Abstract Behavior Types: A Foundation Model for Components and Their Composition

Farhad Arbab

CWI, Amsterdam, The Netherlands
farhad@cwi.nl

Abstract. The notion of Abstract Data Type (ADT) has served as a foundation model for structured and object oriented programming for some thirty years. The current trend in software engineering toward component based systems requires a foundation model as well. The most basic inherent property of an ADT, i.e., that it provides a set of operations, subverts some highly desirable properties in emerging formal models for components that are based on the object oriented paradigm.
We introduce the notion of Abstract Behavior Type (ABT) as a higher-level alternative to ADT and propose 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. The ABT model supports a much looser coupling than is possible with the ADT's operational interface, and is inherently amenable to exogenous coordination. We propose that both of these are highly desirable, if not essential, properties for models of components and their composition.
To demonstrate the utility of the ABT model, we describe Reo: an exogenous coordination language for compositional construction of component connectors based on a calculus of channels. We show the expressive power of Reo, and the applicability of ABT, through a number of examples.

1 Introduction

An Abstract Data Type (ADT) defines an algebra of operations with mathematically well-defined semantics, without specifying any detail about the implementation of those operations or the data structures they operate on to realize them. As such, ADT is a powerful abstraction and encapsulation mechanism that groups data together with their related operations into logically coherent and loosely-dependent entities, such as objects, yielding better structured programs. ADT has served as a foundation model for structured and object oriented programming for some thirty years.

The immense success of object oriented techniques has distracted proper attention away from critical evaluation of some of its underpinning concepts from the perspective of their utility for components. We propose that the most basic inherent property of an ADT, i.e., that it provides a set of operations in its interface, subverts some highly desirable properties in emerging models for component based systems. This is already evident in the current attempts at extending the object oriented models into the realm of components.

We introduce the notion of Abstract Behavior Type (ABT) as a higher-level alternative to ADT and propose 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. In contrast with the algebraic underpinnings of the ADT model, the (generally) infinite streams that are the elements of behavior in the ABT model naturally lend themselves to the coalgebraic techniques and the coinduction reasoning principle that have recently been developed as a general theory to describe the behavior of dynamic systems. The ABT model supports a much looser coupling than is possible with ADT and is inherently amenable to exogenous coordination. We propose that both of these are highly desirable, if not essential, properties for components and their composition.

In our view, a component based system consists of component instances and their connectors (i.e., the "glue code"), both of which are uniformly modeled as ABTs. Indeed, the only distinction between a component and a connector is just that a component is an atomic ABT whose internal structure is unknown, whereas a connector is known to be an ABT that is itself composed out of other ABTs. As a concrete instance of the application of the ABT model, we describe Reo: an exogenous coordination model wherein complex coordinators, called "connectors" are compositionally built out of simpler ones [2, 3]. Reo can be used as a glue language for compositional construction of connectors that orchestrate component instances in a component based system. We demonstrate the surprisingly expressive power of ABT composition in Reo through a number of examples.

The rest of this paper is organized as follows. In Section 2 we motivate our view of components and their composition as a conceptual model at a higher level of abstraction than objects and their composition. Section 3 contains a brief overview of some related work. We review the formal notion of abstract data types in Section 4, and elaborate on its links with and implications on object oriented programming in Section 5. We argue that some of these implications impede the ability of component models based on the object oriented paradigm to support flexible composition and exogenous coordination, both of which, we propose, are highly desirable properties in component based systems. Section 6 is an informal description of our component model, and in Section 7 we describe its accompanying model of behavior. Section 8 is an introduction to Abstract Behavior Types and their composition. In Section 9 we show how channels, connectors, and their composition in Reo are easily expressed as ABTs and their composition. Finally, we close with our concluding remarks in Section 10.

2 A Component Manifesto

The bulk of the work on component based systems is primarily focused on what components are and how they are to be constructed. Relatively little attention has been paid to alternative models and languages for *composing* components into (sub)systems, which is typically considered to be the purpose of the so-called *glue code*, assumed to be written in some scripting language. Clearly, components and their composition are not independent of one another: explicitly emphasizing one defines or at least constrains the other as well, if only implicitly.

A conspicuous driving force behind the upsurge of interest and activity in component based software is the recognition that the object oriented paradigm is not the silver-bullet that some of its over-zealous advocates purported it to be. Nevertheless, presently, the dominant view of what components are or should be reflects a prominent object oriented legacy: components are fortified collections of classes and/or objects, with very similar interfaces. It follows that the interactions among and the composition of components must use mechanisms very similar to those for interactions among and composition of classes and objects. Thus, the method invocation semantics of message passing in object oriented programming becomes the crux of the component composition mechanisms in scripting languages.

This approach to components "solves" some of the problems that are rooted in the inadequacies of the object oriented paradigm simply by shifting them elsewhere. For instance, the relatively tight coupling that must be established between a caller and a callee pair of objects indeed disappears as a concern at the intra-component level when the two objects reside in different component instances, but becomes an issue to be addressed in the glue code and its underlying middleware used to compose those components. As long as components and their interfaces are essentially the same as objects and their interfaces, the (scripting) programs that constitute the glue code end up to be inherently no different than other object oriented software. In complex systems, the body of such specialized glue code can itself grow in size, complexity, intricacy, fragility, and rigidity, rendering the system hard to evolve and maintain, in spite of the fact that this inflexible code wraps and connects otherwise reusable, upgradeable, and replaceable components.

An alternative view of components emerges if we momentarily ignore how they are made or even what they are made of, emphasizing instead what we want to do with them. Beyond fashionable jargon, hype, and merely technical idiosyncrasies, if there is to be any conceptual substance behind the term "component" deserving its minting, it must be that components are less interdependent and are easier and more flexible to compose than objects and classes. The definition of a class or an object specifies the methods it offers to other entities, and the method calls within the code of its methods determine the services and entities it requires to work. This results in a rather tight semantic interdependence among objects/classes and grants each individual a significant degree of control over precisely how it is composed with other classes or objects.

In contrast to objects and classes, it is highly desirable for components to be semantically independent of one another and internally impose no restrictions on the other components they compose with. This yields a level of composition flexibility that is not possible with objects and classes[1] and which is a prerequisite for another highly desirable property in component based systems: we would like for the whole (system) to be more than the mere sum of its (component) parts. This implies that not only it should be generally possible to produce different systems by composing the same set of components in different ways, but also that the difference between two systems composed out of the same set of components (i.e., the difference between the "more" than the "sum of the parts" in each system) must arise out of the actual rules that comprise their two different compositions, i.e., their glue code. The significance of the latter point is that it requires the glue code to contribute to the semantics of the whole system well beyond the mere so-called "wiring-standard-level" support provided by the current popular middleware and component based technologies. On the other hand, we intuitively expect glue code to be void of any application-domain specific functionality: its job is merely to connect components, facilitating their communication and coordinating their interactions, not to perform any application-domain specific computation.

This leads to a subtlety regarding the interaction between glue code and components which fundamentally impacts both. If the contribution of the glue code to the behavior of a composed system is no more than connecting its components, facilitating their communication and coordinating their interactions, then the difference between the behavior of two systems composed out of the same set of components can arise not out of any application-domain specific computation (and certainly not out of the components), but only out of how the glue code connects and coordinates these components to interact with one another. Since glue code is external to the components it connects, this implies that (1) the components must be amenable to external coordination control and (2) the glue code must contain constructs to provide such external coordination. The first implication constrains the mechanisms through which components can interact with their environment. The second implication means that the glue code language must incorporate an *exogenous* coordination model [1].

Finally, if glue code is to have its own non-trivial semantics in a composed system, it is highly desirable both for the glue code itself to be piece-wise explicitly identifiable, and for the semantics of each of its pieces to be independent of the semantics of the specific components that it composes. This promotes the recognition of the glue code as an identifiable, valuable software commodity, emphasizes the importance of its reusability, and advocates glue code construction through composition of reusable glue code pieces.

[1] Observe that generally speaking, it is the code for the methods of an object that determines the other objects it "composes with" to function properly. Thus, objects/classes "decide for themselves" how they compose with each other and their composition generally cannot be determined or influenced from outside.

The notion of compositional construction of glue code out of smaller, reusable pieces of glue code all but eliminates the conceptual distinctions between components and glue code. This behooves us to find conceptual models and formal methods for component based systems wherein the same rules for compositional construction indiscriminately apply to both components as well as their glue code connectors. In such a model, the (perhaps somewhat subjective) distinction between components and their (pieces of glue code) connectors still makes practical sense: although they are indistinguishable when used as primitives to compose more complex constructs, components and connectors are still different in that components are black-box primitives whose internal structures are invisible, whereas the internal structure of a connector shows that it, in turn, is constructed out of other (connector and/or component) primitives according to the same rules of composition.

3 Background and Related Work

In popular models of components (e.g., Enterprise Java Beans [34, 16], CORBA [48, 14], and DCOM [19]) component instances are fortified (collections of) objects. Consequently, they typically use variants of message passing with the semantics of method invocation for inter-component communication. The tight coupling inherent in the method call semantics is more appropriate for intra-component communication. In contrast, inter-component communication invariably requires a minimum level of "control from the outside" of the participating components. In order to break the tight coupling induced by the method call semantics and reduce the interdependence of components on each other, the underlying middleware that supports these component models provides mechanisms or entities (such as the ORB in CORBA) to intercept inter-component messages. Messages may be intercepted to, for instance, provide services (e.g., binding and name servers), enforce imposed constraints (e.g., suppress certain messages in certain states), ensure protocols, and/or enact assigned roles. One way or the other, the middleware's intervention loosens the otherwise tight coupling that would be imposed by targeted active messages (i.e., messages with method-invocation semantics) and furthermore, enforces a certain restricted form of coordination from outside the components.

Coordination languages [17, 40] offer an alternative for inter-component communication, as exemplified by JavaSpaces in the Jini architecture [29, 39, 33]. They impose a stricter sense of temporal and spacial decoupling that supports a looser inter-component semantic dependency, compared with the method invocation semantics of message passing in object oriented paradigms.

Most common component models define components as reusable binary units of software with interfaces that have no more than a syntactic content. This view of components enforces information hiding in only a rather primitive way: the good practice discipline of using questionably suggestive symbolic names in component interfaces non-withstanding, such an interface does not reveal any of the externally relevant semantics of the contents of its component. Such component models cannot support (semi-)formal specification/verification of their external behavior.

A broader definition of components is offered by the Eiffel language [35, 36, 18]: components are client-oriented software with the desirable property that a component, x, can be used by other programs that do not need to be known to x. This property is supported in Eiffel through formal specification techniques which include pre- and post-conditions and invariants. In general, this notion of components requires enhanced specification and verification techniques, as also observed by Hennicker and Wirsing [50, 22].

Our notion of components [8, 5, 15] uses channels as the basic inter-component communication mechanism. A channel is a point-to-point medium of communication with its own unique identity and two distinct ends. A channel supports transfer of passive data only; no transfer of control (e.g., procedure calls, exchange of pointers to internal objects/entities) can take place through a channel. Using channels as the only means of inter-component communication allows a clean, flexible, and expressive model for construction of the glue code for component composition which also supports exogenous coordination.

Our notion of channel is very general and we specifically allow a variety of different channel types (even user-defined ones) to be used simultaneously and composed together. This is different than the way channels are used in virtually all other channel-based models, which typically allow only one or at most a small number of simple predefined channel types. Specifically, our liberal notion of channels and the potency that our model derives from mixing and composing channels of different types are in sharp contrast with the use of channels in the Ptolemy project [13, 32, 31] which ascribes a single interpretation for its connecting channels in each context.

Asynchronous channels form the basis of the dataflow architecture for networks of components as proposed and formally investigated by Broy and his group [11, 23]. In this architectural model, large systems can be realized, allowing programmers to easily understand the input/output behavior of a system as the composition of the behavior of its individual components. Our model of component composition is fundamentally different than (even dynamic) dataflow models because it (1) supports a much wider and more general notion of channels and different channel types; and (2) introduces the notion of channel composition as the construct through which channels are connected to other channels, forming higher level and more sophisticated connectors for component composition.

4 Abstract Data Types

Formally, an ADT is a triplet $\langle \mathcal{S}, \mathcal{O}, \mathcal{A} \rangle$, where $\mathcal{S}$ is a set of *sorts* denoting the required types, $\mathcal{O}$ is a set of *operators* over $\mathcal{S}$, and $\mathcal{A}$ is a set of *axioms* written as algebraic equations defining the results of various combinations of operations in $\mathcal{O}$ on data items of various types in $\mathcal{S}$.

	Stack		Queue
$\mathcal{S}$:	$\mathit{stack}, \mathit{data}, \mathit{boolean}$	$\mathcal{S}$:	$\mathit{queue}, \mathit{data}, \mathit{boolean}$
$\mathcal{O}$:	$\mathit{top}(s) \rightarrow d$	$\mathcal{O}$:	$\mathit{first}(q) \rightarrow d$
	$\mathit{pop}(s) \rightarrow s$		$\mathit{deq}(q) \rightarrow q$
	$\mathit{push}(s, d) \rightarrow s$		$\mathit{enq}(q, d) \rightarrow q$
	$\mathit{empty}(s) \rightarrow b$		$\mathit{empty}(q) \rightarrow b$
$\mathcal{A}$:	$\mathit{empty}(\lambda) = \mathit{true}$	$\mathcal{A}$:	$\mathit{empty}(\lambda) = \mathit{true}$
	$\mathit{empty}(\mathit{push}(s, d)) = \mathit{false}$		$\mathit{empty}(\mathit{enq}(q, d)) = \mathit{false}$
	$\mathit{top}(\mathit{push}(s, d) = d$		$\mathit{first}(\mathit{enq}(\lambda, d)) = d$
	$\mathit{pop}(\mathit{push}(s, d) = s$		$\mathit{first}(\mathit{enq}(\mathit{enq}(q, d_1), d_2)) = \mathit{first}(\mathit{enq}(q, d_1))$
	$\mathit{pop}(\lambda) = \epsilon_1$		$\mathit{deq}(\mathit{enq}(\lambda, d)) = \lambda$
	$\mathit{pop}(\epsilon_1) = \epsilon_1$		$\mathit{deq}(\mathit{enq}(\mathit{enq}(q, d_1), d_2)) = \mathit{enq}(\mathit{deq}(\mathit{enq}(q, d_1), d_2)$
	$\mathit{top}(\lambda) = \epsilon_2$		$\mathit{deq}(\lambda) = \epsilon_1$
			$\mathit{deq}(\epsilon_1) = \epsilon_1$
			$\mathit{first}(\lambda) = \epsilon_2$

Fig. 1. Abstract Data Types for stack and queue

For example, Figure 1 shows the formal ADT definitions for the two common data types stack and queue, in separate columns. The set $\mathcal{S}$ contains *stack*, *data*, and *boolean* types for stack, and *queue*, *data*, and *boolean* types for queue. We use s, q, d, and b to represent items of types *stack*, *queue*, *data*, and *boolean*, respectively. Furthermore, in the stack column in this figure λ is an item of type *stack* representing the empty stack, and likewise in the queue column λ is an item of type *queue* representing the empty queue. Similarly, in each column ϵ_1 and ϵ_2 are special error values of their respective types.

The set $\mathcal{O}$ in each column defines the signature of four operations. For the case of the stack, $\mathit{top}(s)$ is expected to produce the data item at the top of the stack s; $\mathit{pop}(s)$ is expected to produce the stack obtained by removing the data item at the top of the stack s; $\mathit{push}(s, d)$ is expected to produce a stack obtained by pushing the data item d on top of the stack s; and $\mathit{empty}(s)$ is expected to produce a boolean indicating whether or not the stack s is empty. For the case of queue, $\mathit{first}(q)$ is expected to produce the first data item at the head of the queue q; $\mathit{deq}(q)$ is expected to produce the queue obtained by removing the first data item at the head of the queue q (*dequeue*); $\mathit{enq}(q, d)$ is expected to produce a queue obtained by adding the data item d to the tail end of the queue q (*enqueue*); and $\mathit{empty}(q)$ is expected to produce a boolean indicating whether or not the queue q is empty. Of course, the set $\mathcal{O}$ contains only the signatures of these operations and as such it is void of any formal hint of what they (are expected to) do.

It is the set of axioms, $\mathcal{A}$, that formally defines the semantics of the operations in $\mathcal{O}$ in terms of their mutual effects on each other. In the case of the stack, the two axioms for the *empty* operation state that (1) $\mathit{empty}(\lambda) = \mathit{true}$, and (2) *empty* applied to a stack obtained from a *push* operation on any stack yields *false*. The *top* axioms state that (1) *top* applied to the empty stack yields an error (ϵ_2), and (2) *top* applied to a stack obtained from pushing the data item d onto some other stack, yields d. The *pop* axioms state that (1) popping a stack obtained from pushing a data item onto some other stack, s, yields s; (2) popping an empty stack yields an error (ϵ_1); and (3) popping this error value yields the same error value. Any stack is canonically represented as a sequence of *push* operations that add data items on the result of their preceding *push*, starting with the empty stack, e.g., $push(push(push(push(\lambda, d_1), d_2), d_3), d_4)$.

Many of the queue axioms are analogous to their respective stack axioms. The axioms for *first* and *deq* are a bit more interesting. Any queue is canonically represented as a sequence of *enq* operations that add data items on the result of their preceding *enq*, starting with the empty queue; e.g., $enq(enq(enq(enq(\lambda, d_1),$ $d_2), d_3), d_4)$. The *first* axioms state that to find the first element in a queue, we must "peel" it away until we reach the empty queue, at which point we obtain the first data item at the head of the queue. Thus:

$$\begin{aligned} &first(enq(enq(enq(enq(\lambda, d_1), d_2), d_3), d_4)) \\ &\quad = first(enq(enq(enq(\lambda, d_1), d_2), d_3)) \\ &\quad = first(enq(enq(\lambda, d_1), d_2)) \\ &\quad = first(enq(\lambda, d_1)) \\ &\quad = d_1 \end{aligned}$$

Analogously, *deq* peels away the canonical representation of a queue, but it also reconstructs it as it moves inside. For instance:

$$\begin{aligned} &deq(enq(enq(enq(enq(\lambda, d_1), d_2), d_3), d_4)) \\ &\quad = enq(deq(enq(enq(enq(\lambda, d_1), d_2), d_3)), d_4) \\ &\quad = enq(enq(deq(enq(enq(\lambda, d_1), d_2)), d_3), d_4) \\ &\quad = enq(enq(enq(deq(enq(\lambda, d_1)), d_2), d_3), d_4) \\ &\quad = enq(enq(enq(\lambda, d_2), d_3), d_4) \end{aligned}$$

These examples show that an ADT defines a data type in terms of the operations on that data type and how they mutually affect each other. It abstracts away from the implementation of those operations and the data structures they manipulate. The semantics of an ADT is given as algebraic equations. The strong conceptual link between abstract data types and object oriented programming stems from the common manner in which they associate data and the operations that manipulate them together. The ADT for a type, T, defines all operations applicable to entities of type T. It encapsulates the representation of T and the implementation of its operations. This prevents manipulation of the entities of type T in any way other than through its own defined operations.

5 ADT and Object Oriented Programming

Their common aspiration to (1) encapsulate data structures behind operations that manipulate them, and (2) hide the details of those operations as well, has made ADT a suitable foundation model for object oriented programming. An ADT can be seen as a formal description of the interface of an object/class. This encapsulation significantly loosens the coupling between the implementation of an ADT (or object/class) and other code that can use it only through its prescribed operations. The operational interface of an ADT (or object/class) also readily supports extensibility in the form of polymorphism. Extensibility in object oriented programming typically goes beyond mere polymorphism, through some form of inheritance that gives rise to object/class hierarchies. Although a formal semantics of its operations is an integral part of the definition of an ADT, object/class interfaces in object oriented languages are purely syntactic and contain no semantics. Moreover, the explicit definition of the set of all sorts (both provided and required) by an ADT has no correspondence in the object/class interface definitions in main-stream object oriented languages: they do not mention what their respective objects/classes require, but specify only the operations that they provide.

The differences between the ADT model and object oriented programming give rise to a number of problems that have already been discussed in the literature. Some counter-measures for problems such as the conflict between inheritance and encapsulation [49], the purely syntactic nature of interfaces, and their asymmetric specification of offered/required services, have been integrated in the design of certain more advanced object oriented languages and component models. What has not been explored so explicitly and extensively in the literature is how message passing in the object oriented paradigm affects software composition and what alternative mechanisms can be used in its place for components.

The method invocation semantics of object oriented message passing implies a rather tight semantic coupling between the caller and callee pairs of objects. By this semantics, if an object c sends a message $m(p)$ to another object e, then c is invoking the method m of e with the actual parameters p. For this to happen:

- c must know (how to find) e;
- c must know the syntax and the semantics of the method m of e;
- e must (pretend to) perform the activated method m on parameters p, and return its result to c upon its completion (the "pretense" refers to when e delegates the actual execution of m to a third object); and
- c typically suspends between its sending of m and the receiving of its (perhaps null) result.

Not only this "rendezvous semantics" is far from trivial, it is still susceptible to significantly different and mutually incompatible variations (e.g., with synchronous vs. asynchronous message passing, active vs. passive objects, etc.). Underneath the precise semantics of this rendezvous and its various incarnations in different object oriented models, is a strong conceptual link with ADT.

By its virtue of providing a set of operations, all that one can *do* with an ADT is to *perform* one of its operations. Similarly, the fact that an object provides a set of methods in its interface means that one can do nothing with an object but to *invoke* those methods. This *operational interface* (of objects or ADTs) induces an asymmetric, unidirectional semantic dependency of users (of operations) on providers (of those operations). On the one hand, the operations provided by an ADT (or object) can be used by any other entity (that has access to it). On the other hand, an ADT internally decides what operation of what other ADT to perform. This puts users and providers in asymmetric roles. Users internally *make* the decisions on what operations are to be performed, and generally *rely* on some specific semantics that they expect of these operations, while it is left to be the responsibility of the providers to *carry out* the decisions made by the users to *satisfy* their expectations.

Far from a universal pitfall, it can even be argued that the presumed level of intimacy required among a set of objects composed together through message passing, is an advantage in building individual components. However, at the inter-component level, such intimacy subverts independence of components, contributes to breaking of their encapsulation, and leads to a level of interdependence among components that is no looser than that among objects within a component.

6 A Bland Notion of Components

Instead of relying on targeted active messages for inter-component communication, our component model allows a component instance to exchange only untargeted, passive messages with its environment. Passive messages contain only data and carry no control information (e.g., imply no method invocation). Not implying the exchange of any control information makes passive messages more abstract and more flexible than active messages. For instance, because no form of "call" is implied, the receiver of a message need not interpret the message as an operation that it must perform. The receiver of a message is not even obligated to reply. Consequently, the sender does not necessarily suspend waiting for a result either.

Untargeted messages break the asymmetry between senders and receivers that is inherent in models that use targeted messages. With targeted messages, the knowledge of who the receiver of a message is, or at least how it can be identified, must be contained in its sender. The receiver of a message, on the other hand, is not required to know anything about its sender beforehand: it is prepared to receive messages "from its environment" not from any specific sender. This asymmetry makes the sender of a message semantically dependent on (properties and the scheme used to identify) its receiver. This inherent semantic dependency stifles exogenous coordination by severely restricting the ability of a third party to, e.g., set up the interaction of such a sender with a receiver of its own choosing instead of the one prescribed by the sender. With untargeted messages, both senders and receivers symmetrically exchange messages only with their environment, not with any pre-specified entity.

In contrast to the more sophisticated mechanisms necessary for exchanging targeted passive messages, or even more sophisticated ones to support (remote) method invocation for active messages, the mechanism necessary for exchanging untargeted passive messages essentially supports only the mundane I/O primitives: an untargeted message itself is merely some passive data that an entity exchanges with its environment; "sending" such a message is just a write operation; and "receiving" it is just a "read" operation. The I/O operations read and write are performed by a component instance on "contact points" that are recognized by its environment for the purpose of information exchange. We refer to these contact points as the *ports* of a component instance. Without loss of generality, we assume ports are unidirectional, i.e., the information flows through a port in one direction only: either from the environment into its component instance (through read) or from its component instance to the environment (through write). Each I/O operation inherently synchronizes the entity that performs it with its environment: a write operation suspends until the environment accepts the data it has to offer through its respective port; likewise, a read operation suspends until the environment offers the suitable data it expects through its respective port.

This view of component communication leads to a generic component model. In this model, a component instance is a black box that contains one or more active entities. An active entity is one that has its own independent thread of control. Examples of active entities are processes, threads, active objects, agents, etc. No assumption is made in this model about how the active entities inside a component instance communicate with each other. However, simple I/O operations through its ports are the only means of communication for the active entities inside a component instance with any entity not in the same component instance. By this definition, a Unix process, for instance, qualifies as a component instance: it contains one or more threads of control which may even run in parallel on different physical processors, and its file descriptors qualify as ports. A component instance may itself consist of a collection of other component instances, perhaps running in a distributed environment. Thus, by identifying their relevant ports through which they exchange data with their environment, entire systems can be viewed and used as component instances, abstracting away their internal details of operation, structure, geography, and implementation.

Such a simple model of components may at first appear rather banal. Nevertheless, it leads to a simple yet useful notion of behavior and behavioral interface. One of the strengths of this model is that it innately espouses anonymous communication: entities that communicate with each other need not know each other. It makes the model inherently amenable to exogenous coordination and supports highly flexible composition possibilities, yielding a very powerful paradigm for component/behavior composition.

7 Elements of a Behavioral Interface

There are different ways in which one can represent behavior. Given our model of components, the most direct and obvious way to represent the observable behavior of a component instance is to model it as a relation on its observable input and output. Because this input/output takes place through the ports of the component instance, sequences of data items that pass through a port emerge as the key building blocks for describing behavior.

Relating sequences of data items that pass through different ports of a component instance requires a sense of relative temporal order to inter-relate otherwise independent events. We need to state, for instance, that a certain data item passes through this port before or after some other data item passes through that port. The assumption of a global clock is stifling in distributed systems and is an overkill for our purpose. Indeed, what we need is a very diluted notion of time that is much less restrictive than the notion of global time. We need to accommodate for:

- **ordering of events**: stating that the occurrence of a certain event precedes or succeeds that of another;
- **atomicity of a set of events**: stating that a given set of events occur only atomically.
- **temporal progression**: stating that only a finite set of events can occur within any bounded temporal interval.

Observe that we do not speak of *simultaneity* in our list of requirements here. Simultaneity is a rather ambiguous notion in distributed systems. Instead, we speak of *atomicity*. The atomicity of a set of events means that either none of them occurs, or else they all occur before any other event (not in that set) occurs, i.e., the occurrence of an atomic set of events cannot be interleaved with the occurrence of any other event. Stating that a set of events must occur atomically allows but does not require (any subset of) those events to occur simultaneously. It also allows for those events to occur in any nondeterministic order, so long as either they all occur or none occurs at all. Atomicity can be seen as a relaxing generalization of simultaneity. It is as if an atomic set of events all happen "simultaneously," except that we elongate the moment of their occurrence into a temporal interval. The provision that no other event may interleave with the occurrence of those in the set ensures that our "elongation of the time moment into an interval" is not detectable by other entities in the system.

Requiring that only a finite set of events can occur within any bounded temporal interval precludes anomalies such as Zeno's paradox.

We use positive numbers to represent moments in time, with the proviso that it is not the actual numeric values of the time moments, but only their relative ordering that is significant. The numerical less-than relation represents the ordering of events. The numeric equal-to relation represents atomicity, *not* simultaneity. Temporal progression can be enforced by requiring that in every temporal sequence a, for any number $N \geq 0$ there exists an $i \geq 0$ such that the i^{th} element in a exceeds N.

8 Abstract Behavior Types

An ABT defines an abstract behavior as a relation among the observable input/output that occur through a set of "contact points" (e.g., ports of a component instance) without specifying any detail about: (1) the operations that may be used to implement such behavior; or (2) the data types those operations may manipulate for the realization of that behavior.[2] This definition parallels that of an ADT, which abstracts away from the instructions and the data structures that may be used to implement the operational interface it defines for a data type. In contrast, an ABT defines what a behavior is in terms of a relation (i.e., constraint) on the observable input/output of an entity, without saying anything about how it can be realized.

More formally, an ABT is a (maximal) relation among a set of timed-data-streams. The notion of timed-data-streams as well as most of the technical content in this section come from the work of J. Rutten on coalgebras [43, 28], stream calculus [42], and a coalgebraic semantics for Reo [7]. Coalgebraic methods have been used for dynamical systems, automata and formal languages, modal logic, transition systems, hybrid systems, infinite data types, the control of discrete event systems, formal power series, etc. (see for instance [47], [37], [38], [44], [45], [46], [20], [25]). Coalgebras have also been used as models for various programming paradigms, notably for objects and classes (see, e.g., [41], [26], and [24]). One of the first applications of coalgebras to components appears in [10].

Defining observable behavior in terms of input/output implants a dataflow essence within ABTs akin to such dataflow-like networks and calculi as [9], [30], and especially [12]. The coalgebraic model of ABT presented here differs from all of the above-mentioned work in a number of respects. Most importantly, the ABT model is compositional. Its explicit modeling of ordering/timing of events in terms of separate time streams provides a simple foundation for defining complex synchronization and coordination protocols using a surprisingly expressive small set of primitives. The use of coinduction as the main definition and proof principle to reason about both data and time streams allows simple compositional construction of ABTs representing many different generic coordination schemes involving combinations of various synchronous and asynchronous primitives that are not present (and not even expressible) in any of the aforementioned models.

A *stream* (over A) is an infinite sequence of elements of some set A. Streams over sets of (uninterpreted) data items are called *data streams* and are typically denoted as α, β, γ, etc. Zero-based indices are used to denote the individual elements of a stream, e.g., $\alpha(0), \alpha(1), \alpha(2), ...$ denote the first, second, third, etc., elements of the stream α. We use the infix "dot" as the stream constructor: $x.\alpha$ denotes a stream whose first element is x and whose second, third, etc. elements are, respectively, the first and its successive elements of the stream α.

Following the conventions of stream calculus [42], the well-known operations of head and tail on streams are called *initial value* and *derivative*: the initial

[2] The term "Abstract Behavior Type" is a variation of the term "Abstract Behavioral Type" proposed by F. de Boer for a related concept.

value of a stream α (i.e., its head) is $\alpha(0)$, and its (first) derivative (i.e., its tail) is denoted as α'. The k^{th} derivative of α is denoted as $\alpha^{(k)}$ and is the stream that results from taking the first derivative of α and repeating this operation on the resulting stream for a total of k times. Relational operators on streams apply pairwise to their respective elements, e.g., $\alpha \geq \beta$ means $\alpha(0) \geq \beta(0), \alpha(1) \geq \beta(1), \alpha(2) \geq \beta(2), \ldots$.

Time streams are constrained streams over (positive) real numbers, representing moments in time, and are typically denoted as a, b, c, etc. To qualify as a time stream, a stream of real numbers a must be (1) strictly increasing, i.e., the constraint $a < a'$ must hold; and (2) progressive, i.e., for every $N \geq 0$ there must exist an index $n \geq 0$ such that $a(n) > N$.

We use positive real numbers instead of natural numbers to represent time because, as observed in the world of temporal logic [21], real numbers induce the more abstract sense of *dense time* instead of the notion of *discrete time* imposed by natural numbers. Specifically, we sometimes need finitely many steps within any bounded time interval for certain ABT equivalence proofs (see, e.g., [7]). This is clearly not possible with a discrete model of time. Recall that the actual values of "time moments" are irrelevant in our ABT model; only their relative order is significant and must be preserved. Using dense time allows us to locally break strict numerical equality (i.e., simultaneity) arbitrarily while preserving the atomicity of events.

A *Timed Data Stream* is a twin pair of streams $\langle \alpha, a \rangle$ consisting of a data stream α and a time stream a, with the interpretation that for all $i \geq 0$, the input/output of data item $\alpha(i)$ occurs at "time moment" $a(i)$. Two timed data streams $\langle \alpha, a \rangle$ and $\langle \beta, b \rangle$ are equal if their respective elements are equal, i.e. $\langle \alpha, a \rangle = \langle \beta, b \rangle \equiv \alpha = \beta \wedge a = b$.

An *Abstract Behavior Type* (ABT) is a (maximal) relation over timed data streams. Every timed data stream involved in an ABT is tagged either as its *input* or its *output*. The input/output tags of the timed data streams involved in an ABT are meaningless in the relation that defines the ABT. However, these tags are crucial in ABT composition described in Section 8.2.

Generally, we use the prefix notation $R(I_1, I_2, ..., I_m; O_1, O_2, ..., O_n)$ and the separator ";" to designate the ABT defined by the $(m+n)$-ary relation R over the $m \geq 0$ sets of input timed data streams $I_i, 0 \leq i \leq m$ and the $n \geq 0$ sets of output timed data streams $O_j, 0 \leq j \leq n$. As usual, $m+n$ is called the *arity* of R and we refer to m and n individually as the *input arity* and the *output arity* of R. In the special case where $m = n = 1$ it is sometimes convenient to use the infix notation $I\ R\ O$ instead of the standard $R(I;O)$. To distinguish the set of timed data streams that appears in a position in the relation that defines an ABT (i.e., a column in the relation) from a specific timed data stream in that set (i.e., which may appear in a row of the relation in that position) we refer to I_i and O_j as, respectively, the i^{th} input and the j^{th} output *portals* of the ABT.

Formally, a component, as defined in Section 6, with $m \geq 0$ input and $n \geq 0$ output ports is an ABT with m input and n output portals. The set of all possible streams of data items that can pass through each port of the component, together with their respective timing, comprise the set of timed data streams of the ADT's portal that corresponds to that port.

8.1 ABT Examples

In this section we show the utility of the ABT model through a number of examples.

Basic Channels Following is a list of some useful simple binary abstract behavior types. Each has a single input and a single output portal.

1. The behavior of a *synchronous channel* is captured by the `Sync` ABT, defined as
$$\langle \alpha, a\rangle \ \texttt{Sync}\ \langle \beta, b\rangle \equiv \langle \alpha, a\rangle = \langle \beta, b\rangle.$$
Because $\langle \alpha, a\rangle = \langle \beta, b\rangle \equiv \alpha = \beta \wedge a = b$, the `Sync` ABT represents the behavior of any entity that (1) produces an output data stream identical to its input data stream ($\alpha = \beta$), and (2) produces every element in its output at the same time as its respective input element is consumed ($a = b$). Recall that "at the same time" means only that the two events of consumption and production of each data item by a `Sync` channel occur atomically.
2. The behavior of an asynchronous unbounded *FIFO channel* is captured by the `FIFO` ABT, defined as
$$\langle \alpha, a\rangle \ \texttt{FIFO}\ \langle \beta, b\rangle \equiv \alpha = \beta \wedge a < b.$$
The `FIFO` ABT represents the behavior of any entity that (1) produces an output data stream identical to its input data stream ($\alpha = \beta$), and (2) produces every element in its output some time after its respective input element is observed ($a < b$).
3. The behavior of an asynchronous channel with the bounded capacity of 1 is captured by the $\texttt{FIFO}_1$ ABT, defined as
$$\langle \alpha, a\rangle \ \texttt{FIFO}_1\ \langle \beta, b\rangle \equiv \alpha = \beta \wedge a < b < a'.$$
The $\texttt{FIFO}_1$ ABT represents the behavior of any entity that (1) produces an output data stream identical to its input data stream ($\alpha = \beta$), and (2) produces every element in its output some time after its respective input element is observed ($a < b$) but before its next input element is observed ($b < a'$ which means $b(i) < a(i+1)$ for all $i \geq 0$).
4. The behavior of an asynchronous channel with the bounded capacity of 1 filled to contain the data item D as its initial value is captured by the $\texttt{FIFO}_1(D)$ ABT, defined as
$$\langle \alpha, a\rangle \ \texttt{FIFO}_1(D)\ \langle \beta, b\rangle \equiv \beta(0) = D \wedge \alpha = \beta' \wedge b < a < b'.$$
The $\texttt{FIFO}_1(D)$ ABT represents the behavior of any entity that (1) produces an output data stream consisting of the initial data item D followed by

the input data stream of the ABT ($\beta(0) = D \wedge \alpha = \beta'$), and (2) for $i \geq 0$ performs its i^{th} input operation some time between its i^{th} and $i+1^{st}$ output operations ($b < a < b'$).

5. The behavior of an asynchronous channel with the bounded capacity of $k > 0$ is captured by the $\texttt{FIFO}_k$ ABT, defined as

$$\langle \alpha, a \rangle \ \texttt{FIFO}_k \ \langle \beta, b \rangle \equiv \alpha = \beta \wedge a < b < a^{(k)}.$$

   Recall the $a^{(k)}$ is the k^{th}-derivative (i.e., the k^{th}-tail) of the stream a. The $\texttt{FIFO}_k$ ABT represents the behavior of any entity that (1) produces an output data stream identical to its input data stream ($\alpha = \beta$), and (2) produces every element in its output some time after its respective input element is observed ($a < b$) but before its k^{th}-next input element is observed ($b < a^{(k)}$ which means $b(i) < a(i + k)$ for all $i \geq 0$). Observe that $\texttt{FIFO}_1$ is indeed a special case of $\texttt{FIFO}_k$ with $k = 1$.

It is illuminating to compare the `FIFO` ABT defined above with the definition of the queue ADT in Figure 1. They are both mathematically well-defined constructs that describe the same thing: an unbounded FIFO queue. The ADT defines a queue in terms of a set of operations and a set of axioms that constrain the observable mutual effect of those operations on each other. It abstracts away the actual instructions for the implementation of those operations and the data structures that they manipulate. The ABT defines a queue in terms of what data items it exchanges with its environment, when it consumes and produces them, and a set of axioms that constrain their interrelationships. It abstracts away the operations for the realization (or enforcement) of those relationships and the data types that they may utilize to do so.

Merge and Replicate We now define two other ABTs that, as we see in Section 9, form a foundation for an interesting and expressive calculus: merger and replicator. The merger ABT is defined as:

$$
\begin{aligned}
&\mathit{Mrg}(\langle \alpha, a \rangle, \langle \beta, b \rangle; \langle \gamma, c \rangle) \equiv \\
&\begin{cases}
\alpha(0) = \gamma(0) \wedge a(0) = c(0) \wedge \mathit{Mrg}(\langle \alpha', a' \rangle, \langle \beta, b \rangle; \langle \gamma', c' \rangle) & \text{if } a(0) < b(0) \\
\exists t{:}a(0) < t < \mathit{min}(a(1), b(1)) \wedge \exists r, s \in \{a(0), t\} \wedge r \neq s \wedge & \text{if } a(0) = b(0) \\
\quad \mathit{Mrg}(\langle \alpha, r.a' \rangle, \langle \beta, s.b' \rangle; \langle \gamma, c \rangle) & \\
\beta(0) = \gamma(0) \wedge b(0) = c(0) \wedge \mathit{Mrg}(\langle \alpha, a \rangle, \langle \beta', b' \rangle; \langle \gamma', c' \rangle) & \text{if } a(0) > b(0)
\end{cases}
\end{aligned}
$$

Intuitively, the *Mrg* ABT produces an output that is a merge of its two input streams. If $\alpha(0)$ arrives before $\beta(0)$, i.e. $a(0) < b(0)$, then the ABT produces $\gamma(0) = \alpha(0)$ as its output at $c(0) = a(0)$ and proceeds with the tails of the streams in its first input timed data stream. If $\alpha(0)$ arrives after $\beta(0)$, i.e. $a(0) > b(0)$, then the ABT produces $\gamma(0) = \beta(0)$ as its output at $c(0) = b(0)$ and proceeds with the tails of the streams in its second input timed data stream. If the $\alpha(0)$ and $\beta(0)$ arrive "at the same time" (i.e., $a(0) = b(0)$), then in this formulation *Mrg* picks an arbitrary number t in the open time interval

$(a(0), min(a(1), b(1)))$ and uses it to nondeterministically break the tie. The assumption of dense time guarantees the existence of an appropriate t. Recall that the construct $r.a'$ is a stream whose derivative (tail) is a' and whose initial value (head) is r. Thus, for $a(0) = b(0)$ *Mrg* nondeterministically changes the head of one of the two time streams, a or b, thereby "delaying" the arrival of its corresponding data item to break the tie. The finite delay introduced by *Mrg* in this case is justified because although it breaks simultaneity, its value is constrained to preserve atomicity. Observe that $Mrg(\langle\alpha, a\rangle, \langle\beta, b\rangle;\langle\gamma, c\rangle) = Mrg(\langle\beta, b\rangle, \langle\alpha, a\rangle;\langle\gamma, c\rangle)$.

The replicator ABT is defined as:

$$Rpl(\langle\alpha, a\rangle;\langle\beta, b\rangle, \langle\gamma, c\rangle) \equiv \beta = \alpha \wedge \gamma = \alpha \wedge b = a \wedge c = a$$

It is easy to see that this ABT captures the behavior of any entity that synchronously replicates its input stream into its two identical output streams. Observe that $Rpl(\langle\alpha, a\rangle;\langle\beta, b\rangle, \langle\gamma, c\rangle) = Rpl(\langle\alpha, a\rangle;\langle\gamma, c\rangle, \langle\beta, b\rangle)$.

Sum As an example of an ABT that performs some computation, consider a simple dataflow adder. The behavior of such a component is captured by the *Sum* ABT defined as

$$\begin{aligned}
&Sum(\langle\alpha, a\rangle, \langle\beta, b\rangle;\langle\gamma, c\rangle) \equiv \\
&\quad \gamma(0) = \alpha(0) + \beta(0) \wedge \\
&\quad \exists t{:}max(a(0), b(0)) < t < min(a(1), b(1)) \wedge c(0) = t \wedge \\
&\quad Sum(\langle\alpha', a'\rangle, \langle\beta', b'\rangle;\langle\gamma', c'\rangle).
\end{aligned}$$

Sum defines the behavior of a component that repeatedly reads a pair of input values from its two input ports, adds them up, and writes the result out on its output port. As such, its output data stream is the pairwise sum of its two input data streams. This component behaves asynchronously in the sense that it can produce each of its output data items with some arbitrary delay after it has read both of its corresponding input data items ($c(0) = t \wedge t > max(a(0), b(0))$). However, it is obligated to produce each of its output data items before it reads in its next input data item ($t < min(a(1), b(1))$).

Philosophers and Chopsticks The classical dining philosophers problem can be described in terms of $n > 1$ pairs of instances of two components: philosopher instances of *Phil* and chopstick instances of *Chop*. We define the externally observable behavior of each of these components as an ABT. We show in Section 9 how instances of these components can be composed into different component based systems both to exhibit and to solve the famous deadlock problem.

We assume that a chopstick component has two input ports, t (for *take*) and f (for *free*), through which it reads in the timed data streams $\langle\alpha_t, a_t\rangle$ and $\langle\alpha_f, a_f\rangle$, respectively. The data items in α_t and α_f are tokens whose actual values are not of interest to us. In practice, it is a good idea for these tokens to contain the identifier of the entity (e.g., philosopher) who uses the chopstick, but as long as such informative requirements do not affect behavior, they are irrelevant for our ABT definition.

When a chopstick is free (its initial state) it is ready to accept a *take* request and thus reads from its t port the next take request token out of $\langle \alpha_t, a_t \rangle$. Once taken, a chopstick is ready to accept a *free* request and thus reads from its f port the free request token out of $\langle \alpha_f, a_f \rangle$. For the user of the chopstick, the success of its I/O operation on port t or f means the chopstick has accepted its (*take* or *free*) request. This simple behavior is captured by the *Chop* ABT defined as

$$\mathit{Chop}(\langle \alpha_t, a_t \rangle, \langle \alpha_f, a_f \rangle;) \equiv a_t < a_f < a'_t.$$

Because we are not interested in the actual value of the take/free tokens, the *Chop* ABT has nothing to say about the data streams α_t and α_f; it is only the timing that is relevant here. The timing equation simply states that initially, there must be a take, followed by a free, and this sequence repeats.

We assume that a philosopher component has four output ports, *lt* (for *left-take*), *lf* (for *left-free*), *rt* (for *right-take*), and *rf* (for *right-free*), through which it writes the timed data streams $\langle \alpha_{lt}, a_{lt} \rangle$, $\langle \alpha_{lf}, a_{lf} \rangle$, $\langle \alpha_{rt}, a_{rt} \rangle$, and $\langle \alpha_{rf}, a_{rf} \rangle$, respectively. The two ports *lt* and *lf* are "on the left" and two ports *rt* and *rf* are "on the right" of the philosopher component, so to speak. The philosopher's requests to take and free the chopsticks on its left and right are issued through their respective ports.

The externally observable behavior of a philosopher component is as follows. After some period of "thinking" it decides to eat, at which point it attempts to obtain its two chopsticks by issuing take requests on its *lt* and *rt* ports. We assume it always issues a request for its left chopstick before requesting the one on its right. The philosopher component interprets the success of its write operation as the acceptance of its request (e.g., for exclusive access to the chopstick). Once, and if, both of its take requests are granted, it proceeds to "eat" for some time, at the end of which it then issues requests to free its left and right chopsticks by writing tokens to its *lf* and *rf* ports. The philosopher component then repeats the cycle by entering its thinking period again. This behavior is captured by the *Phil* ABT defined as

$$\mathit{Phil}(;\langle \alpha_{lt}, a_{lt} \rangle, \langle \alpha_{lf}, a_{lf} \rangle, \langle \alpha_{rt}, a_{rt} \rangle, \langle \alpha_{rf}, a_{rf} \rangle) \equiv a_{lt} < a_{rt} < a_{lf} < a_{rf} < a'_{lt}.$$

Again, because we are not interested in the actual values of the take/free tokens that this component produces, the *Phil* ABT says nothing about the data streams. All we are interested in is the timing constraints: an arbitrary "thinking" delay; followed by a request to take the left chopstick; once granted, followed by a request to take the right chopstick; once granted, followed by an arbitrary "eating" delay; followed by the requests to free the left and the right chopsticks; and the cycle repeats.

8.2 ABT Composition

Abstract behavior types can be composed to yield other abstract behavior types through a composition similar to the relational join operation in relational databases. Two ABTs can be composed over a common timed data stream if one is the producer and the other the consumer of that timed data stream. The same two ABTs can be composed over zero or more common timed data streams, each ABT playing the role of the producer or the consumer of one of the timed data streams, independent of its role regarding the others. Observe that the producer and the consumer of a timed data stream, $\langle \alpha, a \rangle$, necessarily synchronize their I/O operations on their respective portals for the mutual exchange of the data items in its data stream α, according to the schedule in its twin time stream a. This is accomplished simply by "fusing" their respective portals together such that the timed data stream observed on one is identical to the one observed on the other.

Consider two ABTs B_1 with arity $p = p_i + p_o$ and B_2 with arity $q = q_i + q_o$, where p_i and p_o are, respectively, the input arity and the output arity of B_1, and q_i and q_o, those for B_2. B_1 and B_2 can be composed with $0 \leq k \leq min(p_i, q_o) + min(p_o, q_i)$ pairs of mutually fused portals, where the data items produced through an output portal, O, of one ABT are fed for consumption by the other ABT through its input portal that is fused with O.

We define the *k-dyad composition* of $B_1(I1_1, I1_2, ...I1_{p_i}; O1_1, O1_2, ...O1_{p_o})$ and $B_2(I2_1, I2_2, ...I2_{q_i}; O2_1, O2_2, ...O2_{q_o})$ as a special form of the join of the two relations B_1 and B_2 where k distinct portals (i.e., relational columns) of B_1 are paired each with a distinct portal of B_2 into k dyads such that (1) the two portals in each dyad have opposite input/output tags, and (2) the two timed data streams of the portals in each dyad are equal. The k-dyad composition of B_1 and B_2 yields a new ABT, $B(I_1, I_2, ...I_m; O_1, O_2, ...O_n)$, with arity $m+n = p+q-2\times k$, defined as a relation over those portals of B_1 and B_2 that are not involved in a dyad (i.e., the fused portals disappear from the resulting relation). The list $I_1, I_2, ...I_m$ is obtained from the list $I1_1, I1_2, ...I1_{p_i}, I2_1, I2_2, ...I2_{q_i}$ by eliminating every one of its elements involved in a dyad. Similarly, the list $O_1, O_2, ...O_n$ is obtained from the list $O1_1, O1_2, ...O1_{p_o}, O2_1, O2_2, ...O2_{q_o}$ by eliminating every one of its elements involved in a dyad.

We use the dyad indices $1 \leq l \leq k$ as superscripts to mark the corresponding portals of B_1 and B_2 in their k-dyad composition. For example, $B = B_1(\langle \alpha, a \rangle, \langle \beta, b \rangle^1; \langle \gamma, c \rangle) \circ B_2(\langle \delta, d \rangle; \langle \mu, m \rangle^1)$ denotes the 1-dyad composition of the two abstract behavior types B_1 and B_2 where the output (portal) of B_2 is identical to the second input (portal) of B_1. The resulting ABT is defined through the relation $B \equiv \{\langle \langle \alpha, a \rangle, \langle \delta, d \rangle; \langle \gamma, c \rangle \rangle \mid \langle \langle \alpha, a \rangle, \langle \beta, b \rangle; \langle \gamma, c \rangle \rangle \in B_1 \wedge \langle \langle \delta, d \rangle; \langle \mu, m \rangle \rangle \in B_2 \wedge \langle \beta, b \rangle = \langle \mu, m \rangle\}$. Another example is the ABT $B = B_1(\langle \alpha, a \rangle, \langle \beta, b \rangle^1; \langle \gamma, c \rangle^2) \circ B_2(\langle \delta, d \rangle^2; \langle \mu, m \rangle^1, \langle \nu, n \rangle)$, which denotes the 2-dyad composition of the two abstract behavior types B_1 and B_2 where the first output of B_2 is identical to the second input of B_1 and the output of B_1 is identical to the input of B_2. The resulting ABT is defined as the relation $B \equiv$

$\{\langle\langle\alpha, a\rangle;\langle\nu, n\rangle\rangle \mid \langle\langle\alpha, a\rangle, \langle\beta, b\rangle;\langle\gamma, c\rangle\rangle \in B_1 \wedge \langle\langle\delta, d\rangle;\langle\mu, m\rangle, \langle\nu, n\rangle\rangle \in B_2 \wedge \langle\beta, b\rangle = \langle\mu, m\rangle \wedge \langle\gamma, c\rangle = \langle\delta, d\rangle\}$.

The common case of the 1-dyad composition of B_1 and B_2 where the single output of B_1 is identical to the single input of B_2 is abbreviated as $B_1(...;\langle\alpha, a\rangle) \circ B_2(\langle\beta, b\rangle;...)$ instead of $B_1(...;\langle\alpha, a\rangle^1) \circ B_2(\langle\beta, b\rangle^1;...)$. This abbreviation is particularly convenient together with the infix notation for binary abstract behavior types. For instance, $B = \langle\alpha, a\rangle B_1 \langle\beta, b\rangle \circ \langle\gamma, c\rangle B_2 \langle\delta, d\rangle$ denotes the 1-dyad composition of the two abstract behavior types B_1 and B_2 where the output of B_1 is identical to the input of B_2. Of course, the resulting ABT is defined as the relation $\langle\alpha, a\rangle B \langle\delta, d\rangle \equiv \{\langle\langle\alpha, a\rangle;\langle\delta, d\rangle\rangle \mid \langle\langle\alpha, a\rangle;\langle\beta, b\rangle\rangle \in B_1 \wedge \langle\langle\gamma, c\rangle;\langle\delta, d\rangle\rangle \in B_2 \wedge \langle\beta, b\rangle = \langle\gamma, c\rangle\}$.

For example, consider the binary ABTs defining the basic channels presented in Section 8.1. It is not difficult to see that the (1-dyad) composition of these ABTs produces results that correspond to our intuition. For instance, the composition of two `Sync` ABTs produces a `Sync` ABT. Indeed, composition of a `Sync` ABT with any other ABT (on its left or right) yields the same ABT. More interestingly, the composition of two `FIFO` ABTs produces a `FIFO` ABT. Composing two $\texttt{FIFO}_1$ ABTs produces a $\texttt{FIFO}_2$ ABT. The formal proof of this latter equivalence relies on our notion of dense time (as opposed to discrete time) and is given in [7], together with the formal treatment of many other interesting examples.

9 Reo

The ABT model provides a simple formal foundation for definition and composition of components. The k-dyad composition of ABTs supports a very flexible mechanism for software composition in component based systems. This furnishes the desired level of composition flexibility we expect in a component model. However, composing components directly with one another in this way reduces the glue code to essentially nothing more than repeated applications of the k-dyad composition operator. More importantly, it all but extinguishes the possibility of wielding exogenous coordination through the glue code. The ABT model is too low-level to directly provide any form of non-trivial coordination (beyond the simple synchronization implied by its timed data streams); for that, we need an effective exogenous coordination model.

Reo is a channel-based exogenous coordination model wherein complex coordinators, called *connectors* are compositionally built out of simpler ones [6, 3, 7]. The simplest connectors in Reo are a set of *channels* with well-defined behavior supplied by users. Reo can be used as a language for coordination of concurrent processes, or as a “glue language” for compositional construction of connectors that orchestrate component instances in a component based system. The emphasis in Reo is on connectors and their composition only, not on the entities that connect to, communicate, and cooperate through these connectors. Each connector in Reo imposes a specific coordination pattern on the entities (e.g., component instances) that perform I/O operations through that connector, without the knowledge of those entities.

Channel composition in Reo is a very powerful mechanism for construction of connectors. The expressive power of connector composition in Reo has been demonstrated through many examples in [2, 3, 7]. For instance, exogenous coordination patterns that can be expressed as (meta-level) regular expressions over I/O operations performed by component instances can be composed in Reo out of a small set of only five primitive channel types.

A *mobile channel* allows (physical or logical) relocation of one of its ends without the knowledge or the involvement of the entity at its other end. An efficient distributed implementation of channels supporting this sense of mobility is described in [4]. Both component instances and channels are mobile in Reo. Furthermore, connectors in Reo are dynamically reconfigurable, while they are in use by component instances. As such, Reo resembles dynamically reconfigurable Kahn networks and is also related to Broy's timed dataflow model, although it is more general and more expressive that these and similar models.

It turns out that the ABT model is quite adequate for defining the channel and connector composition operation which is the crux of exogenous coordination in Reo. In the rest of this section we show how connector construction in Reo can be seen as an application of the ABT model.

9.1 Channels and Connectors

Channels are the only primitive medium of communication between two components in Reo. The notion of channel in Reo is far more general than its common interpretation. A channel in Reo has its own unique identity and always has exactly two directed ends, each with its own unique identity. Based on their direction, there are two types of channel ends: *source* and *sink* ends. Data enters through a source channel end into its respective channel, and it leaves through a sink channel end from its respective channel. (Channels themselves have no direction in Reo, only their ends do.)

Beyond a small set of mild obvious requirements, such as enabling I/O operations to read/write data items from/to their ends, Reo places no restrictions on the behavior of channels. This allows an open-ended set of different channel types to be used simultaneously together in Reo, each with its own policy for synchronization, buffering, ordering, computation, data retention/loss, etc. Some typical examples of conventional channels are, e.g., the ones defined in Section 8.1. These channels happen to each have a source end and a sink end. More unconventional channels are also possible in Reo, especially because a channel can also have only two source ends or only two sink ends. A few examples of some such exotic channels appear in Section 9.3; even more examples are presented in [2, 6, 3].

Strictly speaking, Reo itself neither provides nor assumes the availability of any specific set of channel types; it simply assumes that an appropriate assortment of channel types, each with its properly well-defined semantics, is provided by users for it to operate on. Nevertheless, it is reasonable to expect that in practice certain most primitive channel types, e.g., synchronous channels, will always be made available in all cases.

Reo defines a **connector** as a set of channel ends and their connecting channels organized in a graph of **nodes** and *edges* such that:

- Zero or more channel ends coincide on every node.
- Every channel end coincides on exactly one node.
- There is an edge between two (not necessarily distinct) nodes if and only if there is a channel one end of which coincides on each of those nodes.

We use $x \mapsto N$ to denote that the channel end x coincides on the node N, and $\widehat{x}$ to denote the unique node on which the channel end x coincides. For a node N, we define the set of all channel ends coincident on N as $[N] = \{x \mid x \mapsto N\}$, and disjointly partition it into the sets $Src(N)$ and $Snk(N)$, denoting the sets of source and sink channel ends that coincide on N, respectively.

Observe that nodes are neither components nor locations. Although some nodes are attached to component instances to allow their exchange of information, nodes and components are different notions and not every node can be associated with or attached to a component instance. A node is a fundamental concept in Reo representing an important topological property: all channel ends $x \in [N]$ coincide on the same node N. This property entails specific implications in Reo regarding the flow of data among the channel ends $x \in [N]$, irrespective of concern for the location of those channel ends or N, or the possible attachment of N to a component instance.

A node N is called a **source node** if $Src(N) \neq \emptyset \wedge Snk(N) = \emptyset$. Analogously, N is called a **sink node** if $Src(N) = \emptyset \wedge Snk(N) \neq \emptyset$. A node N is called a **mixed node** if $Src(N) \neq \emptyset \wedge Snk(N) \neq \emptyset$.

By the above definition, every channel represents a (simple) connector with two nodes. From the point of view of Reo a port of a component instance is just a node that (initially) contains a single channel end. An input port is (initially a singleton) source node, and an output port is (initially a singleton) sink node. From the point of view of a component instance, each of its ports is merely a simple connector corresponding to a synchronous channel (the node of) one end of which is made publicly accessible for I/O by its environment, while (the node of) its other end is hidden for exclusive use by the component instance itself. An output port of a component instance has the sink node of its synchronous channel public while its source node is available only for I/O operations performed by that component instance. Likewise, an input port has the source node of its synchronous channel public while its sink node is hidden for exclusive use by its component instance.

Reo provides I/O operations on source and sink nodes only; components cannot read from or write to mixed nodes. A component instance can write to a source node and can read from a sink node using node I/O operations of Reo.

The graph representing a connector is *not* directed. However, for each channel end x_c of a channel c, we use the directionality of x_c to assign a *local direction in the neighborhood of* $\widehat{x_c}$ to the edge that represents c. The local direction of the edge representing a channel c in the neighborhood of the node of its source x_c is presented as an arrow emanating from $\widehat{x_c}$. Likewise, the local direction of the edge representing a channel c in the neighborhood of the node of its sink x_c is presented as an arrow pointing to $\widehat{x_c}$. See Figures 2 and 3 for examples.

Complex connectors are constructed in Reo out of simpler ones using its `join` operation. The `join` operation in Reo is defined only on nodes. Joining two nodes N_1 and N_2 destroys both nodes and produces a new node N with the property that $[N] = [N_1] \cup [N_2]$. This single operation allows construction of arbitrarily complex connector graphs involving any combination of channels picked from an open-ended set of channel types. The semantics of a connector is defined as a composition of the semantics of its (1) constituent channels, and (2) nodes. Because Reo does not provide any channels, it does not define their semantics either. What Reo defines is the composition of channels into connectors and the semantics of this composition through the semantics of its (three types of) nodes.

Intuitively, a source node replicates every data item written to it as soon as all of its coincident source channel ends can consume that data item. Reading from a sink node nondeterministically selects one of the data items available through its coincident sink channel ends. A mixed node is a self-contained "pumping station" that combines the behavior of a sink node and a source node in an atomic iteration of an infinite loop: in each atomic iteration it nondeterministically selects an appropriate data item available through its coincident sink channel ends and replicates that data item into all of its coincident source channel ends. A data item is appropriate for selection in an iteration only if it can be consumed by all source channel ends that coincide on that node.

9.2 ABT Models of Nodes and Connectors

Consider a sink node N with $[N] = \{x, y\}$, as in Figure 2.a. The read operations performed on this node induce an output timed data stream, $\langle\alpha_N, a_N\rangle$, for this sink node. We use $\langle\alpha_x, a_x\rangle$ and $\langle\alpha_y, a_y\rangle$ to designate the timed data streams corresponding to the channel ends x and y, respectively. The semantics of this sink node is defined by the ABT $Mrg(\langle\alpha_x, a_x\rangle, \langle\alpha_y, a_y\rangle;\langle\alpha_N, a_N\rangle)$.

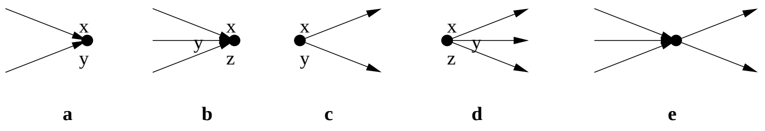

Fig. 2. Representation of nodes in Reo

The semantics of a sink node N where $[N] = \{x, y, z\}$, as in Figure 2.b, is defined as the 1-dyad composition

$$\begin{aligned} &Mrg3(\langle\alpha_x, a_x\rangle, \langle\alpha_y, a_y\rangle, \langle\alpha_z, a_z\rangle;\langle\alpha_N, a_N\rangle)) \equiv \\ &\quad Mrg(\langle\alpha_x, a_x\rangle, \langle\alpha_y, a_y\rangle;\langle\beta_1, b_1\rangle^1) \circ Mrg(\langle\gamma_1, c_1\rangle^1, \langle\alpha_z, a_z\rangle;\langle\alpha_N, a_N\rangle) \end{aligned}$$

where $\langle\alpha_N, a_N\rangle$ is the output timed data stream of the node, as before, and $\langle\beta_1, b_1\rangle$ and $\langle\gamma_1, c_1\rangle$ are internal timed data streams.

Because *Mrg* is associative with respect to its input portals, merging the intermediate result of the merge of x and y with z is the same as merging x with the intermediate result of the merge of y and z; i.e., $\mathit{Mrg}3$ is associative with respect to its input portals. As such, the simple graphical notation of Reo (e.g., in Figures 2.a and b) is quite appropriate because it does not suggest any precedence for the *Mrg* operations. Clearly this scheme can be used to define the semantics of sink nodes with more coincident channel ends in general as the ABT $\mathit{Mrg}\mathtt{k}$ with $\mathtt{k} > 0$ input and one output portals. For completeness, we define $\mathit{Mrg}1(\langle \alpha_x, a_x \rangle; \langle \alpha_N, a_N \rangle)) \equiv \langle \alpha_x, a_x \rangle = \langle \alpha_N, a_N \rangle$ and consider $\mathit{Mrg}2$ to be a pseudonym for *Mrg*.

The write operations performed on a source node N with $[N] = \{x, y\}$, as in Figure 2.c, induce an input timed data stream, $\langle \alpha_N, a_N \rangle$, for N. The semantics of N in this case is defined by the ABT $\mathit{Rpl}(\langle \alpha_N, a_N \rangle; \langle \alpha_x, a_x \rangle, \langle \alpha_y, a_y \rangle$. The semantics of a source node N with $[N] = \{x, y, z\}$, as in Figure 2.d, is defined as the 1-dyad composition

$$\begin{aligned}&\mathit{Rpl}3(\langle \alpha_N, a_N \rangle; \langle \alpha_x, a_x \rangle, \langle \alpha_y, a_y \rangle, \langle \alpha_z, a_z \rangle) \equiv \\ &\quad \mathit{Rpl}(\langle \alpha_N, a_N \rangle; \langle \alpha_x, a_x \rangle, \langle \beta_1, b_1 \rangle^1) \circ \mathit{Rpl}(\langle \gamma_1, c_1 \rangle^1; \langle \alpha_y, a_y \rangle, \langle \alpha_z, a_z \rangle)\end{aligned}$$

where $\langle \alpha_N, a_N \rangle$ is the input timed data stream of the node, as before, and $\langle \beta_1, b_1 \rangle$ and $\langle \gamma_1, c_1 \rangle$ are internal timed data streams. Because *Rpl* is associative with respect to its output portals, the precedence of the *Rpl* operations is irrelevant and $\mathit{Rpl}3$ is also associative with respect to its output portals. Similarly, the general ABT $\mathit{Rpl}\mathtt{k}$ with one input and $\mathtt{k} > 0$ output portals defines the semantics of a source node with $\mathtt{k}$ coincident channel ends. Again, for completeness, we define $\mathit{Rpl}1(\langle \alpha_N, a_N \rangle; \langle \alpha_x, a_x \rangle)) \equiv \langle \alpha_x, a_x \rangle = \langle \alpha_N, a_N \rangle$ and consider $\mathit{Rpl}2$ to be a pseudonym for *Rpl*.

A mixed node, as in Figure 2.e, is a composition of two "half-nodes," a source and a sink. Because no component is allowed to perform an I/O operation on a mixed node, no input/output timed data stream can be defined for a mixed node. A mixed node is a closed entity that does not interact with any component; instead it internally pumps data items from its sink channel ends to its source channel ends. The semantics of a mixed node N with $\mathtt{m} > 0$ sink and $\mathtt{n} > 0$ source channel ends is thus defined as the 1-dyad composition of the two ABTs describing the behavior of each of its half nodes: $\mathit{Mrg}\mathtt{m}(I_1, I_2, \ldots I_{\mathtt{m}}; \langle \beta_1, b_1 \rangle)$ and $\mathit{Rpl}\mathtt{n}(\langle \gamma_1, c_1 \rangle; O_1, O_2, \ldots O_{\mathtt{n}})$. The portals I_i and O_j designate the timed data streams observed at the $\mathtt{m}$ sink and the $\mathtt{n}$ source channel ends coincident on N, respectively. The resulting ABT is thus

$$\begin{aligned}&P\mathtt{m}x\mathtt{n}(I_1, I_2, \ldots I_{\mathtt{m}}; O_1, O_2, \ldots O_{\mathtt{n}}) \equiv \\ &\quad \mathit{Mrg}\mathtt{m}(I_1, I_2, \ldots I_{\mathtt{m}}; \langle \beta_1, b_1 \rangle) \circ \mathit{Rpl}\mathtt{n}(\langle \gamma_1, c_1 \rangle; O_1, O_2, \ldots O_{\mathtt{n}}).\end{aligned}$$

For instance, the behavior of the mixed node in Figure 2.e is captured by the ABT defined as the relation $P3x2(I_1, I_2, I_3; O_1, O_2)$ over the timed data streams of its respective 3 sink and 2 source channel ends.

Every edge of a connector corresponds to a channel whose semantics is defined as an ABT. Since a connector consists of (three types of) nodes and edges, all of whose semantics are now defined as ABTs, the semantics of every connector in Reo can be derived as a composition of the ABTs of its constituent nodes and edges.

9.3 A Cogent Set of Primitive Channels

To demonstrate the utility of Reo we must supply it with a set of primitive channels. The fact that Reo accepts and the ABT model allows definition of an open-ended set of arbitrarily complex channels is interesting. What is more interesting, however, is that connector composition in Reo is itself powerful enough to yield surprisingly expressive complex connectors out of a very small set of trivially simple channels.

A useful set of primitive channels for Reo consists of 7 channel types: `Sync`, `FIFO`, `FIFO`$_1$, `FIFO`$_1(D)$, `Filter`(P), `LossySync`, and `SyncDrain`. This is not a minimal set, in the sense that some of the channel types in this set can themselves be composed in Reo out of others; however, minimality is not our concern here and these channel types turn out to be both simple and frequently useful enough to deserve their own explicit mention. The first four channel types were defined as ABTs in Section 8.1. We define the ABTs for the rest below.

The common characteristics of the last three channels, above, are that they are all (1) synchronous, and (2) *lossy*. Neither channel has a buffer to store data and if necessary, delays the I/O operation on either one of its ends until it is matched with an I/O operation on its other end. A channel is lossy if it does not deliver through its sink end every data item it consumes through its source end. The difference between these three channels is in their loss policy.

1. A `Filter`(P) channel is a synchronous channel with a source and a sink end that takes a *pattern* P parameter upon its creation. It behaves like a `Sync` channel, except that only those data items that match the pattern P can actually pass through it; others are always accepted by its source, but are immediately lost. The behavior of such a channel is captured by the `Filter`(P) ABT defined as

   $$\langle \alpha, a\rangle \; \texttt{Filter}(P) \; \langle \beta, b\rangle \equiv \begin{cases} \beta(0) = \alpha(0) \wedge b(0) = a(0) \wedge \langle \alpha', a'\rangle \; \texttt{Filter}(P) \; \langle \beta', b'\rangle & \text{if } \alpha(0) \ni P \\ \langle \alpha', a'\rangle \; \texttt{Filter}(P) \; \langle \beta, b\rangle & \text{otherwise} \end{cases}$$

   The infix operator $\alpha(0) \ni P$ denotes whether or not the data item $\alpha(0)$ matches with the pattern P. If so, $\alpha(0)$ passes through, otherwise it is lost, and the ABT proceeds with the rest of its timed data streams.
2. A `LossySync` channel is also like a `Sync` channel except that it is always ready to consume every data item written to its source end. If a matching read operation is pending at its sink, the data item written to its source is

actually transferred; otherwise, the written data item is lost. The behavior of this channel is captured by the `LossySync` ABT defined as

$$\langle \alpha, a\rangle \ \texttt{LossySync}\ \langle \beta, b\rangle \equiv \begin{cases} \langle \alpha, a\rangle \ \texttt{LossySync}\ \langle \beta, a(0).b'\rangle & \text{if } a(0) > b(0) \\ \beta(0) = \alpha(0) \wedge \langle \alpha', a'\rangle \ \texttt{LossySync}\ \langle \beta', b'\rangle & \text{if } a(0) = b(0) \\ \langle \alpha', a'\rangle \ \texttt{LossySync}\ \langle \beta, b\rangle & \text{otherwise} \end{cases}$$

3. A `SyncDrain` is a channel with two source ends. Because it has no sink end, it has no way to ever produce any data items. Consequently, every data item written to its source ends is simply lost. `SyncDrain` is synchronous because a write operation on one of its ends remains pending until a write is performed on its other end as well; only then both write operations succeed together. The behavior of this channel is captured by the `SyncDrain` ABT defined as

$$\langle \alpha, a\rangle \ \texttt{SyncDrain}\ \langle \beta, b\rangle \equiv a = b$$

9.4 Coordinating Glue Code

To demonstrate the expressive power of connector composition, in this section we describe a number of examples in Reo. More examples are presented elsewhere [2, 6, 7, 3].

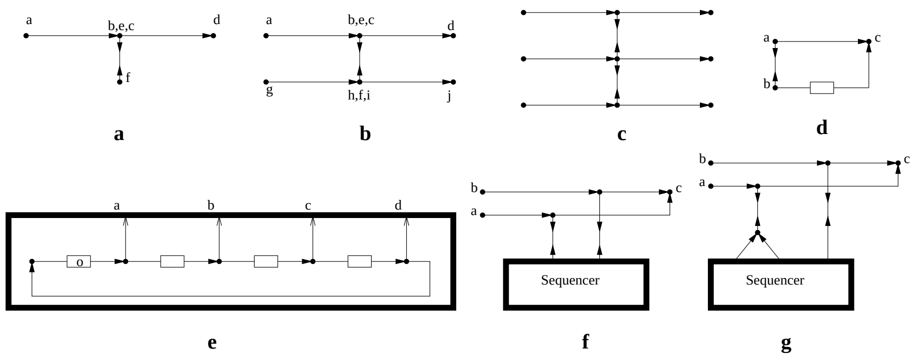

Fig. 3. Examples of connectors in Reo

Write-Cue Regulator Consider the connector in Figure 3.a, composed out of the three channels `ab`, `cd`, and `ef`. Channels `ab` and `cd` are of type `Sync` and `ef` is of type `SyncDrain`. This connector shows one of the most basic forms of exogenous coordination: the number of data items that flow from $\widehat{\texttt{a}}$ to $\widehat{\texttt{d}}$ is the same as the number of write operations that succeeds on $\widehat{\texttt{f}}$. (Recall that $\widehat{\texttt{a}}$ designates the unique node on which the channel end `a` coincides.) The analogy between the behavior of this connector and a transistor in the world of electronic circuits is conspicuous.

A component instance with a port connected to $\widehat{\mathtt{f}}$ can count and regulate the flow of data between the two nodes $\widehat{\mathtt{a}}$ and $\widehat{\mathtt{d}}$ by the timing and the number of write operations it performs on $\widehat{\mathtt{f}}$. The entity that regulates and/or counts the number of data items through $\widehat{\mathtt{f}}$ need not know anything about the entities that write to $\widehat{\mathtt{a}}$ and/or `take` from $\widehat{\mathtt{d}}$, nor that its write actions actually regulate this flow. The two entities that communicate through $\widehat{\mathtt{a}}$ and $\widehat{\mathtt{d}}$ need not know anything about the fact that they are communicating with each other, nor that the volume of their communication is regulated and/or measured by a third entity at $\widehat{\mathtt{f}}$.

Barrier Synchronizers We can build on our write-cue regulator to construct a barrier synchronization connector, as in Figure 3.b. The four channels `ab`, `cd`, `gh`, and `ij` are all of type `Sync`. The `SyncDrain` channel `ef` ensures that a data item passes from $\widehat{\mathtt{a}}$ to $\widehat{\mathtt{d}}$ only simultaneously with the passing of a data item from $\widehat{\mathtt{g}}$ to $\widehat{\mathtt{j}}$ (and vice versa). This simple barrier synchronization connector can be trivially extended to any number of pairs, as shown in Figure 3.c.

Ordering The connector in Figure 3.d consists of three channels: `ab`, `ac`, and `bc`. The channels `ab` and `ac` are `SyncDrain` and `Sync`, respectively. The channel `bc` is of type $\mathtt{FIFO}_1$. The behavior of this connector can be seen as imposing an order on the flow of the data items written to $\widehat{\mathtt{a}}$ and $\widehat{\mathtt{b}}$, through to $\widehat{\mathtt{c}}$: the data items obtained by successive read operations on $\widehat{\mathtt{c}}$ consist of the first data item written to $\widehat{\mathtt{a}}$, followed by the first data item written to $\widehat{\mathtt{b}}$, followed by the second data item written to $\widehat{\mathtt{a}}$, followed by the second data item written to $\widehat{\mathtt{b}}$, etc. See [2, 3] for more detail and [7] for a formal treatment of this connector.

The coordination pattern imposed by our connector can be summarized as $c = (ab)*$, meaning the sequence of values that appear through $\widehat{\mathtt{c}}$ consist of zero or more repetitions of the pairs of values written to $\widehat{\mathtt{a}}$ and $\widehat{\mathtt{b}}$, in that order.

Sequencer Consider the connector in Figure 3.e. The enclosing box represents the fact that the details of this connector are abstracted away and it provides only the four nodes $\widehat{\mathtt{a}}$, $\widehat{\mathtt{b}}$, $\widehat{\mathtt{c}}$, and $\widehat{\mathtt{d}}$ for other entities (connectors and/or component instances) to (in this case) read from. Inside this connector, we have four `Sync`, a $\mathtt{FIFO}_1(\mathtt{o})$, and three $\mathtt{FIFO}_1$ channels connected together. The $\mathtt{FIFO}_1(\mathtt{o})$ channel is the leftmost one and is initialized to have a data item in its buffer, as indicated by the presence of the symbol "`o`" in the box representing its buffer. The actual value of this data item is irrelevant. The read operations on the nodes $\widehat{\mathtt{a}}$, $\widehat{\mathtt{b}}$, $\widehat{\mathtt{c}}$, and $\widehat{\mathtt{d}}$ can succeed only in the strict left to right order. This connector implements a generic sequencing protocol: we can parameterize this connector to have as many nodes as we want, simply by inserting more (or fewer) `Sync` and $\mathtt{FIFO}_1$ channel pairs, as required.

Figure 3.f shows a simple example of the utility of our sequencer. The connector in this figure consists of a two-node sequencer, plus a pair of `Sync` channels and a `SyncDrain` channel connecting each of the nodes of the sequencer to the nodes $\widehat{\mathtt{a}}$ and $\widehat{\mathtt{c}}$, and $\widehat{\mathtt{b}}$ and $\widehat{\mathtt{c}}$, respectively. The connector in Figure 3.f is another connector for the coordination pattern $c = (ab)*$, although there is a subtle difference between the behavior of this connector and the one in Figure 3.d. See [2, 3] for more detail.

It takes little effort to see that the connector in Figure 3.g corresponds to the meta-regular expression $c = (aab)*$. Figures 3.f and g show how easily we can construct connectors that exogenously impose coordination patterns corresponding to the Kleen-closure of any "meta-word" made up of atoms that stand for I/O operations, using a sequencer of the appropriate size.

9.5 Fibonacci Series

A simple example of how a composition of a set of components yields a system that delivers more than the sum of its parts is the computation of the classical Fibonacci series. To assemble a component based application to deliver this series we actually need only one (instance of one) component plus a number of channels. The component we need is a realization of the *Sum* ABT that we already saw in Section 8.1.

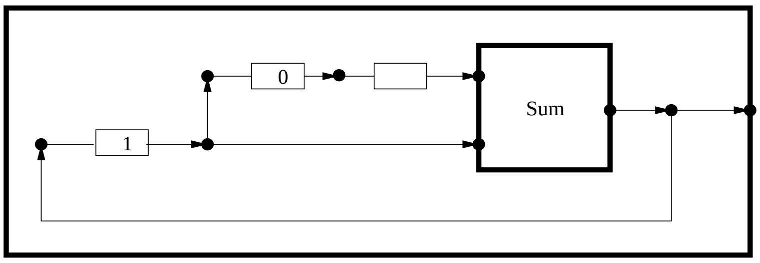

Fig. 4. Computing the Fibonacci series

Figure 4 shows a component (the outermost thick enclosing box) with only one output port (the only exposed node on the right border of the box). This is our component based application for computing the Fibonacci series. Peeking inside this component, we see how it is made out of an instance of *Sum*, a $\mathtt{FIFO}_1(1)$, a $\mathtt{FIFO}_1(0)$, a $\mathtt{FIFO}_1$, and five `Sync` channels.

As long as the $\mathtt{FIFO}_1(0)$ channel is full, nothing can happen: there is no way for the value in $\mathtt{FIFO}_1(1)$ to move out. At some point in time, the value in $\mathtt{FIFO}_1(0)$ moves into the $\mathtt{FIFO}_1$ channel. Thereafter, the $\mathtt{FIFO}_1(0)$ channel becomes empty and the two values in the $\mathtt{FIFO}_1(1)$ and the $\mathtt{FIFO}_1$ channels become available for *Sum* to consume. The intake of the value in $\mathtt{FIFO}_1(1)$ by

Sum inserts a copy of the same value into the $\texttt{FIFO}_1(0)$ channel. When *Sum* is ready to write its computed value out, it suspends waiting for some entity in the environment to accept this value. Transfer of this value to the entity in the environment also inserts a copy of the same value into the now empty $\texttt{FIFO}_1(1)$ channel. At this point we are back to the initial state, but with different values in the buffers of the $\texttt{FIFO}_1(1)$ and the $\texttt{FIFO}_1(0)$ channels.

The ABT models of the component *Sum*, channels, and Reo nodes that we presented earlier suffice for a formal analysis of the behavior of their composition in this example. Observe that all entities involved in this composed application are completely generic and, of course, neither knows anything about the Fibonacci series, nor the fact that it is "cooperating" with other entities to compute it. It is the specific glue code of this application, made by composing 8 simple generic channels in a specific topology in Reo, that coordinates the communication of the components (in this case, only one) with one another (in this case, with itself) and the environment to compute this series.

9.6 Dining Philosophers

We can vividly demonstrate the significance of exogenous coordination in component based system composition through the classical dining philosophers problem. In this section we use instances of two components, each of which is a realizations of one of the two ABTs *Phil* and *Chop* defined in Section 8.1, to (1) compose a dining philosophers application that exhibits the famous deadlock problem; and (2) compose another dining philosophers application that prevents the deadlock.

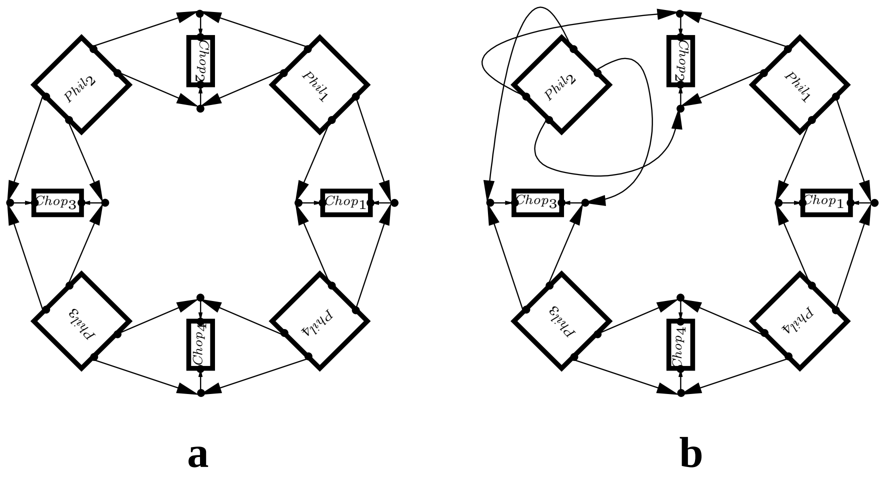

Fig. 5. Dining philosophers in Reo

Figure 5.a shows 4 philosophers and 4 chopsticks around a virtual round table. Each philosopher has 4 output ports, corresponding to the *lt*, *lf*, *rt*, and *rf* portals of the *Phil* ABT in Section 8.1. In this figure, philosophers face the table, thus their sense of left and right is obvious. Each chopstick has two input ports, corresponding to the *t* and *f* input portals of the *Chop* ABT in Section 8.1. In Figure 5.a, chopstick ports on the outer-edge of the table are their *t* ports and the ones closer to the center of the table are their *f* ports. The *t* (take) port of each chopstick is connected to the take ports of its adjacent philosophers, and its *f* port to their respective free ports. All channels are of type `Sync`.

Consider what happens in the node at the three-way junction connected to the *t* port of $Chop_1$. If $Chop_1$ is free and is ready to accept a token through its *t* port, as it initially is, whichever one of the two philosophers $Phil_1$ and $Phil_4$ happens to write its take request token first will succeed to take $Chop_1$. Of course, it is possible for $Phil_1$ and $Phil_4$ to attempt to take $Chop_1$ at the same time. In this case, the semantics of this mixed node (by the definition of the ABT *Mrg*) guarantees that only one of them succeeds, nondeterministically; the write operation of the other remains pending until $Chop_1$ is free again. Because the definition of the ABT *Phil* states that a philosopher frees a chopstick only after it has taken it, there is never any contention at the three-way junction connected to the *f* port of a chopstick.

The composition of channels in this Reo application enables philosophers to repeatedly go through their "eat" and "think" cycles at their leisure, resolving their contentions for taking the same chopsticks nondeterministically. The possibility of starvation is ruled out because the nondeterminism in *Mrg* is assumed to be fair. This simple glue code composed of nothing but common generic `Sync` channels directly renders a faithful implementation of the dining philosophers problem; all the way down to its possibility of deadlock. Because all philosophers are instances of the same component, they all attempt to fetch their chopsticks in the same order. The *Phil* ABT defines this to be left-first. If all chopsticks are free and all philosophers attempt to take their left chopsticks at the same time, of course, they will all succeed. However, this leaves no free chopstick for any philosopher to take before it can eat. No philosopher will relinquish its chopstick before it finishes its eating cycle. Therefore, this application deadlocks, as expected.

Avoiding the Deadlock Interestingly, with Reo, solving the deadlock problem requires no extra code, central authority, or modification to any of the components. In order to prevent the possibility of a deadlock, all we need to do is to change the way in which we compose our application out of the very same components. Figure 5.b shows a slightly different composition topology of the same set of `Sync` channels comprising the glue code that connects the exact same instances of *Phil* and *Chop* as before. We have flipped one philosopher's left and right connections to its adjacent chopsticks (in this particular case, those of $Phil_2$) *without its knowledge*. None of the components in the system are aware of this change, nor is any of them modified in any way to accommodate it. Our flipping of these connections is purely external to all components.

It is not difficult to see why this new topology prevents deadlock. If all philosophers attempt to take their left chopsticks now at the same time, one of them, namely $Phil_2$, will actually reach for the one on its right-hand-side. Of course, $Phil_2$ is unaware of the fact that as it reaches out through its left port to take its first chopstick, it is actually the one on its right-hand-side it competes to take. In this case it competes with $Phil_3$, which is also attempting to take its first chopstick. It makes no difference which one of the two wins this competition, one will be denied access to its first chopstick. This ensures that at least one chopstick will remain free (no philosopher attempts to take $Chop_2$ as its first chopstick) to enable at least one philosopher to obtain its second chopstick as well and complete its eating cycle.

Comparing the composition topologies in Figures 5.a and b, we see that in Reo (1) different glue code connecting the same components produces different system behavior; and (2) coordination protocols are imposed by glue code on components that cooperate with one another through the glue code, without being aware of each other or their cooperation. The two fundamental notions that underpin this pair of highly desirable provisions are:

– The underlying notion of component (Section 6) in the ABT model prevents a component from distinguishing individual entities within its environment directly. Components can exchange only passive data with their environment through communication primitives that (1) do not allow them to discern specific targets as communication partners, and (2) do not entail any further obligation on behalf of the environment. The ABT model of components, thus, grants the environment great flexibility in making late, even dynamic, decisions about how components are composed. This makes ABT components highly susceptible to exogenous coordination, although the ABT model itself offers no non-trivial coordination primitives.
– Reo is a coordination model that takes full advantage of the composition flexibility offered by the ABT model and offers a calculus of connector composition based on a user-defined set of primitive channels, all defined as ABTs. The crux of this calculus is the `join` operator in Reo for composing channel ends into composite nodes, and the specific semantics it defines for these nodes as ABTs (Section 9.2). Connector composition in Reo offers a simple yet surprisingly expressive exogenous coordination model that effectively exploits the flexibility of components in the ABT model.

The two systems in Figures 5.a and b are made of the same number of constituent parts of the same types: the same number of component instances of the same kinds, and the same number of primitive connectors (`Sync` channels). The only difference between the two is in the *topology* of their inter-connections. This topological difference is the only cause of the difference between the “more than sum of the parts” in these two systems.

Making of a Chopstick A moment of reflection reveals that, especially since there is no computation involved in the behavior of a chopstick, it should be easy to realize the behavior defined by the ABT *Chop* through channel composition. The behavior defined as *Chop* is indeed all coordination: it must alternate enabling the write operations on one (t) then on the other (f) of its two input ports. Indeed, we can easily use a two-port sequencer (Figure 3.e) plus two `SyncDrain` channels to realize this behavior. But a much simpler construction is possible as well.

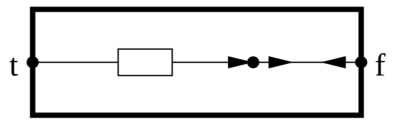

Fig. 6. Inside of a chopstick

The connector hidden inside the enclosing box in Figure 6 is a simplified two-port sequencer which exactly implements the bahavior of the ABT *Chop*. This connector consists of two channels: a $\texttt{FIFO}_1$ and a `SyncDrain`. Initially, the $\texttt{FIFO}_1$ is empty, therefore enabling the first write to its port t to succeed immediately. While this channel is empty, a write to its f port suspends because there is no data item to be "simultaneously" consumed by the opposite (source) end of the `SyncDrain`. Once a write to t succeeds, the $\texttt{FIFO}_1$ channel becomes full and the next write operation on port t will suspend until this channel becomes empty again. When the $\texttt{FIFO}_1$ channel is full, a write to f succeeds, causing the `SyncDrain` channel to consume the contents of the $\texttt{FIFO}_1$ channel as well. This returns the connector to its original state allowing it to cyclically repeat the same behavior.

Adaptation of a Philosopher As a simple example of the usefulness of `Filter`(P) channels, suppose the interface of the philosopher component we acquire for our application does not exactly match that of our *Phil* ABT. The component we obtain, *Philos* has only one output port and it writes all its tokens to the same port. Figure 7 shows how *Philos* can be adapted to fit the interface of *Phil*, using four filter channels.

The wiggly segment in the representation of a filter channel suggests a "resistor" that inhibits the transmission of values that do not match its filter pattern. The text above the wiggly line is the filter pattern. Because *Philos* writes all of its tokens to the same port, it must distinguish them by their values. We assume it writes the four values `lt`, `lf`, `rt`, and `rf` to identify these tokens. Every value written to the output port of *Philos* is automatically replicated into the source ends of the four channel filters that coincide on this node. This copying happens whenever all four source channel ends are ready to consume the replicated value. Whatever the value is, three of the four channels will always be ready to accept it

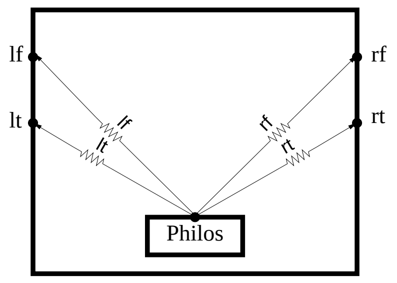

Fig. 7. Adapting *Philos* to appear as *Phil*

unconditionally, because it will not match their filters and they will immediately lose the value. The fourth channel, the one whose pattern matches the written value, is the one whose acceptance triggers the actual replication/transfer. This happens only when the node at the sink end of this filter channel can synchronously dispose of the value, which is possible only when there is a read on that node.

10 Conclusion

The operational interface that is inherent in the Abstract Data Type model and object oriented programming introduces two very different concepts for (1) entities, and (2) the mechanism of their composition. To their outside world, entities are what their interfaces advertise them to be: a set of operations. The mechanism that composes entities is based on *performing* the operations of other entities. This makes composition endogenous (i.e., an entity internally decides what operations of which other entities to perform) and relies on rather strong assumptions about the environment (i.e., the actual availability of appropriate other entities to support those operations with their expected semantics). Unlike the ADT model, main-stream object oriented models do not offer any formal semantics in their object/class interfaces. The purely syntactic nature of their interfaces becomes the weakest link in the reliability of the assumptions that underlie the validity of each composition: unless the entity that invokes the operation knows the entity whose operation it invokes rather intimately, the semantics that one assumes may be different than what the other guarantees; even subtle differences here can sabotage a composition. Furthermore, the composition of two objects does not produce another object.

Components are expected to be independent commodities, viable in their binary forms in the (not necessarily commercial) marketplace, developed, offered, exploited, deployed, integrated, maintained, and evolved by separate autonomous organizations in mutually unknown and unknowable contexts, over very long spans of time. The level of intimacy that is implicitly required of objects that compose by invoking each other's methods, is simply too unrealistic in the world of such components. Component models that rely on (variations of) object oriented programming (e.g., components as fortified collections of objects) and its composition mechanism of method invocation must, on the one hand, ameliorate its inherent endogenous rigidity (e.g., by intercepting, interpreting, retargeting, or suppressing messages), and on the other hand yield quite brittle compositions. Composition of two components, in such models, does not by itself yield another component.

Abstract Behavior Types presented in this paper offer a simpler and far more flexible model of components — and of their composition. An ABT is a mathematical construct that defines and/or constrains the behavior of an entity without any mention of operations or data types that may be used to realize that behavior. This puts the ABT model at a higher-level of abstraction than ADTs and makes it more suitable for components. The endogenous nature of their composition means that it is not possible for a third party, e.g., an entity in the environment, to compose two objects (or two ADTs) "against their own will" so to speak. In contrast, the composition of any two ABTs is always well-defined and yields another ABT.

The building blocks in the mathematical construction of the ABT model are the (generally) infinite streams that represent the externally observable sequences of I/O events that occur at an entity's interaction points (e.g., ports) through which it exchanges data with its environment. Such infinite structures, and thus the ABT model, naturally lend themselves to coalgebraic techniques and the coinduction reasoning principle. The ABT model supports a much looser coupling than is possible with ADT and is inherently amenable to exogenous coordination. We advocate both of these as highly desirable, if not essential, properties for component based systems.

The ABT model provides a simple formal foundation for definition and composition of components. However, direct composition of component ABTs does not generally provide much of an opportunity to systematically wield exogenous coordination. Reo is a channel-based exogenous coordination model that can be used as a glue language for dynamic compositional construction of component connectors in (non-)distributed and/or mobile systems. Connector construction in Reo can be seen as an application of the ABT model. A channel in Reo is just a special kind of an atomic connector (i.e., component): whereas components and connectors offer one or more ports to exchange information with their environment, a channel is an ABT that offers exactly two ports (i.e., its channel-ends) for interaction with its environment. Because all Reo connectors are ABTs, the semantics of channel composition in Reo can be defined in terms of ABT composition.

11 Acknowledgment

I am thankful for the fruitful discussions and the collaboration of all my colleagues at CWI, especially J. Rutten, M. Bonsangue, and F. de Boer, who have contributed to the ideas behind abstract behavior types. I am grateful for the attention and the creative influence of the participants in the ACG seminar series of J. de Bakker at CWI, where various aspects of Reo were presented and discussed in 2001 and 2002. I immensely appreciate the work of my colleagues involved in the development and implementation of Reo. I am particularly grateful for J. Rutten's keen interest in Reo and his inspiring work on a coalgebraic formal semantics for it.

References

1. F. Arbab. What do you mean, coordination? *Bulletin of the Dutch Association for Theoretical Computer Science, NVTI*, pages 11–22, 1998. Available on-line http://www.cwi.nl/NVTI/Nieuwsbrief/nieuwsbrief.html.
2. F. Arbab. A channel-based coordination model for component composition. Technical Report SEN-R0203, Centrum voor Wiskunde en Informatica, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands, February 2002.
3. F. Arbab. Reo: A channel-based coordination model for component composition. *Mathematical Structures in Computer Science*, 2003.
4. F. Arbab, F. S. de Boer, M. M. Bonsangue, and J. V. Guillen Scholten. MoCha: A framework for coordination using mobile channels. Technical Report SEN-R0128, Centrum voor Wiskunde en Informatica, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands, December 2001.
5. F. Arbab, F.S. de Boer, and M.M. Bonsangue. A coordination language for mobile components. In *Proc. ACM SAC'00*, 2000.
6. F. Arbab and F. Mavaddat. Coordination through channel composition. In F. Arbab and C. Talcott, editors, *Coordination Languages and Models: Proc. Coordination 2002*, volume 2315 of *Lecture Notes in Computer Science*, pages 21–38. Springer-Verlag, April 2002.
7. F. Arbab and J. J. M. M. Rutten. A coinductive calculus of component connectors. Technical Report SEN-R0216, Centrum voor Wiskunde en Informatica, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands, September 2002.
8. Farhad Arbab, F. S. de Boer, and M. M. Bonsangue. A logical interface description language for components. In Antonio Porto and Gruia-Catalin Roman, editors, *Coordination Languages and Models: Proc. Coordination 2000*, volume 1906 of *Lecture Notes in Computer Science*, pages 249–266. Springer-Verlag, September 2000.
9. J.W. de Bakker and J.N. Kok. Towards a Uniform Topological Treatment of Streams and Functions on Streams. In W. Brauer, editor, *Proceedings of the 12th International Colloquium on Automata, Languages and Programming*, volume 194 of *Lecture Notes in Computer Science*, pages 140–148, Nafplion, July 1985. Springer-Verlag.
10. L. Barbosa. *Components as Coalgebras.* PhD thesis, Universidade do Minho, Braga, Portugal, 2001.
11. M. Broy. A logical basis for component-based system engineering. Technical report, Technische Universität München, Nov. 2000.
12. M. Broy and K. Stolen. *Specification and development of interactive systems*, volume 62 of *Monographs in Computer Science.* Springer, 2001.
13. J. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt. Ptolemy: a framework for simulating and prototyping heterogeneous systems. *International Journal of Computer Simulation*, special issue on Simulation Software Development(3), January 1990.

14. CORBA. See: http://www.omg.org.
15. F. S. de Boer and M. M. Bonsangue. A compositional model for confluent dynamic data-flow networks. In M. Nielsen and B. Rovan, editors, *Proc. International Symposium of the Mathematical Foundations of Computer Science (MFCS)*, volume 1893 of *Lecture Notes in Computer Science*, pages 212–221. Springer-Verlag, August-September 2000.
16. Enterprise JavaBeans. See: http://java.sun.com/products/ejb.
17. D. Gelernter and N. Carriero. Coordination languages and their significance. *Communication of the ACM*, 35(2):97–107, February 1992.
18. J. Gore. *Object Structures: Building Object-Oriented Software Components*. Addison Wesley, 1996.
19. R. Grimes. *Professional DCOM Programming*. Wrox Press, 1997.
20. H.P. Gumm and T. Schröder. Covarieties and complete covarieties. In *[27]*, 1998.
21. H. Barringer, R. Kuiper, and A. Pnueli. A really abstract current model and its temporal logic. In *Proceedings of Thirteenth Annual ACM Symposium on principles of Programming Languages*, pages 173–183. ACM, 1986.
22. R. Hennicker and M. Wirsing. A formal method for the systematic reuse of specification components. In *Methods of Programming*, volume 544 of *LNCS*, pages 49–75. Springer Verlag, 1991.
23. F. Huber, A. Rausch, and B. Rumpe. Modeling dynamic component interfaces. In M. Singh, B. Meyer, J. Gil, and R. Mitchell, editors, *Proc. Technology of Object-Oriented Languages and Systems (TOOLS'98)*, pages 58–70. IEEE Computer Society, 1998.
24. B. Jacobs. Behaviour-refinement of object-oriented specifications with coinductive correctness proofs. Report CSI-R9618, Computing Science Institute, University of Nijmegen, 1996. Also in the proceedings of TAPSOFT'97.
25. B. Jacobs. Coalgebraic specifications and models of deterministic hybrid systems. In M. Wirsing and M. Nivat, editors, *Algebraic Methods and Software Technology*, number 1101 in Lecture Notes in Computer Science, pages 520–535. Springer-Verlag, 1996.
26. B. Jacobs. Inheritance and cofree constructions. In P. Cointe, editor, *European Conference on Object-Oriented Programming*, number 1098 in Lecture Notes in Computer Science, pages 210–231. Springer-Verlag, 1996.
27. B. Jacobs, L. Moss, H. Reichel, and J.J.M.M. Rutten, editors. *Proceedings of the first international workshop on Coalgebraic Methods in Computer Science (CMCS '98)*, volume 11 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science B.V., 1998. Available at URL: www.elsevier.nl/locate/entcs.
28. B. Jacobs and J.J.M.M. Rutten. A tutorial on (co)algebras and (co)induction. *Bulletin of EATCS*, 62:222–259, 1997. Available on-line http://www.cs.kun.nl/ bart/PAPERS/JR.ps.Z.
29. Jini. See: http://www.sun.com/jini.
30. J.N. Kok. *Semantic Models for Parallel Computation in Data Flow, Logic- and Object-Oriented Programming*. PhD thesis, Vrije Universiteit, Amsterdam, May 1989.
31. E. A. Lee and T. M. Parks. Dataflow process networks. In *Proceedings of the IEEE*, volume 83, pages 773–801, May 1995.
32. Edward A. Lee and David G. Messerschmitt. An overview of the ptolemy project. Technical report, Dept. of Electrical Engineering and Computer Sciences, University of California at Berkeley, 1993.
33. S. Li et al. *Professional Jini*. Mass Market Paperback, 2000.

34. V. Matena and B. Stearns. *Applying Enterprise JavaBeans: Component-Based Development for the J2EE Platform.* Java Series, Enterprise Edition. Addison-Wesley, 2000.
35. B. Meyer. *Eiffel: The Language.* Prentice Hall, 1992.
36. B. Meyer. *Reusable Software: The Base Object-Oriented Component Libraries.* Prentice Hall, 1994.
37. Lawrence S. Moss. Coalgebraic logic. *Annals of Pure and Applied Logic*, 96(1–3):277–317, 1999.
38. Lawrence S. Moss and Norman Danner. On the foundations of corecursion. *Logic Journal of the IGPL*, 5(2):231–257, 1997.
39. S. Oaks and H. Wong. *Jini in a Nutshell.* O'Reilly & Associates, 2000.
40. G.A. Papadopoulos and F. Arbab. Coordination models and languages. In M. Zelkowitz, editor, *Advances in Computers – The Engineering of Large Systems*, volume 46, pages 329–400. Academic Press, 1998.
41. H. Reichel. An approach to object semantics based on terminal coalgebras. *Mathematical Structures in Computer Science*, 5:129–152, 1995.
42. J. J. M. M. Rutten. Elements of stream calculus (an extensive exercise in coinduction. In S. Brookes and M. Mislove, editors, *Proc. of 17th Conf. on Mathematical Foundations of Programming Semantics, Aarhus, Denmark, 23–26 May 2001*, volume 45 of *Electronic Notes in Theoretical Computer Science.* Elsevier, Amsterdam, 2001.
43. J.J.M.M. Rutten. Universal coalgebra: A theory of systems. Technical Report CS-R9652, Centrum voor Wiskunde en Informatica, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands, 1996. Available on-line http://www.cwi.nl/ftp/CWIreports/AP/CS-R9652.ps.Z.
44. J.J.M.M. Rutten. Automata and coinduction (an exercise in coalgebra). Report SEN-R9803, CWI, 1998. Available at URL: www.cwi.nl. Also in the proceedings of CONCUR '98, LNCS 1466, 1998, pp. 194–218.
45. J.J.M.M. Rutten. Automata, power series, and coinduction: taking input derivatives seriously (extended abstract). Report SEN-R9901, CWI, 1999. Available at URL: www.cwi.nl. Also in the proceedings of ICALP '99, LNCS 1644, 1999, pp. 645–654.
46. J.J.M.M. Rutten. Coalgebra, concurrency, and control. Report SEN-R9921, CWI, 1999. Available at URL: www.cwi.nl. Extended abstract in: Discrete Event Systems, R. Boel and G. Stremersch (eds.), Kluwer, 2000.
47. J.J.M.M. Rutten. Universal coalgebra: a theory of systems. *Theoretical Computer Science*, 249(1):3–80, 2000.
48. Jon Siegel. *CORBA: Fundamentals and Programming.* John Wiley & Sons Inc., New York, 1 edition, 1996.
49. Alan Snyder. Encapsulation and inheritance in object-oriented programming languages. In *OOPSLA '86*, pages 38–45, September 1986.
50. M. Wirsing, R. Hennicker, and R. Breu. Reusable specification components. Technical Report MIP-8817, Passau University, 1988.