Abstract Behavior Types: a foundation model for components and their composition

Farhad Arbab*

Centre for Mathematics and Computer Science (CWI), Kruislaan 413, 1098 SJ Amsterdam, The Netherlands

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.
© 2004 Elsevier B.V. All rights reserved.

MSC: 68Q85; 68Q70; 81P05; 68N99; 68Q65; 18C50; 68Q10; 68Q65

* Tel.: +31 20 592 4056; fax: +31 20 592 4199.
E-mail address: farhad@cwi.nl.
URL: http://www.cwi.nl/~farhad.

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 (see, e.g., Sections 3 and 5).

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. The ABT model is not meant as a substitute for ADT. Instead, it is meant as a foundation for *programming with components* as opposed to *programming of components* which is what the ADT model and object oriented programming have proved to be suitable and effective for. 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 [3,4]. 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 abundance of the various (primarily informal) definitions of what a component is makes it difficult to assess the effectiveness of different models for component-based software development. Some differences are often purely technological. Others are more fundamental and sometimes render a side-by-side comparison of model features uninformative, if not meaningless. In this section we present a pre-formal analysis of the concepts and ideas behind the cornucopia of contemporary component models to identify the commonalities of motivations that can in turn be seen as essential requirements of a component model.

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 it should be not only 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

---

[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.

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 [2].

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.

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.

Although connectors contain coordination constructs that do not exist in purely computational components, the ability to use a composition of connectors and components exactly as a (black-box) component in other compositions is crucial. The fact that such (non-purely-computational) components may include connectors whose coordinating effects can be discernible in the externally observable behavior of the components, blurs an absolute distinction between components and connectors and makes it somewhat subjective. It takes an encapsulation mechanism with the power of full abstraction to allow components and connectors to be used indistinguishably. Examples presented in Section 9.7.2 and throughout Section 9.8 demonstrate the practical usefulness of the interchangeability of components and connectors.

3. Background and related work

In popular models of components (e.g., Enterprise Java Beans [47,24], CORBA [70,20], and DCOM [27]) 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.

Intercepting and manipulating messages before they perform the methods that their sender objects intend to invoke is also at the core of one of the most effective contemporary approaches to Aspect Oriented Programming. The so-called Composition Filters superimpose declarative rules for intercepting, filtering, re-routing, and changing the message traffic among objects to support certain inter- and intra-class cross-cutting concerns [13]. Composition Filters are most effective for imposing cross-cutting concerns that can be related to individual message contents. Protocol-related concerns that involve coordination of flows of groups of messages, e.g., synchronization at transaction boundaries, are more difficult, if not impossible, to enforce through Composition Filters. Nevertheless, this and similar approaches to Aspect Oriented Programming underline the wider recognition of the inadequacy of the rigid semantic tie between messages and method invocation even in the world of object oriented programming. They clearly show the advantage of a paradigm based on a more abstract notion of messages as *passive data* over the *active messages* of object oriented programming whose immediate consequences are strictly to invoke the designated methods of their target objects.

Coordination languages [25,58] offer an alternative for inter-component communication, as exemplified by JavaSpaces in the Jini architecture [37,56,45]. They impose a stricter sense of temporal and spatial 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 notwithstanding, 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 [49,50,26]: 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 [73,30].

Our notion of components [7,8,22] 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.

Synchronous channels are the basic primitives in π-calculus [51,52]. Some of the variants of π-calculus and its asynchronous versions [69] have been used in models proposed for component interaction and composition. Notably, Piccola [55] is an experimental component composition language based on a higher-order version of the asynchronous π-calculus, extended with explicit name-spaces called *forms*. Forms in Piccola provide a unified mechanism to address such aspects of component composition as styles, scripts, and glue code. The agents and channels provided by Piccola's underlying calculus support the coordination aspect of component composition.

In contrast to such calculi, 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 differentiates our model from 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, the potency that our model derives from mixing and composing channels of different types, and their consequent harmonious combination of synchrony and asynchrony are unique. For instance, these features of our model are in sharp contrast with the use of channels in the Ptolemy project [18,40,41] 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 [15,31]. 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. They also use data streams containing temporal markers for component composition. 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 of arbitrary types are connected to other channels, forming higher level and more sophisticated connectors for component composition. Unlike Broy's streams, the twin pairs of timed-data streams (see Section 8.1) that we use to model and compose component behavior cleanly separate the flows of time and data, yielding simple, succinct expressions for combining both synchronous and asynchronous behavior, which together with our relational (in contrast to Broy's functional) composition, empower exogenous coordination.

Inspired by both ADTs and imperative programming, a common approach to model externally observable behavior, particularly in object oriented settings, amounts to

	Stack		Queue
S:	*stack, data, boolean*	*S*:	*queue, data, boolean*
O:	$top(s) \rightarrow d$	*O*:	$first(q) \rightarrow d$
	$pop(s) \rightarrow s$		$deq(q) \rightarrow q$
	$push(s, d) \rightarrow s$		$enq(q, d) \rightarrow q$
	$empty(s) \rightarrow b$		$empty(q) \rightarrow b$
A:	$empty(\lambda) = true$	*A*:	$empty(\lambda) = true$
	$empty(push(s, d)) = false$		$empty(enq(q, d)) = false$
	$top(push(s, d) = d$		$first(enq(\lambda, d)) = d$
	$pop(push(s, d) = s$		$first(enq(enq(q, d_1), d_2)) = first(enq(q, d_1))$
	$pop(\lambda) = \epsilon_1$		$deq(enq(\lambda, d)) = \lambda$
	$pop(\epsilon_1) = \epsilon_1$		$deq(enq(enq(q, d_1), d_2)) = enq(deq(enq(q, d_1), d_2))$
	$top(\lambda) = \epsilon_2$		$deq(\lambda) = \epsilon_1$
	$top(\epsilon_1) = \epsilon_3$		$deq(\epsilon_1) = \epsilon_1$
	$empty(\epsilon_1) = \epsilon_4$		$first(\lambda) = \epsilon_2$
			$first(\epsilon_1) = \epsilon_3$
			$empty(\epsilon_1) = \epsilon_4$

Fig. 1. Abstract Data Types for stack and queue.

characterizing behavior in terms of message sequences, where a message represents an (ADT) operation or a method call, as exemplified, e.g., by the notion of behavior used for composition with contracts [29] and behavioral subtyping [46]. In contrast, our notion of behavior is more abstract and dataflow-like, where the externally observable behavior of an entity is characterized in terms of its input/output sequences of passive data. The fact that passive data can be interpreted as messages that invoke methods or operations means that our notion of behavior is at least as expressive as those based on operation sequences. Our model is more abstract because our sequences contain only atomic, uninterpreted, untargeted data that can interchangeably be sent to and received from any source or target. Method invocation sequences, on the other hand, contain typed messages and the structure and type of a message determine the operation it invokes. Each receiver must specify (at least) the set of message types it accepts in its signature. Regardless of how much of the semantics of the operations is reflected in this signature and to what extent it is used in composition, as we argue in Section 5, the operational view of behavior stifles exogenous coordination and composition.

4. Abstract Data Types

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

For example, Fig. 1 shows the formal ADT definitions for the two common data types stack and queue, in separate columns. The set 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 O in each column defines the signature of four operations. For the case of the stack, $top(s)$ is expected to produce the data item at the top of the stack s; $pop(s)$ is expected to produce the stack obtained by removing the data item at the top of the stack s; $push(s, d)$ is expected to produce a stack obtained by pushing the data item d on top of the stack s; and $empty(s)$ is expected to produce a boolean indicating whether or not the stack s is empty. For the case of the queue, $first(q)$ is expected to produce the first data item at the head of the queue q; $deq(q)$ is expected to produce the queue obtained by removing the first data item at the head of the queue q (*dequeue*); $enq(q, d)$ is expected to produce a queue obtained by adding the data item d to the tail end of the queue q (*enqueue*); and $empty(q)$ is expected to produce a boolean indicating whether or not the queue q is empty. Of course, the set 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, A, that formally defines the semantics of the operations in 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) $empty(\lambda) = true$, 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., $push(push(push(push(\lambda, d_1), d_2), d_3), d_4)$. An expression that cannot be transformed into such a canonical form, e.g., $push(push(pop(pop(push(\lambda, d_1))), d_2), d_3)$, is not a legal stack.

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:

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

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

$$deq(enq(enq(enq(enq(\lambda, d_1), d_2), d_3), d_4))$$
$$= enq(deq(enq(enq(enq(\lambda, d_1), d_2), d_3)), d_4)$$
$$= enq(enq(deq(enq(enq(\lambda, d_1), d_2))), d_3), d_4)$$
$$= enq(enq(enq(deq(enq(\lambda, d_1)), d_2), d_3), d_4)$$
$$= enq(enq(enq(\lambda, d_2), d_3), d_4).$$

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 by altering the structure of some canonical representation. 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.

In contrast to the declarative, state-less nature of ADT definitions, popular object oriented programming languages are imperative and carry the semantics of operations of objects/classes in their states. Exposing their states exposes the object/class semantics they carry, but it breaks their encapsulation. Hiding their states enforces encapsulation, but it also obscures the semantics of operations by making their effects invisible. Exposing the semantics of their operations without exposing object/class states becomes a non-trivial issue in imperative object oriented languages. Although a formal semantics of its operations is an integral part of the definition of an ADT, object/class interfaces in popular 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 imperative object oriented programming give rise to a number of problems that have already been discussed extensively in the

literature. Some counter-measures for problems such as the conflict between inheritance
and encapsulation [71], the purely syntactic nature of interfaces, and their asymmetric
specification of offered/required services, have been integrated in the design of certain
more advanced models for object oriented languages, systems, and components [49,50,19,
39,43,44,26,72]. They introduce annotations and features that incorporate some semantics
in the definitions of classes and objects and respect them in their inter-relationships
and use. A notion of behavior as sequences of operations underlies these features, e.g.,
behavioral subtyping [46] and contracts for object composition [29]. In their extreme,
welcome methods that expose and enforce (more of) the semantics of classes and objects
in their interfaces may demystify the semantics of these imperative pieces of software into
declarative formulas as succinct and directly susceptible to formal reasoning as ADTs.
What has not been explored so explicitly and extensively in the literature is how message
passing in the object oriented paradigm, i.e., the operational interface of the ADT model,
affects software composition and what alternative mechanisms can be used in its place for
components. We argue here that the elegance of the ADT model and the usefulness of
the object oriented languages non-withstanding, software construction using components,
their composition, and their coordination involve issues that are not only ill-served, but also
actually subverted by the operational nature of ADTs and its manifestation as the method
invocation semantics of message passing in the object oriented paradigm.

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 is this "rendezvous semantics" 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 inter-dependence among components that is no looser than that among objects within a component. This is not so much a shortcoming or deficiency of the ADT model. Rather, it highlights the inappropriateness of using the ADT model for purposes it was not meant to serve in the first place. To achieve the level of independence that is desirable for inter-component communication, we need a different model based on non-operational interfaces to avoid targeted active messages.

6. A bland notion of components

In a programming paradigm where (instances of) components are its primitive building blocks, all decisions about what components are to be used and how they are to be composed together to build an application must necessarily be made from outside of the application's constituent components themselves. This implies that generally, a component cannot be allowed to internally decide on the components that it composes and communicates with. This simple observation renders ADT unsuitable as a model for components. If a component, like an ADT, provides a set of operations, then the only way to communicate with a component is by *invoking* its operations, and inter-component communication becomes the same as inter-object communication. A formal model for components must provide an inter-component communication mechanism that affords a higher level of mutual independence to components than the ADT model does. In this section, we consider an alternative (to the ADT-style method invocation) communication mechanism as the crux of our definition of a component. The notion of component introduced in this section is then further refined and formalized in Section 8 as Abstract Behavior Types.

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. Passive messages allow interception, filtering, redirection, and manipulation of messages to be done, when necessary, all as legitimate *operations on data* as opposed to active messages which require ad hoc transparent entities outside of or superimposed on a programming model to perform such actions.

Untargeted messages break the asymmetry between senders and receivers that is inherent in models based on 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 (the properties and the scheme used to identify) its receiver. This inherent semantic dependency creates an obligation for the environment to fulfill on behalf of this intended receiver that 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 on passive data exchanged 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 non-deterministic 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.

A partial order over a dense set satisfies the requirements of our diluted notion of time, characterized above. As such, in the terminology of [42], we are primarily interested in an *untimed* model of computation. Imposing the stricter requirement of total order yields a *timed* version of our model, which is not our concern in this paper. Nevertheless, we use positive real numbers to represent our time streams, because of their familiarity and

simplicity, with the proviso that it is not the actual numeric values of the time moments, nor the total order on the real numbers, 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 ith element in a exceeds N.

8. Abstract Behavior Types

An ABT defines an abstract behavior as a relation among the observable input/output that occurs 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.

There are several different ways to formalize the concept of ABT. For instance, process calculi, Petri nets, logic expressions, automata, or labeled transition systems can be used to describe transformations of input to output sequences of observables. Process calculi tend to emphasize processes rather than explicit expression of their input/output behavior. Petri nets are too low level to directly represent the rich set of behavioral relations involving non-determinism, combination of synchrony and asynchrony, and compositionality that we are interested in. Automata can characterize the relation among the observable input/output sequences of an ABT in an operational style. Indeed, *constraint automata* [10] constitute an appropriate formalization of the ABT model, precisely because they were devised to represent the operational semantics of Reo connector circuits for model checking. Constraint automata can be considered generalizations of probabilistic automata, where data constraints, instead of probabilities, label state transitions and influence their firing. Timed-data-streams, which constitute the foundation of the coalgebraic semantics of Reo [6,68], are also the referents in the language of constraint automata. Constraint automata seem to be more useful than labeled transition systems for modeling of systems composed of both synchronous and asynchronous components, because in practice, their composition tends to yield smaller models [48].

In this paper, we do not further elaborate on labeled transition systems or constraint automata. Instead, we formally define an ABT to be a (maximal) relation among a set of timed-data-streams. We prefer this particular formalization here because it emphasizes the relational aspect of the ABT model explicitly, and abstracts away any hint of an underlying operational semantics of its implementation. This helps to focus on behavior specifications and their composition, rather than on operations that may be used to implement entities that exhibit such behavior and their interactions.

---

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

The notion of timed-data-streams as well as most of the technical content in this section comes from the work of J. Rutten on coalgebras [61,35], stream calculus [66], and a coalgebraic semantics for Reo [6,68]. Coalgebraic methods have been used for signal flow graphs, 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 [57,67,65,53,54,62–64,28,32]). Coalgebras have also been used as models for various programming paradigms, notably for objects and classes (see, e.g., [60,34], and [33]). One of the first applications of coalgebras to components appears in [11]. Analogous to the way in which algebraic methods constitute suitable models for the syntactic structure of systems, from its inception, the coalgebraic approach was implicitly recognized as a promising mathematical foundation for modeling the dynamic behavior of (concurrent) systems. The coalgebraic formulation of the ABT model presented here shows this contrast between structure and behavior in comparison with the algebraic formulation of the ADT model in Section 4.

Defining observable behavior in terms of input/output implants a dataflow essence within ABTs akin to such dataflow-like networks and calculi as [21,38], and especially [17]. 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.

8.1. Streams and coinduction

A *stream* (over A) is an infinite sequence of elements of some set A. The set of all streams over A is denoted as A^ω. Streams in $DS = D^\omega$ over a set of (uninterpreted) data items D 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., $\alpha(0), \alpha(1), \alpha(2), \ldots$ 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 [66], the well-known operations of head and tail on streams are called *initial value* and *derivative*: the initial 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 kth 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 $\alpha(0) \geq \beta(0), \alpha(1) \geq \beta(1), \alpha(2) \geq \beta(2), \ldots$.

Constrained streams in $TS = \mathbb{R}_+^\omega$ over positive real numbers representing moments in time are called *time streams* 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 [12], 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., [6]). 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 $\langle \alpha, a \rangle$ in $TDS = DS \times TS$ consisting of a data stream $\alpha \in DS$ and a time stream $a \in TS$, 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 8.3.

Generally, we use the prefix notation $R(I_1, I_2, \ldots, I_m; O_1, O_2, \ldots, O_n)$ 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 $I_i, 0 < i \leq m$ and the $n \geq 0$ sets of output timed data streams $O_j, 0 < j \leq n$. 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 ith input and the jth 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 ABT's portal that corresponds to that port.

8.2. ABT examples

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

8.2.1. 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

$$\langle \alpha, a \rangle \text{ Sync } \langle \beta, b \rangle \equiv \langle \alpha, a \rangle = \langle \beta, b \rangle.$$

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

$$\langle \alpha, a \rangle \text{ FIFO } \langle \beta, b \rangle \equiv \alpha = \beta \wedge a < b.$$

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 FIFO_1 ABT, defined as

$$\langle \alpha, a \rangle \text{ FIFO}_1 \langle \beta, b \rangle \equiv \alpha = \beta \wedge a < b < a'.$$

The 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 $\text{FIFO}_1(D)$ ABT, defined as

$$\langle \alpha, a \rangle \text{ FIFO}_1(D) \langle \beta, b \rangle \equiv \beta = D.\alpha \wedge b < a < b'.$$

The $\text{FIFO}_1(D)$ ABT represents the behavior of any entity that (1) produces an output data stream $\beta = D.\alpha$ consisting of the initial data item D followed by the input data stream α of the ABT, and (2) for $i \geq 0$ performs its ith input operation some time between its ith 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 FIFO_k ABT, defined as

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

Recall the $a^{(k)}$ is the kth-derivative (i.e., the kth-tail) of the stream a. The 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 kth-next input element is observed ($b < a^{(k)}$ which means $b(i) < a(i + k)$ for all $i \geq 0$). Observe that FIFO_1 is indeed a special case of FIFO_k with $k = 1$.

It is illuminating to compare the FIFO ABT defined above with the definition of the queue ADT in Fig. 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.

8.2.2. 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:

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

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 non-deterministically 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* non-deterministically 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 $Rpl(\langle \alpha, a \rangle; \langle \beta, b \rangle, \langle \gamma, c \rangle) = Rpl(\langle \alpha, a \rangle; \langle \gamma, c \rangle, \langle \beta, b \rangle)$.

8.2.3. 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

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

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 ($t < min(a(1), b(1))$).

8.2.4. 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

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

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 the 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

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

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.

8.3. 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 the two ABTs $B_1(I1_1, I1_2, \ldots, I1_{p_i}; O1_1, O1_2, \ldots, O1_{p_o})$ and $B_2(I2_1, I2_2, \ldots, I2_{q_i}; O2_1, O2_2, \ldots, O2_{q_o})$ 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, $B(I_1, I_2, \ldots, I_m; O_1, O_2, \ldots, O_n)$, 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 $I_1, I_2, \ldots, I_m$ is obtained from the list $I1_1, I1_2, \ldots, I1_{p_i}, I2_1, I2_2, \ldots, I2_{q_i}$ by eliminating every one of its elements involved in a dyad. Similarly, the list $O_1, O_2, \ldots, O_n$ is obtained from the list $O1_1, O1_2, \ldots, O1_{p_o}, O2_1, O2_2, \ldots, O2_{q_o}$ 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 $B = B_1(\langle \alpha, a \rangle, \langle \beta, b \rangle^1; \langle \gamma, c \rangle^2) \circ B_2(\langle \delta, d \rangle^2; \langle \mu, m \rangle^1, \langle v, n \rangle)$, 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 $B \equiv \{\langle \langle \alpha, a \rangle; \langle v, n \rangle \rangle \mid \langle \langle \alpha, a \rangle, \langle \beta, b \rangle; \langle \gamma, c \rangle \rangle \in B_1 \wedge \langle \langle \delta, d \rangle; \langle \mu, m \rangle, \langle v, n \rangle \rangle \in B_2 \wedge \langle \beta, b \rangle = \langle \mu, m \rangle \wedge \langle \gamma, c \rangle = \langle \delta, d \rangle \}$.

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 $B_1(\ldots; \langle \alpha, a \rangle) \circ B_2(\langle \beta, b \rangle; \ldots)$ instead of $B_1(\ldots; \langle \alpha, a \rangle^1) \circ B_2(\langle \beta, b \rangle^1; \ldots)$. 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 8.2. 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 `FIFO₁` ABTs produces a `FIFO₂` ABT. The formal proof of this latter equivalence relies on our notion of dense time (as opposed to discrete time) and is given in [6], 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 [5,4,6]. 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 [3,4,6]. 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. Logical mobility changes the topology of the interconnections of communicating entities, while physical mobility can have other implications, e.g., on an entity's (efficiency of) access to various resources. An efficient distributed implementation of channels supporting this notion of mobility is described in [9]. Both component instances and channels are mobile in Reo. Logical mobility of channel ends in Reo allows dynamic reconfiguration of connectors, even while they are being used by component instances. In this respect, Reo resembles dynamically reconfigurable generalized Kahn networks, as in IWIM [1] and Manifold [14], and its dataflow nature is also related to Broy's timed dataflow model, although Reo is more general and more expressive than these and similar models. Much as Reo supports physical mobility through its `move` operation to allow more efficient flow of data, it ascribes no semantic significance to it. The `move` operation does not semantically affect connector topologies, flow of data, or connectivity of components to connectors. In this sense, Reo is orthogonal to the concerns involving the physical mobility of code, e.g., in models such as that of [59].

An important aspect of Reo which is not covered in this paper is that the topology of connectors in Reo is inherently dynamic. This means that the configuration of a component-based system can dynamically change not only due to dynamic construction and connection/disconnection of connectors and component instances, but also – and more interestingly – due to dynamic reconfiguration of instantiated connectors even as they are actively in use. Moreover, Reo supports a very liberal notion of channels. As such, Reo is more general than dataflow models, Kahn-networks, and Petri-nets, all of which can be viewed as specialized channel-based models that incorporate certain specific primitive coordination constructs. Broy's work on timed dataflow channels [16,17] is perhaps closest to Reo. Nevertheless, Reo's more general notion of channels, its inherent dynamic topology, its powerful exogenous coordination that uses a clear separation of

flows of data and time, and the fundamental notion of channel/connector composition that allows, among other things, compositions involving an expressive mix of synchrony and asynchrony, distinguish it from this model as well.

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 8.2. 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 [3,5,4].

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 or read from a sink node using node I/O operations of Reo only if it is *connected* to that node. Connection of a node to a component instance gives the latter the exclusive right to perform I/O operations on that node. Reo provides operations to change the connection of nodes to component instances dynamically, but a node can be connected to at most a single component instance at any given time. This is a prerequisite for the formal notion of compositionality presented in [7].

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 Figs. 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
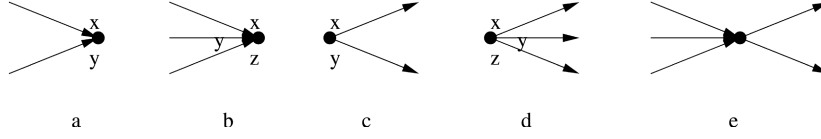
Fig. 2. Representation of nodes in Reo.

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 non-deterministically 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 non-deterministically 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 Fig. 2a. 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 $Mrg(\langle \alpha_x, a_x \rangle, \langle \alpha_y, a_y \rangle; \langle \alpha_N, a_N \rangle)$.

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

$$Mrg3(\langle \alpha_x, a_x \rangle, \langle \alpha_y, a_y \rangle, \langle \alpha_z, a_z \rangle; \langle \alpha_N, a_N \rangle)) \equiv$$
$$Mrg(\langle \alpha_x, a_x \rangle, \langle \alpha_y, a_y \rangle; \langle \psi, p \rangle^1) \circ Mrg(\langle \xi, q \rangle^1, \langle \alpha_z, a_z \rangle; \langle \alpha_N, a_N \rangle)$$

where $\langle \alpha_N, a_N \rangle$ is the output timed data stream of the node, as before, and $\langle \psi, p \rangle$ and $\langle \xi, q \rangle$ 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., *Mrg*3 is associative with respect to its input portals. As such, the simple graphical notation of Reo (e.g., in Fig. 2a 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 *Mrg*k with $k > 0$ input and one output portals. For completeness, we define $Mrg1(\langle \alpha_x, a_x \rangle; \langle \alpha_N, a_N \rangle)) \equiv \langle \alpha_x, a_x \rangle = \langle \alpha_N, a_N \rangle$ and consider *Mrg*2 to be a pseudonym for *Mrg*.

The write operations performed on a source node N with $[N] = \{x, y\}$, as in Fig. 2c, 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 $Rpl(\langle \alpha_N, a_N \rangle; \langle \alpha_x, a_x \rangle, \langle \alpha_y, a_y \rangle)$. The semantics of a source node N with $[N] = \{x, y, z\}$, as in Fig. 2d, is defined as the 1-dyad composition

$$Rpl3(\langle \alpha_N, a_N \rangle; \langle \alpha_x, a_x \rangle, \langle \alpha_y, a_y \rangle, \langle \alpha_z, a_z \rangle) \equiv$$
$$Rpl(\langle \alpha_N, a_N \rangle; \langle \alpha_x, a_x \rangle, \langle \psi, p \rangle^1) \circ Rpl(\langle \xi, q \rangle^1; \langle \alpha_y, a_y \rangle, \langle \alpha_z, a_z \rangle)$$

where $\langle \alpha_N, a_N \rangle$ is the input timed data stream of the node, as before, and $\langle \psi, p \rangle$ and $\langle \xi, q \rangle$ are internal timed data streams. Because Rpl is associative with respect to its output portals, the precedence of the Rpl operations is irrelevant and $Rpl3$ is also associative with respect to its output portals. Similarly, the general ABT $Rplk$ with one input and $k > 0$ output portals defines the semantics of a source node with k coincident channel ends. Again, for completeness, we define $Rpl1(\langle \alpha_N, a_N \rangle; \langle \alpha_x, a_x \rangle)) \equiv \langle \alpha_x, a_x \rangle = \langle \alpha_N, a_N \rangle$ and consider $Rpl2$ to be a pseudonym for Rpl.

A mixed node, as in Fig. 2e, 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 $m > 0$ sink and $n > 0$ source channel ends is given by the ABT $Node_{m \times n}$ defined as the 1-dyad composition of the two ABTs describing the behavior of each of its half nodes: $Mrgm(I_1, I_2, \ldots, I_m; \langle \psi, p \rangle)$ and $Rpln(\langle \xi, q \rangle; O_1, O_2, \ldots, O_n)$. The portals I_i and O_j designate the timed data streams observed at the m sink and the n source channel ends coincident on N, respectively, and $\langle \psi, p \rangle$ and $\langle \xi, q \rangle$ are internal timed data streams.

$$Node_{m \times n}(I_1, I_2, \ldots, I_m; O_1, O_2, \ldots, O_n)$$
$$\equiv Mrgm(I_1, I_2, \ldots, I_m; \langle \psi, p \rangle) \circ Rpln(\langle \xi, q \rangle; O_1, O_2, \ldots, O_n).$$

For instance, the behavior of the mixed node in Fig. 2e is captured by the ABT defined as the relation $Node_{3 \times 2}(I_1, I_2, I_3; O_1, O_2)$ over the timed data streams of its respective 3 sink and 2 source channel ends. For consistency, we use $Node_{m \times 0}$ and $Node_{0 \times n}$ to represent the ABTs for a sink node with m and a source node with n coincident channel ends, respectively:

$$Node_{m \times 0} \equiv Mrgm(I_1, I_2, \ldots, I_m; \langle \alpha, a \rangle)$$
$$Node_{0 \times n} \equiv Rpln(\langle \alpha, a \rangle; O_1, O_2, \ldots, O_n)$$

where I_i and O_j designate the timed data streams observed at the m sink and the n source channel ends coincident on the node and $\langle \alpha, a \rangle$ represents its output or input timed data stream.

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, $\text{FIFO}_1(D)$, 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 8.2. 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

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

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

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

(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
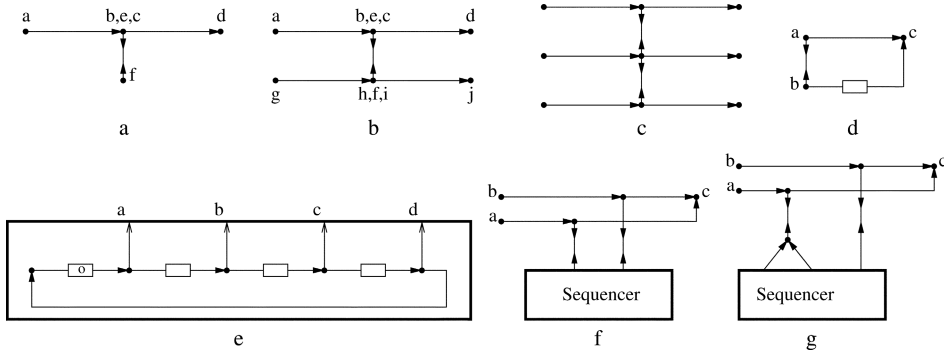
Fig. 3. Examples of connectors in Reo.

both write operations succeed together. The behavior of this channel is captured by the SyncDrain ABT defined as

$$\text{SyncDrain}(\langle \alpha, a \rangle, \langle \beta, b \rangle;) \equiv a = b.$$

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 [3,5,6,4].

9.4.1. Write-cue regulator

Consider the connector in Fig. 3a, 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 $\widehat{a}$ to $\widehat{d}$ is the same as the number of write operations that succeeds on $\widehat{f}$. (Recall that $\widehat{a}$ 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 $\widehat{f}$ can count and regulate the flow of data between the two nodes $\widehat{a}$ and $\widehat{d}$ by the timing and the number of write operations it performs on $\widehat{f}$. The entity that regulates and/or counts the number of data items through $\widehat{f}$ need not know anything about the entities that write to $\widehat{a}$ and/or take from $\widehat{d}$, nor that its write actions actually regulate this flow. The two entities that communicate through $\widehat{a}$ and $\widehat{d}$ 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 $\widehat{f}$.

9.4.2. Barrier synchronizers

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

9.4.3. Ordering

The connector in Fig. 3d consists of three channels: `ab`, `ac`, and `bc`. The channels `ab` and `ac` are `SyncDrain` and `Sync`, respectively. The channel `bc` is of type $FIFO_1$. The behavior of this connector can be seen as imposing an order on the flow of the data items written to $\widehat{a}$ and $\widehat{b}$, through to $\widehat{c}$: the data items obtained by successive read operations on $\widehat{c}$ consist of the first data item written to $\widehat{a}$, followed by the first data item written to $\widehat{b}$, followed by the second data item written to $\widehat{a}$, followed by the second data item written to $\widehat{b}$, etc. See [3,4] for more detail and [6] 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 $\widehat{c}$ consist of zero or more repetitions of the pairs of values written to $\widehat{a}$ and $\widehat{b}$, in that order.

9.4.4. Sequencer

Consider the connector in Fig. 3e. The enclosing box represents the fact that the details of this connector are abstracted away and it provides only the four nodes $\widehat{a}$, $\widehat{b}$, $\widehat{c}$, and $\widehat{d}$ for other entities (connectors and/or component instances) to (in this case) read from. Inside this connector, we have four `Sync`, a $FIFO_1(o)$, and three $FIFO_1$ channels connected together. The $FIFO_1(o)$ 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 $\widehat{a}$, $\widehat{b}$, $\widehat{c}$, and $\widehat{d}$ 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 $FIFO_1$ channel pairs, as required.

Fig. 3f 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 $\widehat{a}$ and $\widehat{c}$, and $\widehat{b}$ and $\widehat{c}$, respectively. The connector in Fig. 3f 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 Fig. 3d. See [3,4] for more detail.

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

9.5. Constant replacer

Fig. 4 shows a Reo connector (encapsulated in the outermost thick box, hiding mixed nodes N1 and N2) with one exposed input (i.e., source node A) and one exposed output (i.e., sink node B) nodes. This connector is composed out of four channels: a `SyncDrain` (A-N1), a `Sync` (N1-B), a $FIFO_1$ (N1-N2), and a filled $FIFO_1(T)$ (N2-N1) that contains an initial value T. Of course, the constructor of this connector can be parameterized to initialize this $FIFO_1$ channel with any supplied value, instead of T, every time it creates a new instance of this circuit.
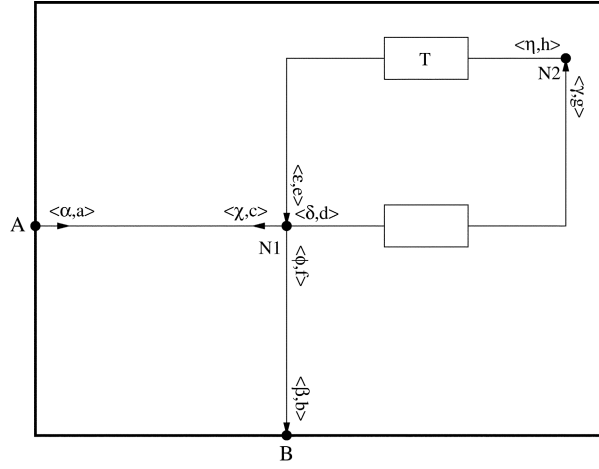
Fig. 4. Constant replacer.

Using the ABT definitions of these channels and those of its nodes, we can find the relationship between the two timed data streams $\langle \alpha, a \rangle$ and $\langle \beta, b \rangle$ that pass through the nodes A and B, respectively. As a side note, it is interesting to observe that this relationship and other insights we gain, below, through a formal treatment of the behavioral equations of this connector, all correspond to and confirm the intuitive impression that we get through an informal reasoning using the schematic of this connector in Fig. 4. This observation underscores the usefulness and the significance of visual representation of Reo connectors.

From the definition of the `SyncDrain` channel in Section 9.3, we have:

$$\texttt{SyncDrain}(\langle \alpha, a \rangle, \langle \chi, c \rangle;) \equiv a = c. \tag{1}$$

By letting $\texttt{m} = 1$ and $\texttt{n} = 3$ in the ABT equation of mixed nodes in Section 9.2, we derive for N1:

$$
\begin{aligned}
&Node_{1 \times 3}(\langle \epsilon, e \rangle; \langle \chi, c \rangle, \langle \phi, f \rangle, \langle \delta, d \rangle) \equiv \\
&\quad Mrg1(\langle \epsilon, e \rangle; \langle \psi, p \rangle) \circ Rpl3(\langle \xi, q \rangle; \langle \chi, c \rangle, \langle \phi, f \rangle, \langle \delta, d \rangle) \equiv \\
&\quad \{\langle \langle \epsilon, e \rangle; \langle \chi, c \rangle, \langle \phi, f \rangle, \langle \delta, d \rangle \rangle \mid \langle \langle \epsilon, e \rangle; \langle \psi, p \rangle \rangle \in Mrg1 \wedge \\
&\quad \langle \langle \xi, q \rangle; \langle \chi, c \rangle, \langle \phi, f \rangle, \langle \delta, d \rangle \rangle \in Rpl3 \wedge \langle \psi, p \rangle = \langle \xi, q \rangle \}.
\end{aligned}
\tag{2}
$$

From the definition of $Mrg1$ in Section 9.2, we get $\langle \langle \epsilon, e \rangle; \langle \psi, p \rangle \rangle \in Mrg1 \equiv \langle \epsilon, e \rangle = \langle \psi, p \rangle$ and from the definition of $Rpl3$ in the same section, we have $\langle \langle \xi, q \rangle; \langle \chi, c \rangle, \langle \phi, f \rangle, \langle \delta, d \rangle \rangle \in Rpl3 \equiv \langle \xi, q \rangle = \langle \chi, c \rangle = \langle \phi, f \rangle = \langle \delta, d \rangle$. Substituting these back in Eq. (2) and simplifying the result yields:

$$
\begin{aligned}
&Node_{1 \times 3}(\langle \epsilon, e \rangle; \langle \chi, c \rangle, \langle \phi, f \rangle, \langle \delta, d \rangle) \equiv \\
&\quad \{\langle \langle \epsilon, e \rangle; \langle \chi, c \rangle, \langle \phi, f \rangle, \langle \delta, d \rangle \rangle \mid \langle \epsilon, e \rangle = \langle \chi, c \rangle = \langle \phi, f \rangle = \langle \delta, d \rangle \},
\end{aligned}
\tag{3}
$$

or

$$Node_{1\times3}(\langle\epsilon, e\rangle; \langle\chi, c\rangle, \langle\phi, f\rangle, \langle\delta, d\rangle) \equiv$$
$$\{\langle\langle\epsilon, e\rangle; \langle\chi, c\rangle, \langle\phi, f\rangle, \langle\delta, d\rangle\rangle \mid \epsilon = \chi = \phi = \delta \wedge e = c = f = d\}. \tag{4}$$

Similarly, for the node N2, we derive:

$$Node_{1\times1}(\langle\gamma, g\rangle; \langle\eta, h\rangle) \equiv \{\langle\langle\gamma, g\rangle; \langle\eta, h\rangle\rangle \mid \gamma = \eta \wedge g = h\}. \tag{5}$$

From the definition of the $\texttt{FIFO}_1$ channel in Section 8.2.1, we have:

$$\langle\delta, d\rangle\ \texttt{FIFO}_1\ \langle\gamma, g\rangle \equiv \delta = \gamma \wedge d < g < d', \tag{6}$$

and the definition of the initialized $\texttt{FIFO}_1$ channel in the same section yields:

$$\langle\eta, h\rangle\ \texttt{FIFO}_1\texttt{(T)}\ \langle\epsilon, e\rangle \equiv \epsilon(0) = \texttt{T} \wedge \eta = \epsilon' \wedge e < h < e'. \tag{7}$$

The definition of the $\texttt{Sync}$ channel in Section 8.2.1 gives:

$$\langle\phi, f\rangle\ \texttt{Sync}\ \langle\beta, b\rangle \equiv \langle\phi, f\rangle = \langle\beta, b\rangle \equiv \phi = \beta \wedge f = b. \tag{8}$$

From Eqs. (1) and (4) we get $d = a$, which together with Eq. (6) yields:

$$a < g < a'. \tag{9}$$

From Eq. (8) we have $f = b$, which together with Eqs. (4) and (1), yields:

$$b = a. \tag{10}$$

From Eq. (7) we have $\epsilon(0) = \texttt{T} \wedge \eta = \epsilon'$ and from Eqs. (4) and (8) we have $\epsilon = \phi = \beta$. This gives us:

$$\beta(0) = \texttt{T} \wedge \beta' = \eta. \tag{11}$$

From Eq. (5) we have $\eta = \gamma$ and from Eq. (6) we have $\gamma = \delta$. But from Eq. (4) we have $\delta = \phi$ and from Eq. (8) we have $\phi = \beta$, therefore $\eta = \beta$, which simplifies Eq. (11) into:

$$\beta(0) = \texttt{T} \wedge \beta' = \beta. \tag{12}$$

Observe that the stream equation $\beta' = \beta$ is just a shorthand for the infinite set of equations $\beta(1) = \beta(0) \wedge \beta(2) = \beta(1) \wedge \beta(3) = \beta(2) \wedge \beta(4) = \beta(3) \wedge \ldots$. This simplifies Eq. (12) into $\beta(0) = \texttt{T} \wedge \beta(1) = \beta(0) \wedge \beta(2) = \beta(1) \wedge \beta(3) = \beta(2) \wedge \beta(4) = \beta(3) \wedge \ldots$, or:

$$\forall i \in \mathbb{N}_+ \quad \beta(i) = \texttt{T}. \tag{13}$$

Eq. (13) clearly shows that there is no relationship between the stream of input values, α, and the stream of output values, β, of this connector: whatever value comes through the node A, its corresponding output value through the node B is the constant value T. On the other hand, Eq. (10) relates the input/output "timings" of this connector: passage of each pair of values through the nodes A and B is atomic.

Eq. (9) shows an internal subtlety of the behavior of this connector. For $i > 0$, the value $\alpha(i)$ (and its corresponding value $\beta(i)$) can pass through A (and B) only after $g(i - 1)$. In other words, the constant value T must cycle through the node N2 once before the next
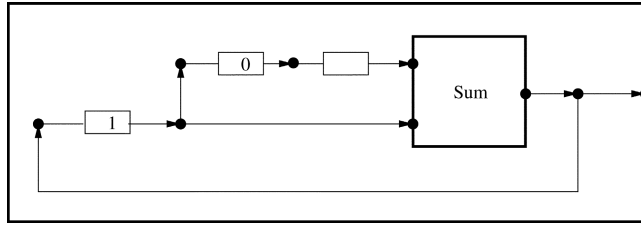
Fig. 5. Computing the Fibonacci series.

pair of values passes through the nodes A and B. In theory, it is always possible to use "fast enough" internal channels such that this cycling of value through the node N2 does not "slow down" the passing of values through A and B. On the other hand, the relational nature of our behavioral equations implies that, in practice, the internal cycling of T will delay value transfers through A and B, if necessary, such that Eq. (9) holds.

9.6. 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 8.2.

Fig. 5 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 $FIFO_1(1)$, a $FIFO_1(0)$, a $FIFO_1$, and five Sync channels.

As long as the $FIFO_1(0)$ channel is full, nothing can happen: there is no way for the value in $FIFO_1(1)$ to move out. At some point in time, the value in $FIFO_1(0)$ moves into the $FIFO_1$ channel. Thereafter, the $FIFO_1(0)$ channel becomes empty and the two values in the $FIFO_1(1)$ and the $FIFO_1$ channels become available for *Sum* to consume. The intake of the value in $FIFO_1(1)$ by *Sum* inserts a copy of the same value into the $FIFO_1(0)$ 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 $FIFO_1(1)$ channel. At this point we are back to the initial state, but with different values in the buffers of the $FIFO_1(1)$ and the $FIFO_1(0)$ 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,
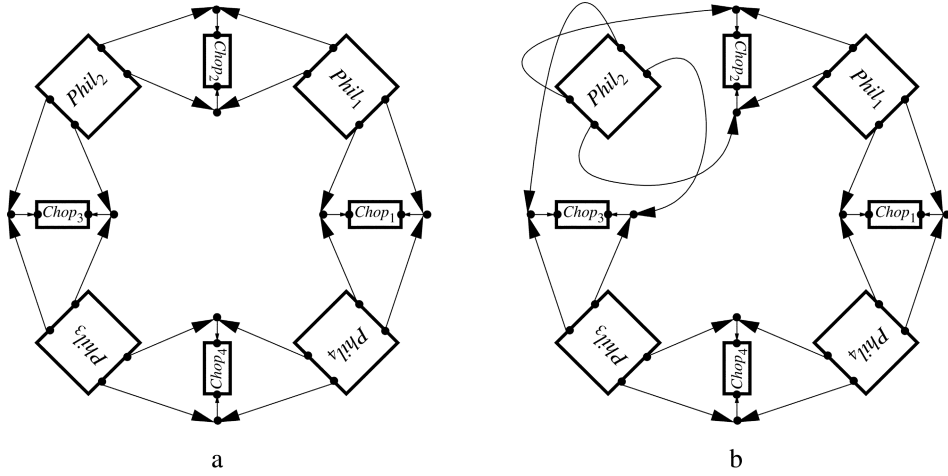
Fig. 6. Dining philosophers in Reo.

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.

9.7. 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 realization of one of the two ABTs *Phil* and *Chop* defined in Section 8.2.4, 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. 6a 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 8.2.4. 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 8.2.4. In Fig. 6a, 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, non-deterministically; 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 non-deterministically. The possibility of starvation is ruled out because the non-determinism 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.

9.7.1. 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. Fig. 6b 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 Fig. 6a 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
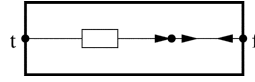
Fig. 7. Inside of a chopstick.

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 inherent in 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 behavior specification in the ABT model.

The two systems in Fig. 6a 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.

9.7.2. 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 (Fig. 3e) plus two `SyncDrain` channels to realize this behavior. But a much simpler construction is possible as well.

The connector hidden inside the enclosing box in Fig. 7 is a simplified two-port sequencer which exactly implements the behavior of the ABT *Chop*. This connector consists of two channels: a $FIFO_1$ and a `SyncDrain`. Initially, the $FIFO_1$ 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 $FIFO_1$ channel becomes full and the next write operation on port t will suspend until this channel becomes empty again. When the $FIFO_1$ channel is full, a write to f succeeds, causing the `SyncDrain` channel to consume the contents of the $FIFO_1$ channel as well. This returns the connector to its original state allowing it to cyclically repeat the same behavior.
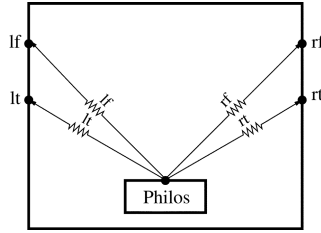
Fig. 8. Adapting *Philos* to appear as *Phil*.

9.7.3. 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. Fig. 8 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 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.

9.8. Microwave oven interface

In this section we present a number of generally useful connectors and show how they can be combined with a number of equally generic – plus some application specific – components to compose the user interface of a simple microwave oven. Of course, any of these connectors can be programmed as a black-box component as well, but in order to show the expressive power of Reo in practice, we construct them here out of the set of primitive channels of Section 9.3. In the sequel, we use the terms "input node" and "output node" of a connector or component as synonyms for, respectively, "source node" and "sink node" through which that component or connector consumes and produces data. For a component, its input and output nodes are its input and output ports, respectively.

9.8.1. Exclusive router

Fig. 9a shows a very useful generic connector, called *exclusive router*. The dashed arrows are our graphical symbols representing LossySync channels. A data item arriving
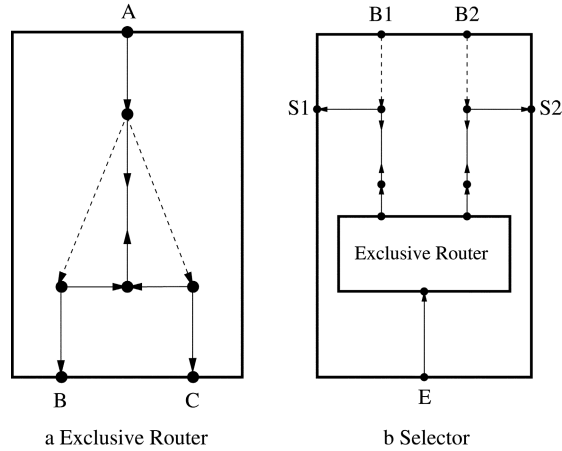
Fig. 9. An exclusive router and a selector.

at the input node A flows through to only one of the output nodes B or C, depending on which one is ready to consume it. If both output nodes are prepared to consume a data item, then one is selected non-deterministically (by the merger ABT inherent in the mixed node in the middle of this connector). The input data is never lost nor replicated to more than one of the output nodes.

The behavior of this connector is the counterpart of the primitive non-deterministic selection inherent in the merge that a Reo (sink or mixed) node performs on its multiple input. In [10] we show this by deriving the constraint automaton for this connector through composition of the constraint automata of its constituent channels.

9.8.2. Selector

Fig. 9b shows another generically useful connector, called *selector*, which uses an exclusive router of Section 9.8.1. Availability of a value at its input node E "enables" this connector to select a value available on one of the input nodes B1 or B2 for transfer through its respective output node S1 or S2. The prerequisite for a transfer of a value from B1 to S1 is the availability of a value at E and another at B1, plus the readiness of a consumer at S1. Likewise, the prerequisite for a transfer of a value from B2 to S2 is the availability of a value at E and another at B2, plus the readiness of a consumer at S2. As long as no value is available on E, all input values on B1 and B2 are lost, while consumers at S1 and S2, if any, wait. If a value is available on E while a value and a ready consumer are available for only one of the two pairs Bi–Si, then a value transfer takes place for that pair and the value on the other Bi node, if any, is lost. If both pairs have available values and ready consumers (when a value is available on E), then the exclusive router non-deterministically enables the value transfer for one of the two pairs and the value on the other Bi node is lost.

9.8.3. Shift-lossy FIFO$_1$

Fig. 10a (due to J. Rutten) shows a connector that is often used as a simple channel in the construction of other connectors. It is called *shift-lossy* FIFO$_1$ because its behavior is
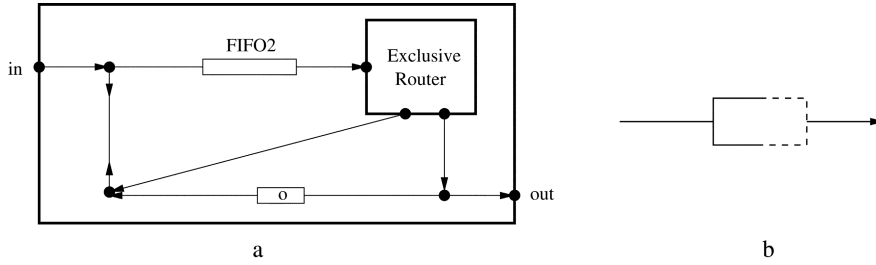
Fig. 10. Shift-lossy `FIFO`$_1$ channel.
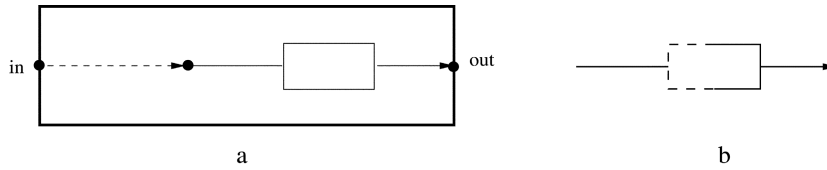
Fig. 11. Overflow-lossy `FIFO`$_1$ channel.

very similar to our `FIFO`$_1$ primitive channel of Section 9.3, except that it loses its current value if its buffer is full to accept a new input value instead. Thus, unlike the case of a `FIFO`$_1$, no write to the `in` node of this connector/channel is ever delayed: as far as the writer is concerned, a shift-lossy `FIFO`$_1$ channel behaves the same as an unbounded `FIFO`. However, this channel keeps only the last of its input values for transfer through its `out` node.

The connector in Fig. 10a is composed out of an exclusive router, a `FIFO`$_1$ channel initialized to contain a token value (o), a `FIFO`$_2$ channel, a `SyncDrain` channel, and 4 `Sync` channels. We derive the constraint automaton describing the behavior of this connector in [10] by composing the constraint automata of its constituent channels.

Because this connector is so commonly useful, we treat it as a channel and use the graphical symbol shown in Fig. 10b to represent it. This symbol is intentionally similar to that of a `FIFO`$_1$ primitive channel. The dashed half of the box representing its buffer suggests data loss. The fact that the sink-side of the box is dashed suggests that older values in the buffer are lost in favor of newer values, i.e., the buffer contents shift to the sink side, losing older values, to make room for newer arrivals.

9.8.4. Overflow-lossy `FIFO`$_1$

The counterpart of the shift-lossy `FIFO`$_1$ channel of Section 9.8.3 is the connector in Fig. 11a. Recall that the dashed arrow used in this connector represents the `LossySync` primitive channel of Section 9.3. Although we do not use this connector to compose our microwave oven interface presented in Section 9.8.9, it is generally useful, interesting, and so complementary to the shift-lossy `FIFO`$_1$ channel, that it deserves to be mentioned here.

The connector in Fig. 11a is composed out of a `LossySync` channel and a `FIFO`$_1$ channel. It, too, behaves as a lossy `FIFO`$_1$ channel, but contrary to the shift-lossy `FIFO`$_1$
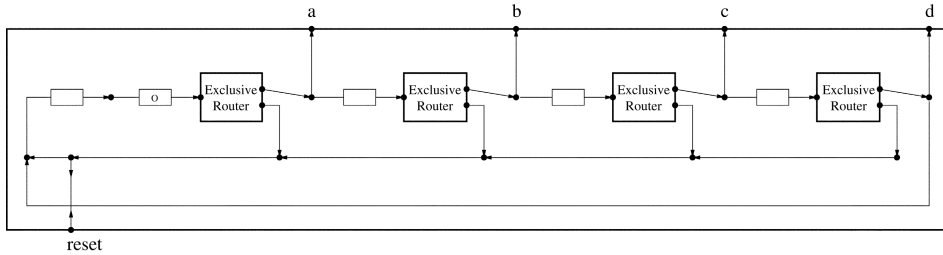
Fig. 12. A sequencer with reset.

channel of Section 9.8.3, its data-loss policy favors retaining older buffer values over newer arrivals. We call this connector/channel *overflow-lossy* $FIFO_1$ and use the graphical symbol in Fig. 11b to represent it. The dashed source-side half of the box representing the buffer of this channel symbol suggests that when its buffer is full, the newer arrivals simply "overflow" and are lost.

9.8.5. Sequencer with reset

Fig. 12 shows yet another generally useful connector that uses the exclusive router of Section 9.8.1. This connector behaves almost the same as our sequencer of Section 9.4.4. As long as no input value is available on its reset node, the exclusive router following the $FIFO_1$ channel that contains the token has no choice but to "decide" to forward the token through its upper output node down the chain, as if it were a simple Sync channel. This makes the transfer of the token conditional only on the availability of a ready consumer on the respective output node of the sequencer connector, exactly as in the case of the sequencer of Section 9.4.4.

When an input value is available on the reset node of this connector, the exclusive router following the token can allow it to "escape" the sequence chain through its lower output node back to the leftmost $FIFO_1$ channel, regardless of whether or not a ready consumer is available on its corresponding output node of the sequencer connector. This "resets" the sequencer to restart its sequence from the left. Observe that a race condition between reset and a ready sequenced output is non-deterministically resolved by the exclusive router.

The sequencer in Fig. 12 has 4 output nodes and thus repeats a sequence of length 4. Clearly, just as the case for the sequencer of Section 9.4.4, the constructor of this connector can be parameterized to instantiate a sequencer that repeats a sequence of any (finite) size.

9.8.6. Cycler

The connector in Fig. 13 is called *cycler*. It uses three instances of the constant replacer connector of Section 9.5, in this case, initialized with constant values High, Med and Low, respectively. It also uses an exclusive router, a ternary version of the sequencer with reset of Section 9.8.5, and a shift-lossy $FIFO_1$ channel (of Section 9.8.3). The exclusive router here has three output nodes. This ternary router is obtained by a simple composition of two (binary) exclusive routers of Fig. 9a, where one of the output nodes of one is connected with a Sync channel to the input node of the other.
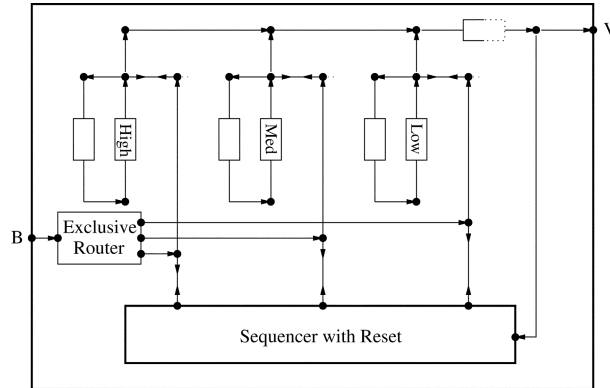
Fig. 13. A cycler connector.

The cycler connector behaves as follows. The first input value through its node B places the value High in its shift-lossy $FIFO_1$ channel, ready for output through the V node of the cycler connector. Successive input values through B "cycle" through the remaining values in the sequence, Med and Low, restarting the cycle again from High, and make each value available, in turn, for output through V, by overriding the previous contents of the shift-lossy $FIFO_1$ channel. Whenever a value is consumed through V, the sequencer resets the connector to restart the cycle from its leftmost, High, value.

Clearly, the constructor of this connector can be parameterized such that it takes a (finite) list of values to create a cycler instance of the proper size, initialized with exactly those values.

9.8.7. Valves

Fig. 14a shows a connector that behaves as a valve.[3] It uses two exclusive routers and three $FIFO_1$ channels, two of which are initialized to contain token values X and O. As long as there is no value available on the input node i of this valve connector, values can flow from its node c to node d; the "valve" is initially open. The first input value through i, say a token value Y, causes the upper SyncDrain channel in this connector to consume the X token, as Y moves to fill the (upper) empty $FIFO_1$ channel. As long as the two $FIFO_1$ channels on the left of this connector are full, no data can flow between c and d: the valve is closed. A second input value through the node i allows the lower SyncDrain channel to consume the Y token, while this second value itself fills the buffer of the $FIFO_1$ channel on the right. Now the connector configuration is back to its original state and the valve is open again. In this way, successive values through the input node i alternately close and open the valve, enabling and disabling the flow of data from c to d, without any data loss.

The valve in Fig. 14b is the same as the one in Fig. 14a, except that it is initially closed. Successive values through its input node i alternately open and close this valve.

---

[3] Different valve connectors are presented in [3]. This particular, more elegant connector circuit is due to Andrei Popescu of Nokia Research Center who is also a Ph.D. student at the Computer Science Department of the University of Helsinki.
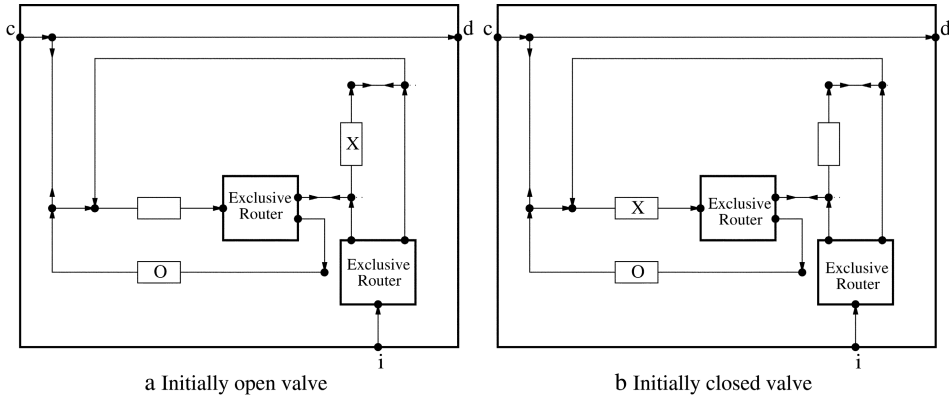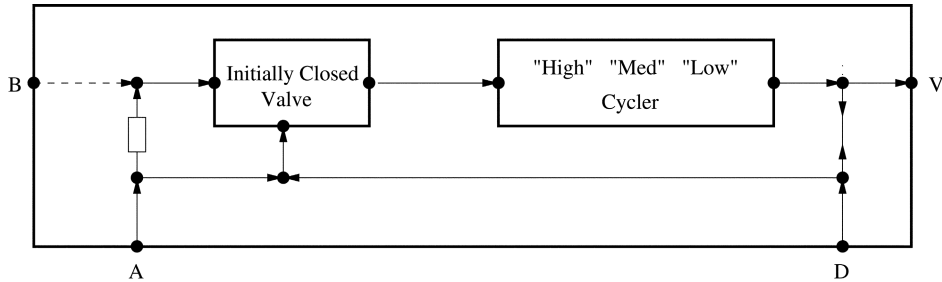
Fig. 14. Valve connectors.

Fig. 15. A value menu connector.

9.8.8. Value menu

The connector in Fig. 15 uses an initially closed valve and a cycler initialized with the parameter vector `High, Med, Low`. Initially, values on node B are lost, because the valve is closed, until an input value through A, say `a`, "activates" this connector. The token `a` first opens the valve, and then passes through the valve to kick the cycler, causing it to make `High` available as the output value of the connector. Arrival of another value on A before an input value on D is unacceptable for this connector; the environment must ensure that input to the nodes A and D alternate. Once thus activated, input tokens through B bump the cycler to make the sequence of values in its list available as the output of the connector. The actual output of the "current value" of the connector through the node V is possible only when a token, say `d`, arrives through the node D. The token `d` allows the output of the connector's current value and simultaneously closes the valve, disabling the connector. Observe that the output of its value also resets the cycler to start again with the first item in its list.

A race condition is possible when two tokens are simultaneously available on the nodes A and B. In this case, the mixed node at the sink of the `LossySync` channel non-deterministically resolves the race condition. However, this resolution is not symmetric:

if the A token is selected, then the B token is lost and the value menu connector will have `High` as its current value. On the other hand, if the B token is selected, then the A token is not lost and it will subsequently bump the cycler to set the current value of the value menu connector to `Med`. While this asymmetric behavior may be a "bug" or a "feature" of this connector when used in different environments, in the context of our intended application, it causes no problem (see Section 9.8.9). It is, of course, possible to eliminate this asymmetric aspect of the behavior of our connector, at the cost of a few more channels (essentially, an exclusive router, a `SyncDrain` and some `Sync` channels), but we prefer our simpler connector here because it is sufficient for our purpose.

9.8.9. Composing the microwave interface

Fig. 16 shows the composition of the interface of a simple microwave oven. Physically, this interface consists of four special-purpose buttons, plus a numeric input device. The four buttons are `Time`, `Defrost`, `Power`, and `Start`. The output of this interface is a pair of values indicating the radiation-time (`Time` node) and the power-setting (`Power` node) that are to be fed to the microwave engine. This interface allows an end-user to first press one of the two buttons `Time` or `Defrost` (if both are pressed, the interface selects one non-deterministically). Pressing the `Time` button indicates that both radiation-time and power-setting will be specified explicitly. Pressing the `Defrost` button, on the other hand, indicates that the microwave oven is to operate in its defrost mode. In this case, the end-user is expected to enter the weight of the item that is to be defrosted and the microwave oven is expected to calculate the proper radiation-time and power-setting automatically. When required, the same numeric input device is used to explicitly enter time as well as weight. Explicit power setting, when required, must be selected from a menu of the three alternative values `High`, `Med`, and `Low`, by successively pressing the power button to cycle through them. Once the proper selections are made and the values are entered, the end-user presses the `Start` button to confirm them and start the microwave engine.

Our interface in Fig. 16 uses four generic button components, a defrost calculator component, and a weight/time input component. These components are shown as solid-color boxes in Fig. 16. A button component is expected to interact with its corresponding physical button and produce a single token through its output node every time its button is pressed. The defrost calculator component has one input and two output nodes. Whenever a value is available on its input node, it consumes it and interprets it as a weight value. It then proceeds to compute the appropriate radiation-time and power-setting for defrosting this weight and makes them available through its respective output nodes. The weight/time input component has two input and a single output nodes. Availability of a token on one of its input nodes (the upper one in Fig. 16) activates this component to interact with the end-user and obtain a weight (e.g., in 100 g incremental units). A token through the other input node (the lower one in the figure) activates the component to interact with the end-user to obtain a radiation-time (e.g., in minutes and seconds). If tokens are available on both of its input nodes (a case that does not arise in our context) then the component non-deterministically consumes one and acts accordingly. The component eventually makes the value it obtains through its interaction with the end-user available through its output node.
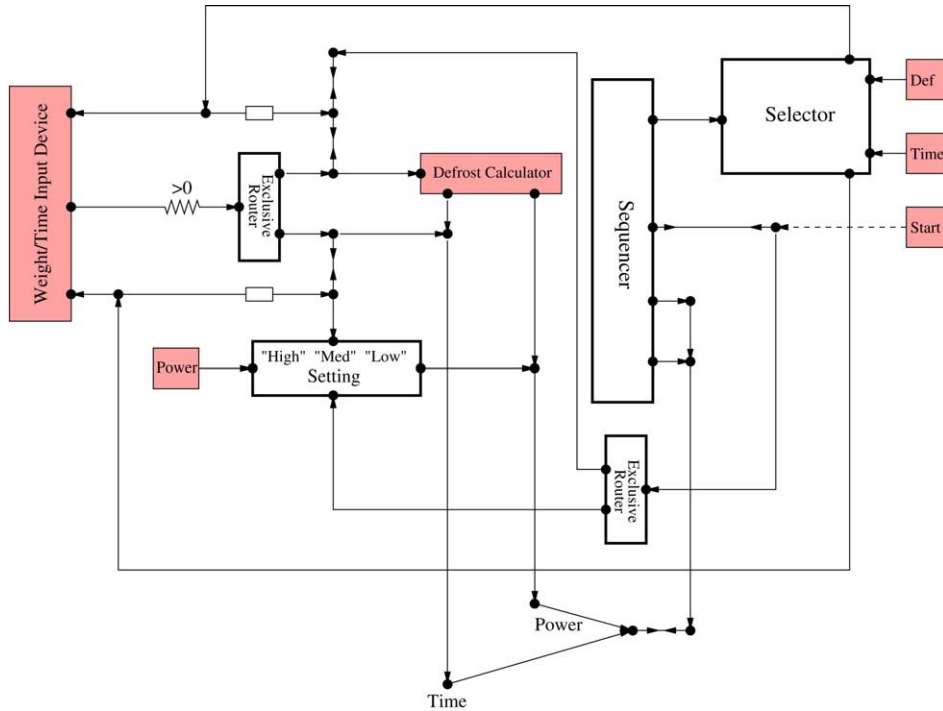
Fig. 16. The interface for a simple microwave oven.

The core of the interface in Fig. 16 is a four-step sequencer connector that produces tokens on its output nodes from top to bottom. In its first step, it enables the selector connector to let the user press one of the two buttons Defrost or Time. Its second step, allows the user to press the Start button. The last two steps of the sequencer allow the consumption of the two output values produced by this interface, in any order. To accomplish this, the Time and Power output nodes are connected to the last two nodes of the sequencer by a variant of the write-cue regulator connector of Section 9.4.1.

In the first step of the sequencer, pressing the Defrost button enables the weight/time input component to produce a weight, and places a token in the top $FIFO_1$ channel to allow the proper disposition of the eventual (weight) output of this component as the input for the defrost calculator component. Similarly, pressing the Time button enables the weight/time input component to produce a time, and places a token in the bottom $FIFO_1$ channel to allow the activation of the power value menu connector after the (time) output of the weight/time input component is available. Observe the use of the barrier synchronization construct of Section 9.4.2 in the coordination of the output of the weight/time input component.

Pressing the Start button has no effect unless the sequencer is in its second step and the lower exclusive router in the figure can dispose of its token one way or the other. If Defrost was pressed in the first step, this exclusive router can dispose of its token as

soon as the (weight) output of the weight/time input component is available. On the other hand, if Time was pressed in the first step, this exclusive router can dispose of its (Start) token only by deactivating the power value menu connector, which implies some value is available through its value output node. In the interval between the pressing of the two buttons Time and Start, the end-user has the option of pressing the Power button to cycle through the three settings of the power value menu. Observe that a default value (of High) is available for output by this value menu connector as soon as it is activated.

An aspect of the interface that is not reflected in our connector shown in Fig. 16 is its display. One can assume that there is a built-in display under the control of the weight/time input component, and connect a separate display to show the current setting of the power menu connector. The race condition mentioned in Section 9.8.8 is possible in this environment only if the end-user presses the Power button exactly when the (time) output of the weight/time input component is available. Assuming that the power setting display shows a value only while the menu connector is enabled, it is reasonable to expect that, normally, the user presses the power button only after this time-output is available (to enable the menu connector). When the Power button is pressed at the same time that the time-output is available, the non-deterministic resolution of the race condition in the menu connector either ignores the pressing of the Power button (as if the menu connector is not activated yet) or behaves as if the Power button was pressed immediately after the menu connector was activated (bumping the power setting to Med). Both of these alternatives are acceptable in this situation.

This example shows the ease with which a coordinator for such non-trivial concurrent behavior can be composed out of "ignorant" components and connectors in Reo. While the generality of the value menu connector of Section 9.8.8 may be somewhat questionable, there is no question that the connectors described in Section 9.8.1 through 9.8.7 are generic and useful in a wide variety of applications. The button components used here are also generic. The only application specific entities in this example are the defrost calculator and to a lesser extent, the weight/time input component. Although they are clearly application specific, these components are "ignorant" of the environment in which they cooperate with each other and other entities to enact the coordinated protocol of their collective behavior.

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.

Acknowledgements

References

[1] F. Arbab, The IWIM model for coordination of concurrent activities, in: P. Ciancarini, C. Hankin (Eds.), Coordination Languages and Models, Lecture Notes in Computer Science, vol. 1061, Springer-Verlag, 1996, pp. 34–56.

[2] F. Arbab, What do you mean, coordination?, Bulletin of the Dutch Association for Theoretical Computer Science NVTI (1998) 11–22. URL: http://www.cwi.nl/NVTI/Nieuwsbrief/nieuwsbrief.html.

[3] F. Arbab, A channel-based coordination model for component composition, Tech. Rep. SEN-R0203, Centrum voor Wiskunde en Informatica, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands, February 2002.

[4] F. Arbab, Reo: A channel-based coordination model for component composition, Mathematical Structures in Computer Science 14 (3) (2004) 1–38.

[5] F. Arbab, F. Mavaddat, Coordination through channel composition, in: F. Arbab, C. Talcott (Eds.), Coordination Languages and Models: Proc. Coordination 2002, Lecture Notes in Computer Science, vol. 2315, Springer-Verlag, 2002, pp. 21–38.

[6] F. Arbab, J. Rutten, A coinductive calculus of component connectors, in: D.P.M. Wirsing, R. Hennicker (Eds.), Recent Trends in Algebraic Development Techniques, Proceedings of 16th International Workshop on Algebraic Development Techniques, WADT 2002, Lecture Notes in Computer Science, vol. 2755, Springer-Verlag, 2003, pp. 35–56. URL: http://www.cwi.nl/ftp/CWIreports/SEN/SEN-R0216.pdf.

[7] F. Arbab, F. de Boer, M. Bonsangue, A logical interface description language for components, in: A. Porto, G.-C. Roman (Eds.), Coordination Languages and Models: Proc. Coordination 2000, Lecture Notes in Computer Science, vol. 1906, Springer-Verlag, 2000, pp. 249–266.

[8] F. Arbab, F. de Boer, M. Bonsangue, A coordination language for mobile components, Proc. ACM SAC'00, 2000, pp. 166–173.

[9] F. Arbab, F. de Boer, M. Bonsangue, J.G. Scholten, MoCha: A framework for coordination using mobile channels, Tech. Rep. SEN-R0128, Centrum voor Wiskunde en Informatica, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands, December 2001.

[10] F. Arbab, C. Baier, J.J.M.M. Rutten, M. Sirjani, Modeling component connectors in Reo by Constraint Automata, Electronic Notes in Theoretical Computer Science (ENTCS), vol. 97, Elsevier, 2004, pp. 25–46. URLs: http://www.elsevier.com/locate/entcs, http://www.cwi.nl/ftp/CWIreports/SEN/SEN-R0304.pdf.

[11] L. Barbosa, Components as coalgebras, Ph.D. Thesis, Universidade do Minho, Braga, Portugal, 2001.

[12] H. Barringer, R. Kuiper, A. Pnueli, A really abstract current model and its temporal logic, in: Proceedings of Thirteenth Annual ACM Symposium on Principles of Programming Languages, ACM, 1986, pp. 173–183.

[13] L. Bergmans, M. Aksit, Composing crosscutting concerns using composition filters, Communications of the ACM 17 (10) (2001) 51–57.

[14] M. Bonsangue, F. Arbab, J. de Bakker, J. Rutten, A. Scutellá, G. Zavattaro, A transition system semantics for the control-driven coordination language Manifold, Theoretical Computer Science 240 (2000) 3–47.

[15] M. Broy, A logical basis for component-based system engineering, Tech. Rep., Technische Universität München, November 2000.

[16] M. Broy, G. Stefanescu, The algebra of stream processing functions, Theoretical Computer Science 258.

[17] M. Broy, K. Stolen, Specification and development of interactive systems, Monographs in Computer Science, vol. 62, Springer, 2001.

[18] J. Buck, S. Ha, E. Lee, D. Messerschmitt, Ptolemy: a framework for simulating and prototyping heterogeneous systems, Simulation Software Development (3), International Journal of Computer Simulation (special issue), January 1990.

[19] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G.T. Leavens, K.R.M. Leino, E. Poll, An overview of JML tools and applications, in: T. Arts, W. Fokkink (Eds.), Eighth International Workshop on Formal Methods for Industrial Critical Systems, FMICS 03, Electronic Notes in Theoretical Computer Science (ENTCS), vol. 80, Elsevier, 2003, pp. 73–89. URL: http://www.elsevier.com/locate/entcs.

[20] CORBA, URL: http://www.omg.org.

[21] J. de Bakker, J. Kok, Towards a uniform topological treatment of streams and functions on streams, in: W. Brauer (Ed.), Proceedings of the 12th International Colloquium on Automata, Languages and Programming, Lecture Notes in Computer Science, vol. 194, Springer-Verlag, Nafplion, 1985, pp. 140–148.

[22] F. de Boer, M. Bonsangue, A compositional model for confluent dynamic data-flow networks, in: M. Nielsen, B. Rovan (Eds.), Proc. International Symposium of the Mathematical Foundations of Computer Science, MFCS, Lecture Notes in Computer Science, vol. 1893, Springer-Verlag, 2000, pp. 212–221.

[23] F. de Boer, M. Bonsangue, S. Graf, W.-P. de Roever (Eds.), Formal Methods for Components and Objects, Lecture Notes in Computer Science, vol. 2852, Springer, 2003.

[24] Enterprise JavaBeans, URL: http://java.sun.com/products/ejb.

[25] D. Gelernter, N. Carriero, Coordination languages and their significance, Communication of the ACM 35 (2) (1992) 97–107.

[26] J. Gore, Object Structures: Building Object-Oriented Software Components, Addison Wesley, 1996.

[27] R. Grimes, Professional DCOM Programming, Wrox Press, 1997.

[28] H. Gumm, T. Schröder, Covarieties and complete covarieties, in: [36], 1998.

[29] R. Helm, I. Holland, D. Gangopadhyay, Contracts: Specifying behavioural compositions in object-oriented systems, in: Proceedings OOPSLA/ECOOP '90, vol. 25, 1990, pp. 169–180.

[30] R. Hennicker, M. Wirsing, A formal method for the systematic reuse of specification components, in: Methods of Programming, LNCS, vol. 544, Springer-Verlag, 1991, pp. 49–75.

[31] F. Huber, A. Rausch, B. Rumpe, Modeling dynamic component interfaces, in: M. Singh, B. Meyer, J. Gil, R. Mitchell (Eds.), Proc. Technology of Object-Oriented Languages and Systems, TOOLS '98, IEEE Computer Society, 1998, pp. 58–70.

[32] B. Jacobs, Coalgebraic specifications and models of deterministic hybrid systems, in: M. Wirsing, M. Nivat (Eds.), Algebraic Methods and Software Technology, Lecture Notes in Computer Science, 1101, Springer-Verlag, 1996, pp. 520–535.

[33] B. Jacobs, Behaviour-refinement of object-oriented specifications with coinductive correctness proofs, Report CSI-R9618, Computing Science Institute, University of Nijmegen. Also in the Proceedings of TAPSOFT '97, 1996.

[34] B. Jacobs, Inheritance and cofree constructions, in: P. Cointe (Ed.), European Conference on Object-Oriented Programming, Lecture Notes in Computer Science, 1098, Springer-Verlag, 1996, pp. 210–231.

[35] B. Jacobs, J. Rutten, A tutorial on (co)algebras and (co)induction, Bulletin of EATCS 62 (1997) 222–259. URL: http://www.cs.kun.nl/~bart/PAPERS/JR.ps.Z.

[36] B. Jacobs, L. Moss, H. Reichel, J. Rutten (Eds.), Proceedings of the First International Workshop on Coalgebraic Methods in Computer Science, CMCS '98, Electronic Notes in Theoretical Computer Science (ENTCS), vol. 11, Elsevier, 1998. URL: www.elsevier.com/locate/entcs.

[37] Jini, URL: http://www.sun.com/jini.

[38] J. Kok, Semantic models for parallel computation in data flow, logic- and object-oriented programming, Ph.D. Thesis, Vrije Universiteit, Amsterdam, May 1989.

[39] G.T. Leavens, K.R.M. Leino, E. Poll, C. Ruby, B. Jacobs, JML: notations and tools supporting detailed design in Java, in: OOPSLA 2000 Companion, ACM, Minneapolis, MN, 2000, pp. 105–106. URL: ftp://ftp.cs.iastate.edu/pub/techreports/TR00-15/TR.ps.gz.

[40] E. Lee, D. Messerschmitt, An overview of the ptolemy project, Tech. Rep., Dept. of Electrical Engineering and Computer Sciences, University of California at Berkeley, 1993.

[41] E. Lee, T. Parks, Dataflow process networks, Proceedings of the IEEE 83 (5) (1995) 773–799. URL: http://ptolemy.eecs.berkeley.edu/publications/papers/95/processNets.

[42] E. Lee, A. Sangiovanni-Vincentelli, A framework for comparing models of computation, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 17 (12) (1998) 1217–1229.

[43] K.R.M. Leino, G. Nelson, An extended static checker for modula-3, in: K. Koskimies (Ed.), Compiler Construction CC'98, Springer LNCS, vol. 1383, Lisbon, 1998, pp. 302–305.

[44] K.R.M. Leino, G. Nelson, J.B. Saxe, ESC/java user's manual, Tech. Rep. SRC-TN-2000-002, Hewlett Packard Laboratories, October 15, 2000.
URL: http://www.hpl.hp.com/techreports/Compaq-DEC/SRC-TN-2000-002.html.

[45] S. Li et al., Professional Jini, Mass Market Paperback, 2000.

[46] B. Liskov, J. Wing, A behavioral notion of subtyping, ACM Transactions on Programming Languages and Systems 16 (6) (1994) 1811–1841.
URL: http://www.acm.org/pubs/articles/journals/toplas/1994-16-6/p1811-liskov/p1811-liskov.pdf.

[47] V. Matena, B. Stearns, Applying Enterprise JavaBeans: Component-Based Development for the J2EE Platform, Java Series, Enterprise Edition, Addison-Wesley, 2000.

[48] N. Mehta, M. Sirjani, F. Arbab, Effective modeling of software architectural assemblies using Constraint Automata, Tech. Rep. SEN-R0309, Centrum voor Wiskunde en Informatica, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands, October 2003. URL: http://www.cwi.nl/ftp/CWIreports/SEN/SEN-R0309.pdf.

[49] B. Meyer, Eiffel: The Language, Prentice Hall, 1992.

[50] B. Meyer, Reusable Software: The Base Object-Oriented Component Libraries, Prentice Hall, 1994.

[51] R. Milner, Elements of interaction, Communications of the ACM 36 (1) (1993) 78–89.

[52] R. Milner, J. Parrow, D. Walker, A calculus of mobile processes, parts I and II, Information and Computation 100 (1) (1992) 1–77.

[53] L. Moss, Coalgebraic logic, Annals of Pure and Applied Logic 96 (1–3) (1999) 277–317.

[54] L. Moss, N. Danner, On the foundations of corecursion, Logic Journal of the IGPL 5 (2) (1997) 231–257.
URL: http://www.math.indiana.edu/home/moss/papers/corecursion-final.ps.gz.

[55] O. Nierstrasz, F. Achermann, A calculus for modeling software components, in [23], pp. 339–360.

[56] S. Oaks, H. Wong, Jini in a Nutshell, O'Reilly & Associates, 2000.

[57] P. Panangaden, F. van Breugel (Eds.), Mathematical Techniques for Analyzing Concurrent and Probabilistic Systems, CRM Monograph Series, American Mathematical Society, ISSN: 1065-8599, 2004.

[58] G. Papadopoulos, F. Arbab, Coordination models and languages, in: M. Zelkowitz (Ed.), Advances in Computers—The Engineering of Large Systems, vol. 46, Academic Press, 1998, pp. 329–400.

[59] D. Pattinson, M. Wirsing, Making components move: A separation of concerns approach, in [23], pp. 487–507.

[60] H. Reichel, An approach to object semantics based on terminal coalgebras, Mathematical Structures in Computer Science 5 (1995) 129–152.

[61] J. Rutten, Universal coalgebra: A theory of systems, Tech. Rep. CS-R9652, Centrum voor Wiskunde en Informatica, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands, 1996.
URL: http://www.cwi.nl/ftp/CWIreports/AP/CS-R9652.ps.Z.

[62] J. Rutten, Automata and coinduction (an exercise in coalgebra), Report SEN-R9803, CWI. Available at URL: http://www.cwi.nl; Also in the Proceedings of CONCUR '98, LNCS 1466, 1998, pp. 194–218.

[63] J. Rutten, Automata, power series, and coinduction: taking input derivatives seriously (extended abstract), Report SEN-R9901, CWI. Available at URL: http://www.cwi.nl. Also in the Proceedings of ICALP '99, LNCS 1644, 1999, pp. 645–654 (1999).

[64] J. Rutten, Coalgebra, concurrency, and control, Report SEN-R9921, CWI. Available at URL: http://www.cwi.nl. Extended abstract in: R. Boel, G. Stremersch (Eds.), Discrete Event Systems, Kluwer, 2000 (1999).

[65] J. Rutten, Universal coalgebra: a theory of systems, Theoretical Computer Science 249 (1) (2000) 3–80.

[66] J. Rutten, Elements of stream calculus (an extensive exercise in coinduction), in: S. Brookes, M. Mislove (Eds.), Proc. of 17th Conf. on Mathematical Foundations of Programming Semantics, Aarhus, Denmark, 23–26 May 2001, Electronic Notes in Theoretical Computer Science, vol. 45, Elsevier, Amsterdam, 2001.

[67] J. Rutten, An application of coinductive stream calculus to signal flow graphs, Tech. Rep. SEN-E0305, Centrum voor Wiskunde en Informatica, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands, October 2003. URL: http://www.cwi.nl/ftp/CWIreports/SEN/SEN-E0305.pdf.

[68] J. Rutten, Component connectors, in [57], pp. 73–87 (Chapter 5).

[69] D. Sangiorgi, Asynchronous process calculi: the first-order and higher-order paradigms (tutorial), Theoretical Computer Science 253.
URL: http://www-sop.inria.fr/meije/personnel/Davide.Sangiorgi/mypapers.html.

[70] J. Siegel, CORBA: Fundamentals and Programming, 1st edition, John Wiley & Sons Inc., New York, 1996.

[71] A. Snyder, Encapsulation and inheritance in object-oriented programming languages, OOPSLA '86, 1986, pp. 38–45.

[72] C. Szyperski, Component Software—Beyond Object-Oriented Programming, Addison-Wesley, 1998.

[73] M. Wirsing, R. Hennicker, R. Breu, Reusable specification components, Tech. Rep. MIP-8817, Passau University, 1988.