*** Reo offers operations to dynamically reconfigure the topology of its connectors, thereby changing the interaction protocol of a running application. A semantic model for Reo cognizant of its reconfiguration capability, a logic for reasoning about reconfigurations, together with its model checking algorithm, are presented in [41]. Graph transformation techniques have been used in combination with the connector coloring model to formalize the dynamic reconfiguration semantics of Reo circuits triggered by dataflow [62,63,70,71].

5.3 Coordination Delegates Characterization

The main benefits of the synthesis method, and of using the notion of Coordination Delegate for distributed coordination, can be summarized as follows.

- ***Compositionality:*** the composition operator that is used to model the coordination logic of the controlled system, i.e., the parallel composition of the synthesized CDs, is based on a enhanced version of the synchronous product of LTSs that is able to deal with aptly defined synchronization messages (additional communication in Fig. 6) that are exchanged synchronously among the CDs in different ways (one-to-one, one-to-many, many-to-one, or many-to-many interactions). Compositionality can be straightforwardly achieved by making the non-synchronized communication observable from outside. This means that the coordination logic can be modeled by composing in parallel the CDs in an incremental way and it does not matter the order in which the composition is performed.
- ***Incrementality:*** it is directly implied by compositionality.
- ***Scalability:*** concerning the CDs synthesis method, the experimental example discussed in [18] show that: (i) the "performance" of the CDs scale, meaning that they are not affected when the number of system consumers increases; (ii) the time required for executing the needed distributed coordination logic is neglectable with respect to the overall collaboration execution time, hence confirming that the CDs enforces the specified global collaboration effectively and efficiently.
- ***Compositional reasoning:*** the coordination logic synthesized by the method supports compositional reasoning for verifying a global property of the controlled system by just performing local checks. Each check considers: (i) a projection (similarly to what is done in step 1) of the property with respect to a set of participants; and (ii) a projection of the coordination logic with respect to the same set of participants. Both (i) and (ii) provide local (to each participant) models of the property and local models of the coordination logic, respectively. Standard model-checking techniques can be then used to singularly check each projection of the property against the related projection of the coordination logic.
- ***Reusability:*** the abstract CDs as synthesized after step 3 (Synthesis) of the method are concrete services independent since they are generated by only looking at the global collaboration specification. This means that, as long as the interaction protocol of the selected concrete participants refines the one of the corresponding abstract participants in the specification, the generated abstract CDs can be reused and only their Concretization (step 4) need to be performed again. Thus, we can conclude that reusability is achieved, although in a limited form.
- ***Evolution:*** in another recent work [19] from the authors of [18], which is based on a slightly revised version of the summarized synthesis method, a novel global collaboration specification is presented where the designer can specify the so called *variation points*. They are points in the specification that

can be realized by alternative collaborations. Each alternative can be dynamically enabled/disabled during system execution depending on the "sensed" context. The CDs that are automatically synthesized out of this new specification are thus able to deal with this (limited) form of dynamic evolution by performing not only exogeneous distributed coordination, as already discussed above, but by also handling the enabling/disabling of the different specified alternatives.

6 Conclusions

Component, connector, and coordination have been key concepts exploited in different communities to manage the complexity of concurrent and distributed system development.

In this paper we discussed three approaches within three different classes: composition in distributed software architectures, exogenous coordination models, and programming abstractions for concurrent and autonomic systems. These classes encompass different perspectives and solutions to face crucial challenges in developing concurrent and distributed systems.

Our discussion considered previous work by De Nicola et al. about the SCEL language [84] for developing autonomic systems, and previous work by the authors about Reo connectors [5] for achieving exogenous coordination and distributed Coordination Delegates [18] for the distributed composition of heterogeneous components.

The approaches have been discussed with respect to some characteristics of interest for the above classes: compositionality, incrementality, scalability, compositional reasoning, reusability, and evolution.

All the three discussed approaches have been found to be representative for the three classes above since they support to some extent all the six dimension of interests for the engineering and development of concurrent and distributed systems.

References

1. Allen, R.: A formal approach to software architecture. Ph.D. thesis, Carnegie Mellon, School of Computer Science, January 1997. Issued as CMU Technical Report CMU-CS-97-144
2. Allen, R., Garlan, D.: A formal basis for architectural connection. ACM Trans. Softw. Eng. Methodol. **6**(3), 213–249 (1997)
3. Arbab, F.: What do you mean, coordination? Bulletin of the Dutch Association for Theoretical Computer Science (NVTI), 19 March 1998
4. Arbab, F.: Reo: a channel-based coordination model for component composition. Math. Struct. Comput. Sci. **14**(3), 329–366 (2004)
5. Arbab, F.: Abstract behavior types: a foundation model for components and their composition. Sci. Comput. Program. **55**(1–3), 3–52 (2005)

6. Arbab, F.: Puff, the magic protocol. In: Agha, G., Danvy, O., Meseguer, J. (eds.) Formal Modeling: Actors, Open Systems, Biological Systems. LNCS, vol. 7000, pp. 169–206. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24933-4_9
7. Arbab, F.: Proper protocol. In: Ábrahám, E., Bonsangue, M., Johnsen, E.B. (eds.) Theory and Practice of Formal Methods. LNCS, vol. 9660, pp. 65–87. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-30734-3_7
8. Arbab, F., Aştefănoaei, L., de Boer, F.S., Dastani, M., Meyer, J.-J., Tinnermeier, N.: Reo connectors as coordination artifacts in 2APL systems. In: Bui, T.D., Ho, T.V., Ha, Q.-T. (eds.) PRIMA 2008. LNCS (LNAI), vol. 5357, pp. 42–53. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-89674-6_8
9. Arbab, F., Chothia, T., Meng, S., Moon, Y.-J.: Component connectors with QoS guarantees. In: Murphy and Vitek [84], pp. 286–304
10. Arbab, F., Chothia, T., van der Mei, R., Meng, S., Moon, Y.-J., Verhoef, C.: From coordination to stochastic models of QoS. In: Field and Vasconcelos [49], pp. 268–287
11. Arbab, F., Kokash, N., Meng, S.: Towards using Reo for compliance-aware business process modeling. In: Margaria, T., Steffen, B. (eds.) ISoLA 2008. CCIS, vol. 17, pp. 108–123. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-88479-8_9
12. Arbab, F., Meng, S.: Synthesis of connectors from scenario-based interaction specifications. In: Chaudron, M.R.V., Szyperski, C., Reussner, R. (eds.) CBSE 2008. LNCS, vol. 5282, pp. 114–129. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-87891-9_8
13. Arbab, F., Meng, S., Moon, Y.-J., Kwiatkowska, M.Z., Qu, H.: Reo2MC: a tool chain for performance analysis of coordination models. In: van Vliet, H., Issarny, V. (eds.) ESEC/SIGSOFT FSE, pp. 287–288. ACM (2009)
14. Arbab, F., Rutten, J.J.M.M.: A coinductive calculus of component connectors. In: Wirsing, M., Pattinson, D., Hennicker, R. (eds.) WADT 2002. LNCS, vol. 2755, pp. 34–55. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-40020-2_2
15. Arvind, A., Gostelow, K.P., Plouffe, W.: Indeterminancy, monitors, and dataflow. In: Rosen, S., Denning, P.J. (eds.) Proceedings of the Sixth Symposium on Operating System Principles, SOSP 1977, Purdue University, West Lafayette, Indiana, USA, 16–18 November 1977, pp. 159–169. ACM (1977)
16. Attie, P., Baranov, E., Bliudze, S., Jaber, M., Sifakis, J.: A general framework for architecture composability. In: Giannakopoulou, D., Salaün, G. (eds.) SEFM 2014. LNCS, vol. 8702, pp. 128–143. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10431-7_10
17. Autili, M., Chilton, C., Inverardi, P., Kwiatkowska, M., Tivoli, M.: Towards a connector algebra. In: Margaria, T., Steffen, B. (eds.) ISoLA 2010. LNCS, vol. 6416, pp. 278–292. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16561-0_28
18. Autili, M., Inverardi, P., Tivoli, M.: Choreography realizability enforcement through the automatic synthesis of distributed coordination delegates. Sci. Comput. Program. **160**, 3–29 (2018)
19. Autili, M., Salle, A.D., Gallo, F., Pompilio, C., Tivoli, M.: On the model-driven synthesis of evolvable service choreographies. In: Proceedings of the 12th European Conference on Software Architecture: Companion Proceedings, ECSA 2018, Madrid, Spain, 24–28 September 2018, pp. 20:1–20:6 (2018)

20. Baier, C., Blechmann, T., Klein, J., Klüppelholz, S.: Formal verification for components and connectors. In: de Boer, F.S., Bonsangue, M.M., Madelaine, E. (eds.) FMCO 2008. LNCS, vol. 5751, pp. 82–101. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04167-9_5
21. Baier, C., Blechmann, T., Klein, J., Klüppelholz, S.: A uniform framework for modeling and verifying components and connectors. In: Field and Vasconcelos [49], pp. 247–267
22. Baier, C., Blechmann, T., Klein, J., Klüppelholz, S., Leister, W.: Design and verification of systems with exogenous coordination using Vereofy. In: Margaria, T., Steffen, B. (eds.) ISoLA 2010. LNCS, vol. 6416, pp. 97–111. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16561-0_15
23. Baier, C., Klein, J., Klüppelholz, S.: Modeling and verification of components and connectors. In: Bernardo, M., Issarny, V. (eds.) SFM 2011. LNCS, vol. 6659, pp. 114–147. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21455-4_4
24. Baier, C., Sirjani, M., Arbab, F., Rutten, J.J.M.M.: Modeling component connectors in Reo by constraint automata. Sci. Comput. Program. **61**(2), 75–113 (2006)
25. Baier, C., Wolf, V.: Stochastic reasoning about channel-based component connectors. In: Ciancarini, P., Wiklicky, H. (eds.) COORDINATION 2006. LNCS, vol. 4038, pp. 1–15. Springer, Heidelberg (2006). https://doi.org/10.1007/11767954_1
26. Basu, A., Bozga, M., Sifakis, J.: Modeling heterogeneous real-time components in BIP. In: SEFM 2006: Proceedings of the Fourth IEEE International Conference on Software Engineering and Formal Methods, Washington, DC, USA, pp. 3–12. IEEE Computer Society (2006)
27. Basu, A., Bozga, M., Sifakis, J.: Modeling heterogeneous real-time components in BIP. In: Proceedings of SEFM 2006, pp. 3–12. IEEE (2006)
28. Basu, S., Bultan, T.: Automated choreography repair. In: Stevens, P., Wąsowski, A. (eds.) FASE 2016. LNCS, vol. 9633, pp. 13–30. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49665-7_2
29. Basu, S., Bultan, T., Ouederni, M.: Deciding choreography realizability. In: POPL. ACM (2012)
30. Benveniste, A., Caspi, P., Le Guernic, P., Halbwachs, N.: Data-flow synchronous languages. In: de Bakker, J.W., de Roever, W.-P., Rozenberg, G. (eds.) REX 1993. LNCS, vol. 803, pp. 1–45. Springer, Heidelberg (1994). https://doi.org/10.1007/3-540-58043-3_16
31. Berry, G.: Esterel and Jazz: two synchronous languages for circuit design (abstract). In: Pierre, L., Kropf, T. (eds.) CHARME 1999. LNCS, vol. 1703, p. 1. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48153-2_1
32. Bhaduri, P., Ramesh, S.: Interface synthesis and protocol conversion. Form. Asp. Comput. **20**(2), 205–224 (2008)
33. Bidinger, P., Schmitt, A., Stefani, J.-B.: An abstract machine for the Kell calculus. In: Steffen, M., Zavattaro, G. (eds.) FMOODS 2005. LNCS, vol. 3535, pp. 31–46. Springer, Heidelberg (2005). https://doi.org/10.1007/11494881_3
34. Bidinger, P., Stefani, J.-B.: The Kell calculus: operational semantics and type system. In: Najm, E., Nestmann, U., Stevens, P. (eds.) FMOODS 2003. LNCS, vol. 2884, pp. 109–123. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-39958-2_8
35. Blechmann, T., Baier, C.: Checking equivalence for Reo networks. Electr. Notes Theor. Comput. Sci **215**, 209–226 (2008)
36. Bliudze, S., Sifakis, J.: The algebra of connectors - structuring interaction in BIP. IEEE Trans. Comput. **57**(10), 1315–1330 (2008)

37. Bruni, R., Lanese, I., Montanari, U.: A basic algebra of stateless connectors. Theor. Comput. Sci. **366**(1), 98–120 (2006)
38. Buck, J.T., Ha, S., Lee, E.A., Messerschmitt, D.G.: Ptolemy: a framework for simulating and prototyping heterogenous systems. Int. J. Comput. Simul. **4**(2), 155–182 (1994)
39. Carriero, N., Gelernter, D.: A computational model of everything. Commun. ACM **44**(11), 77–81 (2001)
40. Caspi, P., Pilaud, D., Halbwachs, N., Plaice, J.: LUSTRE: a declarative language for programming synchronous systems. In: Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages, Munich, Germany, 21–23 January 1987, pp. 178–188. ACM Press (1987)
41. Clarke, D.: A basic logic for reasoning about connector reconfiguration. Fundam. Inform. **82**(4), 361–390 (2008)
42. Clarke, D., Costa, D., Arbab, F.: Modelling coordination in biological systems. In: Margaria, T., Steffen, B. (eds.) ISoLA 2004. LNCS, vol. 4313, pp. 9–25. Springer, Heidelberg (2006). https://doi.org/10.1007/11925040_2
43. de Alfaro, L., Henzinger, T.A.: Interface automata. SIGSOFT Softw. Eng. Notes **26**(5), 109–120 (2001)
44. Dennis, J.B., Gao, G.R.: An efficient pipelined dataflow processor architecture. In: Michael, G.A. (ed.) Proceedings Supercomputing 1988, Orlando, FL, USA, 12–17 November 1988, pp. 368–373. IEEE Computer Society (1988)
45. Dokter, K., Arbab. F.: Treo: textual syntax for Reo connectors. In: Bliudze, S., Bensalem, S. (eds.) Proceedings of the 1st International Workshop on Methods and Tools for Rigorous System Design, MeTRiD@ETAPS 2018. EPTCS, Thessaloniki, Greece, 15th April 2018, vol. 272, pp. 121–135 (2018)
46. Dokter, K., Jongmans, S., Arbab, F., Bliudze, S.: Combine and conquer: relating BIP and Reo. J. Log. Algebr. Meth. Program. **86**(1), 134–156 (2017)
47. Fiadeiro, J.L., Lopes, A., Wermelinger, M.: A mathematical semantics for architectural connectors. In: Generic Programming, pp. 178–221 (2003)
48. Gautier, T., Le Guernic, P., Besnard, L.: SIGNAL: a declarative language for synchronous programming of real-time systems. In: Kahn, G. (ed.) FPCA 1987. LNCS, vol. 274, pp. 257–277. Springer, Heidelberg (1987). https://doi.org/10.1007/3-540-18317-5_15
49. Hirsch, D., Uchitel, S., Yankelevich, D.: Towards a periodic table of connectors. In: Ciancarini, P., Wolf, A.L. (eds.) COORDINATION 1999. LNCS, vol. 1594, p. 418. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48919-3_32
50. Honda, K., Mukhamedov, A., Brown, G., Chen, T.-C., Yoshida, N.: Scribbling interactions with a formal foundation. In: Natarajan, R., Ojo, A. (eds.) ICDCIT 2011. LNCS, vol. 6536, pp. 55–75. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-19056-8_4
51. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. In: Necula, G.C., Wadler, P. (eds.) Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, 7–12 January 2008, pp. 273–284. ACM (2008)
52. Inverardi, P., Tivoli, M.: Automatic synthesis of modular connectors via composition of protocol mediation patterns. In: 35th International Conference on Software Engineering, ICSE 2013, San Francisco, CA, USA, 18–26 May 2013, pp. 3–12 (2013)
53. Jongmans, S.-S., Halle, S., Arbab, F.: Reo: a dataflow inspired language for multicore. In: Proceedings of DFM 2013, pp. 42–50. IEEE (2014)

54. Jongmans, S.-S.T.: Automata-theoretic protocol programming: parallel computation, threads and their interaction, optimized compilation, [at a] high level of abstraction. Ph.D. thesis, Leiden University (2015, submitted)
55. Jongmans, S.-S.T., Arbab, F.: Overview of thirty semantic formalisms for Reo. Sci. Ann. Comput. Sci. **22**(1), 201–251 (2013)
56. Kahn, G.: The semantics of a simple language for parallel programming. In: Rosenfeld, J.L. (ed.) Information Processing, Stockholm, Sweden, pp. 471–475. North Holland, Amsterdam, August 1974
57. Kahn, G., MacQueen, D.B.: Coroutines and networks of parallel processes. In: IFIP Congress, pp. 993–998 (1977)
58. Kemper, S.: SAT-based verification for timed component connectors. Electr. Notes Theor. Comput. Sci. **255**, 103–118 (2009)
59. Kemper, S.: Compositional construction of real-time dataflow networks. In: Clarke, D., Agha, G. (eds.) COORDINATION 2010. LNCS, vol. 6116, pp. 92–106. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-13414-2_7
60. Klein, J., Klüppelholz, S., Stam, A., Baier, C.: Hierarchical modeling and formal verification. An industrial case study using Reo and Vereofy. In: Salaün, G., Schätz, B. (eds.) FMICS 2011. LNCS, vol. 6959, pp. 228–243. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24431-5_17
61. Klüppelholz, S., Baier, C.: Symbolic model checking for channel-based component connectors. Electr. Notes Theor. Comput. Sci **175**(2), 19–37 (2007)
62. Koehler, C., Arbab, F., de Vink, E.: Reconfiguring distributed Reo connectors. In: Corradini, A., Montanari, U. (eds.) WADT 2008. LNCS, vol. 5486, pp. 221–235. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03429-9_15
63. Koehler, C., Costa, D., Proença, J., Arbab, F.: Reconfiguration of Reo connectors triggered by dataflow. In: Ermel, C., Heckel, R., de Lara, J. (eds.) Proceedings of the 7th International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2008), vol. 10, pp. 1–13 (2008). ECEASST. ISSN 1863-2122. http://www.easst.org/eceasst/
64. Koehler, C., Lazovik, A., Arbab, F.: ReoService: coordination modeling tool. In: Krämer et al. [72], pp. 625–626
65. Kokash, N., Arbab, F.: Formal behavioral modeling and compliance analysis for service-oriented systems. In: de Boer, F.S., Bonsangue, M.M., Madelaine, E. (eds.) FMCO 2008. LNCS, vol. 5751, pp. 21–41. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04167-9_2
66. Kokash, N., Arbab, F.: Formal design and verification of long-running transactions with extensible coordination tools. IEEE Trans. Serv. Comput. **6**(2), 186–200 (2013)
67. Kokash, N., Krause, C., de Vink, E.: Data-aware design and verification of service compositions with Reo and mCRL2. In: SAC 2010: Proceedings of the 2010 ACM Symposium on Applied Computing, pp. 2406–2413. ACM, New York (2010)
68. Kokash, N., Krause, C., de Vink, E.P.: Verification of context-dependent channel-based service models. In: de Boer, F.S., Bonsangue, M.M., Hallerstede, S., Leuschel, M. (eds.) FMCO 2009. LNCS, vol. 6286, pp. 21–40. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17071-3_2
69. Kokash, N., Krause, C., de Vink, E.P.: Time and data-aware analysis of graphical service models in Reo. In: Fiadeiro, J.L., Gnesi, S., Maggiolo-Schettini, A. (eds.) SEFM, pp. 125–134. IEEE Computer Society (2010)
70. Krause, C.: Reconfigurable component connectors. Ph.D. thesis, Leiden University (2011). https://openaccess.leidenuniv.nl/handle/1887/17718

71. Krause, C., Maraikar, Z., Lazovik, A., Arbab, F.: Modeling dynamic reconfigurations in Reo using high-level replacement systems. Sci. Comput. Program. **76**(1), 23–36 (2011)
72. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM **21**(7), 558–565 (1978)
73. Lazovik, A., Arbab, F.: Using Reo for service coordination. In: Krämer et al. [72], pp. 398–403
74. Liu, X., Xiong, Y., Lee, E.A.: The Ptolemy II framework for visual languages. In: 2002 IEEE CS International Symposium on Human-Centric Computing Languages and Environments (HCC 2001), Stresa, Italy, 5–7 September 2001, p. 50. IEEE Computer Society (2001)
75. Lopes, A., Wermelinger, M., Fiadeiro, J.L.: Higher-order architectural connectors. ACM Trans. Softw. Eng. Methodol. **12**(1), 64–104 (2003)
76. Manikas, K., Hansen, K.M.: Software ecosystems - a systematic literature review. J. Syst. Softw. **86**(5), 1294–1306 (2013)
77. Milner, R.: A Calculus of Communicating Systems. Springer, New York (1982)
78. Milner, R.: Communicating and Mobile Systems: The Pi-Calculus. Cambridge University Press, Cambridge (1999)
79. Milner, R.: The Space and Motion of Communicating Agents. Cambridge University Press, New York (2009)
80. Moon, Y.-J., Silva, A., Krause, C., Arbab, F.: A compositional semantics for stochastic Reo connectors. In: Mousavi, M.R., Salaün, G. (eds.) FOCLASA. EPTCS, vol. 30, pp. 93–107 (2010)
81. Ng, N., de Figueiredo Coutinho, J.G., Yoshida, N.: Protocols by default - safe MPI code generation based on session types. In: Franke, B. (ed.) CC 2015. LNCS, vol. 9031, pp. 212–232. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46663-6_11
82. Ng, N., Yoshida, N.: Pabble: parameterised scribble for parallel programming. In: 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2014, Torino, Italy, 2–14 February 2014, pp. 707–714. IEEE Computer Society (2014)
83. Nicola, R.D., Duong, T., Inverso, O., Trubiani, C.: AErlang: empowering Erlang with attribute-based communication. Sci. Comput. Program. **168**, 71–93 (2018)
84. De Nicola, R., Loreti, M., Pugliese, R., Tiezzi, F.: A formal approach to autonomic systems programming: the SCEL language. ACM Trans. Auton. Adapt. Syst. **9**(2), 7:1–7:29 (2014)
85. Papadopoulos, G.A., Arbab, F.: Coordination models and languages. Adv. Comput. **46**, 329–400 (1998)
86. Roscoe, A.W.: The Theory and Practice of Concurrency. Prentice-Hall Inc., New York (1998)
87. Schmitt, A., Stefani, J.-B.: The Kell calculus: a family of higher-order distributed process calculi. In: Priami, C., Quaglia, P. (eds.) GC 2004. LNCS, vol. 3267, pp. 146–178. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-31794-4_9
88. Schumm, D., Turetken, O., Kokash, N., Elgammal, A., Leymann, F., van den Heuvel, W.-J.: Business process compliance through reusable units of compliant processes. In: Daniel, F., Facca, F.M. (eds.) ICWE 2010. LNCS, vol. 6385, pp. 325–337. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16985-4_29

89. Sifakis, J.: A framework for component-based construction extended abstract. In: SEFM 2005: Proceedings of the Third IEEE International Conference on Software Engineering and Formal Methods, Washington, DC, USA, pp. 293–300. IEEE Computer Society (2005)
90. Stefani, J.-B.: A calculus of Kells. Electr. Notes Theor. Comput. Sci. **85**(1), 40–60 (2003)
91. Yoshida, N., Hu, R., Neykova, R., Ng, N.: The scribble protocol language. In: Abadi, M., Lluch Lafuente, A. (eds.) TGC 2013. LNCS, vol. 8358, pp. 22–41. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-05119-2_3