A Behavioral Model for Composition of Software Components

Farhad Arbab

Center for Mathematics and Computer Science (CWI)
Kruislaan 413, 1098 SJ Amsterdam, The Netherlands
Leiden Institute for Advanced Computer Science (LIACS), Leiden University
Niels Bohrweg 1, 2333 CA Leiden, The Netherlands
School of Computer Science, University of Waterloo
200 University Ave. West, Waterloo, Ontario ON N2L 3G1, Canada
farhad@cwi.nl

ABSTRACT. Engineering of component based software systems requires a formal model for its foundation. Virtually all contemporary component models use some form of object oriented method invocation for their component composition semantics. In contrast, Abstract Behavior Types (ABT) offer a simpler model of components and their composition. Reo is an exogenous coordination language that provides a middleware for dynamically reconfigurable component composition on distributed, mobile platforms. The ABT model serves as a unifying formalism to express the behavior of Reo connectors, as well as that of components. By examples, we show the expressive power of Reo, and the applicability of the ABT model.

RÉSUMÉ. L’ingénierie des systèmes logiciels à base de composants nécessite un modèle formel comme fondation. Virtuellement tous les modèles actuels de composants utilisent une forme particulière d’appel de méthode orienté-objet comme base de leur sémantique de composition. À l’opposé, les Abstract Behavior Types (ABT) offrent un modèle plus simple aux composants et à leur composition. Reo est un langage de coordination exogène qui fournit un intergiciel permettant la composition dynamiquement reconfigurable de composants sur les plates-formes distribuées et mobiles. Le modèle ABT sert de formalisme unificateur pour exprimer le comportement des connecteurs Reo ainsi que celui des composants. Au travers d’exemples, nous montrons le pouvoir d’expressivité de Reo et l’applicabilité du modèle ABT.

KEYWORDS: Components, Composition, Reo, Abstract Behavior Types, Exogenous Coordination.

MOTS-CLEFS: composants, composition, Reo, types comportementaux abstraits, coordination exogène.
1. Introduction

The object oriented programming paradigm offers powerful abstraction and encapsulation mechanisms whose utilization has dramatically reduced the complexity of software development. Without this reduction in complexity, the abundance of the sophisticated applications and systems of today would not have been practically viable. Although they vary in sometimes significant ways, all object oriented programming paradigms share the notion of Abstract Data Types (ADTs) as their common foundation.

We have introduced the notion of Abstract Behavior Type (ABT) as a higher-level alternative to ADT to model components and their composition (Arbab, 2005). An ABT defines an abstract behavior as a constraining specification of the behavior of an entity by relating the contents and the relative timing of its observable input/output exchanges with its environment, 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 notion of ABT parallels the notion of ADT: an ADT abstracts away the data structures and the algorithmic instructions that manipulate them to offer a set of operations on an abstract data type. An ABT abstracts away the data types and the operations that manipulate them to offer an observable input/output behavior. Constraint automata (Arbab et al., 2004b) and a coalgebraic model of relations on timed-data-streams (Arbab et al., 2003; Rutten, 2004) are two concrete formalizations of the notion of ABT. 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 have proposed that both of these are highly desirable, if not essential, properties for models of components and their composition.

This paper is intended to raise the interest of the object oriented programming community for non-object-oriented models of components and their composition. Method invocation is the only means in the object oriented paradigm by which simpler software can be composed to yield software with more complex behavior. The ability to use components as building blocks for late composition by third parties requires a composition operator on the domain of components. But the semantics of method invocation is incongruent with a general algebraic composition operator for this purpose. Focusing on composition as the primary concern, instead, a model of components and connectors emerges that does not use method invocation, has mathematically precise semantics, and offers a more flexible and more expressive means of composition than is possible in the object oriented paradigm.

The bulk of this paper is a blend of selections from (Arbab, 2004) and (Arbab, 2005), with additional examples, perspective, and other details. We briefly describe composition, behavior, and coordination in Section 2. We define our notion of components and their composition in Section 3. In Section 4, we motivate these definitions in the context of a simple example. Section 5 contains an overview of the coalgebraic model of ABT. While the ABT model provides a simple formal foundation for definition and composition of components, it does not provide primitives
to directly express any form of non-trivial coordination; 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 (Arbab, 2004; Arbab et al., 2003). A summary of Reo is presented in Section 6, with examples that show its flexibility and expressive power. Reo uniquely focuses on compositional construction of connectors that enable and coordinate the interactions among constituents in a concurrent system, without their knowledge. Reo shows how complex behavior in a system can emerge as a composition of primitive interactions. In Section 7, we return to our example of Section 4 and show the construction of a Reo connector circuit for the coordinator glue code of this system. Finally, Section 8 contains the concluding remarks of this paper.

2. Composition, Behavior, and Coordination

We call a software construction compositional only if the properties of the resulting system can be defined as a composition of the properties of its constituent parts. For it to be useful, this definition must be augmented with appropriate definitions of “its constituent parts” and “the properties” that we are interested in. Both of these notions are manifestations of abstraction. Instead of considering individual primitive instructions as the constituents of a complex software system, we must identify parts of the system such that each part consists of a (large) collection of such primitives whose precise number and composition we wish to abstract away as internal details of that part. What properties are abstracted as internal details of a part, versus those that are exposed as the properties of the part, plays a crucial role in defining the effectiveness of an abstraction and the flexibility of compositions using that part. The more properties we hide, the more effective an abstraction we have, which allows more freedom of choice in selecting the precise collection or sequence of instructions that comprise an implementation of a part. On the other hand, the less properties we expose, the less of an opportunity we leave for individual parts to affect and be affected by the exposed properties of other parts, which restricts the possibility of influencing the role that a given part plays in different compositions.

To identify the exposed properties of a part that can and cannot be influenced through its composition with other parts, we distinguish between its behavior versus its semantics. To show the usefulness of this distinction, consider a simple adder as a (software) part (for instance, consider this adder as a process, an agent, an object, a component, etc.). This adder takes two input values, $x$ and $y$, and produces a result, $z$, which is the sum of $x$ and $y$. For this adder to be useful, it must expose its property of how it relates the values $x$, $y$, and $z$, that is $z = x + y$. We call this the semantics of the adder because it reflects the meaning of what it does. In addition to this semantics, successful composition of this adder as a part in any larger system requires the knowledge of certain other properties of the adder that must also be exposed. For instance, we need clear answers to the following questions:
– Does the adder consume $x$ and $y$ in a specific order?
– Does it consume whichever of $x$ and $y$ that arrives first?
– Does it consume $x$ and $y$ only when both are available?
– Does it consume $x$ and $y$ atomically?
– Does it consume $x$ and $y$ in separate steps that can potentially be interleaved with other events?
– Does it compute and produce $z$ atomically together with:
  - the atomic consumption of both $x$ and $y$, or
  - the consumption of $x$ or $y$, whichever is consumed last?
– Does it produce $z$ in a separate step, with possible interleaving of other events, after it consumes $x$ and/or $y$?

The properties that yield answers to such questions define the (externally observable) behavior of the adder, above and beyond its mere semantics. It is clear that even in the simple case of our trivial adder, different alternative answers to the above questions are possible, which means we can have different adders, each with its own different (externally observable) behavior, all sharing (or implementing) the same semantics.

The distinction between behavior and semantics is important in composition of all systems. However, it becomes essential in concurrent systems, especially where independence, autonomy, anonymity, and reuse of parts comprise a primary concern. 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.

A message sent by one object to another is not merely a passive piece of structured data passed from one to the other. A message primarily invokes the operation encoded in a specific method of its recipient and this semantics has certain implications on the flow of control within and through the two objects. The precise semantic and behavioral details of method invocation (e.g., synchronous vs. asynchronous, active vs. passive objects, etc.) still significantly varies within the object oriented world, to the extent of incompatibility in different languages. Nevertheless, method invocation is the only way in the object oriented paradigm in which the behavior of objects can be combined to comprise more complex software. Generally, when an object $c$ sends a message $m(p)$ to another object $e$, this implies that $c$ invokes 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 $e$.

1. The anthropomorphic style we use in our discussion of objects does not imply that we assume any form of awareness or intentionality on the part of objects. Indeed, in most object oriented
– $e$ must (pretend to) perform the activated method $m$ on parameters $p$, and return its result to $e$ upon its completion (the “pretense” refers to when $e$ delegates the actual execution of $m$ to a third object); and

– $e$ typically suspends between its sending of $m$ and the receiving of its (perhaps null) result.

This implies a rather tight semantic coupling between the message sender and receiver objects, involving an asymmetric, unidirectional dependency. On the one hand, the methods provided by an object can be used by any other entity (that has access to it). On the other hand, an object internally decides what operation of what other objects it invokes. 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.

Because of the intricate assumptions involved and the differences among various object-oriented programming languages, it is generally not possible for an object in one language to directly invoke the methods of an object in another language. The intimate coupling inherent in method invocation becomes problematic when combining the behavior of software entities larger than single objects, even in the same language. Furthermore, a substantial body of useful software is written in non-object-oriented languages. On the other hand, regardless of the specific language(s) that various software entities or sub-systems are written in, there is a universal way in which they (potentially can) communicate with one another, as well as with other non-software entities in their environment: exchanges of neutral, pure data through simple input/output operations. These observations behoove us to adopt a definition for the term *component* that is less restrictive and more flexible than the presently popular notions of this term, which predominantly reflect the bias of an object-oriented heritage.

Two factors differentiate component composition from other forms of software composition, such as module interconnections, method invocations, and procedure calls: emphasis on interactions among the constituent components in a concurrent system, and (the need for more) flexibility to influence the behavior of components at the time of their composition. Coordination models and languages address precisely the first of these concerns: management of interactions among the constituents of a system into a coherently coordinated cooperation. However, the different mechanisms that various coordination models offer to manage interaction do not all equally support the increased level of flexibility required in component composition.

Coordination models and languages address such key issues in Component Based Software Engineering as specification, interaction, and dynamic composition of component models, objects are not intentional entities, and as such cannot “know,” “pretend,” etc. Thus, “the object must know the semantics of the method” is merely a short-hand for “the programmer writing the code of the object must know the semantics of the method.” The point is that once this “knowledge” is coded into the object, it accordingly makes the object dependent on its environment.
ponents. (See (Papadopoulos et al., 1998) for a comprehensive survey of coordination models and languages.) Specifically, *exogenous* coordination models and languages provide a very promising basis for the development of effective glue-code languages. Exogenous coordination means coordination from outside and refers to the ability, in a model or language, to coordinate the behavior of a black-box entity, without its knowledge, from outside of the entity itself. This is an essential property for a component composition model to have because it allows building systems with very different emergent behavior out of the exact same components, simply by composing them differently. In such a model, different compositions of the same components create different contexts for the components, each exogenously imposing a different coordination protocol on those components, yielding a different emergent system behavior. We present a concrete example of this in Section 6.8.

3. Components and their Composition

We define a *component* as a template, *e.g.*, any executable code, whose concrete incarnations qualify as *component instances*. A *component instance* is a unique, identifiable execution-time collection that (1) includes at least one active entity; and (2) allows untargeted input/output of passive data as the only means of communication between the entities inside the component instance with any entity not in the same component instance. A component instance may include fragments or modules of sequential code, objects, or classes. An active entity is one that has its own independent thread of control. Examples of active entities include active objects, threads, processes, agents, or (other) component instances. No assumption is made about how the entities inside a component instance communicate with each other. Each component instance has a number of “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. The I/O operations, *e.g.*, read and write, used by (the entities inside) a component instance to communicate with its outside world, are performed on its ports. Without loss of generality, we assume ports are unidirectional, *i.e.*, the information flows through a port in only one direction: 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.²

By this definition, a Unix process, for example, 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

---
² Of course, a component may retract any one of its suspended operations, *e.g.*, due to a time-out.
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.

Clearly, components and component instances are different things: in our Unix process example, a component instance is a running process, whereas the binary file that this process is an instance of, is a component. However, for brevity, we often (mis)use the term “component” instead of “component instance” when the context makes the intention clear.

Restricting the content of inter-component communication to only passive data disallows transfer of control or sharing of pointers to their internal address spaces. Inter-component method invocations and (remote) procedure calls are thus forbidden. It also makes all inter-component communication uniformly undirected and anonymous. Unlike the case of a message sent to a specific target or one that invokes a method, when a component writes a (passive) message through one of its (output) ports, it neither identifies or restricts the consumer of this message (untargeted) nor knows the identity of its consumer (anonymous). Similarly, a component’s reading a (passive) message from one of its (input) ports also constitutes untargeted, anonymous communication. This simpler and more abstract mechanism for inter-component communication makes the coupling of components looser and more flexible than the coupling of ADTs or the dependencies of objects and classes. The fact that passive data can be interpreted as messages that invoke methods or operations means that the object oriented message passing and the ADT’s notion of behavior as operation sequences can be modeled as well, if and when it is necessary. Therefore we do not lose expressiveness for the flexibility and simplicity of our model.

A number of interesting questions immediately arise, for instance: What can one gain by using such a simpler model? How can one characterize the behavior of such component instances? Can such characterizations be formalized? How can one compose such component instances? How flexible and expressive can such a composition paradigm be? Can one use formal models to compositionally reason about the properties of the resulting systems?

Intercepting and manipulating messages before they perform the methods that their sender objects intend to invoke is at the core of the contemporary approaches to Aspect Oriented Programming, as exemplified, e.g., by the so-called Composition Filters (Bergmans et al., 2001). This clearly show the advantage of a paradigm based on a more abstract notion of messages as passive data (that can be freely manipulated and changed, before they are interpreted as a triggers for actions) over the active messages of object oriented programming whose immediate consequences are strictly to invoke the designated methods of their target objects.

Unix pipes constitute an example of a mechanism for composition of our component instances. Useful as they are, their flexibility is limited and they are not expressive enough to allow but the simplest forms of (pipeline) composition. Classical dataflow models and Kahn networks (Kahn, 1974) offer approaches to characterize the
behavior of our component instances. Broy’s dataflow-like characterization of components and incorporation of time tags in data streams comprise a functional paradigm wherein some compositional construction and reasoning about systems of component instances is possible (Broy et al., 2001a; Broy et al., 2001b). In the context of component composition, these models have shortcomings in at least two significant areas. First, they do not allow mixing synchrony and asynchrony in behavioral definitions. Second, they support, at best, only very rudimentary forms of exogenous coordination.

Our components are intrinsically active, do not issue, and do not accept method calls. However, any abstraction, \( X \), offered as a “component” by an alternative contemporary model (e.g., ArchJava (Aldrich et al., 2002b; Aldrich et al., 2002a), JavaBeans (EJB, n.d.), CORBA (COR, n.d.), COM+ (COM, n.d.), etc.) can always be wrapped in a thin layer of adapter code to yield a component in our model. This adapter layer (whose code can even be mechanically generated) creates an active entity, if necessary, and acts as an intermediary that converts the passive input/output messages exchanged between the component and its environment, to the method calls expected and issued by its encapsulated \( X \).

4. System Composition Example

Suppose we have three components, \( C, D, \) and \( T \), as in Figure 1.a. They are all black-box components: we know nothing about what they are made of or how they work internally. They may be made out of hardware, software, or some combination of the two. We can make no assumptions about the language or model used to construct these components. Specifically, they neither provide an interface of methods to call, nor make any method calls to interact with their environment.

The only thing we know about \( C \) is what we can externally observe of its behavior. It has a single port of interaction with its environment, through which it periodically outputs some string of characters. Of course, for the output to take place, (an entity in) the environment of \( C \) must be prepared to accept its output. Assuming an ideally cooperative environment (i.e., always ready to take it whenever \( C \) attempts to output its string), \( C \) produces a string approximately every 15 seconds, with the tolerance margin of \( \epsilon \). The actual content of the strings produced by \( C \) is the current time; so \( C \) is a clock.

\[ \text{(a) } C \rightarrow D \rightarrow T \]
\[ \text{(b) } C \rightarrow D \rightarrow T \]
\[ \text{(c) } C \rightarrow D \rightarrow T \]
\[ \text{(d) } C \rightarrow D \]

**Figure 1. Three components and their various compositions**
The only thing we know about $D$ is that it has a single input port, through which it consumes strings and displays them on its accompanying monitor for approximately 30 seconds. The “processing time” of $D$ is negligible for our purposes.

We observe that $T$ behaves very much the same as $C$, except that its tolerance margin is $\delta$ and the content of its output strings convey the current temperature.

We can construct a few systems out of these components, the simplest ones involving a direct connection, e.g., between $C$ and $D$. Because we cannot alter any of these components, we must make the connection from outside. The simplest connector we can use to compose $C$ and $D$ is what we call a synchronous channel, as in Figure 1.b. A synchronous channel is a medium of communication with two ends. Through one of its ends, it accepts input, and through the other, it dispenses it. We call it “synchronous” because it synchronizes the pair of input and output operations at its opposite ends: the two operations are suspended as necessary to ensure that they succeed together atomically.

If we connect $C$ to $D$ using a synchronous channel whose transfer and synchronization time is negligibly small (compared to the period of $C$), we obtain a composed system that displays the current time, updated approximately every 30 seconds. Similarly, we can construct another system out of $T$ and $D$ connected by a synchronous channel, as in Figure 1.c, to display the current temperature, updated approximately every 30 seconds.

In order to build a system, similar to what one finds on the top of some bank buildings, that alternately displays the current time and temperature, we have all the functional elements that we need in $C$, $D$, and $T$. What we need is a connector to compose them together as in Figure 1.d. This connector must have a more complex behavior than that of a synchronous channel used in the previous compositions: not only it must facilitate the data exchanges among these three components, but it also needs to enforce the coordination protocol that implements the desired alternating behavior. Because the internals of the components cannot be changed, such a connector would have to impose its coordination protocol “from the outside” of the components, which illustrates what we mean by exogenous coordination.

Obviously, such a connector, as well as other even more sophisticated ones, can be developed as programs in any modern programming language; their Turing completeness ensures that. However, it is interesting to ponder if there is a better, higher-level alternative to programming such connectors from scratch. Synchronization and coordination protocols are notoriously complex concurrent programs, and adding provisions to enable them to cope with mobility in distributed environments makes conventional programming models and languages grossly inadequate for their development. There is enough commonality of purpose (facilitating data exchange and exogenous coordination) among such connectors to warrant considering a special connector specification model and a special language for their development. To the extent that they merely connect and coordinate and lack application-specific functionality, each such
connector can be generically designed and reused to compose widely different sets of components into entirely different systems.

What would a special purpose connector specification model be like? Can connectors be reused not just to compose components into (sub)systems, but also to compose more complex connectors? What composition operators are necessary and sufficient to allow connector composition? Is there a set of primitive connectors out of which “all interesting or useful” connectors can be constructed by those connector composition operators? How can one characterize interesting and useful in this context?

Before we can address any of the above questions, we need concrete models to specify the behavior of components and that of the connectors. We briefly describe two such models in the next two sections. We return to these questions in Section 6, where we describe Reo and show how it can serve as a language for compositional construction of reusable coordinating component connectors.

5. A Coalgebraic Formalization of ABT

Coalgebras are simple mathematical structures capturing the behavior of systems (or processes or components) consisting of both observable and unobservable internal dynamics. Although many coalgebraic structures have a long history in both mathematics and computer science (e.g., streams and infinite trees), they were not generally identified as such until the early 1980’s when coalgebras were introduced as formal duals of algebras, to deal with infinite data structures. It has only recently been recognized that coalgebras form the underlying structure of various kinds of dynamical systems, automata, transition systems, infinite data types, object-oriented systems, formal power series, and even various classes of differential equations. See (Jacobs et al., 1997) for a general introduction and (Rutten, 2000) for an extensive overview.

Streams constitute the fundamental coalgebraic concept in our formalization of the ABT model, with coinduction as its primary reasoning principle. An ABT can be defined as a (maximal) relation among a set of timed-data-streams. This particular formalization 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.

The notion of timed-data-streams as well as most of the technical content in this section come from the work of J. Rutten on coalgebras (Rutten, 2000; Jacobs et al., 1997), stream calculus (Rutten, 2001; Rutten, 2005), and a coalgebraic semantics for Reo (Arbab et al., 2003; Rutten, 2004). Analogous to the way in which algebraic methods constitute suitable models for the syntactic structure of systems, the coalgebraic approach is a promising mathematical foundation for modeling the dynamic behavior of (concurrent) systems.
Defining observable behavior in terms of input/output implants a dataflow essence within ABTs akin to such dataflow-like networks and calculi as (de Bakker et al., 1985), (Kok, 1989), and especially (Broy et al., 2001b). 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.

5.1. 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 \( A^\omega \). Streams in \( D S = D^\omega \) over a set of (uninterpreted) data items \( D \) are called data streams and are typically denoted as \( \alpha, \beta, \gamma, \) 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 \( \alpha \). We use the infix “dot” as the stream constructor: \( z.\alpha \) denotes a stream whose first element is \( z \) and whose second, third, etc. elements are, respectively, the first and its successive elements of the stream \( \alpha \).

Following the conventions of stream calculus (Rutten, 2001; Rutten, 2005), the well-known operations of head and tail on streams are called initial value and derivative: the initial value of a stream \( \alpha \) (i.e., its head) is \( \alpha(0) \), and its (first) derivative (i.e., its tail) is denoted as \( \alpha' \). The \( k \)th derivative of \( \alpha \) is denoted as \( \alpha^{(k)} \) and is the stream that results from taking the first derivative of \( \alpha \) 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 (Barringer et al., 1986), 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., (Arbab et al., 2003)). 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” \( a(i) \).\(^3\) Two timed data streams \( \langle \alpha, a \rangle \) and \( \langle \beta, b \rangle \) are equal if their respective elements are equal, i.e. \( \langle \alpha, a \rangle = \langle \beta, b \rangle \Leftrightarrow \alpha = \beta \wedge a = b \).

5.2. Abstract Behavior Types

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 5.4.

Generally, we use the prefix notation \( R(I_1, I_2, ..., I_m; O_1, O_2, ..., O_n) \) and the separator “;” to designate the ABT defined by the \( (m + n) \)-ary relation \( R \) over the \( m \geq 0 \) sets of input timed data streams \( I_i, 0 < i \leq m \) 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 \( i^{th} \) input and the \( j^{th} \) output portals of the ABT.

Formally, a component, as defined in Section 3, 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.

---

\(^3\) The infinity of streams naturally models the infinite behavior of perpetual systems. Finite behavior can be modeled in at least three different ways. First, we can allow finite streams as well. Second, it can be modeled as a special case of infinite behavior, e.g., where after a certain time moment, only the special symbol \( \perp \) appears as values in all time streams. Although viable, we ignore both of these schemes because they do not add conceptual novelty, yet dealing with the special cases that they involve requires a somewhat more complex formalism. The third way to model finite behavior is to ensure that after a certain point in time, the system has no observable behavior. This is possible with or without finite streams. See the discussion at the end of Section 5.4.
The coalgebraic definition of ABTs offers a mathematical model of behavior as a relation, i.e., a set of tuples. We intentionally do not offer any specific language for defining such relations to avoid imposing any restrictions. Many languages can be conceived to define ABTs, each imposing its own structure and restrictions on how and what kinds of ABTs can be expressed. Reo is one such language. Constraint automata offer another. Constraint automata can be considered generalizations of probabilistic automata, where synchronization and data constraints, instead of probabilities, label state transitions and influence their firing (Arbab et al., 2004b). In contrast to the coalgebraic model of ABTs, their incarnations as constraint automata are directly useful for model-checking. Other languages, for instance formal logics, can also be used to define ABTs (see, e.g., (Arbab et al., 2004a)).

5.3. ABT Examples

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

5.3.1. 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 \( \langle \alpha, a \rangle = \langle \beta, b \rangle \equiv \alpha = \beta \land a = b \), 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

\[ \langle \alpha, a \rangle \text{ FIFO } \langle \beta, b \rangle \equiv \alpha = \beta \land a < b. \]

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 ABT, defined as

\[ \langle \alpha, a \rangle \text{ FIFO } _1 \langle \beta, b \rangle \equiv \alpha = \beta \land a < b < a'. \]

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

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

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

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

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

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

5.3.2. Merge and Replicate

We now define two other ABTs that, as we see in Section 6, form a foundation for an interesting and expressive calculus: merger and replicator. The merger ABT defines the behavior of a component with two input and one output ports. Whenever I/O is possible on its output and only one of its input ports, the merger synchronously allows both I/O operations to succeed and transfers the input value as its output. If I/O is possible on all of its three ports, the merger non-deterministically picks one of its input ports and behaves as above, effectively delaying its other input. The merger ABT is defined as:

\[
\begin{align*}
MRg(\langle \alpha, a \rangle, \langle \beta, b \rangle; \langle \gamma, c \rangle) \equiv \\
\{ \alpha(0) = \gamma(0) \land a(0) = c(0) \land MRg(\langle \alpha', a' \rangle, \langle \beta, b \rangle; \langle \gamma', c' \rangle) & \text{ if } a(0) < b(0) \\
\exists t : a(0) < t < \min(a(1), b(1)) \land \exists s r \in \{a(0), t\} \land r \neq s \land a(0) = b(0) & \\
MRg(\langle \alpha, r.a' \rangle, \langle \beta, s.b' \rangle; \langle \gamma, c \rangle) & \\
\beta(0) = \gamma(0) \land b(0) = c(0) \land MRg(\langle \alpha, a \rangle, \langle \beta', b' \rangle; \langle \gamma', c' \rangle) & \text{ if } a(0) > b(0)
\}
\end{align*}
\]

Intuitively, the \(MRg\) ABT produces an output that is a merge of its two input streams. If \(a(0)\) arrives before \(\beta(0)\), \(i.e.\ a(0) < b(0)\), then the ABT produces \(\gamma(0) = a(0)\) as its output at \(c(0) = a(0)\) and proceeds with the tails of the streams in its first input timed data stream. If \(a(0)\) arrives after \(\beta(0)\), \(i.e.\ a(0) > b(0)\), then the ABT produces \(\gamma(0) = b(0)\) 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 \(a(0)\) and \(\beta(0)\) arrive “at the
same time” (i.e., \(a(0) = b(0)\)), then in this formulation \(Mrg\) picks an arbitrary number \(t\) in the open time interval \((a(0), \min(a(1), b(1)))\) and uses it to nondeterministically break the tie. The assumption of dense time guarantees the existence of an appropriate \(t\). Recall that the construct \(r, a'\) is a stream whose derivative (tail) is \(a'\) and whose initial value (head) is \(r\). Thus, for \(a(0) = b(0)\) \(Mrg\) nondeterministically changes the head of one of the two time streams, \(a\) or \(b\), thereby “delaying” the arrival of its corresponding data item to break the tie. The finite delay introduced by \(Mrg\) in this case is justified because although it breaks simultaneity, its value is constrained to preserve atomicity. Observe that \(Mrg(\langle \alpha, a \rangle, \langle \beta, b \rangle; \langle \gamma, c \rangle) = Mrg(\langle \beta, b \rangle, \langle \alpha, a \rangle; \langle \gamma, c \rangle)\).

The replicator ABT defines the behavior of a component with one input and two output ports. If I/O is possible on all of its three ports, the replicator synchronously allows all I/O operations to succeed and copies the input value to both of its output ports. The replicator ABT is defined as:

\[
Rpl(\langle \alpha, a \rangle; \langle \beta, b \rangle, \langle \gamma, c \rangle) \equiv \beta = \alpha \land \gamma = \alpha \land b = a \land c = a
\]

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 \(Rpl(\langle \alpha, a \rangle; \langle \beta, b \rangle, \langle \gamma, c \rangle) = Rpl(\langle \alpha, a \rangle; \langle \gamma, c \rangle, \langle \beta, b \rangle)\).

### 5.3.3. 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

\[
\begin{align*}
Sum(\langle \alpha, a \rangle, \langle \beta, b \rangle; \langle \gamma, c \rangle) & \equiv \\
\gamma(0) & = \alpha(0) + \beta(0) \\
\exists t : \max(a(0), b(0)) < t < \min(a(1), b(1)) \land c(0) = t & \land \\
Sum(\langle \alpha', a' \rangle, \langle \beta', b' \rangle; \langle \gamma', c' \rangle).
\end{align*}
\]

\(Sum\) 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 (\(c(0) = t \land t > \max(a(0), b(0))\)). However, it is obligated to produce each of its output data items before it reads in its next input data item (\(t < \min(a(1), b(1))\)).

### 5.3.4. 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 6 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 $\alpha_t$ and $\alpha_f$ 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 $\text{Chop}$ ABT defined as

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

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

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

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

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

Again, because we are not interested in the actual values of the take/free tokens that this component produces, the $\text{Phil}$ 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.

5.4. 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, \(\alpha, \alpha\), necessarily synchronize their I/O operations on their respective portals for the mutual exchange of the data items in its data stream \(\alpha\), according to the schedule in its twin time stream \(\alpha\). 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. Figure 2.a shows the composition of the two ABTs \(P(x; y)\) and \(Q(u; v)\) where the output portal \(y\) of \(P\) (the bullet on its right edge) is fused with the input portal \(u\) of \(Q\) (the bullet on its left edge), yielding the ABT \(R(x; v)\). Figure 2.b shows the composition of the two ABTs \(P(x, y; z, u, v)\) and \(Q(r, s, t; w)\) where the three output portals \(z, u,\) and \(v\) of \(P\) (the bullets on its right edge) are fused with the three input portals \(r, s,\) and \(t\) of \(Q\) (the bullets on its left edge), yielding the ABT \(R(x, y, w)\). Note that it is perfectly sensible to compose two ABTs, simultaneously fusing some of the output portals of one with input portals of the other, and some of the output portals of the latter with the input portals of the former. For instance, \(P(x, y, z; u, v)\) can be composed with \(Q(w, r; s, t)\), fusing \(v\) with \(w\), and \(s\) with \(y\), yielding \(R(x, z, r; u, t)\).

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 \(B_1(I_1, I_2, \ldots I_{p_i}; O_1, O_2, \ldots O_{p_o})\) and \(B_2(P_1, P_2, \ldots P_{q_i}; O_1, O_2, \ldots O_{q_o})\) 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 $I_1, I_2, \ldots I_{p_1}, I_2, I_2, \ldots I_{2q}$ 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 $O_1, O_1, \ldots O_{p_n}, O_2, O_2, \ldots O_{q_n}$ 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, $B = B_1((\alpha, a), (\beta, b)^1; (\gamma, c)) \circ B_2((\delta, d); (\mu, m)^1)$ 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 $B \equiv \{((\alpha, a), (\delta, d); (\gamma, c)) \mid ((\alpha, a), (\beta, b); (\gamma, c)) \in B_1 \land ((\delta, d); (\mu, m)) \in B_2 \land (\beta, b) = (\gamma, c)\}$. Another example is the ABT $B = B_1((\alpha, a), (\beta, b)^1; (\gamma, c)^2) \circ B_2((\delta, d)^2; (\mu, m)^1; (\nu, n))$, 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 $B \equiv \{((\alpha, a); (\nu, n)) \mid ((\alpha, a), (\beta, b); (\gamma, c)) \in B_1 \land ((\delta, d); (\mu, m), (\nu, n)) \in B_2 \land (\beta, b) = (\gamma, c) \land (\delta, d)\}$. 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 $B_1(...; (\alpha, a)) \circ B_2(...; (\beta, b); \ldots)$ instead of $B_1(...; (\alpha, a)^1) \circ B_2(...; (\beta, b)^1; \ldots)$. This abbreviation is particularly convenient together with the infix notation for binary abstract behavior types. For instance, $B = (\alpha, a)B_1(\beta, b) \circ (\gamma, c)B_2(\delta, d)$ 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 $(\alpha, a)B(\delta, d) \equiv \{((\alpha, a); (\delta, d)) \mid ((\alpha, a); (\beta, b); (\gamma, c)) \in B_1 \land (\gamma, c) \land (\delta, d)\} \in B_2 \land (\beta, b) = (\gamma, c)\}$. For example, consider the binary ABTs defining the basic channels presented in Section 5.3. 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 $\text{Sync}$ ABTs produces a $\text{Sync}$ ABT. Indeed, composition of a $\text{Sync}$ ABT with any other ABT (on its left or right) yields the same ABT. More interestingly, the composition of two $\text{FIFO}$ ABTs produces a $\text{FIFO}$ ABT. Composing two $\text{FIFO}$ ABTs produces a $\text{FIFO}$ ABT. The formal proof of this latter equivalence relies on our notion of dense time (as opposed to discrete time) and is given in (Arbab et al., 2003), together with the formal treatment of many other interesting examples.

Composition of two ABTs requires nothing more than the complementarity of the (input/output) direction tags of their designated portal pairs. In this plain version of the model, all sorts of “inadvertent” nonsensical compositions are possible. A more
elaborate type system can, of course, be devised and imposed on top of this plain ABT model to disallow compositions that are not sensible according to some more refined model of semantics. Work in this direction is already under way, but its discussion is beyond the scope of this paper.

Composition of two ABTs may yield the empty relation, which simply means the result has “no externally observable behavior.” Although “no externally observable behavior” can be interpreted as deadlock, there is nothing inherently wrong with or undesirable about it, because it can also be interpreted as normal termination. Thus, a composition that yields an empty ABT can be a perfectly legitimate way to model finite behavior in an otherwise perpetual systems. An example of such “desired deadlock” situations arises in the inhibitor example in (Arbab, 2004).

An ABT is a mathematical model of the behavior of an active entity. Process algebras, are also models of active entities. However, process algebras such as \(\pi\)-calculus (Milner, 1993; Sangiorgi et al., 2001), CCS (Milner, 1980), CSP (Hoare, 1985), etc., do not directly offer a viable model for our purposes. The emphasis in the ABT model is, not on components, but rather, on composition and coordination of their interactions. True to their moniker, process algebras emphasize processes as the explicit entities constructed by the composition operators they provide, out of their atomic processes that represent primitive actions. Interaction and communication protocols ensue only as ancillary consequences of the unfolding of the collective behavior of the processes involved in a concurrent system and have no direct explicit constructs to represent them; coordination concerns and protocols fair even worse. Thus, strictly speaking, process algebras do not directly cater to the needs of software practitioners for having architecturally relevant constructs and primitives (i.e., components and connectors for composing large-scale distributed systems) that also encapsulate significant coarse-grain behavior, such as generic interaction and coordination protocols. It is, of course, possible to model every such architecturally relevant construct or primitive as a composed process in a suitable process algebra. However, the fine-grained compositionality of process algebras does not directly yield the appropriate composition operators required for the disciplined combination of these derived coarse-grain primitives and constructs. See, e.g., (Guillen-Scholten et al., 2005) for an example of the non-trivial mapping of the high-level concepts of Reo to \(\pi\)-calculus.

6. Reo

The ABT model provides a simple formal foundation for definition and composition of components. The 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 does not immediately confer the power of exogenous coordination to the glue code. The ABT model does not provide primitives to directly express any form of non-trivial coordination; 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 (Arbab, 2004; Arbab et al., 2003). 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.

We propose Reo as a model and language for compositional specification and construction of reusable coordinating glue-code. Pieces of glue-code in Reo can interconnect and exogenously coordinate the behavior of components. They can also just as easily connect and exogenously coordinate the behavior of other pieces of Reo glue-code, thus enabling compositional construction of the glue-code itself. All composition in Reo consists of channel composition. 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 (Arbab, 2004; Arbab et al., 2003). 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 (Arbab, 2004). A Turing machine consists of a finite state automaton for its control, and an unbounded tape. Since an unbounded tape can be simulated by two unbounded FIFO channels, adding \( \pi \) to the above set of channel types makes channel composition in Reo Turing complete.

As far as Reo is concerned, a component instance simply offers a set of channel ends as its contact points or ports. From the point of view of a Reo circuit, it is irrelevant what specific components offer which specific sets of ports: whether every single port that a Reo circuit connects to belongs to a different component instance, or they all belong to the same component instance, or any combination in between, is immaterial; and it makes no difference whether all or only some of the ports of a specific component instance are connected to a given circuit. All that matters is the compatibility of the behavior at connection points. This nicely separates the semantics of connector circuits and component interfaces into two orthogonal issues. Whether or not a subset of ports offered by a component instance can be plugged into a subset of channel ends of a connector can be decided based on the compatibility of the behavior of the two sets. It also enhances adaptability: a set of components, \( A \), can be replaced by another set, \( B \), and as long as the behavior of the subset of the ports of \( A \) and \( B \) that plug into a Reo circuit are compatible, neither the circuit, nor other components connected to it can tell the difference.

All channels in Reo can be mobile. This concept is different than the notion of mobility in \( \pi \)-calculus, which amounts to transmission and sharing of names. 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 (Arbab et al., 2001). The move operation in Reo changes the physical location of channel ends. 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. See (Guillen-Scholten et al., 2005) for the relationship between mobility in Reo and $\pi$-calculus.

Component instances, as well as channels, can be 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 (Arbab, 1996) and Manifold (Bonsangue et al., 2000), 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. Reo’s very liberal notion of channels also makes it more general than dataflow models, Kahn-networks, and Petri nets (Section 6.5), 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 (Broy et al., 2001a; Broy et al., 2001b) is perhaps closest to Reo. Here, components are functions that transform input data streams to output data streams, which represent their interconnecting FIFO channels. The only notion of “time” in this model arises out of sporadic “tick” marks intermixed with the data within the same streams. In contrast to Reo, streams/channels cannot be directly connected or composed together in this model: they can exist only between two components, which use the tick marks to synchronize their various input and output streams. This gives a functional (i.e., uni-directional transformation) flavor to the model. In contrast, Reo circuits are relational (i.e., bi-directional constraints) and have an explicit notion of dense time, represented as separate time streams in its ABT semantics, to express various combinations of synchrony and asynchrony, independent of data. Furthermore, 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 allow, among other things, compositions involving an expressive mix of synchrony and asynchrony, distinguish it from this as well as other models.

It turns out that the ABT model is quite adequate for defining the channel and connector composition operation which is the crux of exogenous coordination in Reo. In the rest of this section we show how connector construction in Reo can be seen as an application of the ABT model.
6.1. 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 5.3. 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 6.3; even more examples are presented in (Arbab, 2002; Arbab, 2004).

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.

Figure 3 shows examples of nodes. Figure 5 shows examples of connectors.

We use \( x \mapsto N \) to denote that the channel end \( x \) coincides on the node \( N \), and \( \hat{x} \) 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 \( S_{rc}(N) \) and \( S_{nk}(N) \), 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 \( \text{Src}(N) \neq \emptyset \land \text{Snk}(N) = \emptyset \). Analogously,
\( N \) is called a **sink node** if \( \text{Src}(N) = \emptyset \land \text{Snk}(N) \neq \emptyset \). A node \( N \) is called a **mixed node** if \( \text{Src}(N) \neq \emptyset \land \text{Snk}(N) \neq \emptyset \).

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 (Arbab et al., 2000).

The graph representing a connector is **not** directed. However, for each channel
end \( x_e \) of a channel \( e \), we use the directionality of \( x_e \) to assign a **local direction in the neighborhood of** \( x_e \) to the edge that represents \( e \). The local direction of the edge representing a channel \( e \) in the neighborhood of the node of its source \( x_e \) is presented as an arrow emanating from \( x_e \). Likewise, the local direction of the edge representing a channel \( e \) in the neighborhood of the node of its sink \( x_e \) is presented as an arrow pointing to \( x_e \). See Figure 5 for examples.

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

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

6.2. ABT Models of Reo Nodes and Connectors

Although there may be good reasons to consider channels, nodes, connectors, components, and systems as ontologically different, they all have behavior, and as such can uniformly be modeled as ABTs. This powerful abstraction allows us to derive the behavior of a system by composing the behavior of its subsystems, each of whose behavior is composed out of the behavior of each of its components and connectors, where the behavior of each connector is composed out of the behavior of each of its constituent nodes and channels.

The behavior of every Reo node can be defined as a composition of two primitive ABTs (Arbab, 2004): a non-deterministic merger, and a replicator, as defined in Section 5.3.2. 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 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.

Consider a sink node \(N\) with \([N] = \{x, y\}\), as in Figure 3.a. The read operations performed on this node induce an output timed data stream, \(\langle a_N, a_N \rangle\), for this sink node. We use \(\langle a_N, a_N \rangle\) and \(\langle a_N, a_N \rangle\) 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 \(\mathcal{M}r(a_N, a_N, a_N, a_N)\).

![Figure 3. Representation of nodes in Reo](image-url)
The semantics of a sink node $N$ where $[N] = \{x, y, z\}$, as in Figure 3.b, is defined as the 1-dyad composition

$$
Mrg(\langle \alpha_x, \alpha_z \rangle, \langle \alpha_y, \alpha_x \rangle; \langle \psi, p \rangle^1) \circ Mrg(\langle \alpha_x, \alpha_z \rangle; \langle \alpha_N, a_N \rangle)
$$

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., $Mrg3$ is associative with respect to its input portals. As such, the simple graphical notation of Reo (e.g., in Figures 3.a and b) is quite appropriate because it does not suggest any precedence for the $Mrg$ operations. Clearly this scheme can be used to define the semantics of sink nodes with more coincident channel ends in general as the ABT $Mrgk$ with $k > 0$ input and one output portals. For completeness, we define $Mrg1(\langle \alpha_x, \alpha_z \rangle; \langle \alpha_N, a_N \rangle) \equiv \langle \alpha_x, \alpha_z \rangle = \langle \alpha_N, a_N \rangle$ and consider $Mrg2$ to be a pseudonym for $Mrg$.

The write operations performed on a source node $N$ with $[N] = \{x, y\}$, as in Figure 3.c, induce an input timed data stream, $\langle \alpha_N, a_N \rangle$, for $N$. The semantics of $N$ in this case is defined by the ABT $Rpl(\langle \alpha_N, a_N \rangle; \langle \alpha_y, a_y \rangle)$. The semantics of a source node $N$ with $[N] = \{x, y, z\}$, as in Figure 3.d, is defined as the 1-dyad composition

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

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 $Rpl1(\langle \alpha_N, a_N \rangle; \langle \alpha_x, \alpha_z \rangle) \equiv \langle \alpha_x, \alpha_z \rangle = \langle \alpha_N, a_N \rangle$ and consider $Rpl2$ to be a pseudonym for $Rpl$.

A mixed node, as in Figure 3.e, is a composition of two “half-nodes,” a source and a sink. Because no component is allowed to perform an I/O operation on a mixed node, no input/output timed data stream can be defined for a mixed node. A mixed node is a closed entity that does not interact with any component; instead it internally pumps data items from its sink channel ends to its source channel ends. The semantics of a mixed node $N$ with $m > 0$ 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 $Rpln(\langle \xi, q \rangle; O_1, O_2, \ldots O_k)$. 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.

\[
\text{Node}_{\text{mix}}(I_1, I_2, \ldots, I_m; O_1, O_2, \ldots, O_n) \equiv M_{\text{g}}(I_1, I_2, \ldots, I_m; \langle \psi, p \rangle) \circ R_{\text{in}}(\langle \xi, q \rangle; O_1, O_2, \ldots, O_n).
\]

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

\[
\text{Node}_{\text{mix}} \equiv M_{\text{g}}(I_1, I_2, \ldots, I_m; \langle \alpha, a \rangle)
\]

\[
\text{Node}_{\text{mix}} \equiv R_{\text{in}}(\langle \alpha, a \rangle; O_1, O_2, \ldots, O_n)
\]

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.

6.3. 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: \( \text{Sync}, \text{FIFO}, \text{FIFO}, \text{FIFO}, \text{Filter}, \text{LossySync}, \) and \( \text{SyncDrain} \). 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 5.3. 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 \text{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 \text{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 \text{Filter}(P) ABT defined as

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

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 \text{ABT} proceeds with the rest of its timed data streams.

2) A \text{LossySync} channel is also like a \text{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 \text{LossySync} ABT defined as

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

3) A \text{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. \text{SyncDrain} is synchronous because a write operation on one of its ends remains pending until a write is performed on its other end as well; only then both write operations succeed together. The behavior of this channel is captured by the \text{SyncDrain} ABT defined as

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

### Figure 4.

A set of primitive channels and their graphical symbols

The graphical representations of a set of simple channels that are commonly used in the construction of Reo circuits, are shown in Figure 4. All of our 7 primitive channels, above, are included in this set. Synchronous spout and asynchronous drain
are included in Figure 4 for completeness, although we do not use them in the examples in this paper. A synchronous spout allows data items to be disposed out of its two ends only synchronously. The actual values it produces through its ends are non-deterministic. The ABT definition of a SyncSpout looks very much like that of a SyncDrain:

\[ \text{SyncSpout}((\alpha, a), (\beta, b)) \equiv a = b. \]

The behavior of an asynchronous drain channel is complementary to the behavior of a SyncDrain:

\[ \text{AsyncDrain}((\alpha, a), (\beta, b);) \equiv a \neq b. \]

Recall that (1) Reo does not define the channels in Figure 4 or any other specific channel, and (2) there is nothing sacrosanct about these channels or their names. This particular set of channels is interesting for our purposes because of its small size (although it is not minimal and some of its channels can be constructed by composing others in the set) and the simplicity of its individual channels. It is interesting to show the expressive power of Reo (in Section 6.4) using such a small set of simple channels. In practice, however, not everything needs to be constructed out of such primitive channels: more complex channels or pre-fabricated connector pieces can be made readily available and used in applications.

6.4. Coordinating Glue Code

To demonstrate the expressive power of connector composition, in this section we describe a number of examples in Reo. More examples appear elsewhere (Arbab, 2002; Arbab et al., 2003; Arbab, 2004). The compositional construction of connector circuits as depicted in their graphical representations, also serve as recipes for compositional verification of these connectors using constraint automata: the constraint automaton of a circuit can be obtained by composing the constraint automata of its sub-circuits, which in turn are derived by composing the constraint automata of their constituent channels (Arbab et al., 2004b). Verification of a circuit, then, consists of verification of the composition of previously verified, simpler sub-circuits.

6.4.1. Write-Cue Regulator

Consider the connector in Figure 5.a, composed out of the three channels ab, cd, and ef. Channels ab and cd are of type Sync and ef is of type SyncDrain. This connector shows one of the most basic forms of exogenous coordination: the number of data items that flow from \( \hat{a} \) to \( \hat{d} \) is the same as the number of write operations that succeeds on \( \hat{e} \). (Recall that \( \hat{a} \) 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 \( \hat{e} \) can count and regulate the flow of data between the two nodes \( \hat{a} \) and \( \hat{d} \) by the timing and the number of write oper-
ations it performs on \( \bar{z} \). The entity that regulates and/or counts the number of data items through \( \bar{z} \) need not know anything about the entities that write to \( \bar{a} \) and/or take from \( \bar{d} \), nor that its write actions actually regulate this flow. The two entities that communicate through \( \bar{a} \) and \( \bar{d} \) 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 \( \bar{z} \).

Figure 5. Examples of connectors in Reo

6.4.2. Barrier Synchronizers

We can build on our write-cue regulator to construct a barrier synchronization connector, as in Figure 5.b. The four channels \( ab, cd, gh \), and \( ij \) are all of type \( Sync \). The SyncDrain channel \( ef \) ensures that a data item passes from \( \bar{a} \) to \( \bar{d} \) only simultaneously with the passing of a data item from \( \bar{g} \) to \( \bar{j} \) (and vice versa). This simple barrier synchronization connector can be trivially extended to any number of pairs, as shown in Figure 5.c.

6.4.3. Ordering

The connector in Figure 5.d consists of three channels: \( ab, ac \), and \( bc \). The channels \( ab \) and \( ac \) are SyncDrain and Sync, respectively. The channel \( bc \) is of type FIFO. The behavior of this connector can be seen as imposing an order on the flow of the data items written to \( \bar{a} \) and \( \bar{b} \), through to \( \bar{c} \): the data items obtained by successive read operations on \( \bar{c} \) consist of the first data item written to \( \bar{a} \), followed by the first data item written to \( \bar{b} \), followed by the second data item written to \( \bar{a} \), followed by the second data item written to \( \bar{b} \), etc. See (Arbab, 2002; Arbab, 2004) for more detail and (Arbab et al., 2003) 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 \( \bar{c} \) consist of zero or more repetitions of the pairs of values written to \( \bar{a} \) and \( \bar{b} \), in that order.
6.4.4. Sequencer

Consider the connector in Figure 5.e. The enclosing box represents the fact that the details of this connector are abstracted away and it provides only the four nodes \( \tilde{a}, \tilde{b}, \tilde{c}, \) and \( \tilde{d} \) for other entities (connectors and/or component instances) to (in this case) read from. Inside this connector, we have four \( \text{Sync} \), a \( \text{FIFO}_1(o) \), and three \( \text{FIFO}_1 \) channels connected together. The \( \text{FIFO}_1(o) \) 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 \( \tilde{a}, \tilde{b}, \tilde{c}, \) and \( \tilde{d} \) 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) \( \text{Sync} \) and \( \text{FIFO}_1 \) channel pairs, as required.

Figure 5.f shows a simple example of the utility of our sequencer. The connector in this figure consists of a two-node sequencer, plus a pair of \( \text{Sync} \) channels and a \( \text{Sync\_Drain} \) channel connecting each of the nodes of the sequencer to the nodes \( \tilde{a} \) and \( \tilde{c} \), and \( \tilde{b} \) and \( \tilde{c} \), respectively. The connector in Figure 5.f is another connector for the coordination pattern \( c = (ab)* \), although there is a subtle difference between the behavior of this connector and the one in Figure 5.d. See (Arbab, 2002; Arbab, 2004) for more detail.

It takes little effort to see that the connector in Figure 5.g corresponds to the meta-regular expression \( c = (aab)* \). Figures 5.f 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.

6.5. Petri Nets

In some respects, Reo circuits resemble Petri nets. However, there are major differences between the two. Petri nets consist of places and transitions with interconnecting arcs, which we can see as a fixed set of building blocks, each with a fixed behavior, for construction of Petri nets. In contrast, Reo defines a fixed set of composition rules and allows an arbitrary set of channels as primitives with arbitrary behavior, on which its composition rules can be applied to construct connector circuits. This readily allows incorporation of arbitrary computational entities into a composed Reo system. More importantly, it allows the harmonious combinations of synchrony and asynchrony in the same model which is not possible in Petri nets.

The similarity of the Petri net construction rules with Reo composition rules allows a direct translation of Petri nets into Reo circuits. Although direct translations of higher-level Petri nets into Reo circuits are also possible, here we consider only Elementary Nets (EN). Figure 6.a shows an example EN. Figure 6.b shows the Reo
equivalent constructs (the right column) for Petri net building blocks (the left column). An empty place corresponds to a **FIFO** channel. A filled place containing a token correspond to a **FIFO**. An arc corresponds to a **Sync** channel. A transition with a single incoming arc and \( n > 0 \) outgoing arcs corresponds to a node with one incoming and \( n \) outgoing channels. A transition with \( m > 1 \) incoming and \( n > 0 \) outgoing arcs corresponds to a degenerate barrier synchronizer (Section 6.4.2) Reo sub-circuit with \( m - 1 \) **SyncDrain** channels, \( m \) input nodes, and a single output node, as shown in the bottom-right of Figure 6.b. All \( n \) output channels are connected to the single output node of this sub-circuit.

![Figure 6](image)

**Figure 6. Translation of Petri nets into Reo circuits**

Using Figure 6.b, it is straight-forward to directly translate a Petri net into a Reo circuit. For example, applying this translation to the Petri net in Figure 6.a yields the Reo circuit in Figure 6.c. In this sense, every Petri net can be trivially considered to be a Reo circuit. The inverse translation, however, is far from trivial.

In Reo, synchrony and exclusion constraints propagate through (the synchronous sub-sections of) circuits. This is generally not the case in Petri nets, because their transitions are local. What sets Petri nets apart from classical automata is their transition nodes, which enable them to directly synchronize otherwise unrelated events (it is no accident that a non-trivial Petri net transition node translates into a barrier synchronizer in Reo). A Petri net transition node enforces synchronous **and** of several arcs/events. However, Petri nets have no primitive for the dual synchronous **or** of several arcs, and there can be no arc between two places, nor between two transitions. The latter disallows nested **ands** of arcs. More significantly, the **or** of several arcs is possible only if they emanate from or end in the same place, which implies the commitment of moving a token from or into that place. This means that events/arcs can be directly **and**-synchronized to compose more complex synchronous transitions (i.e., one-step atomic transactions), but a synchronous **or** of events/arcs is not possible, i.e., two transitions cannot be connected together without an intervening place/commitment. This disallows a direct modeling of composite atomic transactions in Petri nets and prevents arbitrary combinations of synchrony and asynchrony. The ability to construct arbitrarily complex synchronous sub-circuits (representing one-step atomic transactions) with asynchronous behavior in between, is unique in Reo and simplifies expressions of complex behavior. For example in the context of e-commerce, (Zlatev et
al., 2004) and (Diakov et al., 2005) show the construction of non-trivial Reo circuits that implement negotiation protocols for competition and collaboration in electronic auctions. The Petri net models of these same protocols would be substantially more complex and elaborate, because they would have to “simulate” all atomic transactions involved.

6.6. Constant Replacer

Figure 7 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 FIFO1 (N1-N2), and a filled FIFO1(T) (N2-N1) that contains an initial value T. Of course, the constructor of this connector can be parameterized to initialize this FIFO1 channel with any supplied value, instead of T, every time it creates a new instance of this circuit.

Figure 7. Constant replacer

From the definition of the SyncDrain channel in Section 6.3, we have:

\[ \text{SyncDrain}(\langle a, a \rangle, \langle \chi, c \rangle) \equiv a = c. \]  \[1\]

By letting \( m = 1 \) and \( n = 3 \) in the ABT equation of mixed nodes in Section 6.2, we derive for N1:

\[
\begin{align*}
\text{Node}_{1 \times 3}(\langle e, e \rangle; \langle \chi, c \rangle, \langle \phi, f \rangle, \langle \delta, d \rangle) & \equiv Mrg1(\langle e, e \rangle; \langle \psi, p \rangle) \circ Rpl3(\xi, q; \langle \chi, c \rangle, \langle \phi, f \rangle, \langle \delta, d \rangle) \\
& \equiv \{ \langle e, e \rangle; \langle \chi, c \rangle, \langle \phi, f \rangle, \langle \delta, d \rangle \} \cap \{ \langle e, e \rangle; \langle \psi, p \rangle \} \cap Mrg1 \wedge \{ \langle e, e \rangle; \langle \chi, c \rangle, \langle \phi, f \rangle, \langle \delta, d \rangle \} \in Rpl3 \wedge \{ \langle e, e \rangle; \langle \psi, p \rangle \} = \{ \xi, q \}.
\end{align*}
\]  \[2\]

From the definition of \( Mrg1 \) in Section 6.2, we get \( \{ \langle e, e \rangle; \langle \psi, p \rangle \} \in Mrg1 \equiv \langle e, e \rangle = \langle \psi, p \rangle \) and from the definition of \( Rpl3 \) in the same section, we have \( \{ \xi, q \} = \langle \chi, c \rangle = \langle \phi, f \rangle = \langle \delta, d \rangle \). Substituting these back in equation 2 and simplifying the result yields:

\[
\begin{align*}
\text{Node}_{1 \times 3}(\langle e, e \rangle; \langle \chi, c \rangle, \langle \phi, f \rangle, \langle \delta, d \rangle) & \equiv \{ \langle e, e \rangle; \langle \chi, c \rangle, \langle \phi, f \rangle, \langle \delta, d \rangle \} \cap \{ \langle e, e \rangle = \langle \chi, c \rangle = \langle \phi, f \rangle = \langle \delta, d \rangle \},
\end{align*}
\]  \[3\]
Similarly, for the node $N_2$, we derive:

$$\text{Node}_{1×1}((\gamma, g); (\eta, h)) \equiv \{\langle \epsilon, c, \phi, f, \delta, d \rangle \mid \epsilon = \chi = \phi = \delta \wedge c = f = d\}. \tag{4}$$

From the definition of the $\text{FIFO}_1$ channel in Section 5.3.1, we have:

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

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

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

The definition of the $\text{Sync}$ channel in Section 5.3.1 gives:

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

From equations 1 and 4 we get $d = a$, which together with equation 6 yields:

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

From equation 8 we have $f = b$, which together with equations 4 and 1 yields:

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

From equation 7 we have $\epsilon(0) = T \wedge \eta = e'$ and from equations 4 and 8 we have $\epsilon = \phi = \beta$. This gives us:

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

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

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

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 equation 12 into $\beta(0) = T \wedge \beta(1) = \beta(0) \wedge \beta(2) = \beta(1) \wedge \beta(3) = \beta(2) \wedge \beta(4) = \beta(3) \wedge \ldots$, or:

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

Equation 13 clearly shows that there is no relationship between the stream of input values, $\alpha$, and the stream of output values, $\beta$, of this connector: whatever value comes
through the node A, its corresponding output value through the node B is the constant value $\tau$. On the other hand, equation 10 relates the input/output “timings” of this connector: passage of each pair of values through the nodes A and B is atomic.

Equation 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 $\tau$ must cycle through the node N2 once before the next 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 $\tau$ will delay value transfers through A and B, if necessary, such that equation 9 holds.

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 Figure 7. This observation underscores the usefulness and the significance of visual representation of Reo connectors.

### 6.7. 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. Although producing this series (as indeed, any other full application) involves computation, the purpose of our example here is to show the expressiveness of the coordination mechanisms in Reo for composing complex behavior out of simpler ones. This example demonstrates how the coordination protocol for a full application can be constructed to coordinate the ordering of events and the relative timing of activities in that application, without knowing anything about or being specific to its actual computation content. To assemble a component based application to deliver the Fibonacci series we actually need only one (instance of one computational) component plus a number of channels. The component we need is a realization of the $Sum$ ABT that we already saw in Section 5.3.

![Figure 8. Computing the Fibonacci series](image)
Figure 8 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 $\text{Sum}$, a $\text{FIFO}_1(1)$, a $\text{FIFO}_1(0)$, a $\text{FIFO}_1$, and five $\text{Sync}$ channels.

As long as the $\text{FIFO}_1(0)$ channel is full, nothing can happen: there is no way for the value in $\text{FIFO}_1(1)$ to move out. At some point in time, the value in $\text{FIFO}_1(0)$ moves into the $\text{FIFO}_1$ channel. Thereafter, the $\text{FIFO}_1(0)$ channel becomes empty and the two values in the $\text{FIFO}_1(1)$ and the $\text{FIFO}_1$ channels become available for $\text{Sum}$ to consume. The intake of the value in $\text{FIFO}_1(1)$ by $\text{Sum}$ inserts a copy of the same value into the $\text{FIFO}_1(0)$ channel. When $\text{Sum}$ 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 $\text{FIFO}_1(1)$ channel. At this point we are back to the initial state, but with different values in the buffers of the $\text{FIFO}_1(1)$ and the $\text{FIFO}_1(0)$ channels.

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

6.8. 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 $\text{Phil}$ and $\text{Chop}$ defined in Section 5.3.4, to (1) compose a dining philosophers application that exhibits the famous deadlock problem; and (2) compose another dining philosophers application that prevents the deadlock.

Figure 9.a shows 4 philosophers and 4 chopsticks around a virtual round table. Each philosopher has 4 output ports, corresponding to the $lt$, $lf$, $rt$, and $rf$ portals of the $\text{Phil}$ ABT in Section 5.3.4. 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 $\text{Chop}$ ABT in Section 5.3.4. In Figure 9.a, chopstick ports on the outer-edge of the table are their $t$ ports and the ones closer to the center of the table are their $f$ ports. The $t$ (take) port of each chopstick is connected to the take ports of its adjacent philosophers, and its $f$ port to their respective free ports. All channels are of type $\text{Sync}$. 
Consider what happens in the node at the three-way junction connected to the $t$ port of $\text{Chop}_1$. If $\text{Chop}_1$ is free and is ready to accept a token through its $t$ port, as it initially is, whichever one of the two philosophers $\text{Phil}_1$ and $\text{Phil}_4$ happens to write its take request token first will succeed to take $\text{Chop}_1$. Of course, it is possible for $\text{Phil}_1$ and $\text{Phil}_4$ to attempt to take $\text{Chop}_1$ at the same time. In this case, the semantics of this mixed node (by the definition of the ABT $M_{\text{rg}}$) guarantees that only one of them succeeds, nondeterministically; the write operation of the other remains pending until $\text{Chop}_1$ is free again. Because the definition of the ABT $\text{Phil}$ states that a philosopher frees a chopstick only after it has taken it, there is never any contention at the three-way junction connected to the $f$ port of a chopstick.

The composition of channels in this Reo application enables philosophers to repeatedly go through their “eat” and “think” cycles at their leisure, resolving their contentions for taking the same chopsticks nondeterministically. The possibility of starvation is ruled out because the nondeterminism in $M_{\text{rg}}$ is assumed to be fair. This simple glue code composed of nothing but common generic $\text{Sync}$ 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 $\text{Phil}$ 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.
6.8.1. 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. Figure 9.b shows a slightly different composition topology of the same set of sync channels comprising the glue code that connects the exact same instances of Phil and Chop as before. We have flipped one philosopher’s left and right connections to its adjacent chopsticks (in this particular case, those of Phil₂) 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 Phil₂, will actually reach for the one on its right-hand-side. Of course, Phil₂ 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 Phil₃, 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 Chop₂ as its first chopstick) to enable at least one philosopher to obtain its second chopstick as well and complete its eating cycle.

Comparing the composition topologies in Figures 9.a and b, we see that in Reo (1) different glue code connecting the same components produces different system behavior; and (2) coordination protocols are imposed by glue code on components that cooperate with one another through the glue code, without being aware of each other or their cooperation. The two fundamental notions that underpin this pair of highly desirable provisions are:

– The underlying notion of component (Section 3) in the ABT model prevents a component from distinguishing individual entities within its environment directly. Components can exchange only passive data with their environment through communication primitives that (1) do not allow them to discern specific targets as communication partners, and (2) do not entail any further obligation on behalf of the environment. The ABT model of components, thus, grants the environment great flexibility in making late, even dynamic, decisions about how components are composed. This makes ABT components highly susceptible to exogenous coordination, although the ABT model itself offers no non-trivial coordination primitives.

– Reo is a coordination model that takes full advantage of the composition flexibility 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. 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 Figures 9.a and b are made of the same number of constituent parts of the same types: the same number of component instances of the same kinds, and the same number of primitive connectors (Sync channels). The only difference between the two is in the topology of their inter-connections. This topological difference is the only cause of the difference between the “more than sum of the parts” in these two systems.

6.8.2. 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 (Figure 5.e) plus two SyncDrain channels to realize this behavior. But a much simpler construction is possible as well.

![Figure 10. Inside of a chopstick](image)

The connector hidden inside the enclosing box in Figure 10 is a simplified two-port sequencer which exactly implements the behavior of the ABT Chop. This connector consists of two channels: a \( \text{FIFO}_1 \) and a SyncDrain. Initially, the \( \text{FIFO}_1 \) 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 \( \text{FIFO}_1 \) channel becomes full and the next write operation on port \( t \) will suspend until this channel becomes empty again. When the \( \text{FIFO}_1 \) channel is full, a write to \( f \) succeeds, causing the SyncDrain channel to consume the contents of the \( \text{FIFO}_1 \) channel as well. This returns the connector to its original state allowing it to cyclically repeat the same behavior.

Alternatively, one can observe that a chopstick is essentially a two-port sequencer that allows write operations to succeed sequentially on its ports. Thus, we can use a two-port version of our sequencer in Figure 5.e to construct a chopstick, as in Figure 11. The essential addition to the sequencer in this connector consists of the two SyncDrain channels. The two Sync channels exist primarily for aesthetic reasons. The intuitive equivalence of the behavior of the two connectors in Figures 10 and 11 shows the necessity for formal models, e.g., based on constraint automata or timed-data-streams, to allow investigating the equivalence or subsuming relationships among the behavior of seemingly different connectors.
6.9. Synchronous Languages

Synchronous languages (Halbwachs, 1993) like Esterel (Berry, 1999; Berry et al., 1984), Lustre (Halbwachs et al., 1991), and Signal (Le Guernic et al., 1986), emerged in the 1980’s for modeling and programming of reactive systems, signal processing, and critical control software. Because they involve synchronous dataflow networks, they address issues that are also of concern in Reo: specification of non-trivial synchronous behavior. However, there are significant differences between Reo and all synchronous dataflow networks, including synchronous languages.

Esterel (Berry, 1999; Berry et al., 1984) is an imperative program generator language, essentially for defining the behavior of finite state automata. It is used to generate programs that constitute the reactive kernels of reactive systems, with actual interfaces and data handling specified in some other host language. Using Esterel, replication (of transitions, states, etc.) in the automata of complex systems are eliminated by the structural constructs of the language and computation. This makes it more convenient to describe the behavior of complex systems in Esterel, rather than directly as automata. Lustre (Halbwachs et al., 1991) is a declarative dataflow kernel language very close to Esterel. They originally shared intermediate language and compilation tools. Signal (Le Guernic et al., 1986) is another declarative synchronous language. Unlike Lustre, Signal is not a dataflow language, but deals with sequences of input and output signals and their relative order.

These languages have no specific explicit notion of time. Repetition of any event or signal abstractly indicates passage of time. In contrast, Reo has an explicit notion of dense time, represented separately from data, in its time streams. This allows direct expressions of temporal constraints (order, synchrony, and asynchrony) explicitly and independently of the data streams, which is not possible in synchronous languages. Synchronous languages have an implied notion of state, wherein actions and computation take place synchronously, taking “zero time” and transitions between states depend on their input/output data (but not time). Whereas synchronous and asynchronous behavior correspond to states and transitions in synchronous languages, both are specified in Reo uniformly as compositions of temporal and data constraints.
Not only Reo allows asynchrony (as well as synchrony) through composition of, e.g., various FIFO or asynchronous-drain channels, the synchronous merge behavior inherent in its nodes does not have a counterpart in synchronous languages either: it allows two-way propagation of synchrony constraints. In synchronous languages, the (synchronous/asynchronous) behavior of a state cannot be coordinated by external composition: synchronous languages do not support exogenous coordination. This can be explained using the “angelic/demonic” characterization of algebraic calculi of relations by Bergstra and Stefanescu (Bergstra et al., 1998). In the angelic view of composition of two binary relations $R$ and $S$, a pair $(x, z)$ is in the composite relation $R \circ S$ if there exists an intermediary $y$ such that $(x, y) \in R$ and $(y, z) \in S$. This models the usual sequential composition. Compositions with input/output branching points may involve the demonic view of relational composition, which they roughly characterize as: “If in a branching point an input (resp. output) disconnected path exists, then it destroys the other input (resp. output) connected paths of that point.” They obtain three types of demonic calculi by considering the demonic behavior at input, output, or both kinds of branching points: forward-demonic, backward-demonic, and two-way demonic. Forward demonic calculi are sufficient to model the connections in synchronous dataflow languages. In contrast, Reo can be characterized as a declarative dataflow language where independent constraints on data and its flow (i.e., time) are composed to yield the specification of the behavior of a complex system. Composition of these constraints manifests exogenous coordination in Reo, which seems to require a two-way demonic calculus for its model.

Originally, synchronous languages produced executable code for mono-processor platforms only. More recently some are being extended to produce code for distributed architectures as well. The execution model of the code generated by synchronous languages is generally different than the conceptual model of their program specification: they are not designed to allow dynamic changes to program specifications. In contrast, Reo connector circuits are inherently distributed, mobile, and dynamically reconfigurable.

7. Time/Temperature Display Coordinator

The “chopstick” of Section 6.8.2 constitutes the cornerstone of the coordination behavior we need for the component connector in Figure 1.d. We can use either one of the constructions in Figures 10 and 11, or any other with equivalent behavior, as an off-the-shelf component/connector to construct our time/temperature display coordinator, as in Figure 12.

The box labeled “Write-Sequencer” in Figure 12 represents a connector with the behavior of a chopstick. The $C$, $D$, and $T$ ports of the connector in Figure 12 represent the connections with the $C$, $D$, and $T$ components in Figure 1.d, respectively.

Assuming the “Write-Sequencer” is constructed as in Figure 10, we can see that our time/temperature display coordinator can be constructed out of 4 $\text{Sync}$ channels,
A Behavioral Model for Components

Figure 12. Coordinating connector for the Time/Temperature Display system

1 FIPI channel, 1 SyncDrain channel, and 3 mergers (inherent in the Reo nodes C, T, and D). Figure 12, thus, represents a two-dimensional recipe or program describing precisely how the behavior of each constituent primitive (channel or node) must be composed with that of its topologically connected neighbors, to yield the desired behavior of the connector circuit.

8. Conclusions

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. Heterogeneity of the semantics and the technical idiosyncrasies of various (versions of) object oriented models make object oriented composition of third-party black-box software inefficient and impractical, if not impossible. Furthermore, a substantial body of useful software is written in non-object-oriented languages. Nevertheless, this software, and many other not-purely-software systems, communicate with their environment through simple exchanges of passive data, and can sensibly be reused as building block components in the construction of more complex systems. 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 do not offer the looser coupling and the flexible exogenous coordination framework that are necessary to support the composition of the more general non-object-oriented components.

Abstract Behavior Types 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 the operations or the 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 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.

Our current and future work includes development of various tools for (semi-)automatic reasoning, analysis, simulation, and animation of connector circuits, within a visual programming environment for Reo. Constraint automata and tools for their construction, composition, and model checking are an integral part of our on-going work.

9. References


Arbab F., “The IWIM Model for Coordination of Concurrent Activities”, in P. Ciancarini, C. Hankin (eds), Coordination Languages and Models, vol. 1061 of Lecture Notes in Computer Science, Springer-Verlag, p. 34-56, April, 1996.


A Behavioral Model for Components


