Treo: Textual Syntax of Reo Connectors

Kasper Dokter and Farhad Arbab

Leiden University, Leiden, The Netherlands   Centrum Wiskunde & Informatica, Amsterdam, Netherlands

Reo is an interaction-centric model of concurrency for compositional specification of communication and coordination protocols. Formal verification tools exist to ensure correctness and compliance of protocols specified in Reo, which can readily be (re)used in different applications, or composed into more complex protocols. Recent benchmarks show that compiling such high-level Reo specifications produces executable code that can compete with or even beat the performance of hand-crafted programs written in languages such as C or Java using conventional concurrency constructs.

The original declarative graphical syntax of Reo does not support intuitive constructs for parameter passing, iteration, recursion, or conditional specification. This shortcoming hinders Reo's uptake in large-scale practical applications. Although a number of Reo-inspired syntax alternatives have appeared in the past, none of them follows the primary design principles of Reo: a) declarative specification; b) all channel types in Reo are user-defined; and c) channels compose via shared nodes. In this paper, we offer a textual syntax for Reo that respects these principles and supports flexible parameter passing, iteration, recursion, and conditional specification. In on-going work, we use this textual syntax to compile Reo into target languages such as Java, Promela, and Maude.

1 Introduction

The advent of multicore processors has intensified the significance of coordination in concurrent applications. A programmer tackles the coordination concern of an application by specifying a (usually implicit) protocol that defines all possible permissible interactions among different active components of the application. Depending on the language used, programmers define their protocols at different levels of abstraction. A threading library, for instance, generally offers only basic synchronization primitives, such as locks and semaphores, that can be inserted into imperative code to ensure execution follows an implicitly defined protocol. Exogenous coordination languages offer syntax to programmers to explicitly define their interaction protocols at a high level of abstraction.

Reo [3, 4] is an example of such a coordination language that defines an interaction protocol as a *connector*: a graph-like structure that enables (a)synchronous data flow along its edges (cf., Figure 1). Each edge, called a *channel*, has a user-defined type and two channel ends. The type determines the behavior of the channel, specified as a constraint on the flows of data at its two ends. A channel end is either a *source end* through which the channel accepts data, or a *sink end* through which the channel offers data. Multiple channel ends coincident at a vertex of the connector together form a *node*. Nodes have predefined 'merge-replicate' behavior: a node repeatedly accepts a datum from one of its coincident sink ends, chosen non-deterministically, and offers that datum through all of its coincident source ends.

Tools for Reo have been implemented as a collection of Eclipse plugins called the ECT [1]. The main plugin consists of graphical editor that allows a user to draw a connector on a canvas. The graphical editor has an intuitive interface with a flat learning curve. However, it does not provide constructs to express parameter passing, iteration, recursion, or conditional construction. Such language constructs are more easily offered by familiar programming language constructs in a textual representation of connectors.

In the context of Vereofy (a model checker for Reo), Baier, Blechmann, Klein, and Klüppelholz developed the Reo Scripting Language (RSL) and its companion language, the Constraint Automata

Submitted to:
MeTRiD 2018

Reactive Module Language (CARML) [5, 19]. RSL is the first textual language for Reo that includes a construct for iteration, and a limited form of parameter passing. Primitive channels and nodes are defined in CARML, a guarded command language for specification of constraint automata. Programmers then combine CARML specified constraint automata as primitives in RSL to construct complex connectors. In contrast to the declarative nature of the graphical syntax of Reo, RSL is imperative.

Jongmans developed the First-Order Constraint Automata with Memory Language (FOCAML) [12], a textual declarative language that enables compositional construction of connectors from a (pre-defined set of) primitive components. As a textual representation for Reo, however, FOCAML lacks support for two of its basic design principles. First, Reo channels have user-defined types, while FOCAML components have fixed predefined types (i.e., constraint automata with memory [6]). Second, Reo nodes have predefined behavior, while the concept of a node as such does not exist in FOCAML. The absence of nodes forces explicit construction of their 'merge-replicate' behavior in FOCAML specifications.

Jongmans et al. have shown by benchmarks that compiling Reo specifications can produce executable code whose performance competes with or even beats that of hand-crafted programs written in languages such as C or Java using conventional concurrency constructs [18, 14, 16, 15, 17]. A textual syntax for Reo that preserves its declarative, compositional nature, allows user-defined primitives, and faithfully complies with the semantics of its nodes can significantly facilitate the uptake of Reo for specification of protocols in large-scale practical applications.

In this paper, we introduce Treo, a declarative textual language for component-based specification of Reo connectors with user-defined component types and user-defined node behavior. We recall the basics of Reo (Section 2). We describe the structure of a Treo file by means of an abstract syntax (Section 3). In the Appendix, we provide a concrete syntax of Treo as an ANTLR4 grammar [22]. In on-going work, we currently use Treo to compile Reo into target languages such as Java, Promela, and Maude [2].

A user-defined component type consist of a set of components together with a composition operator $\wedge$, a substitution operator $[/]$, and a trivial component $\top$ (Section 4). The composition operator defines the behavior of composite components. The substitution operator binds nodes in the interface or passes values to parameters.

For a given component type, we define the semantics of abstract Treo programs (Section 5). Treo is very liberal with respect to parameter values. A component definition not only accepts the usual (structured) data as actual parameters, but also other component instances and other component definitions. Among other benefits, this flexible parameter passing supports *component sharing*, which is useful to preserve component encapsulation [7, Figure 2].

A given component type may possibly distinguish between inputs and outputs. Thus, not all combinations of components may result in a valid composite component. For example, the composition may not be defined, if two components share an output. In Treo, however, it is safe to compose component on their outputs, because, complying with the semantics of Reo, the compiler inserts special *node components* to ensure well-formed compositions (Section 6).

We conclude by discussing related work (Section 7), and pointing out future work (Section 8).

2 Reo

We briefly recall the basics of the Reo language and refer to [3] and [4] for further details. Reo is a language for specification of interaction protocols, originally proposed with a graphical syntax. A Reo program, called a *connector*, is a graph-like structure whose edges consist of *channels* that enable synchronous and asynchronous data flow and whose vertices consist of *nodes* that synchronously route

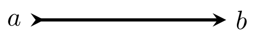
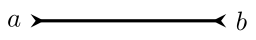
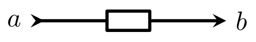
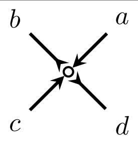
	A Sync channel accepts datum from its source end a, when its simultaneous offer of this datum at its sink end b succeeds.
	A SyncDrain channel simultaneously accepts data from both its source ends a and b and loses the data.
	An empty FIFO_1 accepts data from its source end a and becomes a full FIFO_1. A full FIFO_1 offers its stored data at its sink end b and, when its offer succeeds, it becomes an empty FIFO_1 again.
	A Reo node accepts a datum from one of its coincident sink ends (a or c), when its simultaneous offer to dispense a copy of this datum through every one of its coincident source ends (b and d) succeeds.

Table 1: Informal description of the behavior of nodes and of some channels in Reo.

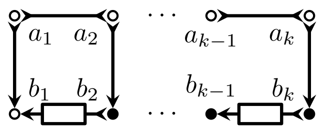


Figure 1: Construction of the $\mathsf{Alternator}_k$ Reo connector, for $k \geq 2$.

data among multiple channels. Each channel has a type and two channel ends. Each channel end is either a *source end*, through which the channel accepts data, or a *sink end*, through which the channel offers data. The type of a channel completely defines the behavior of the channel. Table 1 shows some frequently used channels and an example node together with an informal description of their behavior.

The key concept in Reo is composition, which allows a programmer to build complex connectors out of simpler ones. For example, using the channels in Table 1, we can construct the $\mathsf{Alternator}_k$ connector, for $k \geq 2$, as shown in Figure 1. For $k = 2$, the $\mathsf{Alternator}_2$ consists of four nodes (a_1, a_2, b_1, and b_2) and four channels, namely a SyncDrain channel (between a_1 and a_2), two Sync channels (from a_1 to b_1, and from a_2 to b_2), and a FIFO_1 channel (from b_2 to b_1).

The behavior of the $\mathsf{Alternator}_2$ connector is as follows. Suppose that the environment is ready to offer a datum at nodes a_1 and a_2, and ready to accept a datum from node b_1. According to Table 1, nodes a_1 and a_2 both offer a copy of their received datum to the SyncDrain channel. The SyncDrain channel ensures that nodes a_1 and a_2 accept data from the environment only simultaneously. The Sync channel from a_1 to b_1 ensures that node b_1 simultaneously obtains the datum offered at a_1. By definition, node b_1 either accepts a datum from the connected Sync channel or it accepts a datum from the FIFO_1 channel (but not from both simultaneously), and offers this datum immediately to the environment. The Sync channel from a_2 to b_2 ensures that the value offered at a_2 is stored in the FIFO_1 buffer. In the next step, the environment at node b_1 can retrieve the datum in the buffer, after which the behavior repeats.

3 Abstract syntax

We now present a textual representation for the graphical Reo connectors in Section 2. Table 2 shows the abstract syntax of Treo. We introduce the symbols in the abstract syntax by identifying them in some

K	$::=$	$I \mid KND$	D	$::=$	$V \mid \langle U_0 \rangle (U_1)\{C\}$
L	$::=$	$\varepsilon \mid L,T \mid L,T_0..T_1$	C	$::=$	$V \mid A \mid C_0C_1 \mid \{C \mathbin{\vert} P\} \mid D\langle L \rangle (U)$
U	$::=$	$\varepsilon \mid U,V$	T	$::=$	$V \mid C \mid D \mid [L] \mid T_0 : T_1 \mid T[L] \mid F(L)$
V	$::=$	$N \mid V[L]$	P	$::=$	$V \in T \mid R(L) \mid \neg P \mid P_0 \wedge P_1 \mid P_0 \vee P_1 \mid (P)$

Table 2: Abstract syntax of Treo, with start symbol K (a source file), and terminal symbols for imports (I), primitive components (A), functions (F), relations (R), names (N), and the empty list (ε). The vertical bar in $\{C \mathbin{\vert} P\}$ is just text.

concrete examples. Consider the following Treo file (K in Table 2) representing the $\mathsf{Alternator}_2$:

```
import syncdrain; import sync; import fifo1;
alternator2(a1,a2,b1) { sync(a1,b1) syncdrain(a1,a2) sync(a2,b2) fifo1(b2,b1) }
```

On the first line, we import (I) three different *component definitions*. On the second line, we define the `alternator2` component (ND). Its definition (D) has no parameters ($\langle U_0 \rangle$), and three nodes `a1`, `a2`, and `b1` in its interface ((U_1)). The body ($\{C\}$) of this definition consists of a set of *component instances* that interact via shared nodes. The first component instance `sync(a1,b1)` is an instantiation ($D\langle L \rangle (U)$) of the imported `syncdrain` definition (D) with nodes `a1` and `b1` ((U)) and without any parameters ($\langle L \rangle$).

All nodes that occur in the body, but not in the interface, are hidden. Hiding renames a node to a fresh inaccessible name, which prevents it from being shared with other components. In the case of `alternator2`, node `b2` is not part of the interface and hidden.

Constructed from existing components, `alternator2` is a *composite* component (C_0C_1). However, not every component is constructed from existing components, and we call such components *primitive* (A). The following Treo code shows a possible (primitive) definition of the `fifo1` component.

```
fifo1(a?,b!) { empty -{a},true-> full; full -{b},true-> empty; }
```

The definition of the `fifo1` differs from the definition of the `alternator2` in two ways.

The first difference is that the `fifo1` component is (in this case) defined directly as a *constraint automaton* [6]. The constraint automaton semantics is a popular specification of the type of Reo components, and forms the basis of the Lykos compiler [12]. However, constraint automata are not the *de facto* standard: the literature offers more than thirty different semantic types for specification of Reo components [13], such as the coloring semantics and timed data stream semantics. To accommodate the generality that disparate semantics allow, Treo features *user-defined component types*, which means that the syntax of the implementation of primitive components is user-defined. For example, this means that we may also define the `fifo1` component by referring to a Java file via `fifo1(a?,b!){ "MyFIFO1.java" }`.

The second difference is that the nodes `a` and `b` in the interface are *directed*. That is, each of its interface nodes is either of type input or output, designated by the markers `?` and `!`, respectively. In Reo, it is safe to join two channels on a shared sink node (e.g., node b_1 in Figure 1). However, the composition operators in most Reo semantics do not automatically produce the correct behavior for such nodes (e.g., see [6, Section 4.3] for further details). Therefore, most Reo semantics require *well-formed* compositions, wherein each node has at most one input channel end and at most one output channel end.

The restriction of well-formed compositions can be very inconvenient in practice. To ensure well-formed compositions, a programmer must implement every Reo node with more then one input- or output channel end as a *node component*. The interface of this node component is determined by the *degree*, which is a pair (i,o) giving the numbers of its coincident source and sink ends. Such explicit node components make component constructions verbose and hard to maintain. For convenience, the Treo compiler uses the above input/output annotations to compute the degree of each node in the composition, and subsequently inserts the correct node components component in the construction. We may view the input/output annotations as syntactic sugar that ensures well-formed compositions. This feature allows programmers to remain oblivious to these annotations and well-formed compositions.

The ellipses in Figure 1 signify the parametrized construction of the $\mathsf{Alternator}_k$ connector, for $k > 2$. This notation is informal and not supported in the graphical Reo editor [1], which offers no support for parametrized constructions. In Treo, however, we can define the $\mathsf{Alternator}_k$ connector as:

```
alternator<k>(a[1:k],b[1]) { sync(a[1],b[1])
    { syncdrain(a[i-1],a[i]) sync(a[i],b[i]) fifo1(b[i],b[i-1]) | i in [2..k] } }
```

The definition of the `alternator` depends on a parameter `k`. Since Treo is a strongly typed language with type-inferencing, there is no need to specify a type for the (integer) parameter `k`. The interface consists of an array of nodes `a[1:k]` and the single node `b[1]`. Here, `[1:k]` is an abbreviation for the list `[[1..k]]` that contains a single list of length `k`. The array `a[1:2]` stands for the *slice* `[a[1],a[2]]` of `a`, while the expression `a[1..2]` stands for the element `a[1][2]` in `a` (cf., Equation (2)). For iteration, we write `{ ... | i in [2..k] }` using set-comprehension ($\{C \mid P\}$).

Instead of defining `alternator` iteratively, we may also provide a recursive definition as follows:

```
recursive_alternator(a[1:k],b[1],b[k]) { recursive_alternator(a[1:k-1],b[1],b[k-1])
    { syncdrain(a[k-1],a[k]) sync(a[k],b[k]) fifo1(b[k-1],b[k]) | k > 1 } }
```

The value of `k` is deduced from the size of array `a[1:k]`. We use set-comprehension `{ ... | k > 1 }` for conditional construction. Indeed, the resulting set of component instances is non-empty, only if `k > 1` holds. Although Treo syntax allows recursive definitions, the semantics presented in Section 5 does not yet support recursion, which we leave as future work.

We illustrate the practicality of Treo by providing code for a chess playing program [12, Figure 3.29]. In this program, two teams of chess engines compete in a game of chess. We define a chess team as the following Treo component:

```
import parse; /* and the other imports */
team<engine[1:n]>(inp,out) {
    for (i in [1..n]) {
        engine[i](inp,best[i]) parse(best[i],p[i])
        if (i > 1) concatenate(a[i-1],p[i],a[i]) }
    sync(best[1],a[1]) majority(a[n],b) syncdrain(b,c)
    fifo1(inp,c) move(b,d) concatenate(c,d,out) }
```

The for-loop `for (i in [1..n]) ...` and if-statement `if (i > 1) ...` are just syntactic sugar for set-comprehensions `{ ... | i in [1..n] }` and `{ ... | i > 1 }`, respectively. The `team` component depends on an array `engine[1:n]` of parameters. This array does not contain the usual data values, but consist of Treo component definitions. In the body of the `team` component, these definitions are instantiated via `engine[i](inp,best[i])`. In RSL [5, 19] and FOCAML [12], it impossible to pass a component as a parameter, which shows that Treo is more expressive than those languages.

We may view the `team` component as an example of role-oriented programming [8]. Indeed, the `team` component encapsulates a list of chess engines in a component, so that they can collectively be used as a single participant in a chess match:

```
match() { fifo1full<"">(a,b) fifo1(c,d) team<[eng1, eng2]>(a,d) team<[eng3]>(b,c) }
```

Treo treats not only component definitions but component instances as values. By passing a single component instance as a parameter to multiple components, this feature allows *component (instance) sharing*. Hence, it is straightforward to implement a chess match, wherein a single instance of a chess engine plays against itself.

4 Component types

As noted in Section 3, Reo channels can be defined in many different types of semantics [13], such as the constraint automaton semantics, the colouring semantics, or the timed data stream semantics. Although each type of Reo semantics has its unique properties, they all consists of a collection of composable components with parameters and nodes, which we call a *component type*:

Definition 4.1 (Component types)**.** A component type over a set of names $\mathscr{N}$ with values from $\mathscr{V}$ is a tuple $(\mathscr{C}, \wedge, [/], \top)$ that consists of a set of components $\mathscr{C}$, a composition operator $\wedge : \mathscr{C} \times \mathscr{C} \longrightarrow \mathscr{C}$, a substitution operator $[/] : \mathscr{C} \times (\mathscr{N} \cup \mathscr{V}) \times \mathscr{N} \longrightarrow \mathscr{C}$, and trivial component $\top \in \mathscr{C}$.

We assume that the set of names and the set of values are disjoint, i.e., $\mathscr{N} \cap \mathscr{V} = \emptyset$. For convenience, we write $C \wedge C'$ for $\wedge(C,C')$, and $C[y/x]$ for $[/](C,y,x)$. For any component type T, we write $\mathscr{C}_T$ for its set of components, $\wedge_T$ for its composition operator, $[/]_T$ for its substitution operator, and $\top_T$ for its trivial component. The composition operator $\wedge_T$ ensures that the behavior of finite non-empty compositions is well-defined. To empty compositions we assign the trivial component $\top_T$. The substitution operator $[/]_T$ allows us to change the interface of a component via renaming or instantiation. Let $C \in \mathscr{C}_T$ be a component and $x \in \mathscr{N}$ a name. For a name $y \in \mathscr{N}$, the construct $C[y/x]_T$ renames every occurrence of name x in C to y. For a value $y \in \mathscr{V}$, the construct $C[y/x]_T$ instantiates (parameter) x in C to y.

A component type T implicitly defines an interface for each component $C \in \mathscr{C}$ via the map $\text{supp} : \mathscr{C}_T \longrightarrow 2^{\mathscr{N}}$ defined as $\text{supp}(C) = \{x \in \mathscr{N} \mid C[y/x]_T \neq C, \text{ for some name } y \in \mathscr{N}\}$. If name x does not 'occur' in C, substitution of x by any name y does not affect C, i.e., $C[y/x]_T = C$.

Example 4.1 (Systems of differential equations)**.** The set ODE of systems of ordinary differential equations with variables from $\mathscr{N}$ and values $\mathscr{V} = \{v : \mathbb{R} \longrightarrow \mathbb{R}\}$ constitute a component type. Composition is union, substitution is binding a name or value to a given name, and the trivial component is the empty system of equations. Using the ODE component type, we can define continuous systems in Treo. △

Example 4.2 (Process calculi)**.** Consider the process calculus CSP, proposed by Hoare [10]. The set CSP of all such process algebraic terms comprises a component type. Each process can participate in a number of events, which we can interpret as names from a given set $\mathscr{N}$. We model the composition of CSP processes P and Q by means of the interface parallel operator $P \;|[X]|\; Q$, where $X \subseteq \mathscr{N}$ is the set of event names shared by P and Q. We define substitution as simply (1) renaming the event, if a name is substituted for an event; or (2) hiding the event, if a values is substituted for an event. Since neither STOP nor SKIP shares any event with its environment, we may use either one to denote the trivial component. △

Example 4.3 (I/O-components)**.** Let T be a component type over $\mathscr{N}$ and $\mathscr{V}$. We define the I/O-component type IO_T over T using the notion of a primitive I/O-component of type T.

A *primitive I/O-component* P of type T is a tuple (C,I,O), where $C \in \mathscr{C}_T$ is a component, $I \subseteq \mathscr{N}$ is a set of input names, $O \subseteq \mathscr{N}$ is a set of output names. For $P \subseteq \mathscr{N}$ and $x \in \mathscr{N}$ and $y \in \mathscr{N} \cup \mathscr{V}$, define

$$P[y/x] = \begin{cases} (P - \{x\}) \cup \{y\} & \text{if } x \in P \text{ and } y \in \mathscr{N} \\ P - \{x\} & \text{if } x \in P \text{ and } y \in \mathscr{V} \\ P & \text{otherwise} \end{cases} \tag{1}$$

We define substitution on primitive I/O-components as $(C,I,O)[y/x] = (C[y/x], I[y/x], O[y/x])$, for all $x \in \mathscr{N}$ and $y \in \mathscr{N} \cup \mathscr{V}$. We denote the set of primitive I/O-components over T as $\mathscr{P}_T$.

An *I/O-component* of type T is a sequence $P_1 \cdots P_n \in \mathscr{P}_T^*$, with $n \geq 0$, of primitive I/O-components of type T. Composition of I/O-components is concatenation $\cdot$ of sequences. The trivial I/O-component is the empty sequence ε. We define substitution of composite I/O-components as $(P_1 \cdots P_n)[y/x] = P_1[y/x] \cdots P_n[y/x]$, for all $x \in \mathcal{N}$ and $y \in \mathcal{N} \cup \mathcal{V}$. Hence, $\mathrm{IO}_T = (\mathscr{P}_T^*, \cdot, [/], \varepsilon)$ is a component type. $\triangle$

5 Denotational semantics

We define the denotational semantics of the Treo language over a fixed, but arbitrary, component type T. The main purpose of this denotational semantics is to provide a clear abstract structure that guides the implementation of Treo parsers. The general structure of our denotational semantics is quite standard, and adheres to Schmidt's notation [23].

Although Treo syntax allows recursive definitions, the semantics presented in this section does not support this feature. Since not all recursive definitions define finite compositions of components, extending the current semantics with recursion is not straightforward, and we leave it as future work.

Variables and terms in Treo are structured as non-rectangular arrays. The set of all *(ragged) arrays* over a set X is the smallest set $X^\square$ such that both $X \subseteq X^\square$ and $[x_0, \ldots, x_{n-1}] \in X^\square$, if $n \geq 0$ and $x_i \in X^\square$ for all $0 \leq i < n$. For example, the set $\mathbb{N}^\square$ of ragged arrays over integers contains all natural numbers from $\mathbb{N}$ as 'atomic' arrays, as well as the array $[37, [], [[2, [55], 3]]] \in \mathbb{N}^\square$. Every ragged array has a length, which can be computed via the map $\mathrm{len} : X^\square \longrightarrow \mathbb{N}$ defined inductively as $\mathrm{len}(x) = 0$, if $x \in X$, and $\mathrm{len}([x_0, \ldots, x_{n-1}]) = n$, otherwise. If $x = [x_0, \ldots, x_{n-1}] \in X^\square$ is a ragged array, we access its entries via the function application $x(i) = x_i$, for every $0 \leq i < n$. We extend the access map $\mathbb{N}^\square$ by defining

$$x([i_0, \ldots, i_n]) = \begin{cases} x(i_0)([i_1, \ldots, i_n]) & \text{if } i_0 \in \mathbb{N} \\ [x(i_{00})([i_1, \ldots, i_n]), \ldots, x(i_{0m})([i_1, \ldots, i_n])] & \text{if } i_0 = [i_{00}, \ldots, i_{0m}] \end{cases}, \tag{2}$$

whenever the right-hand side is defined. Two ragged arrays $x \in X^\square$ and $y \in Y^\square$ have the same structure ($x \simeq y$) iff $x \in X$ and $y \in Y$, or $\mathrm{len}(x) = \mathrm{len}(y)$ and $x(i) \simeq y(i)$ for all $0 \leq i < \mathrm{len}(x)$. We can flatten a ragged array from $X^\square$ to a sequence over X via the map $\mathrm{flatten} : X^\square \longrightarrow X^*$ defined as $\mathrm{flatten}(x) = x$, if $x \in X$, and $\mathrm{flatten}([x_0, \ldots, x_{n-1}]) = \mathrm{flatten}(x_0) \cdots \mathrm{flatten}(x_{n-1})$, otherwise.

Suppose that component type T is defined over a set of names $\mathcal{N}$ and a set of values $\mathcal{V}$, with $\mathcal{N} \cap \mathcal{V} = \emptyset$. For simplicity, we assume that, for every component $C \in \mathscr{C}_T$, its support $\mathrm{supp}(C) \subseteq \mathcal{N}$ is finite. Since Treo views components as values, we assume the inclusion $\mathscr{C}_T \subseteq \mathcal{V}$.

We assume that the set of names $\mathcal{N}$ is closed under taking subscripts from $\mathbb{N}$. That is, if $x \in \mathcal{N}$ is a name and $i \in \mathbb{N}$ is a natural number, then we can construct a fresh name $x_i \in \mathcal{N}$. To construct sequences of data with variable lengths, we use a map $\mathrm{lst} : \mathbb{N}^2 \longrightarrow \mathbb{N}^\square$ that constructs from a pair $(i, j) \in \mathbb{N}^2$ of integers a finite ordered list $[i, i+1, \ldots, j]$ in $\mathbb{N}^\square$.

Recall from Section 3 that a component accepts an arbitrary but finite number of parameters and nodes. Therefore, we define a component definition as a map $D : \mathcal{V}^\square \times \mathcal{N}^\square \longrightarrow \mathscr{C}_T \cup \{\lightning\}$ that takes an array of parameter values from $\mathcal{V}^\square$ and an array of nodes from $\mathcal{N}^\square$ and returns a component or an *error* $\lightning$. Let $\mathscr{D} = (\mathscr{C}_T \cup \{\lightning\})^{\mathcal{V}^\square \times \mathcal{N}^\square}$ be the set of all definitions. As mentioned earlier, Treo also allows definitions as values, which amounts to the inclusion $\mathscr{D} \subseteq \mathcal{V}$.[1]

We evaluate every Treo construct in its *scope* $\sigma : N \longrightarrow \mathcal{V}^\square$, with $N \subseteq \mathcal{N}$ finite, which assigns a value to a finite collection of locally defined names. We write $\Sigma = \{\sigma : N \longrightarrow \mathcal{V}^\square \mid N \subseteq \mathcal{N} \text{ finite}\}$ for the set of scopes. For a name $x \in \mathcal{N}$ and a value $d \in \mathcal{V}^\square$, we have a scope $\{x \mapsto d\} : \{x\} \longrightarrow \mathcal{V}^\square$ defined as $\{x \mapsto d\}(x) = d$. For any two scopes $\sigma, \sigma' \in \Sigma$, we have a composition $\sigma\sigma' \in \Sigma$ such that for every

[1] Such a set of values $\mathcal{V}$ exists only if $\mathcal{V} \mapsto \mathscr{C}_T \cup (\mathscr{C}_T \cup \{\lightning\})^{\mathcal{V}^\square \times \mathcal{N}^\square}$ admits a pre-fixed point. In this work, we simply assume that such $\mathcal{V}$ exists.

$x \in \mathrm{dom}(\sigma) \cup \mathrm{dom}(\sigma')$ we have $(\sigma\sigma')(x) = \sigma'(x)$, if $x \in \mathrm{dom}(\sigma')$, and $(\sigma\sigma')(x) = \sigma(x)$, otherwise. The composite scope $\sigma\sigma'$ can be viewed as an extension of σ that includes definitions and updates from σ'.

Let Names be the set of parse trees with root N, and let $\mathbf{N}[\![-]\!] : \text{Names} \longrightarrow \mathscr{N}$ be the semantics of names. We define the semantics of variables as a map $\mathbf{V}[\![-]\!] : \text{Variables} \longrightarrow (\mathscr{N}^{\square} \cup \{\lightning\})^{\Sigma}$, where Variables is the set of parse trees with root V. For a scope $\sigma \in \Sigma$, we define $\mathbf{V}[\![-]\!](\sigma)$ as follows:

1. If V is a name N, we define $\mathbf{V}[\![N]\!](\sigma) = \mathbf{N}[\![N]\!]$;
2. If V is $V[L]$, we define $\mathbf{V}[\![V[L]]\!](\sigma) = x(k) \in \mathscr{N}^{\square}$, if $\mathbf{V}[\![V]\!](\sigma) = x \in \mathscr{N}^{\square}$ and $\mathbf{L}[\![L]\!](\sigma) = k \in \mathbb{N}^{\square}$. Otherwise, we define $\mathbf{V}[\![V[L]]\!](\sigma) = \lightning$. Since $\mathscr{N}$ is closed under taking subscripts, we can define $n(i) = n_i$, for all $n \in \mathscr{N}$ and $i \in \mathbb{N}$, which ensures that $x(k) \in \mathscr{N}^{\square}$ is always defined.

The semantics of arguments is a map $\mathbf{U}[\![-]\!] : \text{Arguments} \longrightarrow (\mathscr{N}^{\square} \cup \{\lightning\})^{\Sigma}$, where Arguments is the set of all parse trees with root U. For a scope $\sigma \in \Sigma$, we define $\mathbf{U}[\![-]\!](\sigma)$ as follows:

1. If U is the empty sequence ε, then we define $\mathbf{U}[\![\varepsilon]\!](\sigma) = []$;
2. If U is U,V, we define $\mathbf{U}[\![U,V]\!](\sigma) = [x_1,\ldots,x_{n+1}]$, whenever $\mathbf{U}[\![U]\!](\sigma) = [x_1,\ldots,x_n]$ and $\mathbf{V}[\![V]\!](\sigma) = x_{n+1}$. Otherwise, we define $\mathbf{U}[\![U,V]\!](\sigma) = \lightning$.

The semantics of terms is a map $\mathbf{T}[\![-]\!] : \text{Terms} \longrightarrow (\mathscr{V}^{\square} \cup \{\lightning\})^{\Sigma}$, where Terms is the set of parse trees with root T. For a scope $\sigma \in \Sigma$, we define $\mathbf{T}[\![-]\!](\sigma)$ inductively as follows:

1. If T is a variable V, we define $\mathbf{T}[\![V]\!](\sigma) = \sigma(\mathbf{V}[\![V]\!](\sigma))$, whenever the latter is defined; and we let $\mathbf{T}[\![V]\!](\sigma) = \lightning$, otherwise;
2. If T is a component C, then we define $\mathbf{T}[\![C]\!](\sigma) = \mathbf{C}[\![C]\!](\sigma) \in \mathscr{C}_T \subseteq \mathscr{V}$;
3. If T is a definition D, then we define $\mathbf{T}[\![D]\!](\sigma) = \mathbf{D}[\![D]\!](\sigma) \in \mathscr{D} \subseteq \mathscr{V}$;
4. If T is a list $[L]$, then we define $\mathbf{T}[\![[L]]\!](\sigma) = \mathbf{L}[\![L]\!](\sigma) \in \mathscr{V}^{\square}$;
5. If T is $T_0 : T_1$, we define $\mathbf{T}[\![T_0 : T_1]\!](\sigma) = \mathrm{lst}(x_0, x_1 - 1)$, whenever $\mathbf{T}[\![T_i]\!](\sigma) = x_i \in \mathbb{N}$; and we let $\mathbf{T}[\![T_0 : T_1]\!](\sigma) = \lightning$, otherwise;
6. If T is $T[L]$, we define $\mathbf{T}[\![T[L]]\!](\sigma) = x(k)$, whenever $\mathbf{T}[\![T]\!](\sigma) = x \in \mathscr{V}^{\square}$ and $\mathbf{L}[\![L]\!](\sigma) = k \in \mathbb{N}^{\square}$; and we let $\mathbf{T}[\![T[L]]\!](\sigma) = \lightning$, otherwise;
7. If T is a function $F(L)$, we define $\mathbf{T}[\![F(L)]\!](\sigma) = \mathbf{F}[\![F]\!](\mathbf{L}[\![L]\!](\sigma))$, whenever $\mathbf{F}[\![F]\!] : \mathscr{V}^k \longrightarrow \mathscr{V}$, with $k = \mathrm{len}(\mathbf{L}[\![L]\!](\sigma))$, is the semantics of F; and we let $\mathbf{T}[\![F(L)]\!](\sigma) = \lightning$, otherwise.

The semantics of lists is a map $\mathbf{L}[\![-]\!] : \text{Lists} \longrightarrow (\mathscr{V}^{\square} \cup \{\lightning\})^{\Sigma}$, where Lists is the set of parse trees with root L. For a given scope $\sigma \in \Sigma$, we define $\mathbf{S}[\![-]\!](\sigma)$ inductively as follows:

1. If L is ε, then we define $\mathbf{L}[\![\varepsilon]\!](\sigma) = []$;
2. If L is L,T, then we define $\mathbf{L}[\![L,T]\!](\sigma) = [x_1,\ldots,x_{n+1}]$, whenever $\mathbf{L}[\![L]\!](\sigma) = [x_1,\ldots,x_n]$ and $\mathbf{T}[\![T]\!](\sigma) = x_{n+1}$; and we let $\mathbf{L}[\![L,T]\!](\sigma) = \lightning$, otherwise;
3. If L is $L, T_0..T_1$, then we define $\mathbf{L}[\![L, T_0..T_1]\!](\sigma) = [x_1,\ldots,x_{n+k}]$, whenever $\mathbf{L}[\![L]\!](\sigma) = [x_1,\ldots,x_n]$ and $\mathrm{lst}(\mathbf{T}[\![T_0]\!](\sigma), \mathbf{T}[\![T_1]\!](\sigma)) = [x_{n+1},\ldots,x_{n+k}]$; and we let $\mathbf{L}[\![L, T_0..T_1]\!](\sigma) = \lightning$, otherwise.

Since we use predicates in Treo for list comprehension, we define the semantics of predicates as a map $\mathbf{P}[\![-]\!] : \text{Predicates} \longrightarrow (2^{\Sigma})^{\Sigma}$, where Predicates is the set of all parse trees with root P. For a scope $\sigma \in \Sigma$, we define the semantics $\mathbf{P}[\![-]\!](\sigma)$ of a predicate P as the set of all extensions of σ that satisfy P. We define $\mathbf{P}[\![-]\!](\sigma)$ inductively as follows:

1. If P is $V \in T$, then we define

$$\mathbf{P}[\![V \in T]\!](\sigma) = \begin{cases} \{\sigma_i \mid 1 \leq i \leq n\} & \text{if } x \notin \mathrm{dom}(\sigma), \mathbf{T}[\![T]\!](\sigma) = [t_1, \dots, t_n] \\ \{\sigma\} & \text{if } \mathbf{T}[\![V]\!](\sigma) \in \mathbf{T}[\![T]\!](\sigma) \\ \emptyset & \text{otherwise} \end{cases},$$

where $\mathbf{V}[\![V]\!](\sigma) = x$, and $\sigma_i = \sigma\{x \mapsto t_i\}$ is the composition of σ and the primitive scope $\{x \mapsto t_i\}$;

2. If P is $R(L)$, we define $\mathbf{P}[\![R(L)]\!](\sigma) = \{\sigma' \in \Sigma \mid \sigma'\sigma = \sigma', \mathbf{L}[\![L]\!](\sigma') \in \mathbf{R}[\![R]\!]\}$;
3. If P is $\neg P$, we define $\mathbf{P}[\![\neg P]\!](\sigma) = \{\sigma' \in \Sigma \mid \sigma'\sigma = \sigma', \neg\mathbf{P}[\![P]\!](\sigma')\}$;
4. If P is $P_0 \wedge P_1$, we define $\mathbf{P}[\![P_0 \wedge P_1]\!](\sigma) = \mathbf{P}[\![P_0]\!](\sigma) \cap \mathbf{P}[\![P_1]\!](\sigma)$;
5. If P is $P_0 \vee P_1$, we define $\mathbf{P}[\![P_0 \vee P_1]\!](\sigma) = \mathbf{P}[\![P_0]\!](\sigma) \cup \mathbf{P}[\![P_1]\!](\sigma)$;
6. If P is (P), we define $\mathbf{P}[\![(P)]\!](\sigma) = \mathbf{P}[\![P]\!](\sigma)$.

For set and list comprehensions, we can iterate over only a finite subset of scopes $\mathbf{P}[\![P]\!](\sigma)$ of P. We ensure this by restricting the set of scopes to those solutions that are minimal with respect to inclusion of domains. Formally, we write $\min \mathbf{P}[\![P]\!](\sigma)$ for the set of all scopes that are minimal with respect to $\leq$ defined as $\sigma_1 \leq \sigma_2$ iff $\mathrm{dom}(\sigma_1) \subseteq \mathrm{dom}(\sigma_2)$, for all $\sigma_1, \sigma_2 \in \mathbf{P}[\![P]\!](\sigma)$.

The semantics of component instances is a map $\mathbf{C}[\![-]\!] : \text{Components} \longrightarrow (\mathscr{C}_T \cup \{\lightning\})^\Sigma$, where Components is the set of parse trees with root C. Recall that Treo views components as values ($\mathscr{C}_T \subseteq \mathscr{V}$). Given a scope $\sigma \in \Sigma$, we define $\mathbf{C}[\![-]\!](\sigma)$ inductively as follows:

1. If C is a variable V, then we define $\mathbf{C}[\![V]\!](\sigma) = \sigma(x)$, whenever $\mathbf{V}[\![V]\!](\sigma) = x \in \mathrm{dom}(\sigma)$ is defined and $\sigma(x) \in \mathscr{C}_T \subseteq \mathscr{V}$ is a component; and we define $\mathbf{C}[\![V]\!](\sigma) = \lightning$, otherwise;
2. If C is a primitive system A, we define $\mathbf{C}[\![A]\!](\sigma) = \mathbf{A}[\![A]\!]$, where $\mathbf{A}[\![-]\!] : \text{Atoms} \longrightarrow \mathscr{C}_T$ gives the semantics of primitive components;
3. If C is a binary composition C_0C_1, we define $\mathbf{C}[\![C_0C_1]\!](\sigma) = \mathbf{C}[\![C_0]\!](\sigma) \wedge_T \mathbf{C}[\![C_1]\!](\sigma)$, whenever $\mathbf{C}[\![C_i]\!](\sigma) \in \mathscr{C}_T$; and we let $\mathbf{C}[\![C_0C_1]\!](\sigma) = \lightning$, otherwise;
4. If C is a parametrized composition $\{C : P\}$, we define $\mathbf{C}[\![\{C : P\}]\!](\sigma) = \top_T$, whenever $\min \mathbf{P}[\![P]\!](\sigma)$ is empty or infinite; $\mathbf{C}[\![\{C \mid P\}]\!](\sigma) = \mathbf{C}[\![C]\!](\sigma_1) \wedge_T \cdots \wedge_T \mathbf{C}[\![C]\!](\sigma_k)$, whenever $\min \mathbf{P}[\![P]\!](\sigma) = \{\sigma_1, \dots, \sigma_k\} \neq \emptyset$ and $\mathbf{C}[\![C]\!](\sigma_i) \in \mathscr{C}_T$; and we let $\mathbf{C}[\![\{C \mid P\}]\!](\sigma) = \lightning$, otherwise;
5. If C is an instantiation $D\langle L\rangle(U)$, we define $\mathbf{C}[\![D\langle L\rangle(U)]\!](\sigma) = \mathbf{D}[\![D]\!](\sigma)(\mathbf{L}[\![L]\!](\sigma), \mathbf{U}[\![U]\!](\sigma))$, if $\mathbf{D}[\![D]\!](\sigma) \in \mathscr{D}$, $\mathbf{L}[\![L]\!](\sigma) \in \mathscr{V}^\square$ and $\mathbf{U}[\![U]\!](\sigma) \in \mathscr{N}^\square$; and we let $\mathbf{C}[\![D\langle L\rangle(U)]\!](\sigma) = \lightning$, otherwise.

The semantics of component definitions is a map $\mathbf{D}[\![-]\!] : \text{Definitions} \longrightarrow (\mathscr{D} \cup \{\lightning\})^\Sigma$, where Definitions is the set of all parse trees with root D. For a scope $\sigma \in \Sigma$, we define $\mathbf{D}[\![-]\!](\sigma)$ as follows:

1. If D is V, we define $\mathbf{D}[\![V]\!](\sigma) = \sigma(\mathbf{V}[\![V]\!](\sigma)) \in \mathscr{D}$, whenever the latter is defined; and we let $\mathbf{D}[\![V]\!](\sigma) = \lightning$, otherwise;
2. If D is a component $\langle U_0\rangle(U_1)\{C\}$, then for an array of parameter values $t \in \mathscr{V}^\square$ and an array of nodes $q \in \mathscr{N}^\square$, we define $\mathbf{D}[\![\langle U_0\rangle(U_1)\{C\}]\!](\sigma)(t, q)$ as follows: Recall from Section 3 that the number of parameters and nodes can implicitly define variables. Suppose that there exists a unique 'index-defining' scope $\sigma' \in \Sigma$ such that for $m = \mathrm{len}(t)$ and $n = \mathrm{len}(q)$. Then we have
    (a) $\mathbf{U}[\![U_0]\!](\sigma') = [s_1, \dots, s_m] \neq \lightning$ satisfies $s_i \simeq t(i)$, for all $1 \leq i \leq m$;
    (b) $\mathbf{U}[\![U_1]\!](\sigma') = [p_1, \cdots, p_n] \neq \lightning$ satisfies $p_i \simeq q(i)$, for all $1 \leq i \leq n$;

$$\begin{tikzcd}
 & P_1 \arrow[dr] & P_2 & P_3 \arrow[dl] & \\
x \arrow[ur] & & y \arrow[u] & & z \arrow[ul]
\end{tikzcd}$$

(a) $P_1 \cdot P_2 \cdot P_3$

$$\begin{tikzcd}
 & & P_1' \arrow[dr] & & P_2' & & P_3' \arrow[dl] & & \\
x \arrow[d] & x_1 \arrow[ur] & & y_1 \arrow[dr] & y_2 \arrow[u] & y_3 \arrow[dl] & & z_3 \arrow[ul] & z \arrow[d] \\
N_x \arrow[ur] & & & & N_y \arrow[u] & & & & N_z \arrow[ul]
\end{tikzcd}$$

(b) $\text{surg}(P_1 \cdot P_2 \cdot P_3)$

Figure 2: Surgery on an I/O-component to remove mixed nodes.

(c) $\text{flatten}([s_1,\dots s_m, p_1,\dots p_n]) \in \mathscr{N}^\square$ has no duplicates;

(d) $\text{dom}(\sigma') \subseteq \mathscr{N}$ is minimal such that properties (a)-(c) are satisfied.

We evaluate the body C of the component definition to the component $\mathbf{C}[\![C]\!](\sigma\sigma')$, where $\sigma\sigma'$ is the composition of σ and σ'. Define the map $r : \text{supp}(\mathbf{C}[\![C]\!](\sigma\sigma')) \longrightarrow \mathscr{N}$ as

$$r(x) = \begin{cases} t_i(k_1)\cdots(k_l) & \text{if } x = s_i(k_1)\cdots(k_l) \\ q_i(k_1)\cdots(k_l) & \text{if } x = p_i(k_1)\cdots(k_l) \\ v \text{ fresh} & \text{otherwise} \end{cases}$$

Observe that r is well-defined, because $\text{flatten}([s_1,\dots s_m, p_1,\dots p_n]) \in \mathscr{N}^\square$ has no duplicates. Note that r is finite, since we assume that $\text{supp}(\mathbf{C}[\![C]\!](\sigma\sigma'))$ is finite. We define $\mathbf{D}[\![\langle U_0\rangle(U_1)\{C\}]\!](\sigma)(t,q)$ as the simultaneous substitutions $\mathbf{C}[\![C]\!](\sigma\sigma')[r(x)/x : x \in \text{dom}(f)]$. If such 'index-defining' scope σ' does not exists or is not unique, then we simply define $\mathbf{D}[\![\langle U_0\rangle(U_1)\{C\}]\!](\sigma)(t,q) = \lightning$.

We define the semantics of files as a map $\mathbf{K}[\![-]\!] : \text{Files} \longrightarrow \Sigma \cup \{\lightning\}$, where Files is the set of parse trees with root K. Let $\mathbf{I}[\![-]\!] : \text{Imports} \longrightarrow \Sigma$ be the semantics of imports. For a scope $\sigma \in \Sigma$, we define $\mathbf{K}[\![-]\!](\sigma)$ inductively as follows:

1. If K is I, we define $\mathbf{K}[\![I]\!](\sigma) = \mathbf{I}[\![I]\!]$;
2. If K is KND, then we define $\mathbf{K}[\![KND]\!](\sigma) = \sigma_0\{x \mapsto c\}$, where $\sigma_0 = \mathbf{K}[\![K]\!](\sigma) \neq \lightning$, $x = \mathbf{N}[\![N]\!]$ is the semantics of names and $c = \mathbf{D}[\![D]\!](\sigma_0)$; and we let $\mathbf{K}[\![KND]\!] = \lightning$, otherwise.

6 Input/output nodes

As mentioned in Section 3, nodes of primitive component definitions require input/output annotations. Treo regards such port type annotations as attributes of the primitive component. For a component type T, we model the input nodes and output nodes of its instances via two maps $I, O : \mathscr{C}_T \longrightarrow 2^{\mathscr{N}}$ satisfying $\text{supp}(C) = I(C) \cup O(C)$, for all $C \in \mathscr{C}_T$. If $x \in I(C) \cap O(C)$, then we call x a *mixed* node.

Example 6.1 (Mixed nodes)**.** Recall the I/O component type from Example 4.3. Let $P_1 = (C_1, \{x\}, \{y\})$, $P_2 = (C_2, \{y\}, \emptyset)$, and $P_3 = (C_3, \{z\}, \{y\})$ be three primitive I/O components. Figure 2(a) shows a graphical representation of composition of P_1, P_2, and P_3. In this figure, an arrow from a node a to a component P indicates that a is an input node of P. An arrow from a component P to a node a indicates that a is an output node of P. Node y is an output node of P_1 and P_3, and it is an input node of P_2. Thus, y is a mixed node in the composition $P_1 \cdot P_2 \cdot P_3$, where $\cdot$ is sequential composition of I/O components. $\triangle$

Most component types that distinguish input and output nodes assume *well-formed* compositions: each shared node in a composition is an output of one component and an input of the other.

Definition 6.1 (Well-formedness)**.** A composition $C_1 \wedge_T \cdots \wedge_T C_n$, with $n \geq 0$, is well-formed if and only if $|\{i \in \{1,\dots,n\} \mid x \in I(C_i)\}| \leq 1$ and $|\{i \in \{1,\dots,n\} \mid x \in O(C_i)\}| \leq 1$, for all $x \in \mathscr{N}$.

For well-formed compositions, the behavior of the composition naturally corresponds to the composition of Reo connectors. However, specification of complex components as well-formed compositions is quite cumbersome, because it requires explicit verbose expression of the 'merge-replicate' behavior of every Reo node in terms of a suitable number of binary mergers and replicators. Reo nodes abstract from such detail and yield more concise specifications. Like Reo, Treo does not impose any restriction on the nodes of constituent components in a composition. Indeed, the denotational semantics of components $\mathbf{C}[\![-]\!]$ in Section 5 unconditionally computes the composition. To define the semantics of $\mathbf{C}[\![-]\!]$ for a component type T where $\wedge_T$ requires well-formedness, parsing a (non-well-formed) Treo composition needs the degree (i.e., the number of coincident input- and output channel ends) of each node to correctly express the 'merge-replicate' semantics of that node. The degree of every node used in a definition can be known only at the end of that definition. The Treo compiler could accomplish this via two-pass parsing.

Alternatively, Treo can delay applying composition $\wedge_T$ in T until parsing completes, Treo accomplishes this by interpreting a Treo program over the I/O-component type IO_T, as defined in Example 4.3, wherein compositions consist of lists of primitive components. First, Treo wraps each primitive component $C \in \mathscr{C}_T$ within a primitive I/O-component $(C, I(C), O(C)) \in \mathscr{P}_T$. Using Section 5, Treo parses the Treo program over the component type IO_T as usual, and obtain a single I/O-component $P_1 \cdots P_n \in \mathrm{IO}_T$.

However, the resulting composition $P_1 \cdots P_n$ may not be well-formed. Therefore, the Treo compiler applies some surgery on $P_1 \cdots P_n$ to ensure a well-formed composition. This surgery consists of splitting all shared nodes in X, and reconnecting them by inserting a *node component*. We model these node components (over component type T) as a map node : $(2^{\mathscr{N}})^2 \times \mathscr{N} \longrightarrow \mathscr{C}_T$. For sets of names $I, O \subseteq \mathscr{N}$ and a default name $x \in \mathscr{N}$, the component $\mathrm{node}(I, O, x) \in \mathscr{C}_T$ has input nodes I (or $\{x\}$, if I is empty) and output nodes O (or $\{x\}$, if O is empty).

Definition 6.2 (Surgery)**.** The surgery map surg : $\mathrm{IO}_T \longrightarrow \mathrm{IO}_T$ is defined as $\mathrm{surg}(P_1 \cdots P_n) = P_1' \cdots P_n' \cdot \prod_{x \in \mathrm{supp}(P_1 \cdots P_n)} N_x$, where $P_i' = P_i[x_i/x : x \in \mathrm{supp}(P_i)]$, for all $1 \leq i \leq n$, and $N_x = (\mathrm{node}(I_x, O_x, x), I_x, O_x)$, with $I_x = \{x_i \mid x \in O(P_i)\}$ and $O_x = \{x_i \mid x \in I(P_i)\}$. The composition $\prod$ is ordered arbitrarily.

Intuitively, the surgery map takes a possibly non-well-formed composition and produces a well-formed composition by inserting node components. Although initially, multiple components may produce output at the same node. After applying the surgery map, these components offer data for the same node component via different 'ports'.

Example 6.2 (Surgery)**.** Figure 2(b) shows the result after applying the surgery map to the I/O-component $P_1 \cdot P_2 \cdot P_3$ from Example 6.1. The surgery map consists of two parts. First, the surgery map splits every node $a \in \{x, y, z\}$ by renaming a to a_i in P_i, for every $1 \leq i \leq n$. Second, the surgery map inserts at every node $a \in \{x, y, z\}$ a node component N_a. Clearly, $\mathrm{surg}(P_1 \cdot P_2 \cdot P_3)$ is a well-formed composition. △

7 Related work

The Treo syntax offers a textual representation for the graphical Reo language [3, 4]. We propose Treo as a syntax for Reo that (1) provides support for parametrized-, recursive-, iterative-, and conditional constructions, and (2) implements basic design principles of Reo more closely than existing languages and reflects its declarative nature. The graphical Reo editor implemented as an Eclipse plugin [1] does not support parametrized-, recursive-, iterative-, or conditional constructions. RSL (with CARML for primitives) [5, 19] is imperative, while Reo is declarative. FOCAML [12], supports only constraint automata [6], while channel types in Reo are user-defined.

Since Treo leaves the syntax for primitive subsystems (i.e., component types) as user-defined, Treo is a "meta-language" that specifies compositional construction of complex structures (using the common core language defined in this paper) out of primitives defined in its arbitrary, user-defined sub-languages. As such, Treo is not directly comparable to any existing language. We can, however, compare the component-based system composition of Treo with the system composition of an existing language.

Treo components are similar to proctype declarations in Promela, the input language for the SPIN model checker developed by Holzmann [11]. However, the focus of Promela is on imperative definitions of processes, while Treo is designed for declarative composition of processes.

SysML is a graphical language for specification of systems [9]. SysML offers 9 types of diagrams, including activity diagrams and block diagrams. Each diagram provides a different view on the same system [20]. Diagram types in SysML are comparable to component types in Treo. The main difference, however, is that Treo requires a well-defined composition operator, using which it allows construction of more complex components, while diagram composition is much less prominent in SysML.

A component model is a programming paradigm based on components and their composition. Our Treo language can be viewed as one such component model with a concrete syntax. Over the past decades, many different component models have been proposed. For example, CORBA [21] is a component model that is flat in the sense that every CORBA component is viewed as a black box, i.e., it does not support composite components. Fractal [7] is an example of a component model that is hierarchical, which means a component can be a composition of subcomponents. Concrete instances of Fractal consist of libraries (API's) for a variety of programming languages, such as Java, C, and OMG IDL [7]. Treo components and Fractal component differ with respect to interaction: Treo components interact via shared names, while Fractal component interact via explicit bindings.

8 Conclusion

We propose Treo as a textual syntax for Reo connectors that allows user-defined component types, and incorporates Reo's predefined node behavior. These features are not present in any of the existing alternative languages for Reo. We provided an abstract syntax for Treo and its denotational semantics based on this abstract syntax. We identify three possible directions for future work.

First, since our semantics disallows recursion, a component in Treo is a composition of finitely many subsystems. Consequently, we cannot express the construction of an unbounded buffer B_ω from a buffer with capacity one, B_1. It seems, however, possible to use simulation and recursion to define B_ω in terms of B_1: buffer B_ω is the smallest (with respect to simulation) component that simulates B_1 and is stable under sequential composition with B_1. These assumptions readily imply that B_ω simulates a buffer of arbitrary length. Semantically, the unbounded buffer would then be defined as a least fixed point of a certain operator on components. An extension of Treo semantics that allows such fixed point definitions would provide a powerful tool to define complex 'dynamic' components.

Second, the current semantics in Section 5 does not support components with an identity. If we instantiate a component definition twice with the same parameters, we obtain two instances of the same component. Ideally, component instantiation should return a component instance with a fresh identity. Allowing components with identities in Treo enables programmers to design systems more realistically.

Finally, a component type T from Definition 4.1 consists of a single composition operator $\wedge_T$. Generally, a component type consists of multiple composition operators (each with it own arity). For example, we may need both sequential composition as well as parallel composition. Extending Treo with (a variable number of) composition operators would enable users to model virtually all types of components.

References

[1] *Extensible Coordination Tools ECT*. `http://reo.project.cwi.nl`. Accessed: 2018-03-23.

[2] *ReoLanguage GitHub repository*. `https://github.com/ReoLanguage/Reo`. Accessed: 2018-03-23.

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

[4] Farhad Arbab (2011): *Puff, The Magic Protocol*. In: Formal Modeling: Actors, Open Systems, Biological Systems, LNCS 7000, Springer, pp. 169–206.

[5] Christel Baier, Tobias Blechmann, Joachim Klein & Sascha Klüppelholz (2009): *A Uniform Framework for Modeling and Verifying Components and Connectors*. In: Proc. of COORDINATION, pp. 247–267.

[6] Christel Baier, Marjan Sirjani, Farhad Arbab & Jan Rutten (2006): *Modeling component connectors in Reo by constraint automata*. Sci. Comput. Programming 61(2), pp. 75–113.

[7] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma & Jean-Bernard Stefani (2006): *The FRACTAL component model and its support in Java*. Softw., Pract. Exper. 36(11-12), pp. 1257–1284.

[8] Philipp Chrszon, Clemens Dubslaff, Christel Baier, Joachim Klein & Sascha Klüppelholz (2016): *Modeling Role-Based Systems with Exogenous Coordination*. In: Theory and Practice of Formal Methods - Essays Dedicated to Frank de Boer on the Occasion of His 60th Birthday, pp. 122–139.

[9] Sanford Friedenthal, Alan Moore & Rick Steiner (2014): *A practical guide to SysML: the systems modeling language*. Morgan Kaufmann.

[10] Charles Antony Richard Hoare (1978): *Communicating sequential processes*. In: The origin of concurrent programming, Springer, pp. 413–443.

[11] Gerard J. Holzmann (2004): *The SPIN Model Checker - primer and reference manual*. Addison-Wesley.

[12] S.-S. T. Q. Jongmans (2016): *Automata-theoretic protocol programming*. Ph.D. thesis, Leiden University.

[13] Sung-Shik T. Q. Jongmans & Farhad Arbab (2012): *Overview of Thirty Semantic Formalisms for Reo*. Sci. Ann. Comp. Sci. 22(1), pp. 201–251.

[14] Sung-Shik T. Q. Jongmans & Farhad Arbab (2015): *Take Command of Your Constraints!* In: Proc. of COORDINATION, pp. 117–132.

[15] Sung-Shik T. Q. Jongmans & Farhad Arbab (2016): *Data optimizations for constraint automata*. Logical Methods in Computer Science 12(3).

[16] Sung-Shik T. Q. Jongmans & Farhad Arbab (2016): *PrDK: Protocol Programming with Automata*. In: Proc. of TACAS, pp. 547–552.

[17] Sung-Shik T. Q. Jongmans & Farhad Arbab (2017): *Centralized coordination vs. partially-distributed coordination with Reo and constraint automata*. Science of Computer Programming.

[18] Sung-Shik T. Q. Jongmans, Sean Halle & Farhad Arbab (2014): *Automata-Based Optimization of Interaction Protocols for Scalable Multicore Platforms*. In: Proc. of COORDINATION, pp. 65–82.

[19] Sascha Klüppelholz (2012): *Verification of Branching-Time and Alternating-Time Properties for Exogenous Coordination Models*. Ph.D. thesis, Dresden University of Technology.

[20] Philippe B Kruchten (1995): *The 4+1 view model of architecture*. IEEE software 12(6), pp. 42–50.

[21] OMG (2006): *CORBA Component Model, v4.0, OMG document formal/06-04-01*. Available at `http://www.omg.org/spec/CCM`. Visited 11-13-2017.

[22] Terence Parr (2013): *The definitive ANTLR 4 reference*. Pragmatic Bookshelf.

[23] David A Schmidt (1997): *Denotational Semantics: A Methodology for Language Development*.

Appendix

Listing 1: Concrete ANTLR4 syntax of Treo language (Treo.g4).

```
grammar Treo;
file      : sec? imp* assg* EOF;
sec       : 'section' name ';' ;
imp       : 'import' name ';' ;
assg      : ID defn ;
defn      : var | params? nodes comp ;
comp      : defn vals? args | var | '{' atom+ '}' | '{' comp* ('|' pred)? '}'
          | 'for' '(' ID 'in' list ')' comp
          | 'if' '(' pred ')' comp ('else' '(' pred ')' comp)* ('else' comp)? ;
atom      : STRING ; /* Example syntax for primitive components */
pred      : 'true' | 'false' | '(' pred ')' | var 'in' list
          | term op=('<=' | '<' | '>=' | '>' | '=' | '!=') term
          | var | 'forall' ID 'in' list ':' pred | 'exists' ID 'in' list ':' pred
          | 'not' pred | pred ('and'|',') pred | pred 'or' pred | pred 'implies' pred ;
term      : var | NAT | BOOL | STRING | DEC | comp | defn | list | 'len(' term ')'
          | '(' term ')' | <assoc=right> term list | <assoc=right> term '^' term
          | '-' term | term op=('*' | '/' | '%' | '+' | '-') term ;
vals      : '<' '>' | '<' term (',' term)* '>' ;
list      : '[' ']' | '[' item (',' item)* ']' ;
item      : term | term '..' term | term ':' term ;
args      : '(' ')' | '(' var (',' var)* ')' ;
params    : '<' '>' | '<' var (',' var)* '>' ;
nodes     : '(' ')' | '(' node (',' node)* ')' ;
node      : var (io=('?' | '!' | ':') ID?)? ;
var       : name list* ;
name      : (ID '.')* ID ;
NAT       : ('0' | [1-9][0-9]*) ;
DEC       : ('0' | [1-9][0-9]*) '.' [0-9]+ ;
BOOL      : 'true' | 'false' ;
ID        : [a-zA-Z_][a-zA-Z0-9_]*;
STRING    : '\"' .*? '\"' ;
SPACES    : [ \t\r\n]+ -> skip ;
SL_COMM   : '//' .*? ('\n'|EOF) -> skip ;
ML_COMM   : '/*' .*? '*/' -> skip ;
```