TinyCORP: A Calculus for Context-Oriented Reactive Programming

Tetsuo Kamina
Oita University
Japan
kamina@acm.org

Tomoyuki Aotani
Tokyo Institute of Technology
Japan
aotani@c.titech.ac.jp

ABSTRACT

Current trend of seamless connections between computing systems and their surrounding environments requires software to be more reactive and adaptable, and reactive programming (RP) and context-oriented programming (COP) have been studied to directly support reactive behavior and dynamic adaptation. Sometimes reactive behavior and dynamic adaptation interact with each other. One issue of such interactions is how to avoid a loop of reactive behavior and dynamic adaptation when there are mutually recursive dependencies between them. This paper proposes TinyCORP, a core calculus for context-oriented reactive programming that is designed in a main-stream, general-purpose language setting. This calculus is expressive enough to represent both features of signals (i.e., time-varying values in RP) and layer-based partial methods in COP, and their interactions including the ability to specify the mutually recursive dependencies between dynamic adaptation and reactive behavior. We also demonstrate that the computation in TinyCORP do not result in the loop of reactive behavior and dynamic adaptation.

CCS CONCEPTS

• **Theory of computation → Program constructs**; *Operational semantics*.

KEYWORDS

context-oriented programming, signals, Featherweight Java

ACM Reference Format:
Tetsuo Kamina and Tomoyuki Aotani. 2019. TinyCORP: A Calculus for Context-Oriented Reactive Programming. In *Workshop on Context-oriented Programming (COP'19), July 15, 2019, London, United Kingdom.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3340671.3343356

1 INTRODUCTION

Current trend of seamless connections between computing systems and their surrounding environments requires software to be more reactive and adaptable, and intensive research efforts to provide appropriate programming language abstractions have been performed. Reactive programming (RP) makes data dependencies and reactive behaviors explicit, and context-oriented programming (COP) modularizes context-dependent behaviors that are adaptable at runtime. There have been lots of RP languages [7, 10, 17, 24, 25, 28] and COP languages [3, 8, 11, 18, 20, 23].

Sometimes reactive behaviors and context-dependent behaviors interact with each other. For example, dynamic adaptation, also known as *layer activation* in COP, can be considered as an effect of reactive behaviors, and thus language mechanisms for reactive layer activation have been proposed [15, 21]. Similarly, reactive behaviors can be context-dependent, and a COP extension to a functional-reactive programming (FRP) language has been proposed [29].

One issue of interactions between RP and COP constructs is cycles of such interactions. The above mentioned research efforts indicate that layer activation itself is a *time-varying value*, and this time-varying value may change the definition of reactive behavior that changes layer activation. This cycle of dependency will make the program stuck in a loop of layer activation and reactive behavior.

This issue was discussed and addressed in a COP extension of tiny FRP language mentioned above [29]. This language, Emfrp [26], is designed for small-scale embedded systems, and adopts the timer-based synchronous execution model. The COP extension to Emfrp supports layer activation based on conditions; i.e., when the condition is `true`, the specified layer is activated. This condition is a time-varying value. To avoid the loop of layer activation and reactive behavior, we cannot use the current values (i.e., values determined in the current time). Instead, each time-varying value used in the condition implicitly refers to the value computed in the *previous* time.

This paper discusses this issue in a more generalized setting. As mentioned above, there is a variety of work in RP and COP languages, and most of them are proposed as their extension to main-stream programming languages, which do not based on the timer-based execution model. Furthermore, in the COP extension to Emfrp, interactions between COP and RP are not fully discussed; e.g., most COP languages provide *partial methods*, which are not supported by Emfrp. These issues still obscures how RP and COP can naturally interact with each other.

In this paper, we propose TinyCORP, a core calculus for context-oriented reactive programming in a main-stream general-purpose language setting. This is designed as an extension to Featherweight Java (FJ) [14], meaning that the dynamic semantics are not timer-based but specified on the basis of beta reduction. TinyCORP supports RP features in that all fields are signals (i.e., time-varying values). It also supports COP features in that layers and partial methods are supported as in ContextFJ [12]. The notable features

COP'19, July 15, 2019, London, United Kingdom

ACM ISBN 978-1-4503-6863-6/19/07…$15.00
https://doi.org/10.1145/3340671.3343356

```
class Car {
  signal int p = ... // proportional gain
  signal int sum = p.sum(); // integral gain

  // setting the gear, the default is 'parking'
  signal int motor = powerDiff(p,sum);
  signal int tilt = ... //tilt sensor
  signal boolean running = false;
  signal boolean second = isHigh(tilt, motor);
}
```

Figure 1: Autonomous vehicle using SignalJ

```
layer NormalRunning {
  class Car {
    int powerDiff(int p, int sum) {
      .. // behavior of the drive gear
    } } }

layer HillRunning {
  class Car {
    int powerDiff(int p, int sum) {
      .. // behavior of the second gear
    } } }
```

Figure 2: Context-dependent behavior for the vehicle

```
contextgroup CarContext (Car c) {
  activate HillRunning if(c.running && c.second);
  activate NormalRunning if(c.running && !c.second);
}
```

Figure 3: Layer activation using signals

are (1) fields (signals) are also layer-dependent, and (2) layer activation depends on signals. Thus, we can express the mutual dependency between signals and layer activation. Nevertheless, the operational semantics are specified not to result in a loop between layer activation and signal evaluations.

The rest of this paper is structured as follows. Section 2 motivates us to develop an RP language model that also supports COP features. Section 3 formalizes that language model as a calculus TinyCORP. Section 4 discusses related work. Finally, Section 5 concludes this paper.

2 CONTEXT-ORIENTED REACTIVE PROGRAMMING

This section revisits the COP and RP mechanisms and describes our motivation.

2.1 Signals

We consider an autonomous vehicle that changes its gear using automotive tilt sensors. In this example, the control of the vehicle is implemented in a class, namely, `Car`, using SignalJ, a Java extension that supports signals [17]. The declaration of this class is shown in Figure 1. In this class, every field is declared as a signal, as indicated by the modifier `signal`. This means that every field is a time-varying value; i.e., if some value in the right-hand side of the declaration is updated, the value of the left-hand side is automatically recalculated. In this example, signals `p`, `tilt`, and `running` do not depend on any other signals. We refer to such a signal as a *source signal*. Other signal declarations refer to other signal in their right-hand side. We refer to such a signal as a *composite signal*. If some source signal (e.g., `p`) is updated, composite signals that depends on that (e.g., `sum`, `motor`, and `second`) are automatically updated.

These signals in `Car` describes the behavior of the vehicle. For example, we assume that the value of `p` is periodically updated. This update is propagated to signals that depend on `p`; thus, the values of `sum` and `motor` are also updated accordingly. Similarly, the updates of `tilt` and `motor` are propagated to `second`, which will be used to change its gear.

2.2 Layer-Based COP

COP provides a modularization mechanism to implement related context-dependent behavior (that crosscuts several existing classes) into a single module. This module, called a *layer*, can be dynamically composed (*activated*) and decomposed (*deactivated*) with the application. Each layer consists of a set of *partial methods*, which change the behavior of the original (base) methods in classes when the specified layer is activated.

For example, in the autonomous vehicle example, we identify two contexts, namely, running on a normal road and running on a steep road, and we have two layers, namely, `NormalRunning` and `HillRunning`, respectively, to implement behaviors that depend on those contexts (Figure 2). These layers implement their own behavior using the partial method `powerDiff` that override the original definition in `Car` (not shown in Figure 1) when the corresponding layer is activated. A partial method can also invoke the overridden behavior using `proceed`, which is similar to `super` that calls the method overridden by the class calling `super`.

2.3 Reactive Layer Activation

One issue in COP is how to specify the layer activation, and our previous work proposed layer activation based on signals [21]. This is described in Figure 3, which declares activation of `HillRunning` and `NormalRunning` using conditional expressions (enclosed in `if`). The construct `contextgroup` declares a set of related layer activation in that they specify layer activation for some specified group of object referring to the same instance `c` of `Car`. As `c.running` and `c.second` are signals, their changes are propagated to layer activation. For example, when `c.second` becomes `true` while the car is running, `HillRunning` is activated and `NormalRunning` is deactivated.

2.4 Context-Dependent Signals

Watanabe proposed a context-oriented FRP language where the definition of the signal changes according to the activated layers [29]. We integrate this feature in our example, to make it possible to rewrite layers in Figure 2 as follows:

```
layer NormalRunning {
  class Car { signal int motor = drive(p,sum); }
}
layer HillRunning {
  class Car { signal int motor = second(p,sum); }
}
```

In this example, methods `drive` and `second` implement the behavior of the drive gear and that of the second gear, respectively. Instead of using partial methods, those layer definitions "switch" the definition of the signal `motor` according to the activated layer. This example indicates that allowing signals change their definition makes the intention of the program clearer.

2.5 Mutual Recursive Dependency between Layers and Signals

So far, we have discussed the integration of COP and RP in two directions: making layer activation reactive and making reactive behavior context-dependent. If those features co-exist in the same language, we encounter the mutual recursive dependency between layer activation and signals.

Our example actually describes this case. The signal `second` depends on `motor`, and `motor` is layer-dependent. The layer activation uses the signal `second`. Thus, there is a cycle of dependencies among them. In general, such a cycle results in a loop of signal update propagations, which makes the program stuck.

This problem can be addressed if layer activation is performed *after* the evaluation of signals. In other words, if the layer activation is determined using the previous values of the signals, this loop of propagations does not occur. This was first discussed by Watanabe in the implementation of the context-oriented FRP language introduced above [29]. As this language is based on the timer-based FRP, it is obvious that what is the previous value. On the other hand, the language discussed in this section is based on a mainstream general-purpose language whose semantics is not provided as timer-based. Thus, our challenge is to discuss the integration of COP and RP in a more generalized setting.

3 CORE CALCULUS

To explain the language mechanisms explained in Section 2, we propose TinyCORP, a core calculus for context-oriented reactive programming. We desire that the calculus has the following features. First, the calculus must be able to explain both COP and RP features and their combinations, in particular partial methods, signals, reactive layer activation, and context-dependent signals. Second, the studied mechanisms should easily be integrated with the mainstream languages; i.e., it is desirable that the calculus is designed as a core of some mainstream language. Meanwhile, to make the study focused, the calculus should be as minimum as possible, and irrelevant features should be abstracted away. Finally,

$$
\begin{array}{lcl}
\mathtt{CL} & ::= & \mathtt{class\ C} \triangleleft \mathtt{C\ \{\ } \overline{\mathtt{T}}\ \overline{\mathtt{s}}\mathtt{=}\overline{\mathtt{e}}\mathtt{;}\ \overline{\mathtt{T}}\ \overline{\mathtt{f}}\mathtt{;\ K\ } \overline{\mathtt{M}}\ \mathtt{\}} \\
\mathtt{K} & ::= & \mathtt{C(}\overline{\mathtt{T}}\ \overline{\mathtt{x}}\mathtt{,}\ \overline{\mathtt{T}}\ \overline{\mathtt{y}}\mathtt{)\ \{\ super(}\overline{\mathtt{y}}\mathtt{);\ this.}\overline{\mathtt{f}}\mathtt{=}\overline{\mathtt{x}}\mathtt{;\ \}} \\
\mathtt{M} & ::= & \mathtt{T\ m(}\overline{\mathtt{T}}\ \overline{\mathtt{x}}\mathtt{)\ \{\ e\ \}} \\
\mathtt{e} & ::= & \mathtt{x} \mid \mathtt{e.f} \mid \mathtt{e.m(}\overline{\mathtt{e}}\mathtt{)} \mid \mathtt{new\ C(}\overline{\mathtt{e}}\mathtt{)} \mid \mathtt{e.f\ =\ e} \mid \mathtt{e;\ e} \mid \epsilon \mid \\
 & & \ell \mid \mathtt{true} \mid \mathtt{false} \mid \mathtt{proceed(}\overline{\mathtt{e}}\mathtt{)} \mid \mathtt{v<C,}\overline{\mathtt{L}}\mathtt{,}\overline{\mathtt{L}}\mathtt{>.m(}\overline{\mathtt{v}}\mathtt{)} \\
\mathtt{v} & ::= & \ell \\
\mathtt{T} & ::= & \mathtt{C} \mid \mathtt{boolean} \mid \mathtt{void} \\
\mathrm{lm} & ::= & \cdot \mid \mathtt{e\ ?}\ \overline{\mathtt{L}}\mathtt{;}\,\mathrm{lm}
\end{array}
$$

Figure 4: TinyCORP: Abstract Syntax

the calculus must have a fundamental property that it does not stuck in the loop of reactive behavior that changes itself.

To stick on the mainstream, basically it is designed as an extension of FJ [14]. The COP features are supported applying the same manner of ContextFJ [12], and the SignalJ-based RP feature [17] is superimposed on that. We found that the resulting calculus is fairly simple but as expressive as possible to explain the aforementioned language mechanisms.

3.1 Syntax

The abstract syntax of TinyCORP is shown in Figure 4. Let the metavariables `C`, `D`, and `E` range over class names; `L` range over layer names; `f` and `g` range over source signals; `s` ranges over composite signals; `m` ranges over method names; `d` and `e` range over expressions; `T` ranges over types; `v` ranges over values; and `x` ranges over variables that include `this`. Overlines denote sequences, e.g., $\overline{\mathtt{f}}$ stands for a possibly empty sequence $\mathtt{f}_1, \cdots, \mathtt{f}_n$. An empty sequence is denoted by $\cdot$. We also use "$\mathtt{this.}\overline{\mathtt{f}}\mathtt{=}\overline{\mathtt{x}}\mathtt{;}$" as shorthand for "$\mathtt{this.f}_1\mathtt{=x}_1\mathtt{;}\cdots\mathtt{;this.f}_n\mathtt{=x}_n\mathtt{;}$" where n denotes the length of $\overline{\mathtt{f}}$. Similar shorthand is applied throughout the syntax.

A class declaration consists of composite signal declarations, source signal declarations, a constructor declaration, and method declarations. Both composite and source signals are declared as fields, and all fields that have their initializers are composite signals. A constructor initializes the source signals. An expression can be either a variable, a field access, a method invocation, a constructor invocation, a field assignment, a concatenation, a location ℓ, an empty expression ϵ, a boolean expression such as `true` and `false`, a `proceed` call, or a runtime expression $\mathtt{v<C,}\overline{\mathtt{L}}\mathtt{,}\overline{\mathtt{L}}\mathtt{>.m(}\overline{\mathtt{v}}\mathtt{)}$, meaning that `m` is going to be invoked on value `v`. A value `v` is actually a location that points to an object. We use the metavariable `v` for our convenience. A type can be either a class name, a boolean type `boolean`, or a unit type `void`, which is the type of the empty expression.

To represent the COP-specific features, the calculus also provides a syntax for the *layer manager* lm, which corresponds to the `contextgroup` declaration shown in Figure 3. This is modeled as a list of layer activation rules. Each layer activation rule is of the form $\mathtt{e\ ?}\ \overline{\mathtt{L}}$, meaning that if `e` is `true`, all layers $\overline{\mathtt{L}}$ are activated. As in ContextFJ, the calculus does not provide syntax for layers. Instead, partial methods are registered in the partial method table *PT* that maps a triple `C`, `L`, and `m` of class, layer, and method to a partial method definition. Similarly, layer-dependent signals are registered in the signal table *ST* that maps a triple `C`, `L`, and `s` of class, layer,

and (composite) signal to the layer-dependent definition of the signal. The calculus also provides the class table CT that maps a class name C to the class definition.

There is one issue regarding the construction of the layer manager lm, which contains boolean expressions to judge layer activation. This construction requires dynamically changing the state of the layer manager, which makes computations complex unnecessarily. To keep the computation rules simple, we assume that the program execution consists of the following three phases, and our calculus only formalizes the final phase:

(1) Construction of the objects that do not depend on lm.
(2) Construction of lm.
(3) Execution of the program that depends on lm. Note that this phase does not reconstruct lm.

A TinyCORP program $(CT, PT, ST, \mathrm{lm}, \mu, \mathsf{e})$ consists of a class table CT, a partial method table PT, a signal table ST, a layer manager lm, an object store μ that maps a location to an object, and an expression e that corresponds to the body of the main method. We also assume the following conditions:

(1) $CT(\mathsf{C}) = \texttt{class C } \cdots$ for any $\mathsf{C} \in dom(CT)$, and no cycles exist in the transitive closure of $\triangleleft$ (extends).
(2) $PT(\mathsf{m}, \mathsf{C}, \mathsf{L}) = \cdots\ \texttt{m}(\cdots)\ \{\cdots\}$ for any $(\mathsf{m}, \mathsf{C}, \mathsf{L}) \in dom(PT)$, and $\overline{\mathsf{C}}, \mathsf{C}_0 = \overline{\mathsf{D}}, \mathsf{D}_0$ if $PT(\mathsf{m}, \mathsf{C}, \mathsf{L}_1) = \mathsf{C}_0\ \texttt{m}(\overline{\mathsf{C}}\ \overline{\mathsf{x}})\ \{\cdots\}$ and $PT(\mathsf{m}, \mathsf{C}, \mathsf{L}_2) = \mathsf{D}_0\ \texttt{m}(\overline{\mathsf{D}}\ \overline{\mathsf{y}})\ \{\cdots\}$ for all m and C (i.e., there is no conflict between partial methods).
(3) $ST(\mathsf{s}, \mathsf{C}, \mathsf{L}) = \mathsf{e}$ for any $(\mathsf{s}, \mathsf{C}, \mathsf{L}) \in dom ST$.
(4) All locations that appear in lm also appear in $dom(\mu)$.
(5) Field hiding and method overloading are not allowed, and all fields in the same class and all parameters in the same method are distinct.

3.2 Computation

The operational semantics of TinyCORP, which is shown in Figure 5, is given by a reduction relation of the form $\mathsf{e} \mid \mu \mid \overline{\mathsf{L}} \longrightarrow \mathsf{e} \mid \mu' \mid \overline{\mathsf{L}}$, which is read as "expression e under object store μ and activated layers $\overline{\mathsf{L}}$ reduces to e′ under μ' and $\overline{\mathsf{L}}'$."

3.2.1 Pull-based signal evaluation. The pull-based behavior of signals is explained by the rule R-CField, which defines the field access where the accessed field is a composite signal. The composite signal s is initialized with an expression e_0, which is eventually evaluated (after layer (de)activation explained below) every time the field is accessed. Thus, this ensures the immediate propagation of a change in the source signal, which is defined by R-Assign. We use $\uplus$ as a relational override of the object store; that is, $(x \uplus y)(k) = y(k)$ if $k \in dom(y)$ or $x(k)$ otherwise. R-Assign ensures that each source signal is updated when the field assignment is performed, and the value of a composite signal is always computed using the up-to-date source signal values. As an assignment reduces to an empty expression, it is assumed to be followed by a concatenation (as shown by R-Cat).

3.2.2 Context-dependent behavior. Following ContextFJ, TinyCORP formalizes context-dependent behavior. First, it provides computation rules for partial methods, which are mostly identical to ContextFJ. Rule R-Invk defines the method invocation where the "cursor", indicating where method lookup should start among the class

$$\boxed{\mathsf{e} \mid \mu \mid \overline{\mathsf{L}} \longrightarrow \mathsf{e}' \mid \mu' \mid \overline{\mathsf{L}}'}$$

$$\frac{\mu(\ell) = \texttt{new C}(\overline{\mathsf{v}}) \qquad \mathit{activeLayers}(\mathrm{lm}, \mu, \overline{\mathsf{L}}) = \overline{\mathsf{L}}'}{\ell.\mathsf{f}_i \mid \mu \mid \overline{\mathsf{L}} \longrightarrow \mathsf{v}_i \mid \mu \mid \overline{\mathsf{L}}'} \quad \text{(R-Field)}$$

$$\frac{\mu(\ell) = \texttt{new C}(\overline{\mathsf{v}}) \qquad \mathit{fdecl}(\mathsf{s}, \mathsf{C}, \overline{\mathsf{L}}') = \mathsf{e}_0 \qquad \mathit{activeLayers}(\mathrm{lm}, \mu, \overline{\mathsf{L}}) = \overline{\mathsf{L}}'}{\ell.\mathsf{s} \mid \mu \mid \overline{\mathsf{L}} \longrightarrow \mathsf{e}_0 \mid \mu \mid \overline{\mathsf{L}}'} \quad \text{(R-CField)}$$

$$\frac{\mu(\ell) = \texttt{new C}(\overline{\mathsf{w}}) \qquad \mathit{activeLayers}(\mathrm{lm}, \mu, \overline{\mathsf{L}}) = \overline{\mathsf{L}}'}{\ell.\texttt{m}(\overline{\mathsf{v}}) \mid \mu \mid \overline{\mathsf{L}} \longrightarrow \ell\texttt{<C},\overline{\mathsf{L}}'\texttt{,}\overline{\mathsf{L}}'\texttt{>.m}(\overline{\mathsf{v}}) \mid \mu \mid \overline{\mathsf{L}}'} \quad \text{(R-Invk)}$$

$$\frac{\mathit{mbody}(\mathsf{m}, \mathsf{C}, \overline{\mathsf{L}}'', \overline{\mathsf{L}}') = \overline{\mathsf{x}}.\mathsf{e}\ \texttt{in C}', \cdot}{\ell\texttt{<C},\overline{\mathsf{L}}''\texttt{,}\overline{\mathsf{L}}'\texttt{>.m}(\overline{\mathsf{v}}) \mid \mu \mid \overline{\mathsf{L}} \longrightarrow \left[\begin{array}{l} \ell\texttt{/this} \\ \overline{\mathsf{v}}/\overline{\mathsf{x}} \end{array}\right] \mathsf{e} \mid \mu \mid \overline{\mathsf{L}}} \quad \text{(R-InvkB)}$$

$$\frac{\mathit{mbody}(\mathsf{m}, \mathsf{C}, \overline{\mathsf{L}}'', \overline{\mathsf{L}}') = \overline{\mathsf{x}}.\mathsf{e}\ \texttt{in C}', (\overline{\mathsf{L}}'''; \mathsf{L}_0)}{\ell\texttt{<C},\overline{\mathsf{L}}''\texttt{,}\overline{\mathsf{L}}'\texttt{>.m}(\overline{\mathsf{v}}) \mid \mu \mid \overline{\mathsf{L}} \longrightarrow \left[\begin{array}{ll} \ell & \texttt{/this} \\ \overline{\mathsf{v}} & /\overline{\mathsf{x}} \\ \ell\texttt{<C}',\overline{\mathsf{L}}'''\texttt{,}\overline{\mathsf{L}}'\texttt{>.m/proceed} & \end{array}\right] \mathsf{e} \mid \mu \mid \overline{\mathsf{L}}} \quad \text{(R-InvkP)}$$

$$\frac{\ell \notin \mu}{\texttt{new C}(\overline{\mathsf{v}}) \mid \mu \mid \overline{\mathsf{L}} \longrightarrow \ell \mid \mu \cup \{\ell \mapsto \texttt{new C}(\overline{\mathsf{v}})\} \mid \overline{\mathsf{L}}} \quad \text{(R-New)}$$

$$\frac{\mu(\ell) = \texttt{new C}(\overline{\mathsf{v}}) \qquad \#(\overline{\mathsf{v}}) = n}{\ell.\mathsf{f}_i = \mathsf{v}_0 \mid \mu \mid \overline{\mathsf{L}} \longrightarrow \epsilon \mid \mu \uplus \{\ell \mapsto \texttt{new C}\ (\mathsf{v}_1, \cdots, \mathsf{v}_{i-1}, \mathsf{v}_0, \mathsf{v}_{i+1}, \cdots, \mathsf{v}_n)\} \mid \overline{\mathsf{L}}} \quad \text{(R-Assign)}$$

$$\epsilon;\ \mathsf{e} \mid \mu \mid \overline{\mathsf{L}} \longrightarrow \mathsf{e} \mid \mu \mid \overline{\mathsf{L}} \quad \text{(R-Cat)}$$

Figure 5: TinyCORP: Computation

hierarchy and currently activated layers, has not been initialized. This cursor is set as the receiver's class and the sequence of activated layers computed by *activeLayers*, which will be explained shortly. Two rules, R-InvkB and R-InvkP, represent invocations of a base method and a partial method, respectively. Both rules use the auxiliary function *mbody*, which is defined in Figure 6 and searches the method body in the order specified by ContextFJ. R-InvkP addresses the case where the method body is found in layer L_0 in class C′. In this case, proceed in the method body is replaced with the invocation of the same method, where the receiver's cursor points to the next layers $\overline{\mathsf{L}}'''$.

The context-dependent behavior originally introduced by TinyCORP is context-dependent signals, which are defined by R-CField. The signal expression e_0 is obtained using *fdecl* defined in Figure 6.

$\boxed{fdecl(\mathtt{s}, \mathtt{C}, \overline{\mathtt{L}}) = \mathtt{e}}$

$$\frac{CT(\mathtt{C}) = \mathtt{class\ C} \lhd \mathtt{D}\ \{\ \cdots\ \mathtt{T\ s{=}e;}\ \cdots\ \}}{fdecl(\mathtt{s}, \mathtt{C}, \cdot) = \mathtt{e}}$$

$$\frac{CT(\mathtt{C}) = \mathtt{class\ C} \lhd \mathtt{D}\ \{\ \overline{\mathtt{T}}\ \overline{\mathtt{s}}{=}\overline{\mathtt{e}}\ \cdots\ \} \qquad \mathtt{s} \notin \overline{\mathtt{s}}}{fdecl(\mathtt{s}, \mathtt{C}, \cdot) = fdecl(\mathtt{s}, \mathtt{D}, \cdot)}$$

$$\frac{ST(\mathtt{s}, \mathtt{C}, \mathtt{L}) = \mathtt{e}}{fdecl(\mathtt{s}, \mathtt{C}, \mathtt{L}; \overline{\mathtt{L}}) = \mathtt{e}}$$

$$\frac{ST(\mathtt{s}, \mathtt{C}, \mathtt{L})\ \text{undefined} \qquad fdecl(\mathtt{s}, \mathtt{C}, \overline{\mathtt{L}}) = \mathtt{e}}{fdecl(\mathtt{s}, \mathtt{C}, \mathtt{L}; \overline{\mathtt{L}}) = \mathtt{e}}$$

$\boxed{mbody(\mathtt{m}, \mathtt{C}, \overline{\mathtt{L}'}, \overline{\mathtt{L}}) = \overline{\mathtt{x}}.\mathtt{e\ in\ D}, \overline{\mathtt{L}''}}$

$$\frac{CT(\mathtt{C}) = \mathtt{class\ C} \lhd \mathtt{D}\ \{\ \cdots\ \mathtt{T\ m}(\overline{\mathtt{T}}\ \overline{\mathtt{x}})\ \{\ \mathtt{e}\ \}\ \cdots\ \}}{mbody(\mathtt{m}, \mathtt{C}, \cdot, \overline{\mathtt{L}}) = \overline{\mathtt{x}}.\mathtt{e\ in\ C}, \cdot}$$

$$\frac{CT(\mathtt{C}) = \mathtt{class\ C} \lhd \mathtt{D}\ \{\ \cdots\ \overline{\mathtt{M}}\ \} \qquad \mathtt{m} \notin \overline{\mathtt{M}} \qquad mbody(\mathtt{m}, \mathtt{D}, \overline{\mathtt{L}}, \overline{\mathtt{L}}) = \overline{\mathtt{x}}.\mathtt{e\ in\ E}, \overline{\mathtt{L}'}}{mbody(\mathtt{m}, \mathtt{C}, \cdot, \overline{\mathtt{L}}) = \overline{\mathtt{x}}.\mathtt{e\ in\ E}, \overline{\mathtt{L}'}}$$

$$\frac{PT(\mathtt{m}, \mathtt{C}, \mathtt{L}_0) = \mathtt{C}_0\ \mathtt{m}(\overline{\mathtt{C}}\ \overline{\mathtt{x}})\ \{\ \mathtt{return\ e;}\ \}}{mbody(\mathtt{m}, \mathtt{C}, (\overline{\mathtt{L}'}; \mathtt{L}_0), \overline{\mathtt{L}}) = \overline{\mathtt{x}}.\mathtt{e\ in\ C}, (\overline{\mathtt{L}'}; \mathtt{L}_0)}$$

$$\frac{PT(\mathtt{m}, \mathtt{C}, \mathtt{L}_0)\ \text{undefined} \qquad mbody(\mathtt{m}, \mathtt{C}, \overline{\mathtt{L}'}, \overline{\mathtt{L}}) = \overline{\mathtt{x}}.\mathtt{e\ in\ D}, \overline{\mathtt{L}''}}{mbody(\mathtt{m}, \mathtt{C}, (\overline{\mathtt{L}'}; \mathtt{L}_0), \overline{\mathtt{L}}) = \overline{\mathtt{x}}.\mathtt{e\ in\ D}, \overline{\mathtt{L}''}}$$

Figure 6: Field and method access

$\boxed{\overline{\mathtt{L}}, \mu \vdash \mathtt{e} \longrightarrow \mathtt{e}'}$

$$\frac{\mu(\ell) = \mathtt{new\ C}(\overline{\mathtt{v}})}{\overline{\mathtt{L}}, \mu \vdash \ell.\mathtt{f}_i \longrightarrow \mathtt{v}_i}\ \text{(R-FieldF)}$$

$$\frac{\mu(\ell) = \mathtt{new\ C}(\overline{\mathtt{v}}) \qquad fdecl(\mathtt{f}, \mathtt{C}, \overline{\mathtt{L}}) = \mathtt{e}_0}{\overline{\mathtt{L}}, \mu \vdash \ell.\mathtt{f} \longrightarrow \mathtt{e}_0}\ \text{(R-CFieldF)}$$

$$\frac{\mu(\ell) = \mathtt{new\ C}(\overline{\mathtt{w}}) \qquad \ell{<}\mathtt{C}, \overline{\mathtt{L}}, \overline{\mathtt{L}}{>}.\mathtt{m}(\overline{\mathtt{v}}) \mid \mu \mid \overline{\mathtt{L}} \longrightarrow \mathtt{e}_0 \mid \mu \mid \overline{\mathtt{L}}}{\overline{\mathtt{L}}, \mu \vdash \ell.\mathtt{m}(\overline{\mathtt{v}}) \longrightarrow \mathtt{e}_0}$$

(R-InvkF)

Figure 7: TinyCORP: Computation with fixed layers

This function searches the signal expression from the class hierarchy and the sequence of activated layers in the order similar to *mbody*. Thus, how the signal s behaves depends on the currently activated layers $\overline{\mathtt{L}'}$.

$$\frac{\overline{\mathtt{L}}, \mu \vdash \mathtt{e} \longrightarrow^* \mathtt{true}}{activeLayers(\mathtt{e?L}_0; \mathtt{lm}, \mu, \overline{\mathtt{L}}) = \mathtt{L}_0, activeLayers(\mathtt{lm}, \mu, \overline{\mathtt{L}})}$$

$$\frac{\overline{\mathtt{L}}, \mu \vdash \mathtt{e} \longrightarrow^* \mathtt{false}}{activeLayers(\mathtt{e?L}_0; \mathtt{lm}, \mu, \overline{\mathtt{L}}) = activeLayers(\mathtt{lm}, \mu, \overline{\mathtt{L}})}$$

$$activeLayers(\cdot, \mu, \overline{\mathtt{L}}) = \cdot$$

Figure 8: Judging activated layers

3.2.3 Judging activated layers. Layer activation is determined by signals in TinyCORP, and the issue is when to determine which layer is activated. One approach could be determining the timing when the state (i.e., μ) is updated. This can be formalized using the event handler mechanism similar to SignalJ. However, to keep the calculus simple, we take another approach, where the layer activation is determined just before the context-dependent behavior is executed. In our case, context-dependent behavior includes signal evaluation (via field access) and method invocation. Thus, R-Field, R-CField, and R-Invk include the judgment of activated layers, which is determined using *activeLayers*, in their premises. We note that we take the strategy of layer activation where the layer activation does not change during the execution of partial methods; i.e., as explained by R-InvkB and R-InvkP, the judgment using *activeLayers* does not appear in their premises. This strategy is also adopted by most COP languages.

The function *activeLayers* returns layers $\overline{\mathtt{L}}$ that should be activated, and this judgment is performed using the boolean expressions in lm. As this judgment might be explained using the aforementioned computation rules, at first the definition of *activeLayers* seems to be formulated as follows:

$$\frac{\mathtt{e} \mid \mu \mid \overline{\mathtt{L}} \longrightarrow^* \mathtt{true} \mid \mu \mid \overline{\mathtt{L}'}}{activeLayers(\mathtt{e?L}_0; \mathtt{lm}, \mu, \overline{\mathtt{L}}) = \mathtt{L}_0, activeLayers(\mathtt{lm}, \mu, \overline{\mathtt{L}'})}$$

$$\frac{\mathtt{e} \mid \mu \mid \overline{\mathtt{L}} \longrightarrow^* \mathtt{false} \mid \mu \mid \overline{\mathtt{L}'}}{activeLayers(\mathtt{e?L}_0; \mathtt{lm}, \mu, \overline{\mathtt{L}}) = activeLayers(\mathtt{lm}, \mu, \overline{\mathtt{L}'})}$$

$$activeLayers(\cdot, \mu, \overline{\mathtt{L}}) = \cdot$$

Actually, this definition does not work, as the premise of each rule may further call *activeLayers*, resulting in a loop of reactive layer activation. This implies that, during the execution of *activeLayers*, any other layer activation must not be handled, and the evaluation of each boolean expression in lm must be performed under the fixed active layers.

For this purpose, we provide computation rules other than the ones in Figure 5. Those are shown in Figure 7. Those rules are given by a reduction relation of the form $\overline{\mathtt{L}}, \mu \vdash \mathtt{e} \longrightarrow \mathtt{e}'$, which is read as "under the fixed activated layers $\overline{\mathtt{L}}$ and object store μ, expression e reduces to e'." We note that boolean expressions in lm are constructed without using side-effective expressions such as object

construction and assignment[1]. Thus, computation rules in Figure 7 consists of only field access (R-FieldF and R-CFieldF) and method invocation (R-InvkF). The field access rules are straightforward; the difference from rules in Figure 5 is that they do not call *activeLayers* in their premises. The rule R-InvkF indicates that if the method body e_0 is found by searching from $\ell.\mathtt{m}(\overline{\mathsf{v}})$ without activating any layers, then $\ell.\mathtt{m}(\overline{\mathsf{v}})$ is reduced to e_0. We note that $\ell\texttt{<C},\overline{\mathtt{L}},\overline{\mathtt{L}}\texttt{>.m}(\overline{\mathsf{v}})$ is reduced to e_0 in a single step (if *mbody* succeeds) by applying R-InvkB or R-InvkP, which do not trigger any layer activation.

Using this reduction relation, *activeLayers* is defined as shown in Figure 8. It checks each boolean expression e in lm, and adds the corresponding layer $\mathtt{L}_0$ to the return value only if e reduces to `true` under $\overline{\mathtt{L}}$ and μ.

3.2.4 Congruence rule. Finally, we show the straightforward congruence rule that enables a reduction of subexpressions. We first introduce the evaluation context E as follows:

$$\begin{array}{rcl} E & ::= & \texttt{[].f} \mid \texttt{[].m(}\overline{\mathsf{e}}\texttt{)} \mid \ell\texttt{.m(}\overline{\ell}\texttt{,[],}\overline{\mathsf{e}}\texttt{)} \mid \texttt{new C(}\overline{\ell}\texttt{,[],}\overline{\mathsf{e}}\texttt{)} \mid \\ & & \texttt{[].f = e} \mid \ell\texttt{.f = []} \mid \texttt{[]; e} \end{array}$$

Each evaluation context is an expression with a hole (written []) somewhere inside it. We write $E[\mathsf{e}]$ for an expression obtained by replacing the hole in E with e.

The congruence rule is defined as follows:

$$\frac{\mathsf{e} \mid \mu \mid \overline{\mathtt{L}} \longrightarrow \mathsf{e}' \mid \mu' \mid \overline{\mathtt{L}}'}{E[\mathsf{e}] \mid \mu \mid \overline{\mathtt{L}} \longrightarrow E[\mathsf{e}'] \mid \mu' \mid \overline{\mathtt{L}}'}$$

The evaluation context defines the evaluation order of arguments to method and constructor invocations, and ensures that the reduction of the right-hand side of ; occurs after the left-hand side reduces to an empty value.

3.2.5 Expected properties. To state the desired progress property, we need to provide a type system, which should be almost identical to that of ContextFJ [12]. Instead of repeating the definitions of type system, in this paper, we informally discuss how the progress property is defined in TinyCORP. First, we need to define the progress property of the reduction that appear in premises of *activeLayers*, as *activeLayers* must *return* some sequence of layers to make the calculus have the progress property. This can be informally stated as that a well-typed conditional expression e in lm reduces to another expression e', or either `true` or `false`. This should be easily proven if we assume a ContextFJ-like type system. We note that this property holds even when there is a cyclic dependency between layer activation rules and layer-dependent signals.

Having this progress property of conditional expressions in lm, we can state the progress property of TinyCORP as follows. If every e in lm eventually reduces to `true` or `false`, well-typed TinyCORP expression e_0 reduces to another expression e_0', or it is a normal form. This can be easily shown that if every e in lm eventually reduces to a boolean value, *activeLayers* always return some sequence of layers. It should be noted that *activeLayers* will not return if there is an infinite loop that is explicitly written (e.g., recursive calls that will not stop) in the evaluation of the conditional expressions.

[1]This restriction is consistent with other reactive layer activation mechanisms such as that found in Emfrp [29].

Definitions:

CT: C $\mapsto$ `class C { boolean s1 = true; boolean m() { return true; } }`

PT: (m, C, L1) $\mapsto$ `boolean m() { return this.s1; }`

ST: (s1, C, L1) $\mapsto$ `false`

lm: ℓ`.s1?L1`

μ: ℓ $\mapsto$ `new C()`

Computation:

$$\begin{array}{lll} & & \cdot, \mu \vdash \ell\texttt{.s1} \longrightarrow \texttt{true} \\ & & \mathit{activeLayers}(\mathrm{lm}, \mu, \cdot) = \texttt{L1} \\ \ell\texttt{.m()} \mid \mu \mid \cdot & \longrightarrow_{\text{(R-Invk)}} & \ell\texttt{<C,L1,L1>.m()} \mid \mu \mid \texttt{L1} \\ & \longrightarrow_{\text{(R-InvkP)}} & \ell\texttt{.s1} \mid \mu \mid \texttt{L1} \\ & \longrightarrow_{\text{(R-CField)}} & \texttt{false} \mid \mu \mid \cdot \end{array}$$

Figure 9: Example execution with mutually dependent signal and layer activation

3.2.6 Example. Instead of providing proofs, we demonstrate how TinyCORP computation proceeds under the existence of mutual dependency between signals and layer activation in Figure 9. In this example, CT contains class C, which declares signal s1 and method m. Those signal and method are overridden by layer L1 as indicated by PT and ST. The layer manager lm uses s1 for the activation of L1; thus, the definition of s1 and the activation of L1 are mutually dependent. The object store μ is initialized to contain the instance of C.

Under these definitions, the evaluation of ℓ`.m()` starts by applying R-Invk. This rule uses *activeLayers* to judge the layer activation. Without any activated layers, the condition ℓ`.s1` reduces to `true`, and thus L1 is activated. Then, the method lookup starts with the activated layer L1, and the body of the partial method is found (R-InvkP). Finally, this body reduces to the layer-dependent value, which is `false`, and the layer L1 is deactivated according to lm.

4 RELATED WORK

There are several research efforts on formalization of both COP and RP. This section introduces those pieces of work, and discusses how TinyCORP is different from them.

Several COP calculi discuss the way of layer activation. Firstly proposed COP calculi formalized their core languages with synchronous layer activation (also called `with`-blocks) [6, 12]. After that, other layer activation mechanisms were also formalized. Examples are event-based layer activation [1], layer activation that depends on other layer activation (also known as composite layers) [19], unified layer activation [2], and generalized layer activation [20]. From this perspective, TinyCORP formalizes layer activation based on signals, but actually it also provides more: formalization of signals that are layer-dependent.

Other research direction discusses the type system of COP, which becomes an interesting issue if we consider layer-introduced base methods [13]. For example, JCop [4], a Java-based COP language, allows a layer introduce new methods and other layer can import them, and its type system is known to be unsafe and a type

safe alternative was proposed [16]. A safe type system for layer-introduced base methods with asynchronous layer deactivation was also proposed [22]. In this paper we do not discuss a type system for TinyCORP, as it will not be interesting because TinyCORP does not provide layer-introduced base methods. We consider that the idea of safe type system for layer-introduced base methods with asynchronous layer deactivation [22] can also be applied to TinyCORP if we extend it with layer-introduced base methods.

The study of FRP originated with Elliott and Hudak [10], who focused on time-varying values in functional programs. RT-FRP [27] is a statically typed language that deals with the use of FRP in real-time applications and identifies a subset of FRP where it can statically guarantee that the time and space costs for a given program are statically bounded. E-FRP [28] further simplifies this idea by generalizing the global clock used in RT-FRP to a set of events. The push-pull FRP [9] determines the evaluation strategy for the efficient implementation to combine both benefits. Those pieces of work are based on functional programming, and their purpose is to enhance their efficiency in resource usage, or to provide an efficient implementation. On the other hand, TinyCORP is based on object-oriented programming, and its purpose is to study its unification with other programming paradigm. The design of RP mechanism in TinyCORP is basically taken from SignalJ [17], which combines the imperative event mechanism with signals where the push and pull strategies are also combined. However, TinyCORP only provides the pull strategy for signals to keep the calculus simple. Instead, it provides fundamental features of COP and the other interesting features of reactive layer activation and layer-dependent signals.

The combination between RP and COP is not new. Inoue and Igarashi proposed layer activation using reactive values [15], and effective layer activation with push-based evaluation was also proposed [21]. On the other hand, Watanabe proposed context-dependent signals [29], which is a COP extension of their FRP language, Emfrp, designed for small-scale embedded systems [26]. The discussion regarding how to avoid the loop of layer activation was also discussed in that paper. The approach is basically the same as the one proposed in this paper; i.e., layer activation is judged using the values computed in the previous time (Emfrp is a timer-based FRP language). TinyCORP discusses this issue in a more generalized setting. Its target is a mainstream general-purpose language where the execution is not timer-based. Thus, TinyCORP provides its semantics based on beta-reduction.

5 CONCLUSION

We proposed TinyCORP, a core calculus for context-oriented reactive programming. This calculus supports both features of COP and RP such as partial methods with `proceed` and pull-based signal evaluation. This calculus also provides interactions between them such as layer activation that is based on signals, and signals that depend on layer activation. By providing two kinds of reduction rules, our calculus avoids the update propagation loop even when there is a mutual dependency between signals and layer activation.

We note that there are still remaining issues. One future research direction that worth discussing is interaction between TinyCORP with other COP features. For example, some COP calculi allows layers introduce new methods, which can be imported by other layers using `requires`. Similarly, we may also consider layer-introduced signals. To extend TinyCORP with those features, we need to develop a type system that ensures calls of layer-introduced base methods and accesses to layer-introduced signals never fail. Another possible research direction is extending the RP mechanism. For example, in TinyCORP, source signals can appear only in classes; i.e., layers cannot override them. If layers can override source signals, updates of source signals become layer-dependent. This means that layers have their own state, which is similar to the interruptible computation [5]. We consider that the study presented in this paper will be a good starting point for those further research directions.

REFERENCES

[1] Tomoyuki Aotani, Tetsuo Kamina, and Hidehiko Masuhara. Featherweight EventCJ: a core calculus for a context-oriented language with event-based per-instance layer transition. In *COP'11*, 2011.

[2] Tomoyuki Aotani, Tetsuo Kamina, and Hidehiko Masuhara. Unifying multiple layer activation mechanism using one event sequence. In *COP'14*, pages 2:1–2:6, 2014.

[3] Malte Appeltauer, Robert Hirschfeld, Michael Haupt, and Hidehiko Masuhara. ContextJ: Context-oriented programming with Java. *Computer Software*, 28(1):272–292, 2011.

[4] Malte Appeltauer, Robert Hirschfeld, Hidehiko Masuhara, Michael Haupt, and Kazunori Kawauchi. Event-specific software composition in context-oriented programming. In *Proceedings of the International Conference on Software Composition 2010 (SC'10)*, volume 6144 of *LNCS*, pages 50–65, 2010.

[5] Engineer Bainomugisha, Jorge Vallejos, Coen De Roover, Andoni Lombide Carreton, and Wolfgang De Meuter. Interruptible context-dependent executions: A fresh look at programming context-aware applications. In *Onward! 2012*, pages 67–84, 2012.

[6] Dave Clarke and Ilya Sergey. A semantics for context-oriented programming with layers. In *COP'09*, 2009.

[7] Gregory H. Cooper. *Integrating Dataflow Evaluation into a Practical Higher-Order Call-by-Value Language*. PhD thesis, Department of Computer Science, Brown University, 2008.

[8] Pascal Costanza and Robert Hirschfeld. Language constructs for context-oriented programming – an overview of ContextL. In *Dynamic Language Symposium (DLS) '05*, pages 1–10, 2005.

[9] Conal Elliott. Push-pull functional reactive programming. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell (Haskell'09)*, pages 25–36, 2009.

[10] Conal Elliott and Paul Hudak. Functional reactive animation. In *Proceedings of the 2nd ACM SIGPLAN International Conference on Functional Programming (ICFP'97)*, pages 263–273, 1997.

[11] Sebastián González, Nicolás Cardozo, Kim Mens, Alfredo Cádiz, Jean-Christophe Libbrecht, and Julien Goffaux. Subjective-C: Bringing context to mobile platform programming. In *SLE'10*, volume 6563 of *LNCS*, pages 246–265, 2011.

[12] Robert Hirschfeld, Atsushi Igarashi, and Hidehiko Masuhara. ContextFJ: a minimal core calculus for context-oriented programming. In *FOAL'11*, pages 19–23, 2011.

[13] Atsushi Igarashi, Robert Hirschfeld, and Hidehiko Masuhara. A type system for dynamic layer composition. In *FOOL'12*, 2012.

[14] Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 2001.

[15] Hiroaki Inoue and Atsushi Igarashi. A library-based approach to context-dependent computation with reactive values. In *MODULARITY Companion'16*, pages 50–54, 2016.

[16] Hiroaki Inoue, Atsushi Igarashi, Malte Appeltauer, and Robert Hirschfeld. Towards type-safe JCop: A type system for layer inheritance and first-class layers. In *COP'14*, 2014.

[17] Tetsuo Kamina and Tomoyuki Aotani. Harmonizing signals and events with a lightweight extension to Java. *The Art, Science, and Engineering of Programming*, 2(3), 2018.

[18] Tetsuo Kamina, Tomoyuki Aotani, and Hidehiko Masuhara. EventCJ: a context-oriented programming language with declarative event-based context transition. In *AOSD '11*, pages 253–264, 2011.

[19] Tetsuo Kamina, Tomoyuki Aotani, and Hidehiko Masuhara. A core calculus of composite layers. In *FOAL'13*, pages 7–12, 2013.

[20] Tetsuo Kamina, Tomoyuki Aotani, and Hidehiko Masuhara. Generalized layer activation mechanism for context-oriented programming. *LNCS Transactions on Modularity and Composition*, 9800:123–166, 2016.

[21] Tetsuo Kamina, Tomoyuki Aotani, and Hidehiko Masuhara. Push-based reactive layer activation in context-oriented programming. In *COP'17*, pages 17–21, 2017.

[22] Tetsuo Kamina, Tomoyuki Aotani, Hidehiko Masuhara, and Atsushi Igarashi. Method safety mechanism for asynchronous layer deactivation. *Science of Computer Programming*, 156:104–120, 2018.

[23] Jens Lincke, Malte Appeltauer, Bastian Steinert, and Robert Hirschfeld. An open implementation for context-oriented layer composition in ContextJS. *Science of Computer Programming*, 76(12):1194–1209, 2011.

[24] Leo A. Meyerovich, Arjun Guha, Jacob Baskin, Gregory H. Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. Flapjax: A programming language for Ajax applications. In *Proceedings of the 24th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Application (OOPSLA'09)*, pages 1–20, 2009.

[25] Guido Salvaneschi, Gerold Hintz, and Mira Mezini. REScala: Bridging between object-oriented and functional style in reactive applications. In *Proceedings of the 13th International Conference on Modularity (MODULARITY'14)*, pages 25–36, 2014.

[26] Kensuke Sawada and Takuo Watanabe. Emfrp: a functional reactive programming language for small-scale embedded systems. In *MODULARITY Companion*, pages 36–44, 2016.

[27] Zhanyong Wan, Walid Taha, and Paul Hudak. Real-time FRP. In *Proceedings of the 6th ACM SIGPLAN International Conference on Functional Programming (ICFP'01)*, pages 146–156, 2001.

[28] Zhanyong Wan, Walid Taha, and Paul Hudak. Event-driven FRP. In *PADL 2002: Practical Aspects of Declarative Languages*, volume 2257 of *LNCS*, pages 155–172, 2002.

[29] Takuo Watanabe. A simple context-oriented programming extension to an FRP language for small-scale embedded systems. In *COP'18*, pages 23–30, 2018.