Part II

Compilation

Chapter 4

Protocol Syntax

The protocol specifications introduced in Part I offer explicit information on the interactions of tasks in a given application, and we can use this information for scheduling. However, scheduling operates on concrete tasks, rather than their formal specifications. Therefore, we now study compilation of these protocol specification, which allows us to develop scheduling techniques for the generated executable.

History shows that the number of compilers for Reo is proportional to the number of PhD students that worked on Reo, and each tool uses its own input language. For example, the Dreams framework [PCdVA12] uses a graphical editor in Eclipse, The Lykos compiler [Jon16] uses FOCAML, and Vereofy uses the RSL [Klü12]. The current chapter aims to unify these toolsets by proposing a syntax for Reo protocols called Treo[1]. A key feature of the Treo language is that it allows tool developers to extend Treo with their own semantics of primitive components, which is an essential feature in view of plethora of semantics models for Reo [JA12].

Many tools for Reo have been implemented as a collection of Eclipse plugins called the ECT [ECT]. The main plugin in this tool set consists of a 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 of connector graphs. 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 Reactive Module Language (CARML) [BBKK09, Klü12]. 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 and/or complete systems. In contrast to the declarative nature of the graphical syntax of Reo, RSL is imperative.

[1]The work in this chapter is based on [DA18b]

Jongmans developed the First-Order Constraint Automata with Memory Language (FOCAML) [Jon16], 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 has poor support for its primary design principle: Reo channels are user-defined, not tied to any specific formalism to express its semantics, and compose via shared nodes with predefined merge-replicate behavior. Although FOCAML components are user-defined, FOCAML requires them to be of the same predefined semantic sort (i.e., constraint automata with memory [BSAR06]). The primary concept of Reo nodes does not exist in FOCAML, which 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 [JHA14, JA15, JA16b, JA16a, JA18]. 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 chapter, we introduce Treo, a declarative textual language for component-based specification of Reo connectors with user-defined semantic sorts and predefined node behavior. We describe the structure of a Treo file by means of an abstract syntax (Section 4.1). In Listing 4.1, we provide a concrete syntax of Treo as an ANTLR4 grammar [Par13]. In on-going work, we currently use Treo to compile Reo into target languages such as Java, Promela, and Maude [Reo]. The construction of the Treo compiler is based on the theory of stream constraints [DA18a].

In order to preserve the agnosticism of Reo regarding the concrete semantics of its primitives, Treo uses the notion of user-defined *semantic sorts*. A user-defined semantic sort consist of a set of component instances together with a composition operator $\wedge$, a substitution operator $[\,/\,]$, and a trivial component $\top$ (Section 4.2). The composition operator defines the behavior of composite components as a composition of its operands. The substitution operator binds nodes in the interface or passes values to parameters.

For a given semantic sort, we define the meaning of abstract Treo programs (Section 4.3). 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 [BCL$^+$06, Figure 2].

A given semantics sort 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 components on their outputs, because, complying with the semantics of Reo, the compiler inserts special *node components* to ensure well-formed compositions (Section 4.4).

We conclude by discussing related work (Section 4.5), and pointing out future work (Section 4.6).

4.1 Treo syntax

We now present a textual representation for the graphical Reo connectors in Section 2.1.2. Table 4.1 shows the abstract syntax of Treo.

$$
\begin{array}{lcl@{\qquad}lcl}
K & ::= & I \mid KND & D & ::= & V \mid \langle U_0\rangle(U_1)\{C\} \\
L & ::= & \epsilon \mid L,T \mid L,T_0..T_1 & C & ::= & V \mid A \mid C_0C_1 \mid \{C \mid P\} \mid D\langle L\rangle(U) \\
U & ::= & \epsilon \mid U,V & T & ::= & V \mid C \mid D \mid [L] \mid T_0 : T_1 \mid T[L] \mid F(L) \\
V & ::= & N \mid V[L] & P & ::= & V \in T \mid R(L) \mid \neg P \mid P_0 \wedge P_1 \mid P_0 \vee P_1 \mid (P)
\end{array}
$$

Table 4.1: 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 bold vertical bar in $\{C \mid P\}$ is just text.

We introduce the symbols in the abstract syntax by identifying them in some concrete examples. These concrete examples are Treo programs that can be parsed using the concrete Treo syntax shown in Listing 4.1.

```
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;
```

Listing 4.1: Concrete ANTLR4 syntax of Treo (Treo.g4).

Consider the following Treo file (K in Table 4.1) representing the 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 `sync` 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 hence hidden.

Constructed from existing components, `alternator2` is a *composite* component ($C_0 C_1$). 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* [BSAR06]. Constraint automata constitute a popular semantic sort for specification of Reo component types, and forms the basis of the Lykos compiler [Jon16]. However, constraint automata are not the *de facto* standard: the literature offers more than thirty different semantic sorts for specification of Reo components [JA12], such as the coloring semantics and timed data stream semantics. To accommodate the generality that disparate semantics allow, Treo features *user-defined semantic sorts*, which means that the syntax for 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 2.3). However, the composition operators in most Reo semantics do not automatically produce the correct behavior for such nodes (e.g., see [BSAR06, 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 than one input or output channel end as a *node component.* The interface of this node component is determined by its *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 a composition, and subsequently inserts the correct node components 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 composition.

The ellipses in Figure 2.3 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 [ECT], 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 (4.2)). For iteration, we write `{ ... | i in [2..k] }` using set-comprehension ($\{C \mid P\}$ in Table 4.1).

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}
}
```

Here, the value of `k` is defined by the size of `a[1:k]`, and we use set-comprehension `{ ... | k > 1 }` for conditional construction, as well. 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 4.3 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 [Jon16, 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 consists of Treo component definitions. In the body of the `team` component, these definitions are instantiated via `engine[i](inp,best[i])`. In RSL [BBKK09, Klü12] and FOCAML [Jon16], it is impossible to pass a component as a parameter, which makes these languages less expressive than Treo.

We may view the `team` component as an example of role-oriented programming [CDB$^+$16]. 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 also component instances as values. By passing a single component instance as a parameter to multiple components, this feature allows *component (instance) sharing* (cf., [BCL$^+$06, Figure 2]). Hence, it is straightforward to implement a chess match, wherein a single instance of a chess engine plays against itself.

4.2 Semantic sorts

As noted in Section 4.1, Reo channels can be defined in many different semantic formalisms [JA12], such as the constraint automaton semantics, the coloring semantics, or the timed data stream semantics. Although each sort of Reo semantics has its unique properties, each of them can be used to define a collection of composable components with parameters and nodes, which we call a *semantic sort*:

Definition 4.2.1 (Semantic sort)**.** A semantic sort over a set of names $\mathcal{N}$ with values from $\mathcal{V}$ is a tuple $(\mathcal{C}, \wedge, [\,/\,], \top)$ that consists of a set of components $\mathcal{C}$, a composition operator $\wedge : \mathcal{C} \times \mathcal{C} \longrightarrow \mathcal{C}$, a substitution operator $[\,/\,] : \mathcal{C} \times (\mathcal{N} \cup \mathcal{V}) \times \mathcal{N} \longrightarrow \mathcal{C}$, and a trivial component $\top \in \mathcal{C}$.

We assume that the set of names and the set of values are disjoint, i.e., $\mathcal{N} \cap \mathcal{V} = \emptyset$. For convenience, we write $C \wedge C'$ for $\wedge(C, C')$, and $C[y/x]$ for $[\,/\,](C, y, x)$. For any semantic sort T, we write $\mathcal{C}_T$ 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 $C \in \mathcal{C}_T$ be a component and $x \in \mathcal{N}$ a name.

For a name $y \in \mathcal{N}$, the construct $C[y/x]_T$ renames every occurrence of name x in C to y. For a value $y \in \mathcal{V}$, the construct $C[y/x]_T$ instantiates (parameter) x in C to y.(See Example 4.2.3 for an example of the distinction between renaming and instantiations.)

A semantic sort T implicitly defines an interface for each component $C \in \mathcal{C}$ via the map $\mathrm{supp} : \mathcal{C}_T \longrightarrow 2^{\mathcal{N}}$ defined as $\mathrm{supp}(C) = \{x \in \mathcal{N} \mid C[y/x]_T \neq C$, for some name $y \in \mathcal{N}\}$. 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 4.2.1 (Systems of differential equations)**.** The set ODE of systems of ordinary differential equations with variables from $\mathcal{N}$ and values $\mathcal{V} = \{v : \mathbb{R} \longrightarrow \mathbb{R}\}$ constitute a semantic sort. 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 semantic sort, we can define continuous systems in Treo. $\Diamond$

Example 4.2.2 (Process calculi)**.** Consider the process calculus CSP, proposed by Hoare [Hoa78]. The set CSP of all such process algebraic terms comprises a semantic sort. Each process can participate in a number of events, which we can interpret as names from a given set $\mathcal{N}$. We model the composition of CSP processes P and Q by means of the interface parallel operator $P \;|[X]|\; Q$, where $X \subseteq \mathcal{N}$ 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. $\Diamond$

Example 4.2.3 (I/O-components)**.** Let T be a semantic sort over $\mathcal{N}$ and $\mathcal{V}$. We define the I/O-component sort IO_T over T using the notion of a primitive I/O-component of sort T.

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

$$P[y/x] \;=\; \begin{cases} (P - \{x\}) \cup \{y\} & \text{if } x \in P \text{ and } y \in \mathcal{N} \\ P - \{x\} & \text{if } x \in P \text{ and } y \in \mathcal{V} \\ P & \text{otherwise} \end{cases} \tag{4.1}$$

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 $x \in \mathcal{N}$ and $y \in \mathcal{N} \cup \mathcal{V}$. We denote the set of primitive I/O-components over T as $\mathcal{P}_T$.

An *I/O-component* of sort T is a sequence $P_1 \cdots P_n \in \mathcal{P}_T^*$, with $n \geq 0$, of primitive I/O-components of sort 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, $\mathrm{IO}_T = (\mathcal{P}_T^*, \cdot, [\,/\,], \epsilon)$ is a semantic sort. $\Diamond$

4.3 Denotational semantics

We define the denotational semantics of the Treo language over a fixed, but arbitrary, semantic sort T. The main purpose of this denotational semantics is to provide a clear abstract structure that guides the implementation of Treo parsers. The syntax to which this denotational semantics applies is the abstract syntax in Table 4.1. The general structure of our denotational semantics is quite standard, and adheres to Schmidt's notation [Sch86].

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 $X^\Box$ such that both $X \subseteq X^\Box$ and $[x_0, \dots, x_{n-1}] \in X^\Box$, if $n \geq 0$ and $x_i \in X^\Box$ for all $0 \leq i < n$. For example, the set $\mathbb{N}^\Box$ of ragged arrays over integers contains all natural numbers from $\mathbb{N}$ as 'atomic' arrays, as well as the array $[37, [], [[2, [55], 3]]] \in \mathbb{N}^\Box$. Every ragged array has a length, which can be computed via the map $\text{len} : X^\Box \longrightarrow \mathbb{N}$ defined inductively as $\text{len}(x) = 0$, if $x \in X$, and $\text{len}([x_0, \dots, x_{n-1}]) = n$, otherwise. If $x = [x_0, \dots, x_{n-1}] \in X^\Box$ 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 $\mathbb{N}^\Box$ by defining $x([i_0, \dots, i_n])$ as

$$\begin{cases} x(i_0)([i_1, \dots, i_n]) & \text{if } i_0 \in \mathbb{N} \\ [x(i_{00})([i_1, \dots, i_n]), \dots, x(i_{0m})([i_1, \dots, i_n])] & \text{if } i_0 = [i_{00}, \dots, i_{0m}] \end{cases}, \tag{4.2}$$

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

Suppose that semantic sort 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 $C \in \mathcal{C}_T$, its support $\text{supp}(C) \subseteq \mathcal{N}$ is finite. Since Treo views components as values, we assume the inclusion $\mathcal{C}_T \subseteq \mathcal{V}$.

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 $\text{lst} : \mathbb{N}^2 \longrightarrow \mathbb{N}^\Box$ that constructs from a pair $(i, j) \in \mathbb{N}^2$ of integers a finite ordered list $[i, i+1, \dots, j]$ in $\mathbb{N}^\Box$.

Recall from Section 4.1 that a component accepts an arbitrary but finite number of parameters and nodes. Therefore, we define a component definition as a map $D : \mathcal{V}^\Box \times \mathcal{N}^\Box \longrightarrow \mathcal{C}_T \cup \{\lightning\}$ that takes an array of parameter values from $\mathcal{V}^\Box$ and an array of nodes from $\mathcal{N}^\Box$ and returns a component or an *error* $\lightning$. Let $\mathcal{D} = (\mathcal{C}_T \cup \{\lightning\})^{\mathcal{V}^\Box \times \mathcal{N}^\Box}$ be the set of all definitions. As mentioned earlier, Treo also allows definitions as values, which amounts to the inclusion $\mathcal{D} \subseteq \mathcal{V}$.[2]

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

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 $x \in \mathrm{dom}(\sigma) \cup \mathrm{dom}(\sigma')$ 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 $\mathbf{N}[\![-]\!] : \text{Names} \longrightarrow \mathcal{N}$ be the semantics of names. We define the semantics of variables as a map $\mathbf{V}[\![-]\!] :$ Variables $\longrightarrow (\mathcal{N}^\square \cup \{\lightning\})^\Sigma$, 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. $\mathbf{V}[\![N]\!](\sigma) = \mathbf{N}[\![N]\!]$;

2. $\mathbf{V}[\![V[L]]\!](\sigma) = \begin{cases} x(k) & \text{if } \mathbf{V}[\![V]\!](\sigma) = x \in \mathcal{N}^\square \text{ and } \mathbf{L}[\![L]\!](\sigma) = k \in \mathbb{N}^\square \\ \lightning & \text{otherwise} \end{cases}.$

Since $\mathcal{N}$ is closed under taking subscripts, we can define $n(i) = n_i$, for all $n \in \mathcal{N}$ and $i \in \mathbb{N}$, which ensures that $x(k) \in \mathcal{N}^\square$ is always defined.

The semantics of arguments is a map $\mathbf{U}[\![-]\!]$: Arguments $\longrightarrow (\mathcal{N}^\square \cup \{\lightning\})^\Sigma$, 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. $\mathbf{U}[\![\epsilon]\!](\sigma) = []$;

2. $\mathbf{U}[\![U, V]\!](\sigma) = \begin{cases} [x_1, \ldots, x_{n+1}] & \text{if } \mathbf{U}[\![U]\!](\sigma) = [x_1, \ldots, x_n] \text{ and } \mathbf{V}[\![V]\!](\sigma) = x_{n+1} \\ \lightning & \text{otherwise} \end{cases}.$

Let Functions be the set of parse trees with root F, and let $\mathbf{F}[\![-]\!]$: Functions $\longrightarrow \{\mathcal{V}^k \longrightarrow \mathcal{V} \mid k \in \mathbb{N}\}$ be the semantics of functions. The semantics of terms is a map $\mathbf{T}[\![-]\!]$: Terms $\longrightarrow (\mathcal{V}^\square \cup \{\lightning\})^\Sigma$, 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. $\mathbf{T}[\![V]\!](\sigma) = \begin{cases} \sigma(\mathbf{V}[\![V]\!](\sigma)) & \text{if defined} \\ \lightning & \text{otherwise} \end{cases};$

2. $\mathbf{T}[\![C]\!](\sigma) = \mathbf{C}[\![C]\!](\sigma)$, which is well-defined since $\mathcal{C}_T \subseteq \mathcal{V}$;

3. $\mathbf{T}[\![D]\!](\sigma) = \mathbf{D}[\![D]\!](\sigma)$, which is well-defined since $\mathcal{D} \subseteq \mathcal{V}$;

4. $\mathbf{T}[\![[L]]\!](\sigma) = \mathbf{L}[\![L]\!](\sigma)$;

5. $\mathbf{T}[\![T_0 : T_1]\!](\sigma) = \begin{cases} \mathrm{lst}(x_0, x_1 - 1) & \text{if } \mathbf{T}[\![T_i]\!](\sigma) = x_i \in \mathbb{N} \text{ for } i \in \{0, 1\} \\ \lightning & \text{otherwise} \end{cases};$

6. $\mathbf{T}[\![T[L]]\!](\sigma) = \begin{cases} x(k) & \text{if } \mathbf{T}[\![T]\!](\sigma) = x \in \mathcal{V}^\square \text{ and } \mathbf{L}[\![L]\!](\sigma) = k \in \mathbb{N}^\square \\ \lightning & \text{otherwise} \end{cases};$

7. $\mathbf{T}[\![F(L)]\!](\sigma) = \begin{cases} \mathbf{F}[\![F]\!](\mathbf{L}[\![L]\!](\sigma)) & \text{if } \mathbf{F}[\![F]\!] : \mathcal{V}^k \longrightarrow \mathcal{V} \text{ and } \operatorname{len}(\mathbf{L}[\![L]\!](\sigma)) = k \\ ⚡ & \text{otherwise} \end{cases}.$

The semantics of lists is a map $\mathbf{L}[\![-]\!] : \text{Lists} \longrightarrow (\mathcal{V}^\Box \cup \{⚡\})^\Sigma$, 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. $\mathbf{L}[\![\epsilon]\!](\sigma) = []$;

2. $\mathbf{L}[\![L, T]\!](\sigma) = \begin{cases} [x_1, \ldots, x_{n+1}] & \text{if } \mathbf{L}[\![L]\!](\sigma) = [x_1, \ldots, x_n] \in \mathcal{V}^\Box \\ & \text{and } \mathbf{T}[\![T]\!](\sigma) = x_{n+1} \in \mathcal{V} \\ ⚡ & \text{otherwise} \end{cases};$

3. $\mathbf{L}[\![L, T_0..T_1]\!](\sigma) = \begin{cases} [x_1, \ldots, x_{n+k}] & \text{if } \mathbf{L}[\![L]\!](\sigma) = [x_1, \ldots, x_n] \in \mathcal{V}^\Box, \\ & \mathbf{T}[\![T_i]\!](\sigma) = a_i \in \mathcal{V}, \text{for } i \in \{0, 1\}, \\ & \text{and } \operatorname{lst}(a_0, a_1) = [x_{n+1}, \ldots, x_{n+k}] \\ ⚡ & \text{otherwise} \end{cases}.$

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. $\mathbf{P}[\![V \in T]\!](\sigma) = \begin{cases} \{\sigma\{x \mapsto t_i\} \mid 1 \leq i \leq n\} & \text{if } \mathbf{V}[\![V]\!](\sigma) = x \notin \operatorname{dom}(\sigma), \\ & \text{and } \mathbf{T}[\![T]\!](\sigma) = [t_1, \ldots, t_n] \\ \{\sigma\} & \text{if } \mathbf{T}[\![V]\!](\sigma) \in \mathbf{T}[\![T]\!](\sigma) \\ \emptyset & \text{otherwise} \end{cases},$

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 $\operatorname{dom}(\sigma_1) \subseteq \operatorname{dom}(\sigma_2)$, for all $\sigma_1, \sigma_2 \in \mathbf{P}[\![P]\!](\sigma)$.

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

1. $\mathbf{C}[\![V]\!](\sigma) = \begin{cases} \sigma(x) & \text{if } \mathbf{V}[\![V]\!](\sigma) = x \in \mathrm{dom}(\sigma) \text{ and } \sigma(x) \in \mathcal{C}_T \\ \lightning & \text{otherwise} \end{cases}$;

2. $\mathbf{C}[\![A]\!](\sigma) = \mathbf{A}[\![A]\!]$, where $\mathbf{A}[\![-]\!] : \text{Atoms} \longrightarrow \mathcal{C}_T$ is the semantics of primitive components;

3. $\mathbf{C}[\![C_0C_1]\!](\sigma) = \begin{cases} \mathbf{C}[\![C_0]\!](\sigma) \wedge_T \mathbf{C}[\![C_1]\!](\sigma) & \text{if } \mathbf{C}[\![C_i]\!](\sigma) \in \mathcal{C}_T\text{, for } i \in \{0,1\} \\ \lightning & \text{otherwise} \end{cases}$;

4. $\mathbf{C}[\![\{C : P\}]\!](\sigma) = \begin{cases} \top_T & \text{if } \min \mathbf{P}[\![P]\!](\sigma) \text{ is empty or infinite} \\ C_1 \wedge_T \cdots \wedge_T C_k & \text{if } \min \mathbf{P}[\![P]\!](\sigma) = \{\sigma_1, \ldots, \sigma_k\} \neq \emptyset, \\ & \text{and } \mathbf{C}[\![C]\!](\sigma_i) = C_i \in \mathcal{C}_T \\ \lightning & \text{otherwise} \end{cases}$;

5. $\mathbf{C}[\![D\langle L\rangle(U)]\!](\sigma) = \begin{cases} \mathbf{D}[\![D]\!](\sigma)(\mathbf{L}[\![L]\!](\sigma), \mathbf{U}[\![U]\!](\sigma)) & \text{if defined} \\ \lightning & \text{otherwise} \end{cases}$.

The semantics of component definitions is a map $\mathbf{D}[\![-]\!] : \text{Definitions} \longrightarrow (\mathcal{D} \cup \{\lightning\})^{\Sigma}$, 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. $\mathbf{D}[\![V]\!](\sigma) = \begin{cases} \sigma(\mathbf{V}[\![V]\!](\sigma)) & \text{if } \mathbf{V}[\![V]\!](\sigma) = x \in \mathrm{dom}(\sigma) \text{ and } \sigma(x) \in \mathcal{D} \\ \lightning & \text{otherwise} \end{cases}$;

2. If D is a component $\langle U_0\rangle(U_1)\{C\}$, then for an array of parameter values $t \in \mathcal{V}^{\Box}$ and an array of nodes $q \in \mathcal{N}^{\Box}$, we define $\mathbf{D}[\![\langle U_0\rangle(U_1)\{C\}]\!](\sigma)(t, q)$ as follows: Recall from Section 4.1 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) $\mathbf{U}[\![U_0]\!](\sigma') = [s_1, \ldots, s_m] \neq \lightning$ 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 $1 \leq i \leq n$;
   (c) $\mathrm{flatten}([s_1, \ldots s_m, p_1, \ldots p_n]) \in \mathcal{N}^{\Box}$ has no duplicates;
   (d) $\mathrm{dom}(\sigma') \subseteq \mathcal{N}$ is minimal such that properties (a)-(c) are satisfied.

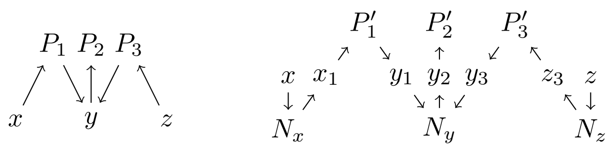
   We evaluate the body C of the component definition to the component $\mathbf{C}[\![C]\!](\sigma\sigma')$, where $\sigma\sigma'$ is the composition of σ and σ'. Define the map

$$r : \mathrm{supp}(\mathbf{C}[\![C]\!](\sigma\sigma')) \longrightarrow \mathcal{N}$$

   as

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

   Map r is well-defined, because $\mathrm{flatten}([s_1, \ldots s_m, p_1, \ldots p_n]) \in \mathcal{N}^{\Box}$ has no duplicates. Note that r is finite, since we assume that $\mathrm{supp}(\mathbf{C}[\![C]\!](\sigma\sigma'))$ is



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

Figure 4.1: Surgery on an I/O-component to remove mixed nodes.

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. $\mathbf{K}[\![I]\!](\sigma) = \mathbf{I}[\![I]\!]$;

2. $\mathbf{K}[\![KND]\!](\sigma) = \begin{cases} \sigma_0\{x \mapsto c\} & \text{if } \sigma_0 = \mathbf{K}[\![K]\!](\sigma) \neq \lightning, x = \mathbf{N}[\![N]\!], \\ & \text{and } c = \mathbf{D}[\![D]\!](\sigma_0) \neq \lightning \\ \lightning & \text{otherwise} \end{cases}$.

4.4 Input/output nodes

As mentioned in Section 4.1, nodes of primitive component definitions require input/output annotations. Treo regards such port type annotations as attributes of the primitive component. For a semantic sort T, we model the input nodes and output nodes of its instances via two maps $I, O : \mathcal{C}_T \longrightarrow 2^{\mathcal{N}}$ satisfying $\text{supp}(C) = I(C) \cup O(C)$, for all $C \in \mathcal{C}_T$. If $x \in I(C) \cap O(C)$, then we call x a *mixed* node.

Example 4.4.1 (Mixed nodes)**.** Recall the I/O component sort from Example 4.2.3. 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 4.1(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. $\Diamond$

Most semantic sorts 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 4.4.1 (Well-formedness)**.** A composition $C_1 \wedge_T \cdots \wedge_T C_n$, with $n \geq 0$, is well-formed if and only if $|\{i \in \{1,\ldots,n\} \mid x \in I(C_i)\}| \leq 1$ and $|\{i \in \{1,\ldots,n\} \mid x \in O(C_i)\}| \leq 1$, for all $x \in \mathcal{N}$.

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 4.3 unconditionally computes the composition. To define the semantics of $\mathbf{C}[\![-]\!]$ for a semantic sort 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 discover the degree of every node 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 sort IO_T, as defined in Example 4.2.3, wherein compositions consist of lists of primitive components. First, Treo wraps each primitive component $C \in \mathcal{C}_T$ within a primitive I/O-component $(C, I(C), O(C)) \in \mathcal{P}_T$. Using Section 4.3, Treo parses the Treo program over the semantic sort IO_T as usual, and obtains 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 semantic sort T) as a map node : $(2^{\mathcal{N}})^2 \times \mathcal{N} \longrightarrow \mathcal{C}_T$. For sets of names $I, O \subseteq \mathcal{N}$ and a default name $x \in \mathcal{N}$, the component $\mathrm{node}(I, O, x) \in \mathcal{C}_T$ has input nodes I (or $\{x\}$, if I is empty) and output nodes O (or $\{x\}$, if O is empty).

Definition 4.4.2 (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 4.4.2 (Surgery)**.** Figure 4.1(b) shows the result of applying the surgery map to the I/O-component $P_1 \cdot P_2 \cdot P_3$ from Example 4.4.1. 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. $\diamondsuit$

4.5 Related work

The Treo syntax offers a textual representation for the graphical Reo language [Arb04, Arb11]. We propose Treo as a syntax for Reo that (1) provides support for parameterization, recursion, iteration, and conditional construction; (2) implements basic design principles of Reo more closely than existing languages; and (3) reflects its declarative nature. The graphical Reo editor implemented as an Eclipse plugin [ECT] does not support parameterization, recursion, iteration, or conditional construction. RSL (with CARML for primitives) [BBKK09, Klü12] is imperative, while Reo is declarative. FOCAML [Jon16], supports only constraint automata [BSAR06], while Treo allows arbitrary user-defined semantic sorts for expressing the behavior of Reo primitives.

Since Treo leaves the syntax for primitive subsystems (i.e., semantic sorts) as user-defined, Treo is a "meta-language" that specifies compositional construction of complex structures (using the common core language defined in this chapter) 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 [Hol04]. 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 [FMS14]. SysML offers 9 types of diagrams, including activity diagrams and block diagrams. Each diagram provides a different view on the same system [Kru95]. Diagram types in SysML are comparable to semantic sorts in Treo. The main difference between the two, 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 [OMG06] 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 [BCL$^+$06] is an example of a component model that is hierarchical, which means a component can be a composition of subcomponents. Concrete instances of Fractal consist of libraries (API's) for a variety of programming languages, such as Java, C, and OMG IDL [BCL$^+$06]. Treo components and Fractal component differ with respect to interaction: Treo components interact via shared names, while Fractal component interact via explicit bindings.

4.6 Discussion

We propose Treo as a textual syntax for Reo connectors that allows user-defined semantic sorts, 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 currently restricted to consist of a composition of finitely many subsystems. Consequently, we cannot, for instance, express the construction of a primitive with an unbounded buffer, B_ω, from a set of primitives with buffer capacity of one, B_1. It seems, however, possible to use simulation and recursion to define B_ω in terms of B_1: 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 primitive with buffer of arbitrary large capacity. 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 4.3 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 semantic sort T from Definition 4.2.1 consists of a single composition operator $\wedge_T$. Generally, a semantic sort 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 semantic sorts.