Composition by Interaction

Rede uitgesproken door

Prof.dr.ir. Farhad Arbab

bij de aanvaardinig van het ambt van hoogleraar
op het gebid van de software compositie
aan de Universiteit van Leiden
op vrijdag 28 oktober 2005

Mijnheer de Rector Magnificus, zeer gewaardeerde toehoorders,

Ladies and gentlemen, my family, friends, and colleagues, you honor me by your presence.

1 Introduction

In the short time since the inception of modern computers, we have come a long way developing hardware and software artifacts that we call computer applications. Although its humble physical manifestation is far less impressive than that of our other spectacular modern artifacts such as bridges, skyscrapers, automobiles, airplanes, and spacecraft, modern software, such as what runs on your common-place laptop, is by far the most complex artifact created by human beings. Indeed, without modern software and computing, much of those other impressive artifacts would have been inconceivable. Software represents a pinnacle of engineering. That we are capable of constructing intricate artifacts with such complex dynamic behavior is amazing. The fact that they work – usually – borders on miracle! That we seem to have a good enough grip to work our way through understanding and explaining such complexity is astonishing.

Be that as it may, to the extent that our formalisms, models, languages, and tools reflect our knowledge of software, I believe our knowledge and engineering methods are as solid as Swiss cheese: there are still some foundational issues on which we do not have an adequate grip.

I believe compositional design, construction, and analysis of modern software – and more generally, of complex discrete systems – is one such fundamental issue. I include as complex discrete systems not only software, but also, for instance, business processes, human organizations, cybernetic systems, and systems biology, among many others.

The name of my chair at Leiden is “Software Composition” and the title of my talk is “Composition by Interaction.” I intend to share with you some of my work and thoughts on composition of complex systems out of simpler parts, using interaction as *the* binding construct.

We have been composing software since the inception of programming. Recognizing the need to go beyond the success of available tools is sometimes more difficult than accepting to abandon what does not work. Our software composition models have served us well-enough to bring us up to a new plateau of software complexity and composition requirements beyond their own effectiveness. In this sense, they have become the victims of their own success. Dynamic composition of behavior by orchestrating the interactions among independent distributed subsystems or services has quickly gained prominence. We now need new models for software composition, on par with those commonly used in more mature engineering disciplines, such as mechanical or electrical engineering. This deficiency is one of the reasons why "software engineering" is sometimes criticized as not-truly-engineering.

We have studied protocols for, and various aspects of, interaction in concurrency theory. However, up to now, we have not considered interaction as a first-class concept in any model that I am aware of. I find this disregard quite curious.

I believe the inadequacy of our contemporary composition techniques and our neglect to treat interaction as a first-class concept are intertwined. After all, interaction arises out of how a composition allows the parts of a system to play against one another.

2 Composition

From houses and bridges to cars, aircraft, and electronic devices, complex systems are routinely constructed by putting simpler pieces together. In spite of claims to the contrary, I believe this can hold for software construction as well.

In every form of construction, the properties of the resulting system, of course, somehow relate to the properties of its constituent parts. However, the nature of this relationship can be rather simple, or quite complex. There is a useful analogy from chemistry here: the distinction between mixtures and compounds. In either case you compose various substances together.

Mixtures combine physically in no definite proportions: they just mix and form no new substance. Each part of a mixture retains its own properties, and can be separated from the others by physical means. On the other hand, compounds require precise proportions of their constituents, and involve a chemical reaction that yields a new substance.

For instance, you can mix hydrogen and oxygen in any proportion to obtain a mixture gas, whose properties can rather simply be derived from those of its constituent gases. This holds until you ignite the mixture with a spark. This starts a chemical reaction which combines hydrogen and oxygen 2-to-1, and yields a new substance, water, with very different properties than that of the two gases. We can no longer consider the properties of the ingredients as gases to derive the properties of the resulting substance. The chemical reaction ignited by that spark blows away the usefulness of “gas” with its macro-level properties as a convenient abstraction. To relate the properties of the resulting compound to those of its ingredients, we must delve into the micro-level relationships that tie the more intricate details of the atomic structures of hydrogen and oxygen into the molecular structure of water.

Mixtures, not compounds, symbolize ideal compositional constructions in computer science. We call a software construction compositional only if the properties of the resulting system can be derived as a composition of the macro-level properties of its constituent parts.

At micro-level, all software construction is compositional: every complex piece of software eventually consists of some composition of a set of primitive instructions, and in principle, its properties can always be derived by applying its relevant rules of composition to the properties of those primitives. This is precisely how one derives the semantic properties of relatively simple programs from those of their primitive instructions.

However, micro-level compositional construction quickly becomes uninteresting and useless for complex concurrent systems, for the same reason that deriving interesting properties of a complex piece of mechanical machinery from those of its constituent atoms is intractable. With only a smidgen of exaggeration, one can say that attempting to derive the dynamic run-time behavior of such software in this way is as hopelessly misguided as trying to derive the properties of a running internal combustion engine from an atomic particle model of the engine, its fuel, air, and electricity.

The term “component” has been used to denote larger units of program code with well-defined properties to serve as building blocks in macro-level composition of software systems. Most contemporary component-based models and systems use composition mechanisms based on method invocation, remote procedure calls, or targeted message passing. We have argued that these mechanisms severely limit reusability and composition possibilities [3, 6]

3 Behavior

We have introduced a more general and more useful notion of “component” based on observable input/output behavior of black-box entities [3]. Such components are not necessarily pieces of software. This has led to the introduction of the important concept of Abstract Behavior Types, as a parallel to the well-known notion of Abstract Data Types, but at a higher level of abstraction.

An Abstract Data Type defines a data type, such as a stack, in terms of a set of operations (such as push, pop, top, and empty) without stating anything about the actual program code that implements those operations, or the data structures that they must manipulate. Analogously, an *Abstract Behavior Type* defines an abstract behavior without stating anything about the actual operations that an actor may perform to manifest that behavior, or the data types that they must manipulate [5]. This is a very powerful concept that leads the way to truly macro-level composition of components through their behavior.

By *behavior*, we mean something very specific and different than semantics. To show the usefulness of this distinction, consider a simple adder as a component. 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 questions such as:

1.Does the adder consume x and y in a specific order?

2.Does it consume whichever of x and y that arrives first?

3.Does it consume x and y only when both are available?

4.Does it consume x and y atomically?

5.Does it compute and produce z atomically together with its last input?

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.

We define a component as an Abstract Behavior Type, which specifies its observable behavior in terms of a relationship on the relative timing and order of its exchanges of passive data with its environment [5].

For instance, the Abstract Behavior Type defining an asynchronous unbounded queue merely relates its observable input and output through the following pair of constraints [11]. First, the sequence of data items that goes in is exactly the same as the sequence of data items that comes out: nothing is lost, the queue generates no data of its own, and the order of the data items is preserved. Second, every data item can come out only after it goes in.

This Abstract Behavior Type precisely defines the behavior of a queue without saying anything about any operations or data types that may be used to implement it. This sort of abstract behavior specification is all we need to define the behavior of our components and to compose them through interaction.

In its coalgebraic formalization, we use stream calculus [17, 18] to define an Abstract Behavior Type, such as our queue, as a mathematical relation on timed data streams [5]. We have also introduced *constraint automata* [9] that can be used to specify Abstract Behavior Types.

Whichever definition of "component" one uses, composing them into a system requires understanding and prescribing how they interact. However, thus far, our models for construction and analysis of composed systems have treated interaction only as a secondary, derived phenomenon. In a sense, all these models are "models of things that interact" rather than "models of interaction."

Process algebras, for instance, are models for constructing processes. They offer operators for composing atomic processes or primitive actions into more complex processes. Interaction ensues only as a consequence of the unfolding of the behavior of the processes involved in a concurrent system. For example, as a process *p* unfolds and performs its actions, one of its primitive actions, such as a send, collides with a compatible primitive action, such as a receive, performed by another process *q*. It is this collision of actions that forms an interaction. Whether this collision occurs by dumb luck, divine intervention, or intelligent design, is irrelevant. A split-second earlier or later, perhaps in a different run, the same two actions could have collided with other actions of other processes, yielding entirely different interactions. Actions and their composition have explicit constructs used to define a system. Interaction is ephemeral and implicit, and plays no structural role in the construction of a system. Other contemporary models for software composition fair no better than process algebras in this regard.

But, interaction is the most interesting and the most difficult aspect of composed systems. So, why not use it as *the* first-class concept in system composition?

4 Interaction

What does it mean for interaction to be a first-class concept?

A model wherein interaction is a first-class concept must provide two things: first, primitive interactions; and second, rules of composition for combining interactions into more complex ones.

An interaction is a constraint: an explicit relation that holds among a set of actors, and constrains each to coordinate their collective behavior. As constraints, interactions can be composed together in various ways to yield more complex constraints (or, interaction protocols), without any reference to the action sequences or the states of actors.

Consider some simple examples of system composition, using 3 black-box components: a clock, a thermometer, and a display. All we know about these components is what we can externally observe of their exchange of data with their environment. For the clock, we observe that it has an output port through which it periodically produces a string of characters that represents the current time. Similarly, the thermometer has an output port through which it periodically produces a string of characters that represents the current temperature. The display has an input port through which it periodically consumes a string of characters and displays it.

We can build various systems by composing these components. For instance, if we connect the output of the clock, with a connector like a pipe, to the input of the display, we construct a system that periodically displays the current time. Similarly, if we compose the thermometer and the display by connecting their ports, we obtain a system that periodically displays the current temperature.

Even in these simple compositions, interaction is not as trivial as it first seems. The black-box components know nothing about each other, and are not necessarily designed to work with one another. Therefore, the chance that they have compatible periods is indeed very slim. This means that the connector that composes each pair, needs to implement an interaction protocol that somehow compensates for the mismatch of their periods.

Now consider a third system – like what you see on top of some tall bank buildings – that alternately displays the current time and current temperature. We have all the functional ingredients that we need for this system in our three components. We should be able to compose them together in the right way to obtain this system.

Clearly, our connector cannot consist of a simple pipe, or two, in this case. It must do more to coordinate the three components, from outside of the components and “without their knowledge” so to speak, to yield the alternating behavior. This is an important concept, and we have coined the term *exogenous coordination* to refer to it [1, 16]. The word “exogenous” means “from outside” and exogenous coordination means coordination from outside. Exogenous coordination clearly separates computation from coordination, yielding concrete, tangible pure coordination code, side-by-side with pure computation code, each of which can be reused in different systems.

Of course, we can write the code for the connector in our example in any programming language, like C or Java. But, if we consider such connector programs for various coordination protocols in a number of different systems, certain patterns emerge. All exogenous coordination protocols deal with a relatively small set of basic concepts: synchrony, exclusion, asynchrony, buffering, atomicity, ordering, etc.
It is sensible, then, to study the elements of exogenous coordination, and to design a formal model and a programming language that offer appropriate composition operators to allow construction of exogenous coordination protocols directly in terms of this very set of primitive concepts. This leads to my work on Reo. Reo is just such a language.

5 Reo

The name Reo is a Greek word which means “[I] flow” —- as water in streams. I started the work on Reo on my own a few years ago [2, 10, 4], but was soon joined by many of my colleagues who have since made significant contributions to its development and implementation. Most notably, coalgebraic semantics for Reo based on stream calculus [11], constraint automata for model-checking of Reo circuits [9, 8], temporal logic specifications of their behavior [7], and elegant schemes for distributed solution of synchronization and exclusion constraints in the actual implementation of Reo [13], are only a few of the many interesting and useful results that I certainly would not have been able to achieve on my own. It is a pleasure for me to acknowledge the contributions of my colleagues, especially today.

Reo offers a channel-based exogenous coordination model wherein complex coordinators, called connectors are compositionally built out of simpler ones. Each connector explicitly represents an interaction that constrains the success of the input/output actions of its engaged actors. The simplest connectors in Reo are a set of channels supplied by users.

5.1 Channels

Reo defines a channel as a medium of communication with exactly two ends, and a constraint that defines its interaction protocol through these ends. Reo recognizes two types of channel ends: source ends, through which data enter channels, and sink ends through which data come out of channels. It is important to note that Reo does not define anything else about channels.

Reo does not define any specific channels either. Users define different channel types and all aspects of their behavior in terms of specific constraints that relate their data exchanges through their respective ends. These constraints define, for example, whether a channel is synchronous or asynchronous, whether or not it has a buffer, whether or not its buffer is bounded, whether or not it retains the order of the data items it receives, whether it loses some of its data, or generates fresh data items, etc.

Reo does not even require a channel to have a source and a sink. It is perfectly content with a channel that has two sources or two sinks, with whatever behavior a user may define for it.

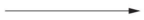
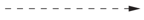
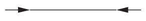
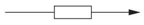
	Synchronous Channel	I/O operations on its ends succeed synchronously (i.e., atomically)
	LossySync Channel	A take on its sink blocks waiting for a write on its source; every write immediately succeeds; if a pending take exists, data is transferred otherwise, the data is lost
	SyncDrain Channel	I/O operations on its ends succeed synchronously (i.e., atomically)
	Asynchronous FIFO1 Channel	A write succeeds if the buffer is empty, a take succeeds if the buffer is full. Buffer is initially empty and has capacity of 1.

Figure 1: A sample of simple channels

To work with Reo, a user must define a set of primitive channels, and their precise behavior. To demonstrate what Reo can do, we define a small set of channels shown in Figure 1.

A Synchronous channel, graphically represented as a solid arrow, has a source- and a sink-end. This channel represents the primitive interaction enforced by the constraint of synchronizing the success of the two I/O operations on its two ends. In other words, it blocks a write operation on its source end or a take operation on its sink end, as necessary, to ensure that these two operations succeed atomically.

FIFO1 is an asynchronous channel with a source end and a sink end and a bounded buffer with the capacity to contain at most 1 data item. Its buffer is initially empty. With an empty buffer, a write operation on its source end succeeds and fills the buffer. With a non-empty buffer, a take on the sink end of this channel succeeds and removes the data. Otherwise, I/O operations block waiting for the status of the buffer to change.

SyncDrain is a synchronous channel with two source ends; it has no sink end. This means no one can ever take any data out of this channel. Therefore, all data entered into this channel are lost. SyncDrain is a synchronous channel in exactly the same sense as an ordinary Synchronous channel: it represents a primitive interaction enforced by the constraint that synchronizes the two I/O operations on its ends. In this case they must both be write operations, and SyncDrain blocks either of the two, as necessary, to ensure that they succeed atomically.

LossySync is a synchronous channel with a behavior very similar to that of the Synchronous channel. Just as for a Synchronous channel, a take operation on the sink end of a LossySync blocks until a write is performed on its source end. Unlike the case of the Synchronous channel, all write operations on the source end of a LossySync immediately succeed: if there is a pending take on its sink end, then the written data item is transferred; otherwise, the write operation succeeds, but the written data item is lost.

We now have a useful set of primitive connectors to use in Reo. How do we use them?

5.2 Nodes

Reo defines a composition operator to construct complex connectors out of simpler ones, yielding a calculus of connector composition, where channels constitute atomic connectors. For this purpose, Reo introduces the concept of a node. Initially, every single channel end is a singleton node. The join operator in Reo composes nodes into more complex nodes. Recall that there are only two types of channel ends: source and sink. Thus, there can be only two types of singleton nodes, which when combined, can yield only three types of nodes.

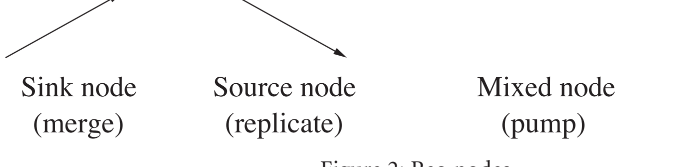


Figure 2: Reo nodes

As we see in Figure 2, a node that contains only source channel ends is called a source node. One that contains only sink channel ends is called a sink node. And a node that contains both kinds of channel ends is called a mixed node.

Reo merely defines the behavior of these three types of nodes. It states that components can perform write operations on source nodes, and take operations on sink nodes, but no I/O operation is allowed on a mixed node.

Reo defines the behavior of a source node as follows. A write operation on a source node blocks until all channel ends that coincide on that node are prepared to consume the written data. Then, the write succeeds and the data item is automatically replicated into all those channel ends. Thus, a source node replicates.

Reo defines the behavior of a sink node as follows. A take operation on a sink node blocks until there is at least one channel end coincident on that node that has a suitable data item to offer for the take. If there is exactly one such coincident channel end, then that channel end is selected. If more than one such channel end exists, one of them is selected non-deterministically. Subsequently, the data item offered by the selected channel end is consumed by the take operation. Other channel ends and the data that they contain are not affected. Thus, a sink node non-deterministically merges the data items available through its channel ends.

Reo defines the behavior of a mixed node as a combination of the behavior of the other two types of nodes. A mixed node is a self-contained pumping station that repeatedly selects a value through one of its sink ends and replicates it to all of its source ends. The subtlety is that nodes have no buffer to hold any data. Therefore, before a mixed node selects a value out of one its coincident sink ends, it must ensure that this value can be replicated into all of its coincident source ends.

5.3 Connector circuits

Given these as the rules of the game, what sorts of connectors can we compose in Reo out of our primitive channels in Figure 1?

5.3.1 Regulator

Figure 3 shows a simple, yet very useful connector composed out of three channels: two Synchronous channels and one SyncDrain. This connector represents an interaction that enforces a three-way synchronization constraint on the flow of data through its loose ends. For a write to `a` to succeed, node `m` must be able to replicate the written data into its two coincident source channel ends. The source end of the Synchronous channel coincident on m cannot accept any data item unless its sink end at b can synchronously dispense it. The source end of the SyncDrain channel coincident on m cannot accept any data item unless it can synchronously consume another data item through its opposite end at c.

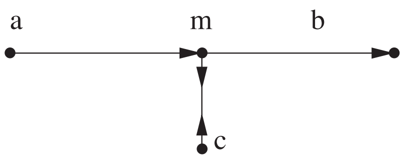


Figure 3: Write-cue regulator

The connector in Figure 3 shows one of the most basic forms of exogenous coordination: the number of data items that flow from a to b is the same as the number of data items that flow through c. The behavior of this connector is analogous to that of a transistor in the world of electronic circuits. Our transistor shows up frequently in construction of other Reo circuits, such as the one in Figure 5.

A component instance connected to c can count and regulate the flow of data from a to b by the timing and the number of write operations that it performs on c. The entity connected to c need not know anything about the entities at a and b, nor that its own write actions actually regulate this flow. The two entities that communicate through a and b need not know anything about the fact that they are communicating with each other, nor that the rate and the volume of their communication are regulated and/or measured by a third entity at c. This blissful ignorance is the essence of exogenous coordination.

5.3.2 Ordering

The connector in Figure 4 consists of three channels: a SyncDrain ab, a Synchronous channel ac, and a FIFO1 bc. This connector represents the interaction protocol imposed by a constraint that orders the flow of data from a and b through to c. The constraint imposed by this circuit ensures that the sequence of data items obtained through c consists of the first data item written to a, followed by the first data item written to b, followed by the second data item written to a, followed by the second data item written to b, etc.

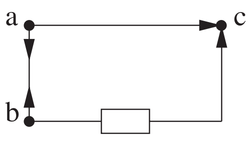


Figure 4: Alternator

We can use this circuit to compose two writers and a reader that know nothing about one another, each attempting to exchange data at its own pace. Unbeknownst to them all, this connector ensures that the reader alternately obtains data from one and then the other writer. This is another simple useful example of exogenous coordination.

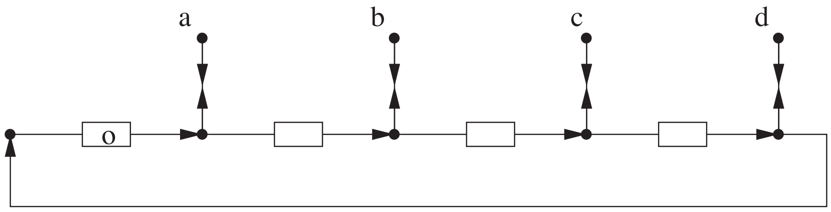


Figure 5: Sequencer

5.3.3 Sequencer

The connector in Figure 5 represents an interaction protocol that constrains the write operations on the nodes `a`, `b`, `c`, and `d`, to succeed only in the strict left to right order. This connector implements a sequencer protocol.

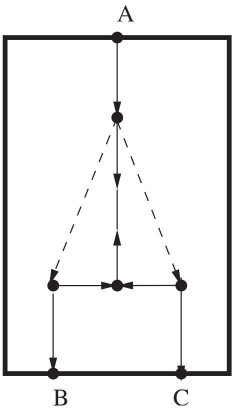


Figure 6: Exclusive router

5.4 Expressiveness

Connector composition in Reo is a very powerful mechanism for construction of complex interactions. Interaction protocols that can be expressed as ω-regular expressions over I/O operations can be composed in Reo out of a small set of only five primitive channel types [4]. Adding unbounded FIFO to the above set of channel types makes channel composition in Reo Turing complete [6].

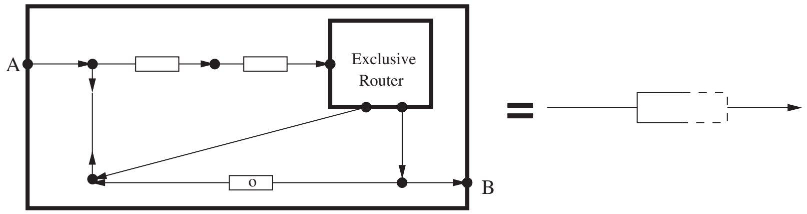


Figure 7: ShiftLossyFIFO1 channel and its graphical symbol

In principle, all computation can be done in Reo through coordination. Thus, the distinction between computing components and coordinating connectors becomes purely a subjective matter of convenience. Reo does not dictate what should be a component as opposed to a connector. Each application makes this distinction based on its own specific requirements.

Our tools not only influence how we solve problems, they also change our very notion of those problems. Process algebras explicitly compose and construct processes making the interaction relations amongst them ephemeral and implicit. Reo, on the other hand, explicitly composes constraints that represent interaction protocols and makes processes that engage in those relations implicit. Reo's liberal notion of channels and its fundamental notion of connector composition allow interaction protocols involving an expressive arbitrary mix of synchrony and asynchrony.

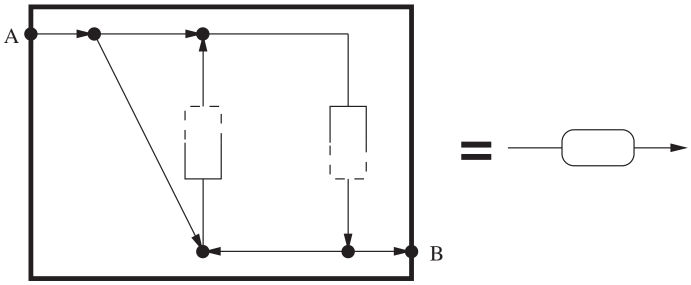


Figure 8: Variable and its graphical symbol

6 Coordinated composition

Returning to our simple examples of system composition, Figure 9 shows a connector that compensates for the mismatch of the periods of the two components in our time-display system. This connector consists of a single instance of a dataflow variable, whose construction is shown in Figure 8. The variable in Figure 8 uses two instances of the ShiftLossy FIFO1 connector shown in Figure 7, which in turn uses an instance of the exclusive router of Figure 6.

These connector circuits represent commonly useful generic interaction protocols, and the ease with which they can be composed in Reo to construct connectors that exogenously coordinate their engaged actors in more complex interactions.

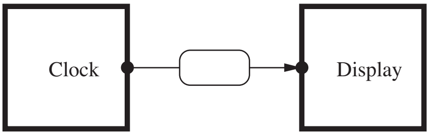


Figure 9: Time-Display system

Figure 10 shows our clock, thermometer, and display components composed together. The connector circuit in this figure consists of a two-node sequencer, two variables, and other channels. This circuit exogenously coordinates the components to manifest our desired alternating protocol.

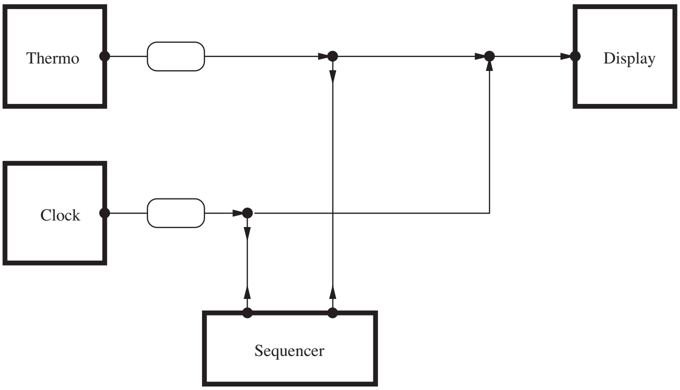


Figure 10: Time-Temperature-Display system

7 Dining philosophers

A Reo circuit is an explicit, tangible piece of specification or program code that represents an interaction. The same Reo circuit can be employed to engage entirely different sets of actors in its exogenous coordination, to yield entirely different systems. Perhaps more interestingly, the same set of actors can be composed together with different circuits, producing systems with very different emergent behavior. To demonstrate this, we use the classical dining-philosophers problem.

Figure 11 shows 4 philosophers and 4 chopsticks around a virtual round table. Each philosopher has 4 output ports through which it attempts to “pick” and “free” its right and left chopsticks. Each chopstick has two input ports, through which it is picked and freed. All channels are of type Synchronous.

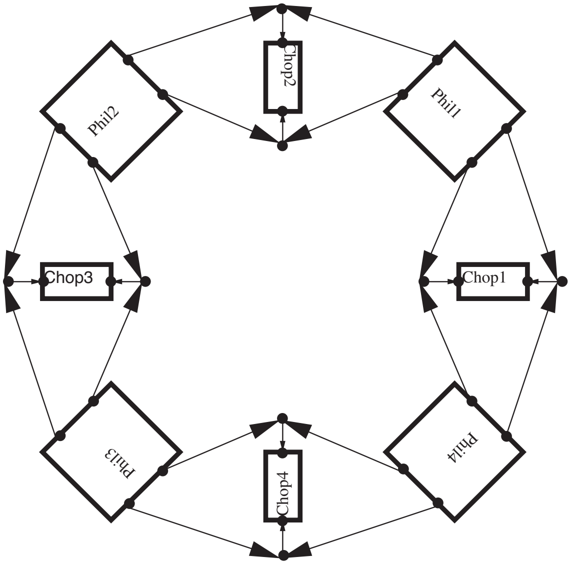


Figure 11: Dining philosophers may deadlock

A philosopher attempts to pick or free a chopstick on its either side by performing a write operation on its own respective port. The success of a write operation on its front-right or front-left port means that it has obtained the chopstick on its right or left, respectively. The success of a write operation on its rear-right or rear-left port means that it has released the chopstick on its right or left, respectively. Every philosopher attempts to pick its right chopstick before its left one.

A chopstick is initially free and performs a take operation on its front port, making itself available to be picked. The success of this take operation means the chopstick is in use, after which it performs a take operation on its rear port, waiting to be released. The success of this take operation means that the chopstick is free, and it repeats the cycle.

Consider what happens in the node at the three-way junction connected to the front port of Chop1. If Chop1 is free, either one of the two philosophers Phil1 and Phil4 that happens to write its pick request first will succeed to pick Chop1. If Phil1 and Phil4 attempt to pick Chop1 at the same time, the behavior of this mixed node guarantees that only one of them succeeds, nondeterministically; the write operation of the other remains pending until Chop1 becomes free again.

The Reo circuit in this application enables philosophers to repeatedly go through their “eat” and “think” cycles at their leisure, resolving their contentions for picking the same chopsticks nondeterministically. This composition is a faithful implementation of the dining-philosophers problem; all the way down to its possibility of deadlock.

If all chopsticks are free and all philosophers attempt to pick their first chopsticks at the same time, of course, they will all succeed. However, this leaves no free chopstick for any philosopher to pick before it can eat. No philosopher will relinquish its chopstick before it finishes its eating cycle. Therefore, this application deadlocks, as expected.

Clearly, this deadlock is an emergent property of the composed system, not an inherent property of any of its components. It is natural, then, to wonder if it is possible to compose the exact same components differently as to obtain a different emergent behavior, namely one that precludes the possibility of deadlocks.

Exogenous coordination makes this possible in Reo without any extra code, additional components, central authority, or modification to any of the existing components.

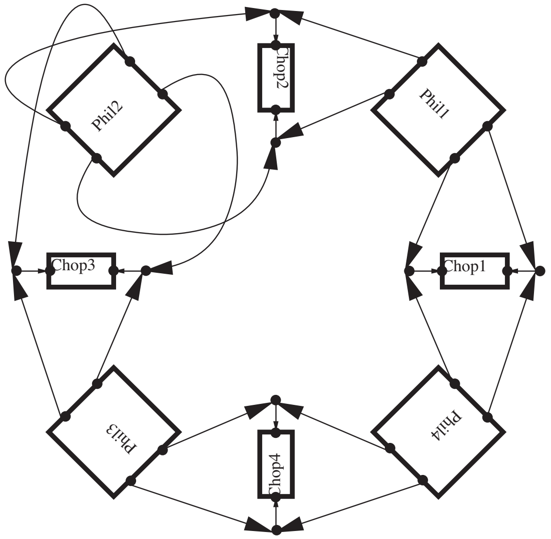


Figure 12: Dining philosophers cannot deadlock

Figure 12 shows a slightly different composition topology of the same set of Synchronous channels that connect the exact same instances of philosophers and chopsticks as before.

We have flipped the left and right connections of Phil2 to its adjacent chopsticks without their knowledge. None of the components in the system is aware of this change, nor are they modified in any way to accommodate it. Our flipping of these connections is purely external to all components. This is exogenous coordination. Interestingly, deadlock is now impossible.

We see that exogenous coordination in Reo allows different compositions of the same components yield systems with entirely different emergent behavior.

8 Conclusion

Reo is a simple, rich, versatile, and surprisingly expressive language for compositional construction of (distributed) systems out of black-box components. Its unique emphasis on interaction as the only first-class concept, allows composition of primitive interactions into reusable coordinating connectors.

Reo allows arbitrary user-defined channels as primitive interactions, yielding complex interaction protocols that involve an arbitrary mix of synchrony and asynchrony. This collection of features for definition of interaction protocols and atomic transactions is quite unique in Reo, and makes it more directly expressive than, for instance, dataflow models or Petri nets [6].

Together with my colleagues, we have used Reo to build a collection of commonly used coordinators; define interaction protocols in user-interfaces; express e-business process models for electronic auction protocols [21, 14]; and formalize a model in systems biology representing metabolization of galactose in yeast [12].

A number of tools already exist around Reo, and more is under development. For instance, we can translate Reo circuits into constraint automata and feed them to engines that run or animate them. Our colleagues in Bonn are extending existing algorithms to constraint-automata for model-checking of Reo circuits.

Our scalable distributed implementation of Reo is on-going. Currently, as the base layer for Reo, a middleware of distributed mobile channels exists [19, 20, 15]. Relatively simple distributed connector circuits, such as the one for the dining philosophers, can easily be implemented directly on top of this layer. More complex circuits require a full implementation of the generic behavior of Reo nodes.

Nodes involved in a circuit must propagate its synchrony and exclusion constraints to effectively solve a distributed constraint-satisfaction problem without a central authority or global view, using no means of communication other than the user-defined channels that interconnect them. This is a non-trivial problem for which my colleagues have devised an interesting solution algorithm [13].

Exogenous coordination makes interaction protocols tangible and concrete. Reo allows explicit construction of such interaction protocols out of a small set of primitive interactions in the form of connectors for composition of distributed systems. This is composition by interaction.

Mijnheer de Rector Magnificus,

my family, friends, and colleagues, thank you for your attention.

Ik heb gezegd.

References

[1] ARBAB, F. The IWIM model for coordination of concurrent activities. In *COORDINATION* (1996), P. Ciancarini and C. Hankin, Eds., vol. 1061 of *Lecture Notes in Computer Science*, Springer, pp. 34-56.

[2] ARBAB, F. Coordination of mobile components. *Electronic Notes in Theoretical Compututer Science 54* (2001).

[3] ARBAB, F. Abstract Behavior Types: A foundation model for components and their composition. In *FMCO* (2002), F. S. de Boer, M. M. Bonsangue, S. Graf, and W. P. de Roever, Eds., vol. 2852 of *Lecture Notes in Computer Science*, Springer, pp. 33-70.

[4] ARBAB, F. Reo: a channel-based coordination model for component composition. *Mathematical Structures in Computer Science 14*, 3 (2004), 329-366.

[5] ARBAB, F. Abstract Behavior Types: A foundation model for components and their composition. *Science of Computer Programing 55*, 1-3 (2005), 3-52.

[6] ARBAB, F. A behavioral model for composition of software components. *L'Objet* (2006). To appear.

[7] ARBAB, F., BAIER, C., DE BOER, F. S., AND RUTTEN, J. J. M. M. Models and temporal logics for timed component connectors. In *SEFM* (2004), IEEE Computer Society, pp. 198-207. Extended version to appear in *International Journal on Software and Systems Modeling* in 2006.

[8] ARBAB, F., BAIER, C., DE BOER, F. S., RUTTEN, J. J. M. M., AND SIRJANI, M. Synthesis of Reo circuits for implementation of component-connector automata speci - cations. In *COORDINATION* (2005), J.-M. Jacquet and G. P. Picco, Eds., vol. 3454 of *Lecture Notes in Computer Science*, Springer, pp. 236-251.

[9] ARBAB, F., BAIER, C., RUTTEN, J., AND SIRJANI, M. Modeling component connectors in Reo by Constraint Automata. In *Proc. International Workshop on Foundations of Coordination Languages and Software Architectures (FOCLASA 2003)* (July 2004), vol. 97 of *Electronic Notes in Theoretical Computer Science (ENTCS)*, Elsevier, pp. 25-46. Extended version to appear in *Science of Computer Programming* in 2006.

[10] ARBAB, F., AND MAVADDAT, F. Coordination through channel composition. In *COORDINATION* (2002), F. Arbab and C. L. Talcott, Eds., vol. 2315 of *Lecture Notes in Computer Science*, Springer, pp. 22-39.

[11] ARBAB, F., AND RUTTEN, J. J. M. M. A coinductive calculus of component connectors. In *WADT* (2002), M. Wirsing, D. Pattinson, and R. Hennicker, Eds., vol. 2755 of *Lecture Notes in Computer Science*, Springer, pp. 34-55.

[12] CLARKE, D., ARBAB, F., AND COSTA, D. Modeling coordination in biological systems. In *Proc. of the International Symposium on Leveraging Applications of Formal Methods (ISoLA 2004)* (Nov. 2004).

[13] Clarke, D., Costa, D., and Arbab, F. Connector colouring I: Synchronisation and context dependency. In *Proc. International Workshop on Foundations of Coordination Languages and Software Architectures (FOCLASA 2005)* (2005), Electronic Notes in Theoretical Computer Science (ENTCS), Elsevier. To appear.

[14] Diakov, N., Zlatev, Z., and Pokraev, S. Composition of negotiation protocols for e-commerce applications. In *The 2005 IEEE International Conference on e-Technology, e-Commerce and e-Service* (March 2005), W. Cheung and J. Hsu, Eds., pp. 418-423.

[15] Guillen-Scholten, J., Arbab, F., de Boer, F., and Bonsangue, M. Mocha-pi: An exogenous coordination calculus based on mobile channels. In *Proc. of the 20th ACM Symposium on Applied Computing (SAC 2005), Special Track on Coordination Models, Languages, and Applications* (Santa Fe, New Mexico, USA, Mar. 13-17 2005), ACM.

[16] Papadopoulos, G., and Arbab, F. Coordination models and languages. In *Advances in Computers – The Engineering of Large Systems*, M. Zelkowitz, Ed., vol. 46. Academic Press, 1998, pp. 329-400.

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

[18] Rutten, J. A coinductive calculus of streams. *Mathematical Structures in Computer Science 15*, 1 (February 2005), 93-147.

[19] Scholten, J. G., Arbab, F., de Boer, F. S., and Bonsangue, M. M. Mobile channels, implementation within and outside components. *Electronic Notes in Theoretical Computer Science 66*, 4 (2002).

[20] Scholten, J. G., Arbab, F., de Boer, F. S., and Bonsangue, M. M. A channel-based coordination model for components. *Electronic Notes in Theoretical Computer Science 68*, 3 (2003).

[21] Zlatev, Z., Diakov, N., and Pokraev, S. Construction of negotiation protocols for E-Commerce applications. *ACM SIGecom Exchanges 5*, 2 (November 2004), 11-22.