Cooperative Decoupled Processes

The E-Calculus and Linearity

Andi Bejleri[1] Mira Mezini[1,2] Patrick Eugster[1,3]

[1]TU Darmstadt, Germany [2]Lancaster University, UK [3]Purdue University, USA

{bejleri, mezini, peugster}@cs.tu-darmstadt.de

Abstract

Event-driven programming has become a major paradigm in developing concurrent, distributed systems. Its benefits are often informally captured by the key tenet of "decoupling", a notion which roughly captures the ability of modules to join and leave (or fail) applications dynamically, and to be developed by independent parties. Programming models for event-driven programming either make it hard to reason about global control flow, thus hampering sound execution, or sacrifice decoupling to aid in reasoning about control flow.

This work fills the gap by introducing a programming model – dubbed cooperative decoupled processes – that achieves both decoupling and reasoning about global control flow. We introduce this programming model through an event calculus, loosely inspired by the Join calculus, that enables reasoning about cooperative decoupled processes through the concepts of pre- and postconditions. A linear type system controls aliasing of events to ensure uniqueness of control flow and thus safe exchange of shared events. Fundamental properties of the type system such as subject reduction, migration safety, and progress are established.

Categories and Subject Descriptors D.1.3 [*Concurrent Programming*]; D.2.4 [*Software/Program Verification*]

Keywords Event-driven programming, decoupling, control flow, aliasing, linear type system

1. Introduction

Events representing happenings such as sensor readings, communication actions, or state changes in software components have long been used for driving interactions in concurrent [37] and distributed [33] systems. Ever since the advent of concurrent operating systems and interrupts, events (and event handlers respectively) have also served as a programming abstraction. In the last decade, event-driven programming [1, 6, 10, 12, 15, 16, 19, 25, 27, 28, 34, 35, 40, 43] has become a major paradigm for developing distributed systems, as demonstrated by the 20-fold increase of the licenses for so-called "messaging" middleware systems [20].

Modularity in event-driven systems. Event-driven systems are organized as collections of decoupled concurrent modules that communicate through *shared events*. *Decoupling* [4, 8, 13, 21, 26, 29, 30, 38] is a key tenet in advocating for event-driven over point-to-point, synchronous interaction. Decoupling is desired in terms of both development-time and runtime characteristics. It enables modules to be developed independently, e.g., without being explicitly bound to other modules but also makes it possible to add new module instances at runtime, or continue operating as existing module instances fail or are removed. However, one cannot just throw together modules that were independently developed. At the least they need to agree on the content of the events they exchange to accomplish some common goal.

Reasoning about global control flow. Decoupling makes it hard to reason about global control flow of event-driven modules because (1) shared events as a means of communication leads to competing recipients [7], and (2) structuring modules as collections of event handlers leads to the problem of *stack management* [42], as interaction logic is fragmented across multiple event handlers. This has led to the emergence of a family of event-driven systems [1, 6, 10, 16, 25, 27, 28, 34, 35, 40, 43] that address these challenges by either using "call-return" [1, 6, 10, 35, 43] or coroutine [16, 25, 27, 28, 34, 40] primitives. Both approaches sacrifice decoupling, i.e., modules are not treated anonymously but hold references to each other. They lead to one-event-one-handler definitions ("call-return" approach), as in rule based systems e.g. event condition action, and a non-general model, i.e., language API dependent (coroutines approach).

To summarise, event-driven programming approaches either emphasise decoupling but do not allow for reasoning about control flow, or address the latter by sacrificing the former. In this paper, we bridge this gap by proposing the model of *cooperative decoupled processes* (CDP for

[Copyright notice will appear here once 'preprint' option is removed.]

short), which supports reasoning about control flow without sacrificing decoupling. Concretely, we propose:

(I) Global control flow for event-driven modules. We model modules as asynchronous concurrent processes. They can express interest in an event or pattern of events and can subsequently receive any event generated by another module that matches their interest. On top of this event-driven communication semantics, called *migration*, CDP models control flow by attaching *pre-/postcondition* transitions to handlers. Preconditions control which handler may take place and postconditions drive the occurrence of the next one. We present E-Calculus, a formal account of the CDP model, which is loosely inspired by the Join calculus [17], contributing towards leveraging the popularity and expressivity of process calculi for the characterisation of modular event-driven systems. We study the control flow formally as a preorder relation over pre-/postconditions attached to handlers. We have encoded a variant of the Join calculus to the E-Calculus. It turns out that (1) preordering of handlers of more than one postconditions in our model is analogous to reflexion in the core (recursive) Join calculus without message exchange over explicit channels, and (2) migration of events between processes is analogous to nesting of processes modulo scoping on free product names.

(II) Safe composition of event-driven modules. Aliasing of events is created when modules are composed together. It is an important feature to model shared events, but may break the uniqueness of control flow. Multiple references to pre-/postconditions means multiple control flows, consequently introducing also conflicts on shared events. Thus, control of aliasing is needed to ensure uniqueness of control flow and safe exchange of shared events when composing modules. This is achieved through a *linear type system* that gives the capability *linear* to pre-/postconditions and *nonlinear* to (possible) shared events, where terms linear and nonlinear are used as in Girard's linear logic [24]. We establish that handler occurrence and exchange of linear and nonlinear events is done in accordance with linearity constraints; control flow is unique and there are no conflicts when shared events are exchanged; the evaluation of a simple and well-typed process will never get stuck.

To recap, the CDP model with the linear type system is a (true) concurrent one and does not suffer from (1) fragmentation of interactions across multiple handlers and (2) explicit conflict resolution of competing recipients on shared events. Pre-/postconditions control the interactions and their linearity prevents conflicts from happening, hence, solving the stack management and shared event problems.

The paper is structured as follows. Section 2 illustrates the problems of stack management and shared events. Section 3 introduces the CDP model and indicates how it addresses these problems. Section 4 presents the E-Calculus and the encoding of a variant of the Join calculus. Section 5 introduces the linear type system and Section 6 states subject reduction, migration safety, and progress. Section 7 surveys related work and Section 8 concludes with an outlook on future work. Omitted definitions, theorems, and proofs can be found in a longer version [3] of this paper.

2. Background and Motivation

In this section, we briefly illustrate the issues mentioned with stack management and shared events.

2.1 Stack Management

We use the example of a fire protection system [2], which consists of two modules: fire alarm and fire suppression. The former includes the interaction of heat and smoke sensors to warn personnel in the event of fire; the latter includes the interaction of smoke sensors to suppress fire by releasing chemical agents. It is crucial that the second interaction occurs after the first: the warning alarm must occur before agents are released to advise personnel for immediate evacuation.

Event-based systems that support reasoning about control flow have traditionally employed manual stack management [1, 6, 10, 35, 43]. Programs are organised as collections of event handlers. Control is defined through handlers that "call" other handlers passing along state, initiating so a new task, and expect a "return" from those to return the control to the originating task. As a result, the control flow for a task and its state are broken across several handlers, discarding so locality features. Programmers must match "call-return" pairs to reason about flow even in scenarios where handlers are present in different parts of the code.

Figure 1(a) gives the model[1] of the fire alarm relying on the syntax of representing events as objects and handlers as functions in a C++ syntax. The task is responsible for detecting smoke and heat in a room by receiving events *smokeDetected* and *heatDetected*, and for driving their transformation into two new notification events *light* and *horn*. Receiving an event is expressed by reading the pointer to the event object. Triggering new events is expressed through the new operator, i.e., creating new event objects. Consumption of an event is expressed by invoking the delete function with the event as an argument; this is specified in the body of the scheduler. Method *myPrint* prints the name (string) of an event object. For presentation reasons, we assume that events received by a handler are present, i.e. we do not check in the code. The model uses a continuation object *Cont* to store (1) the function ($func$) to call, when this continuation is scheduled, (2) the state consisting of events (e), and (3) the continuation object ($\_main$) that stores the function where the control will be returned.

```
class Cont{
  void (*func)(Event *e, Cont *cont);
  Event *e;
  Cont *_main;
  ...//constructor
}
```

[1] The manual stack is modelled as in [1].

```
void H1(Event *e, Cont *c){
  myPrint(smokeDetected);
  Cont *cont = new Cont(&H2, smokeDetected, c);
  scheduler(cont);
}
```

```
void H2(Event *e, Cont *c){
  myPrint(heatDetected);
  Cont *cont = new Cont(&H3, heatDetected, c);
  scheduler(cont);
}
```

```
void H3(Event *e, Cont *c){
  light = new Event("light");
  horn = new Event("horn");
  Cont *cont = new Cont(&H4, null, c);
  scheduler(cont);
}
```

(a)

```
void H4(Event *e, Cont *c){
  myPrint(smokeDetected);
  Cont *cont = new Cont(&H5, heatDetected, c);
  scheduler(cont);
}
```

```
void H5(Event *e, Cont *c){
  releaseGas = new Event("releaseGas");
  scheduler(c);
}
```

(b)

Figure 1. Fire protection in C++: (a) Fire alarm and (b) Fire suppression tasks. * and & denote respectively pointers and addresses of values in C++.

Function H_1 in Figure 1(a) receives event *smokeDetected*, prints it, and creates a continuation that stores the next handler function in the call chain H_2, the event state *smokeDetected*, and the function *c*, to which the control needs to be returned at the end of the handler call chain. H_1 than passes control to the scheduler with *cont* as an argument. The scheduler invokes H_2 along H_1's state (*smokeDetected*, c) as an argument. Similarly, H_2 receives event *heatDetected* and returns control to the scheduler, which invokes H_3 along H_2's state. H_3 triggers events *light*, *horn*. Concurrency between the several tasks is managed by the scheduler.

H_3 does not return the control to the function in *c* that triggered the chain of handlers of the fire alarm task, which would capture the end of the fire alarm task. Instead, it passes *c* to the functions of the fire suppression task (shown in Figure 1(b)) to thus modeling the composition of the two tasks. The fire suppression task consists of receiving event *smokeDetected* and triggering *releaseGas*. The received event denotes detection of smoke in a room and the triggered event denotes the release of gas, i.e., reading, triggering, and consuming events are expressed similarly as in the fire alarm; also for the continuation object *Cont*. Function H_4 receives event *smokeDetected* and returns control to the scheduler, which invokes H_5 along H_4's state. H_5 triggers event *releaseGas* and returns the control to the scheduler on *c*.

The above design fragments the interaction logic over five handlers (functions). It imposes strong coupling between handlers even across tasks: Each handler holds a reference to the next in the flow. Also the handlers explicitly manage control and state of the execution ("manual stack management"). The *Cont* object *c* stores the function, where the control will be return and it is passed on to the other functions.

2.2 Shared Events

Handler functions may compete for shared events, which calls for a mechanism to handle potential conflicts, which, as we will see later, can lead to a series of issues from erroneous behaviours to deadlocks. To illustrate the problem with shared events, consider a scenario when functions H_1 and H_4 depicted above are both active, e.g. the client invokes the scheduler concurrently on both functions. Hence, H_1 and H_4 will compete for event *smokeDetected*.

The manual control of the flow as shown in Figure 1 allows H_4 to be active only after H_1 has consumed its *smokeDetected* and moved to a program state that will not compete with H_4. Thus, the explicit modeling of the control flow supports programmers in handling conflicts between cooperating tasks on shared events, i.e., programmers do not employ any mutexes to model synchronisation. However, the approach is tedious and error prone, as programmers must programmatically match points, where shared events are consumed and new tasks are scheduled. Also, reasoning about control flow does not support programmers in handling conflicts between tasks that are not of the same flow, i.e., two tasks of independent flows can access on some shared state.

3. Cooperative Decoupled Processes

This section discusses how CDP addresses stack management and shared events, while retaining decoupling.

3.1 Decoupling

CDP provides concepts that support decoupling along its three dimensions: space, synchronization and time.

3.1.1 Space Decoupling

The key concept for modelling space decoupling – which roughly posits that modules should not refer to each other [22] – is that of a *site*. A site composes events and handlers into a scoping construct. Handlers of a site describe the logic over many events (a), and compose handlers in disjunction (b). Feature (a) is in the spirit of many modern event-driven systems [4, 26, 29]. Feature (b) models choice, including non-deterministic choice, allowing terms that specify various ways of behaviour. These features allow many events in a site to transform only according to handlers associated to that site, and the site's handlers to drive the transformation of only the site's events. Site is an abstraction that limits the visibility of events, offering a characteristic for modules. Abstraction and information hiding are the basic principles that influence structured programming [14, 36].

For illustration, consider the site of the fire alarm below. The first part (the line following the *def* clause) presents the definition of a handler and the second represents events which currently occur within the site (the code in the *in* clause).

(*Fire alarm site 1*)

```
def smokeDetected & heatDetected ▷ light & horn
in  (smokeDetected & heatDetected)
```

In the handler (1) the & operator specifies parallel composition, (2) names (*smokeDetected*, *heatDetected*) and (*light*, *horn*) are called respectively *reactants* and *products*, (3) the infix operator ▷ distinguishes names denoting reactants from names denoting products, and (4) the arrangement of reactants *smokeDetected* & *heatDetected* is called *join pattern*; in the events in parallel, names (*smokeDetected*, *heatDetected*) denote events and & specifies parallel composition of events. In the example site above, events *smokeDetected* and *heatDetected* occurring in the body are transformed only according to the handler *smokeDetected* & *heatDetected* ▷ *light* & *horn*; in turn, this handler can transform only the events present in the site.

Consider a variant of the fire suppression system that controls fire through a sprinkler or a gaseous system (see below). The suppression system controls fire by using the component detecting it first.

```
def  smokeDetected ▷ releaseGas     (Fire suppression site 1)
  or heatDetected ▷ releaseAgent
in   smokeDetected
```

The infix operator or defines the two different ways the suppression system can behave. If the smoke detector of the gaseous system and the heat detector of the sprinkler system detect an event of fire at the same time, the choice to release inert gases or chemical agents is done non-deterministically. Non-determinism in our model is not limited to the choice of event patterns; handlers having the same reactants but different products are also chosen non-deterministically.

The example illustrates how CDP supports space decoupling: Neither site has a reference to the other and knows with how many instances it is interacting.

3.1.2 Synchronisation and Time Decoupling

Synchronisation [13] and time [22] decoupling is supported through the way in which we model the interaction between sites. Sites communicate by (1) letting out (also reversible) events that do not participate in any of their handlers, and (2) letting in events that can participate in any of their handlers. To achieve this form of interaction, we allow to express events outside a site, called *global events*, that model events shared by handlers of different sites, in contrast to events that are local that model events shared by handlers of a single site.

Events that are bound by reactants are trapped within a site by handlers and can be transformed by parallel composition. Conversely, events that are not bound, including product events, are not trapped in the site and so, cannot transform, giving them the ability to *leave and (re)enter* the site. For an event to leave a site means that it composes in parallel with the site. For example, after a *handler occurrence* in site *(1)*, the updated site contains the new product events

(*Fire alarm site 2*)

```
def smokeDetected & heatDetected ▷ light & horn
in  (light & horn)
```

where events *light* and *horn* are not bound by any reactant so they can leave and reenter the site without restriction. Below, event *light* has left the site, becoming global

(*Fire alarm site 3*)

```
(def smokeDetected & heatDetected ▷ light & horn
 in  horn) & light
```

The other fundamental feature to achieve migration is by *dynamically adding* global events into a site. A global event can be added into a site only if it is bound by one of the reactants present in the handlers. Consider a fire alarm site where the event *smokeDetected* is stated as global and so is composed in parallel with the site as in:

```
  smokeDetected
& (def smokeDetected & heatDetected ▷ light & horn
   in  heatDetected)
```

smokeDetected can be added to the site since it is bound by a reactant of the handler. This results in the *Fire alarm site 1* that now contains both events locally.

In this form of interaction, the possible recipient sites of a global event may not be active in the moment when the global event is produced; i.e., those sites may not be yet "connected" with the others. Also, when an event leaves or enters a site, handlers in those sites can occur in the meantime.

3.2 Enabling Declarative Safe Composition

In here, we present the features of our composition model: (a) modelling control flow between handlers, (b) exploiting (multiprocessor) parallelism and (c) implicitly avoiding conflicts between recipients of shared events and its safety properties.

def [*init*] & *smokeDetected* & *heatDetected* ▷ [$fireAlarm$] & *light* & *horn*
in *smokeDetected* & *smokeDetected*

(a)

def [$fireAlarm$] & *smokeDetected* ▷ [$final1$] & *releaseGas*
or [$fireAlarm$] & *heatDetected* ▷ [$final2$] & *releaseAgent*
in *smokeDetected*

(b)

Figure 2. Fire protection in CDP: (a) Fire alarm (*Fire alarm site 4*) and (b) Fire suppression tasks (*Fire suppression site 2*).

To model control flow, handlers in our model do not only capture observed events to trigger new events; they also capture *control events* that express a flow through their occurrences. The control events are also called pre-/postconditions events; they are declared within square brackets to distinguish them from other reactants and products. Brackets regard the syntax of handlers and the static semantics, and have no meaning at runtime.

Figure 2 gives the model of the fire protection system in CDP. We extend the handler of the *Fire alarm site 1* with the *init* precondition on reactants and $fireAlarm$ postcondition on products. *init* is an event that is always bound to the first handler in the system, denoting the precondition before any handler occurrence. Also, we extend the handlers of the *Fire Suppression 1* with the $fireAlarm$ precondition, and $final1$ and $final2$ postconditions. The flow between the fire alarm and suppression handlers is represented by including the postcondition of the alarm handler as a precondition in the suppression handlers. Postcondition events use the concept of migration to leave their birth sites and join the sites of the next handlers that declare them as precondition events.

A handler h_1 occurrence precedes another h_2 if some of h_1's postconditions are preconditions of h_2. For example, handler [*init*] & *smokeDetected* & *heatDetected* ▷ [$fireAlarm$] & *light* & *horn* precedes ($\precsim$) [$fireAlarm$] & *smokeDetected* ▷ [$final1$] & *releaseGas*. Our control flow is formalised as a preorder relation, i.e., "almost" a partial order, since anti-symmetry does not hold. Consider the ordering of three handlers as [*a*] ▷ [*b*] $\precsim$ [*b*] ▷ [*c*] $\precsim$ [*c*] ▷ [*a*] by definition of $\precsim$. By transitivity, we conclude that [*a*] ▷ [*b*] $\precsim$ [*c*] ▷ [*a*] and, by definition of $\precsim$, we have that [*c*] ▷ [*a*] $\precsim$ [*a*] ▷ [*b*]. However, handlers [*c*] ▷ [*a*] and [*a*] ▷ [*b*] are not equal. In our system, two handlers are equal iff they have the same preconditions, reactants, postconditions, and products. Hence, our ordering does not hold anti-symmetry, meaning that three handlers forming a loop do not imply that the first and third are the same.

Control events establish the flow of a program. Therefore, the following uniqueness invariant must hold for them:

(I) a precondition is always unique to only one site,
(II) a postcondition is always unique to only one handler, and
(III) there is always a unique link postcondition-precondition.

The invariant allows a precondition to appear in different handlers of only one site to model non-determinism (see Figure 2(b)); for a postcondition, it can appear only in one handler. To illustrate the importance of establishing this invariant, we consider two versions of the fire suppression system, one that has a duplicated control event and another one that lacks control due to lack of a unique postcondition-precondition link. An example for the first version is the system defined over the following two handlers:

[*init*] & *smokeDetected* & *heatDetected* ▷ [*init* & $fireAlarm$] & *light* & *horn*

[$fireAlarm$] & *smokeDetected* ▷ [*init* & $fireAlarm$] & *releaseGas*

In this version, there is a copy of control event in each site. As there may be an arbitrary number of concurrently running clients of fire alarm and suppression, the two sites will run forever without any cooperation ever being established.

An example of the second version of the system is the one defined over the handlers below, in which the precondition is not *init* and does not relate to any postcondition. In such a system, clients are blocked within the sites.

[*a*] & *smokeDetected* & *heatDetected* ▷ [*b*] & *light* & *horn*

[$fireAlarm$] & *smokeDetected* ▷ [$final$] & *releaseGas*.

Unlike in systems that dot not support establishing the uniqueness invariant, the CDP model ensures this statically by a type system. Hence, pre-/postconditions are placed within brackets. One can easily verify that the fire protection at the beginning holds the invariant.

In addition to modelling control flow, our model involves further aspects: Exploiting (multiprocessor) parallelism and avoiding conflicts on shared events. Two features of our model address these aspects: (1) modules are allowed to yield control to more than one module, and (2) only modules that have control can receive shared events, i.e., control events serve as an implicit lock mechanism to prevent other modules of a flow from receiving shared events. Feature (1) is modelled by having transitions from many preconditions to many postconditions. Feature (2) is modelled by supporting addition of global events to sites that contain the precondition events of the handler referencing them. This means that the invariant on uniqueness of control events also guarantees safety of conflict resolution between recipients, similarly to the uniqueness of locks.

Below, we illustrate how these features address parallelism and conflicts on shared events, again by looking at variants of the fire suppression system.

To illustrate support for parallelism consider how one can model a variant of the fire suppression with two components, one using a sprinkler and another one using a gaseous system, both running in case of fire, consequently exploiting parallelism. To enable this, their handlers can be modelled with different preconditions, e.g. $fireAlarm1$ and $fireAlarm2$, and the postconditions of the handler of *Fire alarm site 4* must reflect both names as [$fireAlarm1$ & $fireAlarm2$].

$P ::=$ Processes: | $\mathsf{def}\ D\ \mathsf{in}\ M$ Sites

$\quad | \ l$ Global events | $P \ \& \ P$ Parallel

$D ::= [J] \ \& \ J \triangleright [J] \ \& \ J$ Handlers | $D\ \mathsf{or}\ D$ Disjunction

$J ::= l \mid J \ \& \ J$ Join names

$M ::= \mathbf{0} \mid l \mid M \ \& \ M$ Local events

Figure 3. Syntax of processes.

To illustrate support for issues of shared events consider a fire protection system, where the sensors of smoke are not local to the modules of fire alarm and suppression. This is modelled as two sites composed in parallel with the global event *smokeDetected*. Also, the precondition event *init* is present in the site of the fire alarm. That is, the following module

$$\begin{aligned} &\mathit{smokeDetected}\ \&\ (\mathsf{def}\ [\mathit{init}]\ \&\ \mathit{smokeDetected}\ \&\ \mathit{heatDetected} \\ &\qquad \triangleright [\mathit{fireAlarm}]\ \&\ \mathit{light}\ \&\ \mathit{horn} \\ &\quad \mathsf{in}\ \ \mathit{init}\ \&\ \mathit{heatDetected}) \end{aligned}$$

in parallel with *Fire suppression site 2*. A conflict between the two modules raises on adding the shared event. Such conflicts can lead to a deadlock; e.g., if the event is erroneously added in the fire suppression site, no handler will be able to occur. However, this cannot happen, since events are only added to sites that have a precondition that holds, which is not the case with the fire suppression site. This event is thus added into the fire alarm site, where the precondition is present, permitting the fire alarm handler to occur.

4. The E-Calculus

The E-Calculus is based on a small-step operational semantics, defined over inference rules and a relation of structural congruence.

4.1 Syntax

The metavariable P stands for processes; l ranges over names of events, reactants, products, preconditions and postconditions; D stands for handler declarations; M stands for local events; J stands for join names. Metavariables with subscript denote the same class of terms; e.g., P_1 denotes a process.

Figure 3 gives the syntax of processes. A global event represents a process. Parallel composition is standard, composing in parallel the behaviour of two or more processes. A site is a process definition inspired by the Join calculus [17], that, in contrast, does not allow reflexion and nesting of sites to ensure decoupling. A handler is a blueprint that describes transformation of events. Pre-/postconditions denote join names J. Local events include events in parallel or $\mathbf{0}$. The latter is used to express a site without local events. Join names J denote non-empty patterns of names (preconditions, postconditions, reactants, or products). The association of the parallel operator & is the weakest over $\mathsf{def}\ D\ \mathsf{in}\ M$ as defined below.

Definition 1. The set of free names of a process P, written $fn(P)$, and the set of defined names of a declaration D, written $dn(D)$, are defined as follows:

$$\begin{aligned} &fn(l) = \{l\} \quad fn(P_1 \ \& \ P_2) = fn(P_1) \bigcup fn(P_2) \\ &fn(\mathsf{def}\ D\ \mathsf{in}\ M) = fn(D) \bigcup fn(M) \setminus dn(D) \\ &fn([J_1] \ \& \ J_2 \triangleright [J_3] \ \& \ J_4) = dn(J_1) \bigcup dn(J_2) \bigcup fn(J_3) \bigcup fn(J_4) \\ &fn(J_1 \ \& \ J_2) = dn(J_1) \bigcup fn(J_2) \\ &fn(D_1\ \mathsf{or}\ D_2) = fn(D_1) \bigcup fn(D_2) \\ &fn(M_1 \ \& \ M_2) = dn(M_1) \bigcup fn(M_2) \quad fn(\mathbf{0}) = \emptyset \\ &dn([J_1] \ \& \ J_2 \triangleright [J_3] \ \& \ J_4) = dn(J_1) \bigcup dn(J_2) \\ &dn(D_1\ \mathsf{or}\ D_2) = dn(D_1) \bigcup dn(D_2) \\ &dn(l) = \{l\} \quad dn(J_1 \ \& \ J_2) = dn(J_1) \bigcup dn(J_2) \end{aligned}$$

The formal definition of control flow is a chain of handlers defined over a preordering relation:

Definition 2 (Preorder). Write $h, h', ...$ for handlers of a program. Then we write $h \precsim h'$ when the occurrence of h precedes the one of h'. Formally $\precsim$ is the preorder such that $[J_1] \ \& \ J_2 \triangleright [J_3] \ \& \ J_4 \precsim [J_1'] \ \& \ J_2' \triangleright [J_3'] \ \& \ J_4'$ if $dn(J_3) \cap dn(J_1') \neq \emptyset$. For all h, h', h'' in a program, we have:

- (reflexivity) $h \precsim h$
- (transitivity) if $h \precsim h'$ and $h' \precsim h''$ then $h \precsim h''$

We write h^t for the transition associated to handlers h, i.e., parts containing only the pre- and postconditions: $[J] \triangleright [J']$.

Definition 3 (Chain of handlers). A chain of handlers, written A, is defined as $[h_1^t, ..., h_n^t]$, such that for $\forall i \in [2, .., n]$, we have that $h_j \precsim h_i$ for $j \in [1, ..., i-1]$.

$$P \ \& \ Q \equiv Q \ \& \ P \ \lfloor \textsc{Str-P-Com} \rfloor$$
$$P \ \& \ (Q \ \& \ R) \equiv (P \ \& \ Q) \ \& \ R \ \lfloor \textsc{Str-P-Assoc} \rfloor$$
$$\mathsf{def}\ D\ \mathsf{in}\ M \equiv \mathsf{def}\ D\ \mathsf{in}\ M' \ \text{if}\ M \equiv M' \ \lfloor \textsc{Str-D-P} \rfloor$$
$$\mathsf{def}\ D\ \mathsf{or}\ E\ \mathsf{in}\ M \equiv \mathsf{def}\ E\ \mathsf{or}\ D\ \mathsf{in}\ M \ \lfloor \textsc{Str-D-Com} \rfloor$$
$$\mathsf{def}\ D\ \mathsf{or}\ (E\ \mathsf{or}\ F)\ \mathsf{in}\ M \equiv \mathsf{def}\ (D\ \mathsf{or}\ E)\ \mathsf{or}\ F\ \mathsf{in}\ M \ \lfloor \textsc{Str-D-Assoc} \rfloor$$
$$\mathsf{def}\ D\ \mathsf{in}\ M_1 \ \& \ M_2 \equiv \mathsf{def}\ D\ \mathsf{in}\ (M_1 \ \& \ M_2) \quad \text{if}\ \ dn(D) \cap fn(M_2) = \emptyset \ \lfloor \textsc{Str-Def} \rfloor$$
$$M \ \& \ \mathbf{0} \equiv M \ \lfloor \textsc{Str-M-Inact} \rfloor \quad M_1 \ \& \ M_2 \equiv M_2 \ \& \ M_1 \ \lfloor \textsc{Str-M-Com} \rfloor$$
$$M_1 \ \& \ (M_2 \ \& \ M_3) \equiv (M_1 \ \& \ M_2) \ \& \ M_3 \ \lfloor \textsc{Str-M-Assoc} \rfloor$$

Figure 4. Structural congruence.

Structural congruence is the smallest congruence on processes that satisfies the axioms shown in Figure 4. It defines all processes that have the same behaviour but different syntax. $\lfloor$Str-P-Com$\rfloor$, $\lfloor$Str-P-Assoc$\rfloor$, $\lfloor$Str-D-Com$\rfloor$, $\lfloor$Str-D-Assoc$\rfloor$, $\lfloor$Str-M-Com$\rfloor$, and $\lfloor$Str-M-Assoc$\rfloor$ define that parallel composition of processes, declarations, and local events is commutative and associative. $\lfloor$Str-D-P$\rfloor$ defines structural congruence, i.e. associativity, commutativity, and identity, over local events in a site. $\lfloor$Str-Def$\rfloor$[2] allows an event not bound by any handler to leave and (re)enter the birth site, reading the rule from the left to the right and from the right to the left. $\lfloor$Str-M-Inact$\rfloor$ defines $\mathbf{0}$ as the identity element of local events with respect to parallel composition.

[2] For presentation reasons without affecting the correctness of the operational and static semantics, we write M_2 instead of writing l_1 & & l_n. This style is similar to the reduction rule in the Join calculus [17].

$$\mathsf{def}\ [J_1]\ \&\ J_2 \rhd [J_3]\ \&\ J_4\ \mathsf{in}\ (J_1\ \&\ J_2\ \&\ M) \longrightarrow \mathsf{def}\ [J_1]\ \&\ J_2 \rhd [J_3]\ \&\ J_4\ \mathsf{in}\ (J_3\ \&\ J_4\ \&\ M) \quad fn(J_3) \cap fn(M) = \emptyset \ \lfloor \text{R-Occ} \rfloor$$

$$l\ \&\ \mathsf{def}\ [J_1]\ \&\ J_2 \rhd [J_3]\ \&\ J_4\ \mathsf{in}\ M \longrightarrow \mathsf{def}\ [J_1]\ \&\ J_2 \rhd [J_3]\ \&\ J_4 \mathsf{in}\ (l\ \&\ M) \quad l \in dn(J_1) \quad \lfloor \text{R-AddPre} \rfloor$$

$$l\ \&\ \mathsf{def}\ [J_1]\ \&\ J_2 \rhd [J_3]\ \&\ J_4\ \mathsf{in}\ M \longrightarrow \mathsf{def}\ [J_1]\ \&\ J_2 \rhd [J_3]\ \&\ J_4 \mathsf{in}\ (l\ \&\ M) \quad dn(J_1) \subseteq fn(M) \text{ and } l \in dn(J_2) \quad \lfloor \text{R-AddRea} \rfloor$$

$$\mathsf{def}\ D\ \mathsf{or}\ E\ \mathsf{in}\ M \longrightarrow \mathsf{def}\ D\ \mathsf{or}\ E\ \mathsf{in}\ M' \ \text{ if } \ \mathsf{def}\ D\ \mathsf{in}\ M \longrightarrow \mathsf{def}\ D\ \mathsf{in}\ M' \ \lfloor \text{R-DisO} \rfloor$$

$$l\ \&\ \mathsf{def}\ D\ \mathsf{or}\ E\ \mathsf{in}\ M \longrightarrow \mathsf{def}\ D\ \mathsf{or}\ E\ \mathsf{in}\ (l\ \&\ M) \ \text{ if } \ l\ \&\ \mathsf{def}\ D\ \mathsf{in}\ M \longrightarrow \mathsf{def}\ D\ \mathsf{in}\ (l\ \&\ M) \ \lfloor \text{R-DisA} \rfloor$$

$$P\ \&\ Q \longrightarrow P'\ \&\ Q \ \text{ if } \ P \longrightarrow P' \ \lfloor \text{R-Par} \rfloor \qquad P \equiv Q \longrightarrow Q' \equiv P' \ \text{ if } \ P \longrightarrow P' \ \lfloor \text{R-Str} \rfloor$$

Figure 5. Operational semantics.

4.2 Operational Semantics

Figure 5 gives the operational semantics of the E-Calculus via the reduction relation $\longrightarrow$ where the state of the machine is defined only by terms of the calculus, written $P \longrightarrow P'$; this reads "*process P reduces to process P′ in one step*". A handler occurrence is the act of checking for the presence of a pattern from a set of events that matches the pattern of preconditions and reactants, and substituting the event pattern with postcondition and product events as defined in rule ⌊R-Occ⌋. Local events M define additional events that are not copies of postcondition events, allowing terms of larger patterns to reduce. The side condition is necessary in proving the coherence of the dynamic and static semantics. Global events can be added to a site following rules ⌊R-AddPre⌋, ⌊R-AddRea⌋: the first rule expresses addition of precondition events and the second expresses addition of reactant events while solving conflicts between recipients, i.e., a reactant event (possible shared) is added if the control events are present in the site. Rules ⌊R-DisO⌋, ⌊R-DisA⌋ allow to infer the handler that, respectively, matches a pattern of events and allows the addition of a global event from the composed handlers. ⌊R-Par⌋ defines computation in processes composed in parallel, reducing first each subprocess. ⌊R-Str⌋ defines reduction on structurally congruent terms.

4.3 Encoding a Variant of the Core (Recursive) Join calculus

The E-Calculus can express control of extending the machine with reducible sites (reflexion) as in the core (recursive) Join calculus [17] without message-passing over explicit channels. Reflexion in the Join calculus is the ability of reactions (our handlers) to extend the machine with new molecules (our patterns of events) along reaction rules. Our purpose here is to show that (1) reducible sites can be enabled through the pre-/postcondition mechanism rather than dynamically adding them into the solution, and (2) communication between sites can be done using migration rather than nested definitions, providing a model of computation that offers decoupling.

We firstly translate the syntax of (a) the core Join calculus into (b) a variant of it without message (l) exchange over explicit channels using the function $[\![P]\!]_l = \mathcal{P}$. Intuitively, the translation removes names attached to reactants and products and names associated to global names. For presentation reasons, the encoding of reflexion and nesting is done in two steps. That is, we encode reflexion of the variant $(\mathcal{P}, \mathcal{Q})$ of the Join calculus into a variant (P) of the E-Calculus without disjunction but with nesting and then encode nesting as in P into the E-Calculus. The first encoding, written as $[\![\mathcal{P}]\!]_J = \mathrm{P}$, is given below

$$\mathrm{P, Q} ::= l \mid \mathrm{P}\ \&\ \mathrm{Q} \mid \mathsf{def}\ [J_1]\ \&\ l\ \&\ k\ \rhd [J_2]\ \&\ J_3\ \mathsf{in}\ \mathrm{P}$$

$$\text{(I)}\ [\![l]\!]_J = l \qquad \text{(II)}\ [\![\mathcal{P}_1 \mid \mathcal{P}_2]\!]_J = [\![\mathcal{P}_1]\!]_J\ \&\ [\![\mathcal{P}_2]\!]_J$$

$$\begin{aligned}\text{(III)}\ [\![&\mathsf{def}\ l \mid k\ \rhd\ (J'\ \&\ \mathsf{def}\ l_1 \mid k_1\ \rhd \mathcal{P}_1\ \mathsf{in}\ \mathcal{Q}_1\ \&\ \ldots \\ &\qquad \&\ \mathsf{def}\ l_n \mid k_n\ \rhd \mathcal{P}_n\ \mathsf{in}\ \mathcal{Q}_n) \\ &\mathsf{in}\ \mathcal{P}]\!]_J = \mathsf{def}\ [J]\ \&\ l\ \&\ k\ \rhd [l'_1\ \&\ \ldots \& l'_n]\ \&\ J'\ \mathsf{in}\ [\![\mathcal{P}]\!]_{l'}\ \& \\ &[\![\mathsf{def}\ l_1\ \&\ k_1\ \rhd \mathcal{P}_1\ \mathsf{in}\ \mathcal{Q}_1]\!]_{l'_1}\ \&\ \ldots\ \&\ [\![\mathsf{def}\ l_n\ \&\ k_n\ \rhd \mathcal{P}_n\ \mathsf{in}\ \mathcal{Q}_n]\!]_{l'_n}\end{aligned}$$

where $n \geq 0$ and the index J denotes the preconditions when encoding a Join-like reaction into a handler. Rules (I) and (II) define encodings of a name and parallel composition. Rule (III) defines encoding for a Join-like site, where the product is defined by names J' and sites $\mathsf{def}\ l_i\ \&\ k_i\ \rhd \mathcal{P}_i\ \mathsf{in}\ \mathcal{Q}_i$ for $i \in [1..n]$ in parallel; if $n = 0$ then there are no sites in the product. The encoded "main" Join reaction contains J as precondition and n postconditions (fresh names) l'_i as n sites; the process $\mathcal{P}$ in the scope of the reaction is not part of the reflexion semantics so it is encoded inductively with a different index l'. Sites are encoded in the solution with an index that corresponds to one of the postconditions. Each of the postconditions relates to the precondition for the handler of each site. Intuitively, this means that sites are reducible only after the "main" handler occurs. The image of the translation is the *forking subset of the* E-Calculus, i.e., transitions associated to handlers are defined by one precondition and many postconditions. As expected, reductions in this subset only produce new flows. The initial encoding index is *init*: $[\![\mathcal{P}]\!]_{init}$.

Nested process definitions can be distilled into flat definitions composed in parallel up to the scoping of free product names as:

$$\begin{aligned}&[\![\mathsf{def}\ [J_1]\ \&\ l\ \&\ k\ \rhd [J_2]\ \&\ J_3\ \mathsf{in}\ (\mathrm{P}_1\ \&\ \mathrm{P}_2)]\!] = \\ &(\mathsf{def}\ [J_1]\ \&\ l\ \&\ k\ \rhd [J_2]\ \&\ J_3\ \mathsf{in}\ [\![\mathrm{P}_1]\!])\ \&\ [\![\mathrm{P}_2]\!]\end{aligned}$$

where $\mathrm{P}_2 \equiv \mathsf{def}\ [J'_1]\ \&\ l'\ \&\ k'\ \rhd [J'_2]\ \&\ J'_3\ \mathsf{in}\ \mathrm{P}$. This rule is similar to our congruence rule ⌊Str-Def⌋. One can observe that names $\{J_1, l, k\}$ loose the scope on $[\![\mathrm{P}_2]\!]$. They can gain the scope on free events in $[\![\mathrm{P}_2]\!]$ using migration but not on the free products (J'_2 and J'_3).

5. The Linear Type System

This section gives an informal description of the challenges addressing aliasing in the calculus and a static type system.

5.1 The Addressed Challenges

In the following, we consider scenarios, where uncontrolled aliasing can break the uniqueness of control flow and safe exchange of shared events. Aliased events can also be called *nonlinear*, in contrast to nonaliased events called *linear*. For presentation reason, we present challenges through examples consisting of one precondition and one postcondition per handler without diluting their complexity.

Consider a global event k coming from an unbound product of a previous handler occurrence or stated as global. k is referenced by a reactant and also a precondition of two different sites (regardless of the same or different flows) as

$$k \;\&\; \mathsf{def}\ [k] \;\&\; M' \rhd [k']\ \mathsf{in}\ M'$$
$$\&\; \mathsf{def}\ [l] \;\&\; k \;\&\; M \rhd [l']\ \mathsf{in}\ (l \;\&\; M)$$

The operational semantics allows to add k in one of the two sites, i.e., it can be added in the first since it is referenced by a precondition and in the second since it is referenced by a reactant. Hence, two possible control flows rise. Since k comes from an unbound product, it is not a control event. Thus, adding it to the first site breaks the control flow of both handlers. Here, aliasing can be controlled by distinguishing between consumers (recipients) of events, preconditions and reactants, and producers of events, postconditions and products. This problem is relevant also if both handlers are part of a single site

$$k \;\&\; \mathsf{def}\quad [k] \;\&\; M' \rhd [k']$$
$$\mathsf{or}\ [l] \;\&\; k \;\&\; M \rhd [l']\ \mathsf{in}\ (M' \;\&\; l \;\&\; M)$$

or when k is stated as local to the first site. In the latter, stated events do not define control events with the exception of *init* so they can be distinguished as produced by products.

A similar scenario is when event k is produced by a postcondition, i.e., it is a control event (in here, handler are part of different flows). Thus, adding it to the second site breaks the control flow of both handlers. Aliasing here can be controlled with the same mechanism as above.

A shared event t that is referenced by reactants of two handlers of the same flow

$$t \;\&\; \mathsf{def}\ [l] \;\&\; t \rhd [k]\ \mathsf{in}\ l$$
$$\&\; \mathsf{def}\ [k] \;\&\; t \rhd [k']\ \mathsf{in}\ \mathbf{0}$$

presents a challenge of aliasing that is resolved by the operational semantics; i.e., t is added only in the first site, which has control of the flow at that point (precondition event). As expected, the first handler returns control to the second handler. The expectations will also hold for the case when both handlers are part of a single site. However, the operational semantics cannot resolve a similar scenario where handlers are part of different flows

$$t \;\&\; \mathsf{def}\ [l] \;\&\; t \rhd [l']\ \mathsf{in}\ l$$
$$\&\; \mathsf{def}\ [k] \;\&\; t \rhd [k']\ \mathsf{in}\ k$$

The operational semantics allows to add t in one of the two sites, i.e., both sites contain the precondition events. Thus, both sites are possible recipients, rising a conflict between them. Here, aliasing can be controlled by distinguishing between recipients and producers of events of different flows. This problem is relevant also if both handlers are part of the same site. However, if t is stated local to each site, i.e., it is not shared, then no issues arise.

$$\mathsf{def}\ [l] \;\&\; t \rhd [l']\ \mathsf{in}\ (l \;\&\; t)$$
$$\mathsf{def}\ [k] \;\&\; t \rhd [k']\ \mathsf{in}\ (k \;\&\; t)$$

To address the challenges presented here and Section 3.2, a type system is introduced. The type system should not only control aliasing of existing events at definition time but also of the ones that may be created during evaluation to ensure uniqueness of control events and safe exchange (no conflict) of shared events in all steps of computation. So, static control of aliasing should address also blueprints of events. In short, at different handlers, the type system forbids terms of the following form:

1. the same name for more than one precondition or by more than one postcondition,
2. a name for precondition without a copy for a postcondition, except the name *init*,
3. products referenced by preconditions,
4. postconditions referenced by reactants,
5. products of a flow referenced by more than one reactant of other flows, and
6. precondition of the first handler in each flow is not *init*.

5.2 Typing Rules

On the typing level, we introduce linear types L and nonlinear types N for a name. A linear type gives the capability to a name to be used only once and a nonlinear type gives a capability with no multiplicity constraints. L can take two forms: l_r, l_p to distinguish between preconditions and postconditions. Similarly, N can take two forms: r, p to distinguish between reactants and products. We introduce also flow typing $\Delta ::= \emptyset \mid A, \Delta$ to track the flows where a name is used.

Type environment and judgments A type environment Γ is a finite set of linear (L) and nonlinear (N) type assignments to names (l) of the form $l : L@\Delta$, $l : N@\Delta$, where all names are different. Following, we give the definition: $\Gamma ::= \emptyset \mid \Gamma, l : l_r@\Delta \mid \Gamma, l : l_p@\Delta \mid \Gamma, l : r@\Delta \mid \Gamma, l : p@\Delta$. We write Γ_1, Γ_2 for the union of two type environments, whereby the names of Γ_1 and Γ_2 are disjoint. Type judgments are of the form $\Gamma \vdash P$ for processes, $\Gamma \vdash D$ for declarations, $\Gamma \vdash_{pre} C$ for preconditions, $\Gamma \vdash_r M$ for reactants, $\Gamma \vdash_{post} C$ for postconditions, and $\Gamma \vdash_p M$ for products. In addition to union (defined informally by ","), we define the combination operations on types and environments. They ensure that linear capabilities are exercised to guarantee uniqueness of control events and a safe exchange of shared events in the presence of different flows. We distinguish between combination operators for type environments of preconditions ($\uplus_{pre}$), reactants ($\uplus_r$), preconditions and reactants ($\uplus_{rea}$), postconditions ($\uplus_{post}$), products ($\uplus_p$), postconditions and products ($\uplus_{pro}$), handlers ($\uplus_H$), disjunction

$l_r@A \uplus_{pre} l_r@A \triangleq \text{err}$ $\quad l_p@A \uplus_{post} l_p@A \triangleq \text{err}$ $\quad r@A \uplus_r r@A \triangleq r@A$ $\quad p@A \uplus_p p@A \triangleq p@A$

$l_r@A \uplus_{rea} r@A \triangleq \text{err}$ $\quad l_p@A \uplus_{pro} p@A \triangleq \text{err}$

$\uplus_H$	$l_p@A$	$p@A$
$l_r@A$	$l_r l_p@A$	err
$r@A$	err	$rp@A$

$\uplus_S$	$l_r@\Delta$	$l_r l_p@A$	$r@\Delta$	$l_p@A$	$p@\Delta$	$rp@\Delta$
$p@[]$	err	err	err/$rp@A_{if A=\Delta}$	err	$p@\Delta$	$rp@\Delta$

$\uplus_D$	$l_r@\Delta$	$l_r l_p@A$	$r@\Delta$	$l_p@A$	$p@\Delta$	$rp@\Delta$
$l_r@\Delta'$	$l_r@\Delta, \Delta'$	err	err	$l_r l_p@A \circ \Delta'$	err	err
$l_r l_p@B$	err	err/$l_r l_p@A_{if A=B}$	err	err	err	err
$r@\Delta'$	err	err	$r@\Delta, \Delta'$	err	err/$rp@\Delta, B_{if \Delta'=B}$	err
$l_p@B$	$l_r l_p@B \circ \Delta$	err	err	err	err	err
$p@\Delta'$	err	err	err/$rp@\Delta', A_{if \Delta=A}$	err	$p@\Delta, \Delta'$	$rp@\Delta, \Delta'$
$rp@\Delta'$	err	err	err	err	$rp@\Delta, \Delta'$	err

$l_r@\Delta, l_r l_p@A \uplus l_r@\Delta', l_r l_p@B \triangleq \text{err}$ $\quad l_r@\Delta, l_p@A \uplus r@\Delta', p@\Delta'', rp@\Delta''' \triangleq \text{err}$ $\quad l_r l_p@A \uplus r@\Delta, p@\Delta', rp@\Delta'' \triangleq \text{err}$

$p@\Delta \uplus r@\Delta' \triangleq \text{err} \quad |\Delta'| \geq 2$ $\quad l_r l_p@A, l_p@B \uplus l_p@C \triangleq \text{err}$

Figure 6. Combination of types.

($\uplus_D$), site ($\uplus_S$) and processes ($\uplus$) of the same and different flows. Below, $\biguplus$ denotes $\uplus_{pre}, \uplus_r, \uplus_{rea}, \uplus_{post}, \uplus_p, \uplus_{pro}, \uplus_H, \uplus_D, \uplus_S$ and $\uplus$.

Definition 4 (Combinations of type environments). $\Gamma_1 \biguplus \Gamma_2$ is defined as:

$$(\Gamma_1 \biguplus \Gamma_2)(l) \triangleq \begin{cases} \Gamma_1(l) & \text{if } l \in dom(\Gamma_1) \text{ and } l \notin dom(\Gamma_2) \\ \Gamma_2(l) & \text{if } l \in dom(\Gamma_2) \text{ and } l \notin dom(\Gamma_1) \\ \Gamma_1(l) \biguplus \Gamma_2(l) & \text{if } l \in dom(\Gamma_1) \text{ and } l \in dom(\Gamma_2) \\ \text{undefined} & \text{otherwise} \end{cases}$$

These operations are associative and commutative. They are not always defined if exists $l \in dom(\Gamma_1)$ and $l \in dom(\Gamma_2)$ such that $\Gamma_1(l) \biguplus \Gamma_2(l) = \text{err}$. This is the case, whenever $\Gamma_1(l)$ and $\Gamma_2(l)$ capture harmful aliasing scenarios listed in Section 5.1. These operations are similar to the product rule in Girard's linear logic [24].

Figure 6 gives the combination of types coming from environments in typing rules. Rules of $\uplus_{pre}$ and $\uplus_{post}$ prohibit the same name in more than one respectively precondition and postcondition. These rules enforce only one copy of precondition and postcondition in a handler. Rules of $\uplus_r$ and $\uplus_p$ combine names denoting respectively reactant and products, allowing multiple copies. These rules do not enforce any constraint on the number of copies of a reactant and product in a handler. Rules of $\uplus_{rea}$ and $\uplus_{pro}$ prohibit combination of preconditions and reactants types, and postconditions and products types. Rules of $\uplus_H$ (a) combine precondition and postcondition types into a linear type $l_r l_p$ and reactants and products into a nonlinear type rp and (b) prohibit combination of preconditions with products and postconditions with reactants types. The rules so far are defined over the flow consisting of only the handler in consideration. The following ones consider more than one handler; hence, (possibly) involving many flows.

Rules of $\uplus_D$ (1) ensure linearity of pre-/postconditions in handlers composed in disjunction, (2) prohibit combination of linear and nonlinear types, and (3) prohibit conflicts between recipients of different flows and a product (however the rules allow the combination of one reactant and many products of different flows). In here, l_r types can combine since names are composed in disjunction, not breaking so their single-copy capability. The concatenation of two flow typing is defined as $\{A_1, ..., A_n\} \circ B = \{A_1 \circ B, ..., A_n \circ B\}$ where $[h_1, ..., h_n] \circ [h'_1, ..., h'_m] = [h_1, ..., h_n, h'_1, ..., h'_m]$ such that $h_n \precsim h'_1$.

Rules of $\uplus_S$ prohibit combination of linear types to nonlinear type p, prohibiting so conflicts between recipients of different flows and a product, and combine nonlinear types. For presentation reasons, we present the rules that prohibit combination of types for processes in parallel. Rules of $\uplus$ prohibit (I) the same name for more than one precondition or by more than one postcondition on different sites, (II) referencing of products by preconditions and postconditions referenced by reactants, and (III) products of a flow referenced by reactants of other flows. So, Γ is extended with new non-/linear types: $\Gamma ::= ... \mid \Gamma, l : l_r l_p@\Delta \mid \Gamma, l : rp@\Delta$.

Typing rules for terms of the E-Calculus are presented in Figure 7. Rule ⌊T-Init⌋ assigns a linear type $l_p@[]$ to the special event *init* in the environment: the empty flow typing [] is assigned to every stated event as a distinguished typing. Rule ⌊T-GName⌋ assigns a nonlinear type $p@[]$ to a stated event, either local or global, in the environment. For two processes in parallel (⌊T-Par⌋) to be well-typed, the subprocesses must be well-typed and the environments must combine accordingly $\uplus$. For a site (⌊T-Site⌋) to be

$$init : l_p@[] \vdash init \ \lfloor\text{T-Init}\rfloor \qquad l : p@[] \vdash l \ \lfloor\text{T-GName}\rfloor$$

$$\frac{\Gamma \vdash P \quad \Gamma' \vdash Q}{\Gamma \uplus \Gamma' \vdash P \,\&\, Q} \ \lfloor\text{T-Par}\rfloor \qquad \frac{\Gamma \vdash D \quad \Gamma' \vdash M}{\Gamma \uplus_S \Gamma' \vdash \mathsf{def}\ D\ \mathsf{in}\ M} \ \lfloor\text{T-Site}\rfloor \qquad \frac{\Gamma_1 \vdash D_1 \quad \Gamma_2 \vdash D_2}{\Gamma_1 \uplus_D \Gamma_2 \vdash D_1\ \mathsf{or}\ D_2} \ \lfloor\text{T-Dis}\rfloor \qquad \emptyset \vdash \mathbf{0} \ \lfloor\text{T-Inact}\rfloor$$

$$\frac{\Gamma_1@[J_1 \rhd J_3] \vdash_{pre} J_1 \quad \Gamma_2@[J_1 \rhd J_3] \vdash_r J_2 \quad \Gamma_3@[J_1 \rhd J_3] \vdash_{post} J_3 \quad \Gamma_4@[J_1 \rhd J_3] \vdash_p J_4}{(\Gamma_1 \uplus_{rea} \Gamma_2) \uplus_H (\Gamma_3 \uplus_{pro} \Gamma_4)@[J_1 \rhd J_3] \vdash [J_1] \,\&\, J_2 \rhd [J_3] \,\&\, J_4} \ \lfloor\text{T-Hand}\rfloor$$

$$l : l_r/r/li_p/p@\Delta \vdash_{pre/r/post/p} l \ \lfloor\text{T-Pre/R/Post/P Name}\rfloor \qquad \frac{\Gamma \vdash_{pre/r/post/p} J \quad \Gamma' \vdash_{pre/r/post/p} J'}{\Gamma \uplus_{pre/r/post/p} \Gamma' \vdash_{pre/r/post/p} J \,\&\, J'} \ \lfloor\text{T-Pre/R/Post/P Par}\rfloor$$

Figure 7. Static semantics.

well-typed, the declaration and the events must be well-typed and the environments must combine accordingly $\uplus_S$. Handlers composed in disjunction (⌊T-Dis⌋) are well-typed if each handler is well-typed and the environments must combine accordingly $\uplus_D$. Inaction is well-typed with an empty type environment (⌊T-Inact⌋). A handler (⌊T-Hand⌋) is well-typed if preconditions, reactants, postconditions and products are well-typed according to their typing judgment and type environments combine with the flow typing of the handler. The four judgment distinguish between the different classes of names to maintain a minimal syntax. Parallel composition of join names (⌊T-PrePar⌋, ⌊T-RPar⌋, ⌊PostPar⌋, ⌊T-PPar⌋) is well-typed if the single names are well-typed (⌊T-PreName⌋, ⌊T-RName⌋, ⌊PostName⌋, ⌊T-PName⌋) and the type environments combine. Pre-/postconditions are typed as linear, and reactants and products as nonlinear.

However the typing rules so far do not ensure that the precondition of the first handler for every flow is *init* and the final type assigned to a name is combined $l_r l_p, rp$ or l_p, p. Thus, we introduce a new type judgment $\emptyset \vdash_{flows} P$ and a typing rule that captures that constraint as

$$\frac{\Gamma \vdash P \quad \Gamma = \{l_i : l_r l_p/rp/l_p/p@\Delta_i\}^{i\in I}}{\emptyset \vdash_{flows} P} \ \lfloor\text{T-flows}\rfloor$$

where $\Delta = \{[[init] \ \rhd \ [J_i], ..., h_i^t]\}^{i\in J}$. That is, a process P is well-flowed if it is well-typed, for every consumer there is a producer, there may be free postconditions and products, and every flow in it starts with the precondition *init*. This rule is decided in linear time.

6. Properties of Typing

Subject reduction states that the type of a process is preserved following the reductions.

Theorem 1 (Subject congruence and reduction).

1. If $P \equiv Q$ then $\Gamma \vdash P$ iff $\Gamma \vdash Q$.
2. If $\Gamma \vdash P$ and $P \longrightarrow Q$ then $\Gamma \vdash Q$.
3. If $\emptyset \vdash_{flows} P$ and $P \longrightarrow Q$ then $\emptyset \vdash_{flows} Q$

These properties are proved by induction on the typing judgment assuming respectively $P \equiv P'$ and $P \to P'$.

Migration safety ensures exchange of events. To establish it, we give the reduction context: $\mathcal{E} ::= \mathcal{E} \,\&\, P \mid P \,\&\, \mathcal{E}$. We write $P_{pre}\langle l\rangle$ for precondition l contained at a handler of a site in P; analogously, we write for $P_{post}\langle l\rangle$, $P_r\langle l\rangle$ and $P_p\langle l\rangle$. We say *l is added to P* if l is a global event referenced by a precondition or reactant.

Theorem 2 (Migration safety). Suppose $\Gamma \vdash P$ then for adding l to P where $P \equiv \mathcal{E}[l]$ such that either $\mathcal{E}$ contains:

1. exactly one $\mathcal{E}_{pre}\langle l\rangle$ and $l = init$
2. exactly one pair of $\mathcal{E}_{post}\langle l\rangle$ and $\mathcal{E}_{pre}\langle l\rangle$
3. exactly one $\mathcal{E}_{post}\langle l\rangle$ and many $\mathcal{E}_{pre}\langle l\rangle$ in only one site
4. many $\mathcal{E}_r\langle l\rangle$ of the same flow
5. many $\mathcal{E}_r\langle l\rangle$ and many $\mathcal{E}_p\langle l\rangle$ of the same flow
6. exactly one $\mathcal{E}_r\langle l\rangle$ and many $\mathcal{E}_p\langle l\rangle$ of different flows

This property is proved by the derivation of the typing judgment, analysing all possible cases. The observation is that a type combination represents an exchange of shared events between same or different flows, e.g., a type combination for an event will represent an exchange of that shared event.

By Theorems 1 and 2, we have that a typed process of many flows will not go wrong, i.e., (1) a handler occurs over events coming from postconditions and products of previous occurrences (or stated products), and an event is added to a site if it comes from a postcondition or a product of previous occurrences and (2) an exchange of a shared event occurs between processes of the same flow or of two different flows (in the latter, only if the "sending" flow has no recipient process).

Progress asserts that the reduction of a simple, and well-typed process denoting a tree of flows (flows starting with precondition *init*) will never get stuck, i.e., the process is *inactive* or can make a reduction step.

Definition 5 (Inactive). A process P is *inactive* if defined as a composition of sites where no handlers can occur and no events are exchanged.

Definition 6 (Simple). A process P is *simple* if (1) event *init* is present, (2) reactant events of the first handler are present, either as global or in the handler's site, (3) there are as many reactants for a handler as declared events and unbound products of handlers in simple processes.

Theorem 3 (Progress). If $\emptyset \vdash_{flows} P$ and P is simple then $P \longrightarrow P'$ or inactive.

7. Related Work

Automatic stack management and decoupling. The automatic stack management is used [16, 25, 27, 28, 34, 40] to address the problem of storing and communicating the state between tasks without requiring programers to model the stack. Every task is expressed as a single procedure that blocks on an I/O operation while keeping the current state. Closures are used to encapsulate data into callback functions. We address the same problem through sites and migration based on decoupling and without using complex, non-mainstream primitives. In contrast to our proposal, closures, e.g. continuations [11], are based on strong coupling between tasks, i.e., pointers are kept between them, and are not present in low-level languages, e.g. C, C++. This sacrifices program readability and maintainability, and is dependent of the language API.

Decoupling is used to increase scalability of distributed asynchronous applications. It is implemented in various communication models [19]: push-pull, aperiodic-periodic, and unicast-multicast. In our work, decoupling is used to address the stack management problem and is defined over unicast. Extending decoupling to multicast is ongoing work.

Formal languages. The closely related calculi to the E-Calculus are: the Join calculus [17], the M calculus [23, 39], and the Distributed Join (DJ) calculus [18]. The relation of the join-calculus to the E-Calculus is given through the encoding, so we focus here on the M and DJ calculi.

The M calculus describes a distributed, higher-order version of the Join calculus. Higher-order communication along programmable localities are introduced to address the strong locality of the Join calculus where names are permanently bound to a single locality (our sites). It comes with a cost of complex reduction rules and behavioural type system, based on concepts such as membrane, coming from the CHAM [5], and so-called routing rules.

DJ integrates concepts such as hierarchical localities, transparent mobility and communications. The semantics is defined over multiple solutions where every name is defined in at most one solution. This is similar to our parallel sites but more restrictive to our migration strategy. Despite their expressiveness in modelling distribution, none of these calculi can express and guarantee the challenges of two main tenets in event-driven programming: stack management, shared state and decoupling.

Recently [9], concepts coming from the Join calculus are used to investigate typestate oriented programming in a concurrent setting. At a high level, our pre-/postconditions can be viewed as roughly analogous to typestates [41]. In contrast, that work does not address reasoning of control flow and decoupling in event-driven programming.

Type systems. The area of validation for event-driven programming is relatively novel and not explored. Recently, a model checker [12] was introduced for a DSL that supports asynchronous events. A type system based on session types [31] is designed for a variant of the π calculus extended with event register.

The π calculus type system [32], which this type system is inspired, controls the number of times a name is used in a term by imposing polarity and multiplicity constraints. Linear and nonlinear types take three forms i/o/# to capture the action of receiving, sending and both; in contrast, our type system captures the behaviour of pre-/postconditions, reactants, products, events and the union of them. A fundamental difference with our work is that no one of these systems captures and guarantees the uniqueness of control flow and safe addition of global events in the presence of aliasing.

8. Conclusions and Future Work

This paper introduced an event calculus and a linear type system to address the problems of reasoning about control flow in event-driven programming without hampering decoupling in the presence of aliasing. The E-Calculus and the type system add another fundamental stone to the formal study and validation of event-driven systems by proposing a practical model of cooperative decoupled processes. Our system demonstrates the symbiosis between these two main tenets through a concise calculus and type system, including number of constructs, rules of operational semantics and type system, and proofs of properties. Our type system ensures subject reduction, migration safety and progress. We are currently using this work as basis for two further investigations:

Multicasting [15] is a distinguishing feature in several event-driven system, i.e., an event is distributed across many sites in paradigms like implicit invocations/publish-subscribe. Multicasting depends on the interested sites of the event and the application semantics. We plan to make the choice of multicasting vs unicasting an integral part of an event's definition rather than that of individual producers or consumers.

Implementation of a safe event-driven library in a mainstream language represents the next step in the implementation of the E-Calculus and its type system. The library must support constructs for join patterns and sites. We are using Scala for this implementation, given its rich existing support for event-driven programming [21, 29].

Acknowledgments

This work has been supported by the European Research Council, grants No. 321217 and 617805.

References

[1] Atul Adya, Jon Howell, Marvin Theimer, William J. Bolosky, and John R. Douceur. Cooperative task management without manual stack management. In *USENIX*, pages 289–302, 2002.

[2] National Fire Protection Association. Codes and standards, November 2015. `http://www.nfpa.org/`.

[3] Andi Bejleri, Mira Mezini, and Patrick Eugster. *Cooperative Decoupled Processes: The E-Calculus and Linearity*, 2015. `http://andibejleri.net/papers/BME15.pdf`.

[4] Nick Benton, Luca Cardelli, and Cédric Fournet. Modern concurrency abstractions for c#. *ACM Trans. Program. Lang. Syst.*, 26(5):769–804, 2004.

[5] Gérard Berry and Gérard Boudol. The chemical abstract machine. *Theor. Comput. Sci.*, 96(1):217–248, 1992.

[6] William J. Bolosky et al. Feasibility of a serverless distributed file system deployed on an existing set of desktop pcs. In *SIGMETRICS '00*, pages 34–43, 2000.

[7] Brian Chin and Todd Millstein. Responders: Language support for interactive applications. In *ECOOP*, 2006.

[8] Simon Courtenage. Specifying and detecting composite events in content-based publish/subscribe systems. In *ICDCS Workshops*, pages 602–610, 2002.

[9] Silvia Crafa and Luca Padovani. The chemical approach to typestate-oriented programming. In *OOPSLA*, 2015.

[10] Ryan Cunningham and Eddie Kohler. Making events less slippery with eel. In *HOTOS*, 2005.

[11] Christopher T. Haynes Daniel P. Friedman and Eugene E. Kohlbecker. Programming with continuations. In *PTPE*, pages 263–274, 1984.

[12] Ankush Desai, Vivek Gupta, Ethan Jackson, Shaz Qadeer, Sriram Rajamani, and Damien Zufferey. P: Safe asynchronous event-driven programming. *SIGPLAN Not.*, 48(6), June 2013.

[13] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, June 2003.

[14] Ludger Fiege, Mira Mezini, Gero Mühl, and Alejandro P. Buchmann. Engineering event-based systems with scopes. In *ECOOP '02*, pages 309–333, 2002.

[15] Sally Floyd et al. A reliable multicast framework for light-weight sessions and application level framing. *IEEE/ACM Trans. Netw.*, 5(6), December 1997.

[16] Adam Foltzer, Abhishek Kulkarni, Rebecca Swords, Sajith Sasidharan, Eric Jiang, and Ryan Newton. A meta-scheduler for the par-monad: Composable scheduling for the heterogeneous cloud. *SIGPLAN Not.*, 47(9), September 2012.

[17] Cédric Fournet and Georges Gonthier. The reflexive cham and the join-calculus. In *POPL*, pages 372–385, 1996.

[18] Cédric Fournet, Georges Gonthier, Jean-Jacques Lévy, Luc Maranget, and Didier Rémy. A calculus of mobile agents. In *CONCUR*, pages 406–421, 1996.

[19] Michael Franklin and Stanley Zdonik. A framework for scalable dissemination-based systems. *SIGPLAN Not.*, 32(10), October 1997.

[20] J. Garcia, D. Popescu, G. Safi, W.G.J. Halfond, and N Medvidovic. Identifying Message Flow in Distributed Event-Based Systems. In *ESEC/FSE*, pages 367–377, 2013.

[21] V. Gasiunas, L. Satabin, M. Mezini, A. Nú nez, and J. Noyé. EScala: Modular Event-driven Object Interactions in Scala. In *AOSD*, pages 227–240, 2011.

[22] David Gelernter. Generative communication in linda. *ACM Trans. Program. Lang. Syst.*, 7(1), January 1985.

[23] Florence Germain, Marc Lacoste, and Jean-Bernard Stefani. An abstract machine for a higher-order distributed process calculus. *Elec. Not. Theor. Comput. Sci.*, 66(3):145–169, 2002.

[24] Jean-Yves Girard. Linear logic. *Theor. Comp. Sci.*, 50:1–102, 1987.

[25] Andreas Gustafsson. Threads without the pain. *Queue*, 3(9), November 2005.

[26] Philipp Haller and Tom Van Cutsem. Implementing joins using extensible pattern matching. In *COORDINATION*, 2008.

[27] Philipp Haller and Martin Odersky. Event-based programming without inversion of control. In *JMLC*, 2006.

[28] Philipp Haller and Martin Odersky. Scala actors: Unifying thread-based and event-based programming. *Theor. Comput. Sci.*, 410(2-3), February 2009.

[29] Jurgen M. Van Ham, Guido Salvaneschi, Mira Mezini, and Jacques Noyé. Jescala: modular coordination with declarative events and joins. In *MODULARITY*, pages 205–216, 2014.

[30] Annika Hinze and Agnès Voisard. A parameterized algebra for event notification services. In *TIME*, pages 61–63, 2002.

[31] Raymond Hu, Dimitrios Kouzapas, Olivier Pernet, Nobuko Yoshida, and Kohei Honda. Type-safe eventful sessions in java. In *ECOOP*, 2010.

[32] Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner. Linearity and the pi-calculus. *ACM Trans. Program. Lang. Syst.*, 21(5):914–947, September 1999.

[33] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.

[34] Peng Li and Steve Zdancewic. Combining events and threads for scalable network services implementation and evaluation of monadic, application-level concurrency primitives. In *PLDI*, 2007.

[35] J. K. Ousterhout. Why threads are a bad idea (for most purposes). In *Usenix (Invited talk)*, January 1996.

[36] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12), December 1972.

[37] A. Pnueli. The Temporal Logic of Programs. In *FOCS*, pages 46–57, 1977.

[38] César Sánchez, Sriram Sankaranarayanan, Henny Sipma, Ting Zhang, David L. Dill, and Zohar Manna. Event correlation: Language and semantics. In *EMSOFT*, pages 323–339, 2003.

[39] Alan Schmitt and Jean-Bernard Stefani. The m-calculus: a higher-order distributed process calculus. In *POPL*, pages 50–61, 2003.

[40] Eugene Shih, Paramvir Bahl, and Michael J. Sinclair. Wake on wireless: An event driven energy saving strategy for battery operated devices. In *MobiCom '02*, pages 160–171, 2002.

[41] Robert E. Strom and Shaula Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Software Eng.*, 12(1):157–171, 1986.

[42] J. Robert von Behren, Jeremy Condit, and Eric A. Brewer. Why events are a bad idea (for high-concurrency servers). In *HotOS*, pages 19–24, 2003.

[43] Matt Welsh, David E. Culler, and Eric A. Brewer. Seda: An architecture for well-conditioned, scalable internet services. In *SOSP*, pages 230–243, 2001.