Analysis of synchronisation patterns in stateful active objects

Ludovic Henrio, Cosimo Laneve, Vincenzo Mastandrea

LABORATOIRE

INFORMATIQUE, SIGNAUX ET SYSTÈMES DE SOPHIA ANTIPOLIS
UMR 7271

Analysis of synchronisation patterns in stateful active objects

Ludovic Henrio, Cosimo Laneve, Vincenzo Mastandrea
EQUIPE Scale, Focus

Rapport de Recherche

06-2017

Analysis of synchronisation patterns in stateful active objects

Ludovic Henrio [1], Cosimo Laneve [2], Vincenzo Mastandrea [3],

EQUIPE Scale
06-2017 - 27 pages

Abstract : This paper presents a static analysis technique based on effect and behavioural types for deriving synchronisation patterns of stateful active objects and verifying their safety – e.g. absence of deadlocks. This is challenging because active objects use futures to refer to results of pending asynchronous invocations and because these futures can be stored in object fields, passed as method parameters, or returned by invocations. Our effect system traces the access to object fields, thus allowing us to compute behavioural types that express synchronisation patterns in a precise way. The behavioural types are thereafter analysed by a solver that discovers potential deadlocks.

Key-words : Deadlock detection, type system, behavioral types, stateful active objects

1. Laboratoire I3S – CNRS – ludovic.henrio@cnrs.fr
2. INRIA, University of Bologna - cosimo.laneve@unibo.it
3. Laboratoire I3S – Université Côte d'Azur, INRIA – mastandr@i3s.unice.fr

Analysis of synchronisations in stateful active objects

Ludovic Henrio[1], Cosimo Laneve[2], and Vincenzo Mastandrea[3]

[1] Université Côte d'Azur, CNRS, I3S, France, ludovic.henrio@cnrs.fr
[2] University of Bologna, Italy & INRIA-Focus, France, cosimo.laneve@unibo.it
[3] Université Côte d'Azur, CNRS, I3S, INRIA-Focus, France, mastandr@i3s.unice.fr

Abstract. This paper presents a static analysis technique based on effect and behavioural types for deriving synchronisation patterns of stateful active objects and verifying their safety – e.g. absence of deadlocks. This is challenging because active objects use futures to refer to results of pending asynchronous invocations and because these futures can be stored in object fields, passed as method parameters, or returned by invocations. Our effect system traces the access to object fields, thus allowing us to compute behavioural types that express synchronisation patterns in a precise way. The behavioural types are thereafter analysed by a solver that discovers potential deadlocks.

1 Introduction

Active objects are a programming model that unifies the models of actors and objects. In this model, method invocations are *asynchronous*: an object that invokes a method does not release the control and is free to continue processing – the invocation is "not blocking". The returned value of an invocation is bound to a pointer, called *future*, which is used by the caller to access the value. The access to a future triggers a synchronisation [13,4,17]. Active objects are gaining prominence because they provide a high-level multitasking paradigm easier to program than explicit threads. For this reason, they are a pervasive Symbian OS idiom [16] and have been adopted in several languages and libraries, such as `Akka` [19], an actor library for `Java` and `Scala` [11], or in `ABS` [13], and in `ProActive` [4]. In active object languages, futures are first class values; therefore they can be sent as arguments of method invocations, returned by methods, or stored in object fields. In this context, the analysis of synchronisation patterns is challenging because the context where synchronisation, i.e. future access, occurs can be different from the context where the future is created. For example, the synchronisation of a future stored in a field happens when the value stored in the field is necessary;at this point, the execution of the corresponding method must finish before the value of the future can be accessed. This paper presents a static analysis technique for finding synchronisation patterns and detecting deadlocks in stateful active objects. Our analysis is expressed on an active model called `gASP` that features implicit synchronisation on futures (called *wait-by-necessity*) and does not require any specific type for futures. With wait-by-necessity, the execution is only blocked when a value to be returned by a method is needed to evaluate a term. This programming abstraction allows the programmer not to worry about placing synchronisation points: the synchronisation will always occur as late as possible. The strengths of this analysis are: the precise management of object states and their update, the tracking of futures passed by method invocations or stored in fields, and the support for infinite states. This paper extends previous works [9,7] with the handling of stateful objects by tracing the effects of methods on fields, including the storage of futures inside object fields. To illustrate synchronisation in active objects, consider the example below.

```
Int n

addToStore(Int x){
  count = n + 1;
  n = this.store(x,count);
  return count }

store(Int x, Int y){
  /* storing x */
  return y }
```

```
//MAIN
{ Store = new Act(0);
  x = Store.addToStore(1);
  x = x + 1; // needed to
             avoid conflicts
  k = Store.addToStore(4) }
```

This program creates an active object, calls the `addToStore` method asynchronously twice. To prevent non-deterministic results, and to ensure the order of execution of requests, we synchronise on the result of the first invocation (line 14) before triggering the second one. Synchronisation is expressed by any operation accessing the method result, a specific synchronisation operation is not necessary in `gASP` even if it could be added. The `addToStore` method triggers an invocation to the `store` method and counts the number of stored elements. Our analysis is able to detect that a deadlock is possible if the second invocation to `addToStore` is executed before the method `store`. The analysis reveals by a circular dependency where the single thread of the active object is waiting for the value of `n` inside `addToStore`, the effect analysis reveals that `n` contains the result of the `store` method, and thus `store` must be executed to resolve the dependency. The analysis also discovers that if line 14 is omitted then the two concurrent `addToStore` requests lead to a non-deterministic object state (one of the states being undesired).

The typing technique is based on an *effect system* that traces the accesses to fields (e.g. read and write access to `n` in the example), and a *behavioural system* that discovers the synchronisation patterns of active objects. The effect type records if a field is read or write, and which parameters are used by each method. It is used to identify conflicting field accesses, e.g. one invocation reading a field and a parallel one writing a new future in the same field. The effect type records the usage of parameters because they correspond to synchronisations that create a dependency between tasks. Also we mark an accessed future as "already synchronised" to avoid synchronising it multiple times. Because futures are implicit and pervasive we use a novel technique where "everything is a future", this enables precise tracking of futures and prevent multiple synchronisation of the same future hold by several variables. The analysis detects and excludes program with non-deterministic effects. It could be extended to non-deterministic programs by associating multiple values to each variable, merging the different environments when non-determinacy is detected. This is not studied here, it would make the analysis less precise and the formalisation more complex.

The behavioural types define the synchronisation patterns. They are expressed in a modelling language that is an extension of *lams* [8,14], which are conjunctions and disjunctions of object dependencies and method invocations. Like in [7], to deal with method returning futures, we use a place-holder that represents the object that will access a future. Actually, our types extend those of [7] with so-called *delegations* that represent side-effects of methods on argument fields. If a method stores a future f in the field of an argument, then the next access to the field should occur after the end of the method (to prevent read/write conflicts) and should be bound to the future. As the future f is generally not known when typing, we create a delegation which represents this future. We introduce the notation $\mathit{method}\rightsquigarrow \mathit{object}.\mathit{field\_name}$ for delegations.

The analysis of the behavioural type is performed by the solver defined in [7], which detects circularities in the graph of dependencies, highlighting potential *deadlock* caused by an erroneous synchronisation pattern. The behavioural type system specifies a set of pairwise dependencies between futures, some of them being delegations; the analysis unfolds this set of dependencies to find the potential circularities in the program execution. We prove that our analysis finds all the potential deadlocks of a program.

Section 2 presents `gASP`. Section 3 describes our type system and Section 4 presents our analysis technique. Section 5 provides related work and a conclusion.

2 The active object model gASP

Syntax. For simplicity, programs in `gASP` have a single class, called `Act`. Extending this work to several classes is not problematic. Types T may be either integers `Int` or active object `Act`. We use x, y, z, $\cdots$ to range over variable names. The notation $\overline{T\ x}$ denotes any finite sequence of *variable declarations* $T\ x$, separated by commas. A `gASP` program is a sequence of variable declarations $\overline{T\ x}$ (fields) and method definitions $T\ \mathtt{m}(\overline{T\ y})\ \{\, s \,\}$, plus a main body $\{\ s'\ \}$. The syntax of `gASP` body is defined by the following grammar:

$$
\begin{array}{ll}
s ::= \mathtt{skip} \mid x = z \mid \mathtt{if}\ e\ \{\, s \,\}\ \mathtt{else}\ \{\, s \,\} \mid s\ ;\ s \mid \mathtt{return}\ v & \text{statements} \\
z ::= e \mid v.\mathtt{m}(\overline{v}) \mid \mathtt{new\ Act}(\overline{v}) & \text{expressions with side effects} \\
e ::= v \mid v \oplus v & \text{expressions} \\
v ::= x \mid \mathtt{null} \mid \textit{integer-values} & \text{atoms}
\end{array}
$$

Expressions with side effects include asynchronous method call $v.\mathtt{m}(\overline{v})$, where v is the invoked object and $\overline{v}$ are the arguments of the invocation. Operations taking place on different active objects occur in

$$\frac{w \text{ is not a variable}}{[\![w]\!]_\ell \;=\; w} \qquad \frac{x \in \mathrm{dom}(\ell)}{[\![x]\!]_\ell \;=\; \ell(x)} \qquad \frac{\begin{array}{c}[\![v]\!]_\ell = k \quad [\![v']\!]_\ell = k' \\ k, k' \;\mathit{values} \quad k'' = k \oplus k'\end{array}}{[\![v \oplus v']\!]_\ell \;=\; k''}$$

Fig. 1: The evaluation function

parallel, while operations in the same active object are sequential. Terms z also include `new Act`$(\overline{v})$ that creates a new active object whose fields contain the values $\overline{v}$. A (pure) expression e may be a simple term v or an arithmetic or relational expression; the symbol $\oplus$ range over standard arithmetic and relational operators. Without loss of generality, we assume that fields and local variables have distinct names.

Semantics. The semantics of `gASP` uses two sets of names: *active object names*, ranged over by α, β, ..., and *future names*, ranged over by f, f', g, g' .... The runtime syntax of `gASP` is:

$$\begin{array}{rcll} cn & ::= & f(w) \;\mid\; f(\bot) \;\mid\; \alpha(a, p, \overline{q}) \;\mid\; cn\ cn & \text{configurations} \\ w & ::= & \alpha \;\mid\; f \;\mid\; v & \text{values and names} \\ p, q & ::= & \{\ell \mid s\} & \text{processes} \\ a, \ell & ::= & \overline{x} \mapsto \overline{w} & \text{memories} \end{array}$$

Configurations, denoted cn, are non empty sets of active objects and futures. Active objects $\alpha(a, p, \overline{q})$ contain a name α, a memory a recording fields, a running process p, and the set of processes waiting to be scheduled $\overline{q}$. The element $f(\cdot)$ represents a *future* which may be an actual value (called *future value*) or $\bot$ if the future has not yet been computed. A name, either active object or future, is *fresh* in a configuration if it does not occur in the configuration. Memories a and ℓ (where ℓ stores local variables) map variables into values or names. The following auxiliary functions are used: $\mathit{dom}(\ell)$ return the domain of ℓ; $\mathit{fields}(\texttt{Act})$ is the list of fields of `Act`; $\ell[x \mapsto v]$ is the standard map update; $a + \ell$ merges the mappings a and ℓ, it is undefined if $a(x) \neq \ell(x)$ for some x. We use the following notation: $(a + \ell)[x \mapsto w] = a' + \ell'$ implies $a' = a[x \mapsto w]$, if $x \in \mathrm{dom}(a)$, or $\ell' = \ell[x \mapsto w]$, otherwise. The evaluation of an expression, denoted $[\![e]\!]_{a+\ell}$, returns the value of e by computing the expression, retrieving the values stored in $a + \ell$; $[\![\overline{e}]\!]_{a+\ell}$ returns the tuple of values of $\overline{e}$. Finally, if m is defined by $T\,\texttt{m}(\overline{T\ x})\ \{\, s \,\}$ then: $\mathrm{bind}(\alpha, m, \overline{w}, f) = p$ where p is a process in the following shape $\{\, [\,\mathrm{destiny} \mapsto f,\ \texttt{this} \mapsto \alpha,\ \overline{x} \mapsto \overline{w}\,] \mid s \,\}$. The special variable destiny records the name of the future currently computed.

The operational semantics of `gASP` is defined by a transition relation between configurations Figure 2. Most of the rules are standard; we discuss those that deserve comments. Rule Update performs the update of a future when the corresponding value has been computed. The new value may be also a Rule Serve schedules a new process to be executed, which is taken from the set q of waiting processes. Rule Assign stores a value or a name into a local variable or a field (*cf.* definition of $a + \ell$). The evaluation of $[\![e]\!]_{a+\ell}$ may require synchronizations. Indeed, if e contains arithmetic operations, then the operands must be evaluated to integers. Therefore, if an operand is a future, the rule can only be applied *after this future has been evaluated and updated.* Also, in rule Invk, the evaluation of $[\![e]\!]_{a+\ell}$ must return an object name. If, instead, it returns a future then the rule cannot be applied and a synchronisation occurs. Similarly, the `if` statement is omitted here but the evaluation of the condition must result in a boolean which may trigger a synchronisation. Note that this semantics naturally ensures the strong encapsulation featured by actors and active-objects: an active object can only assign its own field and cannot access the fields of other active objects directly. The initial configuration of a `gASP` program with main body $\{s\}$ is:

$$\mathit{main}(\,[\,\overline{x} \mapsto \overline{0}\,], \{\,[\,\mathrm{destiny} \mapsto f_{\mathit{main}},\ \texttt{this} \mapsto \mathit{main}\,] \mid s\,\}, \varnothing)$$

where main is a special active object, $\overline{x} = \mathit{fields}$,and f_{main} is a future name. As usual, $\rightarrow^*$ is the reflexive and transitive closure of $\rightarrow$.

Analysed Programs. In order to simplify the technical details, we only consider `gASP` programs that verify the following restrictions:

(i) object fields and method returned values are of type `Int` (at runtime they can be either futures or integer values);

$$\textsc{Serve}\quad \alpha(a,\varnothing,\overline{q}\cup\{p\}) \to \alpha(a,p,\overline{q})$$

$$\textsc{Update}\quad \frac{\begin{array}{c}(a+\ell)(x)=f\\ (a+\ell)[x\mapsto w]=a'+\ell'\end{array}}{\begin{array}{c}\alpha(a,\{\ell\mid s\},\overline{q})\ f(w)\\ \to \alpha(a',\{\ell'\mid s\},\overline{q})\ f(w)\end{array}}
\qquad
\textsc{Assign}\quad \frac{\begin{array}{c}[\![e]\!]_{a+\ell}=w\\ (a+\ell)[x\mapsto w]=a'+\ell'\end{array}}{\begin{array}{c}\alpha(a,\{\ell\mid x=e\ ;\ s\},\overline{q})\\ \to \alpha(a',\{\ell'\mid s\},\overline{q})\end{array}}$$

$$\textsc{New}\quad \frac{[\![\overline{v}]\!]_{a+\ell}=\overline{w}\qquad \beta\ \text{fresh}\qquad \overline{y}=\mathit{fields}(\texttt{Act})}{\begin{array}{c}\alpha(a,\{\ell\mid x=\texttt{new Act}(\overline{v})\ ;\ s\},\overline{q})\\ \to \alpha(a,\{\ell\mid x=\beta\ ;\ s\},\overline{q})\ \ \beta([\overline{y}\mapsto\overline{w}],\varnothing,\varnothing)\end{array}}$$

$$\textsc{Invk}\quad \frac{\begin{array}{c}[\![v]\!]_{a+\ell}=\beta\qquad [\![\overline{v}]\!]_{a+\ell}=\overline{w}\qquad \beta\neq\alpha\\ f\ \text{fresh}\qquad \text{bind}(\beta,m,\overline{w},f)=p'\end{array}}{\begin{array}{c}\alpha(a,\{\ell\mid x=v.\texttt{m}(\overline{v})\ ;\ s\},\overline{q})\ \beta(a',p,\overline{q'})\\ \to \alpha(a,\{\ell\mid x{=}f\,;s\},\overline{q})\ \beta(a',p,\overline{q'}\cup\{p'\})\ f(\bot)\end{array}}$$

$$\textsc{Invk-Self}\quad \frac{\begin{array}{c}[\![v]\!]_{a+\ell}=\alpha\qquad [\![\overline{v}]\!]_{a+\ell}=\overline{w}\\ f\ \text{fresh}\qquad \text{bind}(\alpha,m,\overline{w},f)=p'\end{array}}{\begin{array}{c}\alpha(a,\{\ell\mid x=v.\texttt{m}(\overline{v})\ ;\ s\},\overline{q})\\ \to \alpha(a,\{\ell\mid x=f\ ;\ s\},\overline{q}\cup\{p'\})\ f(\bot)\end{array}}$$

$$\textsc{If-True}\quad \frac{[\![e]\!]_{a+\ell}\neq 0}{\begin{array}{c}\alpha(a,\{\ell\mid \texttt{if}\ e\ \{\,s_1\,\}\ \texttt{else}\ \{\,s_2\,\}\ ;\ s\},\overline{q})\\ \to \alpha(a,\{\ell\mid s_1\ ;\ s\},\overline{q})\end{array}}
\qquad
\textsc{If-False}\quad \frac{[\![e]\!]_{a+\ell}= 0}{\begin{array}{c}\alpha(a,\{\ell\mid \texttt{if}\ e\ \{\,s_1\,\}\ \texttt{else}\ \{\,s_2\,\}\ ;\ s\}\},\overline{q})\\ \to \alpha(a,\{\ell\mid s_2\ ;\ s\}\},\overline{q})\end{array}}$$

$$\textsc{Return}\quad \frac{[\![v]\!]_{a+\ell}=w\qquad \ell(\text{destiny})=f}{\begin{array}{c}\alpha(a,\{\ell\mid \texttt{return}\ v\},\overline{q})\ f(\bot)\\ \to \alpha(a,\varnothing,\overline{q})\ f(w)\end{array}}
\qquad
\textsc{Context}\quad \frac{cn\to cn'}{cn\ cn''\to cn'\ cn''}$$

Fig. 2: Evaluation function and semantics of `gASP` (excerpt).

(*ii*) the futures created in a method must be either returned or synchronised or stored in a field of a parameter (or *this*).

The constraint (*i*) can be checked by a standard type checker, and (*ii*) can be verified by a simple static analyser. In particular, (*ii*) prevents computations running in parallel without any mean to synchronise on them. Technically, admitting futures that are never synchronised requires to collect the corresponding behaviours and add them to any possible continuation, like in [9].

2.1 Deadlocks and queues with deterministic effects

In `gASP`, when computing an expression, if one of the elements of the expression is a future then the current active object waits until the future has been updated. If the waiting relation is *circular* then no progress is possible. In this case all the active objects in the circular dependency are *deadlocked.* We formalise the notion of deadlock below. Let *contexts* $C[\ ]$ be the following terms

$$\begin{array}{rcl}C[\ ] & ::= & x=[\ ]\oplus v\ ;\ s\ \mid\ x=v\oplus[\ ]\ ;\ s\ \mid\ \texttt{if}\ [\ ]\ \{\,s'\,\}\ \texttt{else}\ \{\,s''\,\}\ ;\ s\\ & \mid & \texttt{if}\ [\ ]\oplus v\ \{\,s'\,\}\ \texttt{else}\ \{\,s''\,\}\ ;\ s\ \mid\ \texttt{if}\ v\oplus[\ ]\ \{\,s'\,\}\ \texttt{else}\ \{\,s''\,\}\ ;\ s\end{array}$$

As usual, $\mathcal{C}[e]$ is the context where the hole [] of $\mathcal{C}[\ ]$ is replaced by e.
Let $f\in \mathit{destinies}(\overline{q})$ if there is $\{\ell|s\}\in\overline{q}$ such that $\ell(\text{destiny})=f$.

Definition 1 (Deadlocked configuration). *Let cn be a configuration containing* $\alpha_0(a_0,p_0,\overline{q_0})$, $\cdots$, $\alpha_{n-1}(a_{n-1},p_{n-1},\overline{q_{n-1}})$. *If, for every* $0\le i<n$,

1. $p_i=\{\ell_i\mid C[v]\}$ *where* $[\![v]\!]_{a_i+\ell_i}=f_i$ *and*
2. $f_i\in \mathit{destinies}(p_{i+1},\overline{q_{i+1}})$, *where* $+$ *is computed modulo* n

then cn is deadlocked.

A program is deadlock-free if, denoting cn its initial configuration, for every cn' *s.t.* $cn\to^* cn'$, cn' *is deadlock free.*

Definition 1 is about runtime entities that have no static counterpart. Therefore we consider a notion weaker than deadlocked configuration. This last notion will be used in the Appendices B to demonstrate the correctness of the type system.

Definition 2. *A configuration cn has*

i) a dependency (α, β) *if:*

- $\alpha(a, \{\ell \mid C[f]\}, \overline{q})\ \beta(a', p', \overline{q'}) \in cn$
- $f \in destinies(p', \overline{q'})$.

ii) a dependency (α, α) *if*

- $\alpha(a, \{\ell \mid C[f]\}, \overline{q}) \in cn$
- $f \in destinies(\overline{q})$.

Given a set D of dependencies, let D^+ *be the transitive closure of D. A configuration cn contains a circularity if the transitive closure of its set of dependencies has a pair* (α, α).

Proposition 1. *If a configuration is deadlocked then it has a circularity. The converse is false.*

Since gASP is stateful, it is possible to store futures in object fields and to pass them around during invocations. Therefore, computing the value of a field is difficult and, sometimes, not possible because of the nondeterminism caused by the concurrent behaviours. To be precise enough, we restrict the analysis to programs where concurrent methods have no conflicting access to same fields, i.e. if one method writes a field of an object, any method that can execute in parallel cannot access to the same field. This constraint is defined below.

Let $x^{\mathtt{w}} \in \{\ell \mid s\}$ whenever x occurs as a left-hand side variable in an assignment of s; $x^{\mathtt{r}} \in \{\ell \mid s\}$ whenever x occurs as an atom in s.

Definition 3 (Queue with deterministic effects). *An active object* $\alpha(a, p, \{q_1, \cdots, q_n\})$ *has a* queue with deterministic effects *if, for every* $x \in dom(a)$*, there are no* $i \neq j$ *such that either (*i*)* $x^{\mathtt{w}} \in q_i$ *and* $x^{\mathtt{w}} \in q_j$ *or (*ii*)* $x^{\mathtt{r}} \in q_i$ *and* $x^{\mathtt{w}} \in q_j$.

A configuration cn has deterministic effects *if every active object of this configuration has a* queue with deterministic effects. *A* gASP *program* has deterministic effects *if* $cn \to^* cn'$, cn' has deterministic effects*, where cn is the initial configuration for this program.*

Example. This simple program proposed in the introduction starts creating an active object (α associated to the variable `Store`), where the only field `n` is initialized to 0 (rule New); and it continues invoking the method `addToStore` on the active object just created (rules Invk, Serve). This method invocations creates the future f that is stored in x. Then we reach the configuration:

$$main\big(\,[\,n \mapsto 0\,], \big\{\,[\,\text{destiny} \mapsto f_{main},\ \text{Store} \mapsto \beta,\ \text{x} \mapsto f\,] \,|\, \texttt{x = x + 1;}\ \cdots\,\big\}, \varnothing\,\big)$$
$$\alpha\big(\,[\,n \mapsto 0\,], \big\{\,[\,\text{destiny} \mapsto f, \text{x} \mapsto 0\,] \,|\, \texttt{count = n + 1;}\ \cdots\,\big\}, \varnothing\,\big) \qquad f(\bot)$$

At this point the execution of the main function stops because the result of the method previously invoked is needed to compute the expression `x + 1`. The active object α can continue to execute the method `addToStore`, which can compute the expression on line 4 (all the values needed to compute the expression are known). After the evaluation of that expression the active object α invokes the method `store` on itself. This invocation creates a new future g, which will be stored in the field `n`. The method `addToStore` ends returning `count`.

$$main\big(\,[\,n \mapsto 0\,], \big\{\,[\,\text{destiny} \mapsto f_{main},\ \text{Store} \mapsto \alpha,\ \text{x} \mapsto 1\,] \,|\, \texttt{int k = Store.addToStore(4)}\,\big\}, \varnothing\,\big)$$
$$\alpha\big(\,[\,n \mapsto g\,], \varnothing, \{\textit{body-of-}\texttt{store}\}\,\big) \quad f(1) \quad g(\bot)$$

At this point the main function can compute the expression on line 14 and just after it invokes again the method `addToStore` that will be computed by α. A new future called h is created by this method invocation.

$$main\big(\,[\,n \mapsto 0\,], \varnothing, \varnothing\,\big)$$
$$\alpha\big(\,[\,n \mapsto g\,], \varnothing, \{\textit{body-of-}\texttt{store}\}, \{\textit{body-of-}\texttt{addToStore}\}\,\big) \quad f(1) \quad g(\bot) \quad h(\bot)$$

As we can se in the configuration just above the active object α can run one of the two request that will produce two different scenarios.

From this point, if α serves the invocation of `addToStore` we reach a deadlock, because to execute the expression of line 4 the value of the field `n` is needed, but the method `store` can only be served after the termination of the current method.

On the contrary, if `store` is served first, then when the execution of `addToStore` occurs, the future stored in the field `n` is already computed therefore the expression `n + 1` can be solved and the program terminates.

3 Behavioral Type System

Behavioral types are abstract descriptions that are associated to gASP programs by a type system. This is done by recording several informations: (1) *effects on object fields* to enforce consistency of read/write operations between methods invoked in parallel on the same active object; (2) *dependencies between active objects* and *between futures and active objects* to enforce consistency of synchronization patterns. The analysis is performed following the program structure and verifying that the types of methods match previously declared types. From the explicit type system presented below, an inference system can be defined in a standard way. Note that it is syntactically not possible to infer which variables might contain a future. Consequently, we consider all stored values as futures. However some of these future values will be already synchronized when created. It is therefore important to distinguish *future names* that are identifiers and *future value* that are values corresponding to futures; the environment will map future identifiers to future types.

Analysed Properties. The goal of the type system is to verify the deadlock freedom of gASP programs. Since gASP is statefull, deadlocks might be caused by access to futures stored in object fields. Therefore, the type system must also compute the *effects* of statements on active object fields (and expose them in types of methods so that the analysis is compositional). It is worth to notice that in gASP, because of concurrency, the computations are non-deterministic and the effects on fields may be indeterminate. Our type system also verifies whether the analysed program might exhibit such a non-deterministic behaviour.

Types. We use a set of future names, ranged over by f, g, $\cdots$. Types are either basic types, future types or behavioural types. They are defined as follows:

$$\begin{array}{lll}
\mathbb{b} ::= \square \mid \alpha[\overline{x:f}] & & \text{basic type}\\
\mathbb{f} ::= \mathbb{b} \mid \lambda X.\mathtt{m}(f, \overline{g}, X, \Gamma, E) \mid f \rightsquigarrow g.x & & \text{future type}\\
\kappa ::= \star \mid \alpha \mid X & & \text{synchronizers}\\
\mathsf{L} ::= \mathsf{0} \mid (\kappa, \alpha) \mid f_\kappa \mid \mathsf{L} + \mathsf{L} \mid \mathsf{L} \,\&\, \mathsf{L} & & \text{behavioral type}
\end{array}$$

Basic types $\mathbb{b}$ are used for values or parameters; they may be either primitive type, i.e. integer, $\square$ or an object type $\alpha[\overline{a:f}]$. Future types $\mathbb{f}$ include basic types, invocation results, and delegations. The invocation result $\lambda X.\mathtt{m}(f, \overline{g}, X, \Gamma, E)$ represents the value computed by a method invocation. where $f, \overline{g}$ are the arguments of the invocation (f is the future of the called object), X, called *handle*, is a place-holder for the object that will synchronize with the invocation, the environment Γ and the effects E record the state changes peroformed by the methd, they are discussed in the following. The delegation $f \rightsquigarrow g.x$ represents a method side effect, namely the value that is written by the method corresponding to f in the field x of the argument g. In the type system we also use "checkmarked" future types, noted $\mathbb{f}^{\checkmark}$, to represent a future value that has been already synchronized. We use $\mathbb{f}^{[\checkmark]}$ to range over both future types and "checkmarked" future types.

Behavioral types include $\mathsf{0}$, the empty dependency, and (κ, α) that means: if κ is instantiated by an object β, then β will need α to be available in order to proceed its execution. Behavioral types also include *synchronisation commitments* f_κ. The precise meaning of f_κ depends on the value of κ: $f_\star$ means that the invocation related to f is potentially running in parallel; f_α means that the active object α is waiting for the result of the invocation corresponding to f; f_X represents the return of a future f, where the handle X will be replaced with the name of the object that will synchronize on the result of f. The types $\mathsf{L} \,\&\, \mathsf{L}'$ is the behaviour of two statements of types L and L' running in parallel; $\mathsf{L} + \mathsf{L}'$ is the behaviour of two statements (of types L and L') running in sequence (regardless of the order). We will shorten $\mathsf{L}_1 \,\&\, \cdots \,\&\, \mathsf{L}_n$ into $\&_{i\in\{1..n\}} \mathsf{L}_i$ and $\mathsf{L}_1 + \cdots + \mathsf{L}_n$ into $\sum_{i\in\{1..n\}} \mathsf{L}_i$. The operations "&" and "+" on behavioural types are associative, commutative with $\mathsf{0}$ being the identity for $\&$ and $+$. The operator "&" has precedence over "+".

Environments. Environments, noted Γ, Γ', $\cdots$, are mappings from variables to future names ($x \mapsto f$) and from future names to future types, checkmarked or not ($f \mapsto \mathbb{f}^{[\checkmark]}$). Environments also map method names to their signatures.

We use the standard notations $\mathrm{im}(\Gamma)$ for denoting the image. We also use a few additional operations on mappings: the restriction operation is denoted $\Gamma|_S$; the difference operation $\Gamma \setminus x$ is defined as $\Gamma|_{\mathrm{dom}(\Gamma)\setminus x}$. The following functions on Γ will also be used:

$$E[f.x \mapsto^{\sqcup} \mathtt{h}](f.x) \; = \begin{cases} \mathtt{h} \sqcup \mathtt{h}' & \text{if } E(f.x) = \mathtt{h}' \\ \mathtt{h} & \text{if } x \notin E(f) \text{ and } x \in \mathit{fields}(\mathtt{Act}) \\ \mathit{undefined} & \text{otherwise} \end{cases} \tag{1}$$

$$(E \sqcup E')(f.x) = \begin{cases} E(f.x) \sqcup E'(f.x) & \text{if } x \in E(f) \text{ and } x \in E'(f) \\ E(f.x) & \text{if } x \in E(f) \text{ and } x \notin E'(f) \\ E'(f.x) & \text{if } x \notin E(f) \text{ and } x \in E'(f) \end{cases} \tag{2}$$

$$\mathit{Effects}(\Gamma) = \bigsqcup \{E \mid \Gamma(f) = \lambda X.\mathtt{m}(\overline{g}, X, \Gamma_{\mathtt{m}}, E)\} \tag{3*}$$

$$x^{\mathtt{h}} \# y^{\mathtt{h}'} \; = \begin{cases} \mathit{true} & \text{if } x \neq y \text{ or } (x = y \text{ and } \mathtt{h}' = \mathtt{r} = \mathtt{h}) \\ \mathit{false} & \text{otherwise} \end{cases} \tag{4}$$

$$\{x_1^{\mathtt{h}_1}, \cdots, x_n^{\mathtt{h}_n}\} \# \{y_1^{\mathtt{h}'_1}, \cdots, y_m^{\mathtt{h}'_m}\} \; = \bigwedge_{i \in 1..n, j \in 1..m} x_i^{\mathtt{h}_i} \# y_j^{\mathtt{h}'_j} \tag{5**}$$

$$\mathit{instanceof}(E, \sigma)(f) \; = \begin{cases} \bigsqcup_{g \in \sigma^{-1}(f)} E(g) & \text{if } \forall f_1, f_2 \in \sigma^{-1}(f). f_1 \neq f_2 \Rightarrow E(f_1) \# E(f_2) \\ \mathit{undefined} & \text{otherwise} \end{cases} \tag{6}$$

Fig. 3: Auxiliary functions for effects.

- $\mathit{names}(\Gamma) = \mathrm{dom}(\Gamma) \cup \{\alpha \mid \alpha[\overline{x : f}]^{[\checkmark]} \in \mathrm{im}(\Gamma)\}$;
- $\mathit{obj}(\overline{f})$ and $\mathit{int}(\overline{f'})$ are subsets of $\overline{f}$ such that for each $f' \in \mathit{obj}(\overline{f})$ or $f' \in \mathit{int}(\overline{f})$ we have $\Gamma(f') = \alpha[\cdots]$ or $\Gamma(f') \neq \alpha[\cdots]$ for some α respectively;
- $\mathit{Fut}(\Gamma)$ is the set of future names in $\mathrm{dom}(\Gamma)$; $\mathit{aFut}(\Gamma)$ and $\mathit{sFut}(\Gamma)$ are the subset of $\mathit{Fut}(\Gamma)$ that contain future names f such that $\Gamma(f)$ is not "checkmarked" or checkmarked respectively;
- $\mathit{unsync}(\Gamma) = \&_{f \in \mathit{aFut}(\Gamma)} f_\star$ is the parallel behaviour of the not-yet-synchronized method invocations;
- $\Gamma[f^\checkmark]$ returns the environment $\Gamma[f \mapsto \mathbb{f}^\checkmark]$ when $\Gamma(f)$ is either $\mathbb{f}$ or $\mathbb{f}^\checkmark$;
- $\Gamma(f.x) = \begin{cases} g & \text{if } \Gamma(f) = \alpha[\cdots, x : g, \cdots] \\ \mathit{undefined} & \text{otherwise} \end{cases}$
- $\Gamma[f.x \mapsto g]$ returns the environment such that $\Gamma(f.x) = g$, assuming that $f \in \mathrm{dom}(\Gamma)$ and $x \in \mathit{fields}(\mathtt{Act})$; $\Gamma[f.x \mapsto g]$ is defined like Γ elsewhere;
- $\Gamma_1 =_{\mathtt{unsync}} \Gamma_2$ whenever $\Gamma_1(f) = \Gamma_2(f)$ for every f in $\mathit{aFut}(\Gamma_1) \cup \mathit{aFut}(\Gamma_2)$.
- We define a flattening function on environments:

$$\mathit{flat}(f, \overline{f'}, \Gamma) \; = \begin{cases} [\,\alpha, f, \overline{g}\,] :: \mathit{flat}(\overline{f'}, \Gamma) & \text{if } \Gamma(f) = \alpha[\overline{x : g}] \\ [\,f\,] :: \mathit{flat}(\overline{f'}, \Gamma) & \text{if } \Gamma(f) \neq \alpha[\overline{x : g}] \\ \mathit{undefined} & \text{otherwise} \end{cases}$$

Effects. Effects are functions, noted E, E', A, A', $\cdots$, that map future names to a set of field names labelled either with $\mathtt{r}$ (read) or with $\mathtt{w}$ (write). For example, consider m a method with effect E, and f one of its arguments, $E(f) = \{x^{\mathtt{w}}, y^{\mathtt{r}}\}$ means that $\mathtt{m}$ writes on the field x of the object that is the value of f and reads on the field y. Let $\mathtt{h}$ range over $\{\mathtt{r}, \mathtt{w}\}$; if $x^{\mathtt{h}} \in E(f)$, we use the notation $E(f.x) = \mathtt{h}$. With an abuse of notation, we also write $x \in E(f)$ if $E(f) = \{x_1^{\mathtt{h}_1}, \cdots, x_n^{\mathtt{h}_n}\}$ and $x \in \{x_1, \cdots, x_n\}$ (therefore $x \notin E(f)$ also when $E(f)$ is undefined).

The set $\{\mathtt{r}, \mathtt{w}\}$ with the ordering $\mathtt{r} < \mathtt{w}$ is a lattice, therefore we use the operation $\sqcup$ for least-upper bound. We also use few auxiliary operations that are shown in Figure 3: *update operation with upper bound*$^{(1)}$; *merge of effects*$^{(2)}$; *effects of unsynchronized methods*$^{(3)}$; *compatibility*$^{(4-5)}$; effect instantiation taking into account effect compatibility$^{(6)}$.

Judgements. The judgements used in the type system are:

- $\vdash m : (f, \overline{g}, \Gamma_{\mathtt{m}}, X) \to (E, A)$ for instantiating the method signature of $\mathtt{m}$, where $f, \overline{g}, X$ are the *formal parameters*, $\Gamma_{\mathtt{m}}$ is the part of environment accessible from the method parameters which are objects: $\Gamma_{\mathtt{m}} = (\Gamma|_{f \cup \mathit{obj}(\overline{g})})$, where Γ is the environment at invocation point. E, A are two environments that

* We notice that $\Gamma(f)$ is not checkmarked

** It is compatible to either read several times or to write once.

values, variables and method names: $\Gamma \vdash x : \mathbb{b}$ and $\Gamma \vdash \mathtt{m} : (\overline{f}, X, \Gamma') \to (E, A)$

$$\text{(T-Val)}\quad \frac{v \quad \textit{integer-value} \text{ or } \mathtt{null}}{\Gamma, E \vdash v : \square \ \triangleright \ E} \qquad \text{(T-Var)}\quad \frac{\Gamma(x) = f}{\Gamma, E \vdash x : f \ \triangleright \ E} \qquad \text{(T-Fut)}\quad \frac{\Gamma(f) = \mathbb{f}^{[\checkmark]}}{\Gamma \vdash f : \mathbb{f}^{[\checkmark]}}$$

$$\text{(T-Field)}\quad \frac{\begin{array}{c}\Gamma(\textit{this}.x) = f \\ E' = E[\textit{this}.x \mapsto^{\sqcup} \mathtt{r}]\end{array}}{\Gamma, E \vdash x : f \ \triangleright \ E'} \qquad \text{(T-Method-Sign)}\quad \frac{\begin{array}{c}\Gamma(\mathtt{m}) = (\overline{f}, X, \Gamma') \to (E, A) \qquad \sigma \text{ renaming} \\ \Gamma'' = \sigma(\Gamma') \qquad E' = \textit{instanceof}(E, \sigma) \quad A' = \textit{instanceof}(A, \sigma)\end{array}}{\vdash m : (\sigma(\overline{f}), \sigma(X), \Gamma'') \to (E', A')}$$

synchronizations: $\Gamma, E \ {}^{\oplus}\!\!\vdash_S \ e : \mathsf{L} \ \triangleright \ \Gamma', E'$

$$\text{(T-Synchronized)}\quad \frac{\Gamma, E \vdash v : f \ \triangleright \ E' \qquad \Gamma \vdash f : \mathbb{f}^{\checkmark}}{\Gamma, E \ {}^{\oplus}\!\!\vdash_S \ v : \mathsf{0} \ \triangleright \ \Gamma, E'}$$

$$\text{(T-Sync-Invk)}\quad \frac{\begin{array}{c}\Gamma \vdash \textit{this} : \alpha[\cdots]^{\checkmark} \qquad \Gamma, E \vdash x : f \ \triangleright \ E' \\ \Gamma \vdash f : \lambda X.\mathtt{m}(\overline{f'}, X, \Gamma_{\mathtt{m}}, E_{\mathtt{m}}) \qquad \Gamma' = \Gamma[f^{\checkmark}][h^{\checkmark}]^{h \in \mathrm{dom}(E_{\mathtt{m}})} \\ \Gamma'' = \Gamma'([g.y \mapsto g'][g' \mapsto f \rightsquigarrow g.y])^{y^{\mathtt{w}} \in E_{\mathtt{m}}(g),\ g' \text{ fresh}}\end{array}}{\Gamma, E \ {}^{\oplus}\!\!\vdash_S \ x : f_\alpha \,\&\, \textit{unsync}(\Gamma'') \ \triangleright \ \Gamma'', E' \sqcup E_{\mathtt{m}}|_S}$$

$$\text{(T-Sync-Field)}\quad \frac{\begin{array}{c}\Gamma \vdash \textit{this} : \alpha[\cdots]^{\checkmark} \qquad \Gamma, E \vdash x : f \ \triangleright \ E' \\ \Gamma \vdash f : g \rightsquigarrow \textit{this}.x \qquad \Gamma' = \Gamma[f^{\checkmark}]\end{array}}{\Gamma, E \ {}^{\oplus}\!\!\vdash_S \ x : f_\alpha \,\&\, \textit{unsync}(\Gamma') \ \triangleright \ \Gamma', E'} \qquad \text{(T-Sync-Param)}\quad \frac{\begin{array}{c}\Gamma \vdash \textit{this} : \alpha[\cdots]^{\checkmark} \qquad \Gamma, E \vdash x : f \ \triangleright \ E' \\ \Gamma \vdash f : \mathbb{f} \qquad f \in S \qquad \Gamma' = \Gamma[f^{\checkmark}]\end{array}}{\Gamma, E \ {}^{\oplus}\!\!\vdash_S \ x : f_\alpha \,\&\, \textit{unsync}(\Gamma') \ \triangleright \ \Gamma', E' + [f \mapsto \varnothing]}$$

Fig. 4: Typing rules for names and synchronizations

store two kinds of effects of $\mathtt{m}$: E stores the effects that happen before $\mathtt{m}$ is synchronized, A stores the effects of the methods invoked by $\mathtt{m}$ and not synchronized in its body;

- $\Gamma, E \vdash x : f \ \triangleright \ E'$ for typing values and variables with future names, where E' is the update of E
- $\Gamma \vdash f : \mathbb{f}$ for typing future names with future types;
- $\Gamma, E \ {}^{\oplus}\!\!\vdash_S \ e : \mathsf{L} \ \triangleright \ \Gamma', E'$ for typing synchronizations, where S is the set of arguments of the method, L is the behavioral type, and Γ' and E' are the updates of Γ and E respectively;
- $\Gamma, E, A \vdash_S z : f, \ \mathsf{L} \ \triangleright \ \Gamma', E', A'$ for typing expressions with side effects z;
- $\Gamma, E, A \vdash_S s : \mathsf{L} \ \triangleright \ \Gamma', E', A'$ for typing statements s.

Type System. In the type system the environments Γ are always defined on future names $\square$ and *this*, such that $\Gamma(\square) = \square^{\checkmark}$ and $\Gamma(\textit{this}) = \alpha[\cdots]$ where α is the active object running the current method. The typing rules are presented below and the most significant ones are discussed.

The rules for values, variables and method names are listed on top of Figure 4. Rule (T-Field) models the reading of a field (of the *this* actor). The preconditions verify that the access is compatible with the effects of not yet synchronized invocations in Γ and those in A (that will not be synchronized). We notice that there is no compatibility check with effects in E and E is updated with the new access (performing the upper bound with the old value). Rule (T-Method-Sign) instantiates a method signature according to the invocation parameters. In particular, the rule also covers the case when actual parameters are not linear and deals with them through the use of the *instanceof* function. In the signature, each parameter has a fresh name, but upon invocation, new conflicts might be created by the fact that two different parameters are actually the same object. In this case, we prevent the instantiation of the invocation if a conflict might occur. For example, if the signature of a method $\mathtt{m}$ is such that $\Gamma(\mathtt{m}) = (f, f', X, \Gamma') \to ([f \mapsto \{x^{\mathtt{r}}\}, f' \mapsto \{x^{\mathtt{w}}\}]$ or $\Gamma(\mathtt{m}) = (f, f', X, \Gamma') \to ([f \mapsto \{x^{\mathtt{w}}\}, f' \mapsto \{x^{\mathtt{w}}\}]$, the type system is not able to instantiate the method invocation $\lambda X.\mathtt{m}(g, g, X, \Gamma'', E_{\mathtt{m}})$ because of potential conflicts: two operations of write on the same object appeared due to the aliasing created between parameters.

The rules for typing synchronizations are defined at the bottom of Figure 4. In `gASP`, synchronizations are due to the evaluation of expressions e that are not variables. We use the notation ${}^{\oplus}\!\!\vdash$ for these judgments. Overall, we parse the expression and the leaves have two cases: either the future is synchronized (checkmarked) or not. In this last case, there are three sub-cases, according to the future corresponds to an invocation – rule (T-Sync-Invk) –, or to a field – rule (T-Sync-Field) –, or to a method's argument – rule (T-Sync-Param). We discuss (T-Sync-Invk), the other ones are similar. In this case, the future

expressions with side effects: $\Gamma, E, A \vdash_S z : f,\ \mathsf{L}\ \rhd\ \Gamma', E', A'$

(T-Atom)

$$\dfrac{\Gamma, E \vdash v : f\ \rhd\ E'}{\Gamma, E, A \vdash_S v : f,\ \mathtt{0}\ \rhd\ \Gamma, E', A}$$

(T-Expression)

$$\dfrac{\begin{array}{c}\Gamma, E\ {}^{\oplus}\!\vdash_S\ v : \mathsf{L}\ \rhd\ \Gamma', E' \\ \Gamma', E'\ {}^{\oplus}\!\vdash_S\ v' : \mathsf{L}'\ \rhd\ \Gamma'', E''\end{array}}{\Gamma, E, A \vdash_S v \oplus v' : \square,\ \mathsf{L} + \mathsf{L}'\ \rhd\ \Gamma', E'', A}$$

(T-New)

$$\dfrac{\begin{array}{c}\Gamma, E \vdash \overline{v} : \overline{g}\ \rhd\ E' \qquad \beta, f \text{ fresh} \\ \overline{x} = \mathit{fields}(\mathtt{Act}) \quad \Gamma' = \Gamma[f \mapsto \beta[\overline{x : g}]^{\checkmark}]\end{array}}{\Gamma, E, A \vdash_S \mathtt{new\ Act}(\overline{v}) : f,\ \mathtt{0}\ \rhd\ \Gamma', E', A}$$

(T-Invk)

$$\dfrac{\begin{array}{c}\Gamma, E \vdash v : f\ \rhd\ E \qquad \Gamma \vdash f : \beta[\cdots]^{\checkmark} \qquad \Gamma, E \vdash \overline{v} : \overline{f'}\ \rhd\ E' \qquad \overline{h} = f \cup obj(\overline{f'}) \\ \vdash \mathtt{m} : (f, \overline{f'}, X, \Gamma|_{\overline{h}}) \rightarrow (E_{\mathtt{m}}, A_{\mathtt{m}}) \qquad g \text{ fresh} \qquad \overline{g'} = \overline{f'}[^{\square}/_{int(sFut(\Gamma))}] \qquad \Gamma_{\mathtt{m}} = (\Gamma|_{\overline{h}})[^{\square}/_{int(sFut(\Gamma))}] \\ \Gamma' = \Gamma[g \mapsto \lambda X.\mathtt{m}(f, \overline{g'}, X, \Gamma_{\mathtt{m}}, E_{\mathtt{m}})] \qquad \left(\mathit{Effects}(\Gamma')(h') \# y^{(E_{\mathtt{m}} \sqcup A)(h'.y)}\right)^{h' \in \mathrm{dom}(E_{\mathtt{m}} \uplus A)\ \wedge\ y \in \mathit{fields}(\mathtt{Act})}\end{array}}{\Gamma, E, A \vdash_S v.\mathtt{m}(\overline{v}) : g,\ g_{\star}\ \&\ \mathit{unsync}(\Gamma)\ \rhd\ \Gamma', E', A \sqcup A_{\mathtt{m}}|_S}$$

statements $\Gamma, E, A \vdash_S s : \mathsf{L}\ \rhd\ \Gamma', E', A$

(T-Assign-Var-Exp)

$$\dfrac{\begin{array}{c}x \notin \mathit{fields}(\mathtt{Act}) \\ \Gamma, E, A \vdash_S z : f,\ \mathsf{L}\ \rhd\ \Gamma', E', A'\end{array}}{\Gamma, E, A \vdash_S x = z : \mathsf{L}\ \rhd\ \Gamma'[x \mapsto f], E', A'}$$

(T-Assign-Field-Exp)

$$\dfrac{\begin{array}{c}x \in \mathit{fields}(\mathtt{Act}) \qquad \Gamma, E, A \vdash_S z : f,\ \mathsf{L}\ \rhd\ \Gamma', E', A' \\ \mathit{Effects}(\Gamma')(\mathit{this}) \# x^{\mathtt{w}} \qquad A'(\mathit{this}) \# x^{\mathtt{w}}\end{array}}{\Gamma, E, A \vdash_S x = z : \mathsf{L}\ \rhd\ \Gamma'[\mathit{this}.x \mapsto f], E'[\mathit{this}.x \mapsto^{\sqcup} \mathtt{w}], A'}$$

(T-Skip)

$$\Gamma, E, A \vdash_S \mathtt{skip} : \mathtt{0}\ \rhd\ \Gamma, E, A$$

(T-Seq)

$$\dfrac{\begin{array}{c}\Gamma, E, A \vdash_S s_1 : \mathsf{L}_1\ \rhd\ \Gamma_1, E_1, A_1 \\ \Gamma_1, E_1, A_1 \vdash_S s_2 : \mathsf{L}_2\ \rhd\ \Gamma_2, E_2, A_2\end{array}}{\Gamma, E, A \vdash_S s_1 ; s_2 : \mathsf{L}_1 + \mathsf{L}_2\ \rhd\ \Gamma_2, E_2, A_2}$$

(T-Return-Fut)

$$\dfrac{\Gamma, E \vdash v : f\ \rhd\ E' \qquad \Gamma \vdash f : \mathbb{f} \qquad \Gamma(\text{future}) = X}{\Gamma, E, A \vdash_S \mathtt{return}\ v : f_X\ \&\ \mathit{unsync}(\Gamma \setminus f)\ \rhd\ \Gamma, E', A}$$

(T-Return-Val)

$$\dfrac{\Gamma, E \vdash v : f\ \rhd\ E' \qquad \Gamma \vdash f : \mathbb{f}^{\checkmark}}{\Gamma, E, A \vdash_S \mathtt{return}\ v : \mathtt{0}\ \rhd\ \Gamma, E', A}$$

(T-If)

$$\dfrac{\begin{array}{c}\Gamma, E, A \vdash_S e : \square,\ \mathsf{L}\ \rhd\ \Gamma', E', A' \qquad \Gamma', E', A' \vdash_S s_1 : \mathsf{L}_1\ \rhd\ \Gamma_1, E_1, A_1 \\ \Gamma', E', A' \vdash_S s_2 : \mathsf{L}_2\ \rhd\ \Gamma_2, E_2, A_2 \qquad \Gamma_1 =_{\mathtt{unsync}} \Gamma_2\end{array}}{\Gamma, E, A \vdash_S \mathtt{if}\ e\ \{\ s_1\ \}\ \mathtt{else}\ \{\ s_2\ \} : \mathsf{L} + \mathsf{L}_1 + \mathsf{L}_2\ \rhd\ \Gamma_1 + \Gamma_2, E_1 \sqcup E_2, A_1 \sqcup A_2}$$

methods and programs: $\Gamma \vdash \mathtt{m}\ (\overline{T\ x})\{s\} : (\overline{x'}, X) \rightarrow (\nu\, \overline{\varkappa})(\Gamma' \cdot \Gamma'' \cdot \mathsf{L})$ and $\Gamma \vdash \overline{\mathtt{Int}\ a}, \overline{M}\ \{s\} : (\mathcal{L}, \Gamma' \cdot \mathsf{L})$

(T-Method)

$$\dfrac{\begin{array}{c}\Gamma(\mathtt{m}) = (\mathit{this}, \overline{f}, X, \Gamma_{\mathtt{m}}) \rightarrow (E, A) \qquad \overline{g} = int(\overline{f} \cup names(\Gamma_{\mathtt{m}})) \qquad \Gamma' = \Gamma + \Gamma_{\mathtt{m}} + \overline{x} : \overline{f} + \overline{g} : \square + \text{future} : X \\ \Gamma', [\mathit{this} \mapsto \varnothing], \varnothing \vdash_{\{\mathit{this}, \overline{f}\}} s : \mathsf{L}\ \rhd\ \Gamma'', E, A' \qquad \overline{w} = \mathit{flat}(\mathit{this}, \overline{f}, \Gamma_{\mathtt{m}}) \\ \overline{\varkappa} = names(\Gamma'') \setminus names(\Gamma') \qquad A = A' \sqcup \bigsqcup_{h \in \mathrm{dom}(\Gamma'')} \left\{ \left(E_{\mathtt{m}'}|_{\{\mathit{this}, \overline{f}\}} \right) \mid \Gamma''(h) = \lambda Y.\mathtt{m}'(\overline{f}, Y, \Gamma_{\mathtt{m}'}, E_{\mathtt{m}'}) \right\}\end{array}}{\Gamma \vdash \mathtt{m}\ (\overline{T\ x})\{s\} : (\overline{w}, X) \rightarrow (\nu\, \overline{\varkappa})(\Gamma''|_{\overline{\varkappa}} \cdot \Gamma''|_{obj(\overline{f})} \cdot \mathsf{L}\ \&(X, \alpha))}$$

(T-Program)

$$\dfrac{\left(\Gamma \vdash \mathtt{m}\ (\overline{T\ x})\{s\} : \mathcal{L}(\mathtt{m})\right)^{(\mathtt{m}\ (\overline{T\ x})\{s\}) \in \overline{M}} \qquad \Gamma + \mathit{this} : \mathit{main}[\overline{x : \square}]^{\checkmark}, \varnothing, \varnothing \vdash_{\varnothing} s : \mathsf{L}\ \rhd\ \Gamma', E, A}{\Gamma \vdash \{\overline{\mathtt{Int}\ x}, \overline{M}\}\ \{s\} : (\mathcal{L}, \Gamma'|_{Fut(\Gamma')} \cdot \mathsf{L})}$$

Fig. 5: Typing expressions with side effects, statements, methods, and programs

f bound to x is synchronized – henceforth its result is check-marked in the environment. Correspondingly, the futures that are synchronized by f, namely those that are recorded in the effect $E_{\mathtt{m}}$, are synchronized as well. Finally, the rule records in the environment the updates of arguments' fields. Technically this is done using the delegation future type. The behavioural type collects the futures of methods that are running in parallel *and* f, which is annotated with the synchronizing actor name α. This type will let us to compute the dependencies of the parallel methods during the analysis.

Figure 5 shows the rules for expressions with side effects. The rule (T-Invk) creates a new future g corresponding to the invocation and stores it in Γ, after having computed the instance of the method signature. The last premise verifies the compatibility between the effects of the invoked method and those of the other running methods (the current one and the not-yet synchronised ones). The behavioural type collects futures of methods that are running in parallel, including g, which is created by the rule. The future g is not annotated with any actor name because this information is not known here. The substitution on second line replaces synchronised futures by $\Box$ to prevent additional synchronisations on these futures.

The rules for statements are collected in the bottom of Figure 5. The behavioural type of statements is a sum of types that are parallel composition of synchronization dependencies and unsynchronized behaviors. The rules are almost standard. We discuss the rule for returning a future – rule (T-Return-Fut). In this case, the returned value is an unsynchronised future f, therefore the synchronisation of f is bound to the synchronisation of the method under analysis. For this reason, the behavioural type is f_X, where X is the place-holder for the active object synchronising the method currently analysed. The rest of the behavioural type collects the unsynchronised behaviour.

The rules in Figure 5 type methods and programs. In (T-Method), the premises verify the consistency of the typing of `m` in the environment with the typing of its body. In particular, the asynchronous effects of `m` must be the sum of the asynchronous ones in its body, i.e. A', plus the effects of the invocations that have not been synchronized. We notice that the behavioural type of the method has arguments that are structureless: object's fields are removed and the corresponding values are lifted as arguments – see the function *flat*. We also notice that the behavioural type of the body s is extended with a dependency (X, α). This dependency will be instantiated by the synchronising object when it is known. The behavioural type of a method has the shape $(\Gamma \cdot \Gamma' \cdot \mathsf{L})$. The environment Γ defines the fresh names created in the body of the method, it maps future names to either future results $\lambda X.\mathtt{m}(\overline{g}, X, \Gamma'', E)$ or delegations $f \rightsquigarrow g.x$ or object types $\alpha[\overline{a : f}]$. The environments we will use in the behavioural type analysis are a simplified form of the foregoing ones where future results are $\lambda X.\mathtt{m}(\overline{g}, X, \Gamma'')$, it will then be denoted Θ instead of Γ. The environment Γ' records the updates to the arguments $\overline{f}$ performed by the method, we denote it Φ.

Finally, a *behavioral type program* is a pair $\big(\mathcal{L},\ \Theta \cdot \mathsf{L}\big)$, where $\mathcal{L}$ maps *method names* `m` to *method behaviors* $(\overline{f}, X) \to (\nu\, \overline{\varkappa})(\Theta' \cdot \Phi \cdot \mathsf{L}')$, $\overline{f}, X$ are the *formal parameters* of `m`, Θ' and Φ are the *environments* defining respectively the futures created (bound by $\varkappa$) and the updates done to the arguments $\overline{f}$, and L' is the behavior of the body of `m`. The last two elements of a behavioral type program, namely Θ and L, are the *environment* and the *type* of the main body.

Example. The behavioural type of the program of Section 1 is of the form: $(\mathcal{L}, \Theta \cdot f_\star + f_{main} + f'_\star)$ where:

$$\begin{aligned}\Theta = [\, & f \mapsto \lambda X.\mathtt{addToStore}(g, \Box, X, [\, g \mapsto \alpha[n : \Box]^{\checkmark}\,], [\, g \mapsto [n^{\mathtt{W}}]\,]\,),\ g' \mapsto f \rightsquigarrow g.n,\\ & f' \mapsto \lambda X.\mathtt{addToStore}(g, \Box, X, [\, g \mapsto \alpha[n : g']^{\checkmark}\,], [\, g \mapsto [n^{\mathtt{W}}]\,]\,)\,].\end{aligned}$$

We observe that the behavioural type of the main function performs two invocations of `addToStore`. The first invocation is performed on the object α where the field n stores a value $(g \mapsto \alpha[n : \Box]^{\checkmark})$, indeed at that point $n = 0$. The second invocation is performed on the same object but n stores the value written by the first invocation: in Θ we have the delegation $g' \mapsto f \rightsquigarrow g.n$ and in the second method invocation the object field n maps to g'. We can also notice that the first invocation has been synchronized, indeed the presence of the delegation in the environment indicates that the rule (T-Synch-Invk) has been applied. Both invocations of the `addToStore` method write on the field n of the object g, and the effect of both invocations is $[\, g \mapsto [n^{\mathtt{W}}]\,]$.

As stated above, $\mathcal{L}$ stores the behavioural type for each method of the program, then we have an entry for `addToStore` and `store`.

$$\begin{aligned}&\mathcal{L}(\mathtt{addToStore}) = (\beta, \mathit{this}, g, f, X) \to (\nu\, f')(\Theta_{\mathsf{add}} \cdot \Phi_{\mathsf{add}} \cdot \mathsf{L}_{\mathsf{add}}) \text{ where}\\ &\mathsf{L}_{\mathsf{add}} = (\, g_\alpha + f'_\star + f'_X\,) \,\&(X, \beta) \qquad \Phi_{\mathsf{add}} = [\, \mathit{this} \mapsto \beta[n : f']\,]\\ &\Theta_{\mathsf{add}} = [\ f' \mapsto \lambda X.\mathtt{store}(\mathit{this}, f, \Box, X, [\ \mathit{this} \mapsto \beta[n : g]^{\checkmark}\ ], \varnothing)\,]\end{aligned}$$

The behavioural type shows that the method `addToStore` performs three main actions. The first action is the possible synchronization, expressed by g_α, where g is one of the parameters. The second action is the invocation of the method `store` corresponding to future f'. The third action returns the result of the invocation of `store`; expressed by the term f'_X stating that the f' is returned.

Concerning `store` we have: $\mathcal{L}(\mathtt{store}) = (\gamma, \mathit{this}, f, g, X) \to (\varnothing \cdot \varnothing \cdot (X, \gamma))$.

$$\textsc{BT-fun}\quad \frac{\begin{array}{c}\mathit{\Theta}(f) = \lambda X.\mathtt{m}(\overline{f}, X, \mathit{\Gamma}) \\ \mathcal{L}(\mathtt{m}) = (\overline{w}, Y) \to (\nu\,\overline{\varkappa})(\mathit{\Theta}' \cdot \mathit{\Phi} \cdot \mathsf{L}) \\ \kappa \text{ is either } \star \text{ or an object name} \\ \overline{\varkappa'} \text{ fresh} \quad \mathit{\Theta}'' = \mathit{\Theta} + \mathit{\Theta}'[^{\overline{\varkappa'}}/_{\overline{\varkappa}}][^{\mathit{flat}(\overline{f},\mathit{\Gamma})}/_{\overline{w}}] \\ \mathsf{L}' = \mathsf{L}[^{\overline{\varkappa'}}/_{\overline{\varkappa}}][^{\kappa}/_{Y}][^{\mathit{flat}(\overline{f},\mathit{\Gamma})}/_{\overline{w}}]\end{array}}{\mathit{\Theta} \cdot \mathcal{C}[\, f_\kappa \,] \to \mathit{\Theta}'' \cdot \mathcal{C}[\, \mathsf{L}' \,]}
\qquad
\textsc{BT-field}\quad \frac{\begin{array}{c}\mathit{\Theta}(f) = f' \rightsquigarrow g.x \quad \mathit{\Theta}(f') = \lambda X.\mathtt{m}(\overline{f}, X, \mathit{\Gamma}) \\ \mathcal{L}(\mathtt{m}) = (\overline{w}, Y) \to (\nu\,\overline{\varkappa})(\mathit{\Theta}' \cdot \mathit{\Phi} \cdot \mathsf{L}) \\ \mathit{\Phi}' = \mathit{\Phi}[^{\overline{\varkappa'}}/_{\overline{\varkappa}}][^{\mathit{flat}(\overline{f},\mathit{\Gamma})}/_{\overline{w}}] \quad \mathit{\Phi}'(g.x) = h\end{array}}{\mathit{\Theta} \cdot \mathcal{C}[\, f_\kappa \,] \to \mathit{\Theta} \cdot \mathcal{C}[\, h_\kappa \,]}$$

Fig. 6: Behavioural type reduction rules

4 Behavioural type soundness and analysis

The soundness of the type system is demonstrated by a subject reduction theorem expressing that if a runtime configuration cn is well typed and $cn \to cn'$ then cn' is well typed as well. While the theorem is almost standard, we cannot guarantee type-preservation, instead we exhibit a relation between the type $\mathit{\Theta} \cdot \mathsf{L}$ of cn and the type $\mathit{\Theta}' \cdot \mathsf{L}'$ of cn'. Informally, this relation connects (i) the presence of a deadlock in a configuration with the presence of circularity in a type and (ii) the presence of a circularity in the evaluation of $\mathit{\Theta}' \cdot \mathsf{L}'$ with the circularities of the evaluation of $\mathit{\Theta} \cdot \mathsf{L}$. The formal proof of type soundness is similar to that in [7], it is available in the full version [12].

We focus here on the way we address delegation types that are new relatively to [7]. The evaluation of a behavioural types is defined by a transition relation between types $\mathit{\Theta} \cdot \mathsf{L}$ that follows the rules in Figure 6 and includes a specific rule for delegation types. We use *type contexts*:

$$\mathcal{C}[\;] \;::=\; [\;] \mid \mathsf{L} \,\&\, \mathcal{C}[\;] \mid \mathcal{C}[\;] \,\&\, \mathsf{L} \mid \mathsf{L} + \mathcal{C}[\;] \mid \mathcal{C}[\;] + \mathsf{L}$$

Overall, BT-FUN and BT-FIELD indicate that the behavioural type semantics is simply the unfolding of function invocations and the evaluation of delegations. More precisely, rule BT-FUN replaces a future with the the body of the corresponding invocation. The environment $\mathit{\Theta}$ is augmented with the names defined in this body. Note that $\mathit{\Theta}''$ is well-defined because $\mathrm{dom}(\mathit{\Theta}) \cap \mathrm{dom}(\mathit{\Theta}'[^{\overline{\varkappa'}}/_{\overline{\varkappa}}][^{\mathit{flat}(\overline{f},\mathit{\Gamma})}/_{\overline{w}}]) = \varnothing$ and $(\mathit{flat}(\overline{f}, \mathit{\Gamma}) \cup \overline{w}) \cap \overline{\varkappa'} = \varnothing$. The behavioural type L' is defined by a classical substitution. The substitution $[^{\mathit{flat}(\overline{f},\mathit{\Gamma})}/_{\overline{w}}]$ replaces active object and future namesin $\overline{w}$. This substitution can generate terms of the form $\square_\alpha$, those terms can safely be replaced by $\mathsf{0}$. Rule BT-FIELD computes futures f bound to delegations $f' \rightsquigarrow g.x$, i.e. when the invocation corresponding to f' has updated the field x of the argument g; it retrieves the instance of $\mathit{\Phi}$ in the method of f' and infers h, the future written in the accessed field.

Definition 4. *Let $\mathsf{L} \equiv_d \mathsf{L}'$ whenever L and L' are equal up-to commutativity and associativity of "&" and "+", identity of $\mathsf{0}$ for & and +, and distributivity of & over +, namely $\mathsf{L} \,\&(\mathsf{L}' + \mathsf{L}'') = \mathsf{L} \,\&\, \mathsf{L}' + \mathsf{L} \,\&\, \mathsf{L}''$.*

The behavioural type L has a circularity *if there are $\alpha_1, \cdots, \alpha_n$ and $\mathcal{C}[\;]$ such that* $\mathtt{n}\mathsf{L} \equiv_d \mathcal{C}[\,(\alpha_1, \alpha_2) \,\&\, \cdots \,\&(\alpha_n, \alpha_1)\,]$.

A type $\mathit{\Theta} \cdot \mathsf{L}$ has a circularity *if $\mathit{\Theta} \cdot \mathsf{L} \to^* \mathit{\Theta}' \cdot \mathsf{L}'$ and L' has a circularity.*

Below we write $\mathit{\Gamma} \vdash cn : \mathit{\Theta} \cdot \mathsf{L}$ to say that the configuration cn has type $\mathit{\Theta} \cdot \mathsf{L}$ in the environment $\mathit{\Gamma}$. This judgment requires an extension of the type system in Figures 4 to configurations (see [12]). The main properties of the type system and its extension to configurations are stated below.

Theorem 1. *Let P be a* gASP *program and suppose that $\mathit{\Gamma} \vdash P : (\mathcal{L}, \mathit{\Theta} \cdot \mathsf{L})$, then:*
1. *$\mathit{\Gamma} \vdash cn : \mathit{\Theta} \cdot \mathsf{L}$ where cn is the initial configuration;*
2. *if $cn \to^* cn'$ then there are $\mathit{\Gamma}'$, $\mathit{\Theta}'$ and L' such that $\mathit{\Gamma}' \vdash cn' : \mathit{\Theta}' \cdot \mathsf{L}'$ and if $\mathit{\Theta}' \cdot \mathsf{L}'$ has a circularity then also $\mathit{\Theta} \cdot \mathsf{L}$ has a circularity.*
3. *if $\mathit{\Theta} \cdot \mathsf{L}$ has no circularity then P is deadlock-free.*

Our technique reduces the problem of detecting deadlocks in a gASP program to that of detecting circularities in a behavioural type. It is worth to notice that these types have models that are infinite states because of recursion and creation of new names. Notwithstanding this fact, the problem of absence of circularities in a behavioural type is decidable. The solver uses a fixpoint technique that is defined in [14,8], which has been adapted to the types of this paper in [7].

Example. We show how a circularity appears when we apply the reduction rule on the illustrative example. The behavioural type of the example was shown in Section 3, we start from the behavioral type of the main function and describe the main reduction steps.

We focus on the third term ($f'_\star$) that refers to the second method invocation of `addToStore`. The rule BT-Fun replaces the behavioural type of method invocation $f'_\star$ with the body of `addToStore` properly instantiated. Here the method invocation related to f' is $\Theta(f') = \lambda X.\texttt{addToStore}(\cdots)$, we take the behavioural type $\mathsf{L}_{\texttt{add}}$, build the substitution $[^{h}/_{f'}][^{\star}/_{X}][^{\alpha,g,g',\square}/_{\beta,this,g,f}]$ that instantiates the parameters adequately, and obtain the behaviour: $(g'_\alpha + h_\star + h_X\ ) \,\&(\star, \alpha)$, additionally $\Theta' = \Theta + \Theta'_{\texttt{add}}$ where $\Theta'_{\texttt{add}}$ is obtained from $\Theta_{\texttt{add}}$ applying the same substitution. Finally we can apply BT-Fun and obtain the reduction $\Theta \cdot (f_\star + f_{main} + f'_\star) \to \Theta' \cdot (f_\star + f_{main} + (g'_\alpha + h_\star + h_X\ ) \,\&(\star, \alpha))$.

We then focus on the term g'_α that refers to the synchronization of the field `n` during the execution of the second invocation of `addToStore`. The type associated to g' ($\Theta'(g') = f \rightsquigarrow g.n$) denotes that, when typing, we don't know the method invocation related to the future stored in `n`, we only know that the method invocation related to f has stored a future inside `n`. To solve this delegation and then discover the name of the future stored in the that field we apply the rule Bt-Field and obtain: $\Theta' \cdot (\cdots + (g'_\alpha + h_\star + h_X\ ) \,\&(\star, \alpha)) \to \Theta' \cdot (\cdots + (h'_\alpha + h_\star + h_X\ ) \,\&(\star, \alpha))$. This reduction only replaces g'_α with h'_α where $h' = \Phi'_{\texttt{add}}(g.n)$ and $\Phi'_{\texttt{add}}$ corresponds to the instantiation of $\Phi_{\texttt{add}}$ accordingly to the invocation related to f: $\Theta(f) = \lambda X.\texttt{addToStore}(g, \square, X, [\, g \mapsto \alpha[n\!:\!\square]^{\checkmark}\ ]])$ with the substitution $[^{h}/_{f'}][^{\alpha,g,\square,\square}/_{\beta,this,g,f}]$.

Now we focus on the term h'_α and, as in the first step, we can apply the rule BT-Fun we replace h'_α with the behavioral type of `store` opportunely instantiated and obtain: $\Theta' \cdot (\cdots + (h'_\alpha + h_\star + h_X\ ) \,\&(\star, \alpha)) \to \Theta' \cdot (\cdots + ((\alpha, \alpha) + h_\star + h_X\ ) \,\&(\star, \alpha))$ as the behavioural type of `store` is reduced to a pair.

The circularity (α, α) highlights a potential deadlock in our program. Indeed the method `store` is called on α and then the result of this invocation is awaited in the method `addToStore` in α, as no further order is ensured on the execution of these requests, this circularity indeed reveals a potential deadlock.

5 Concluding remarks

This article defines a technique for analysing deadlocks of stateful active objects that is based on behavioural type systems. The technique also takes into account stateful objects that store futures in their fields. This required us to analyse synchronisation patterns where the future synchronisation occurs in a different context from the asynchronous invocation that created the future. The behavioural types that are obtained by the type system are analysed by a solver that detects circularities and identifies potential deadlocks.

To deal with implicit futures, we use a novel paradigm in our analyses, that consider "*every element as a future*". This also allows us to deal with aliasing and with the fact that the future updates are performed on place at any time.

Related Work. Up-to our knowledge, the first paper proposing effect systems for analysing data races of concurrent systems dates back to the late 80's [15]. In fact, our approach of annotating the types to express further intentional properties of the semantics of the program is very similar to that of Lucassen and Gifford. The first application of a type and effect system to deadlock analysis is [3]. In that case programmers must specify a partial order among the locks and the type checker verifies that threads acquire locks in the descending order. In our case, no order is predefined and the absence of circularities in the process synchronisations is obtained in a post-typing phase. In [6], the authors generate a finite graph of program points by integrating an effect and point-to analysis for computing aliases with an analysis returning (an over-approximation of) points that may run in parallel. In the model presented in [6], future are passed (by-value) between methods only as parameters or return values, the possibility of storing future in object field is treated as a possible extension and not formalized. Furthermore this aspect is not considered combined to the possibility of having infinite recursion. However, [6] analyses *finite* abstraction of the computational models of the language. In our case, the behavioural type model associated to the program handles unbounded states.

Model checking is often used to verify stateful distributed systems. In particular, [18] uses the characteristics of actor languages to limit, by partial order reduction, the model to check. [1] provides an parametrised model of an active object application that is abstracted into a finite model afterwards. Contrarily to us, these results are restricted to a finite abstraction of the state of the system. Two articles [2,10] translate active objects into Petri-nets and model-check the generated net; these approaches cannot verify infinite systems because they would lead to an infinite Petri-net or an infinite set of colours for the tokens.

We refer the interested reader to [9] (Section 8) for a further comparison of alternative analysis techniques.

References

1. R. Ameur-Boulifa, L. Henrio, O. Kulankhina, E. Madelaine, and A. Savu. Behavioural semantics for asynchronous components. *Journal of Logical and Algebraic Methods in Programming*, 89:1 – 40, 2017.
2. Frank S. De Boer, Mario Bravetti, Immo Grabe, Matias David Lee, Martin Steffen, and Gianluigi Zavattaro. A Petri Net Based Analysis of Deadlocks for Active Objects and Futures. In *FACS 2012*, Lecture Notes in Computer Science. Springer.
3. Chandrasekhar Boyapati, Robert Lee, and Martin C. Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *Proc. OOPSLA 2002*.
4. Denis Caromel, Ludovic Henrio, and Bernard P. Serpette. Asynchronous sequential processes. *Inf. Comput.*, 207(4):459–495, 2009.
5. B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order.* Cambridge University Press, 2002.
6. Antonio Flores-Montoya, Elvira Albert, and Samir Genaim. May-happen-in-parallel based deadlock analysis for concurrent objects. In *Proc. FORTE/FMOODS 2013.* Springer, 2013.
7. Elena Giachino, Ludovic Henrio, Cosimo Laneve, and Vincenzo Mastandrea. Actors may synchronize, safely! In *Proceedings of PPDP 2016.* ACM, 2016.
8. Elena Giachino, Naoki Kobayashi, and Cosimo Laneve. Deadlock analysis of unbounded process networks. In *Proceedings of CONCUR 2014.* Springer, 2014.
9. Elena Giachino, Cosimo Laneve, and Michael Lienhardt. A framework for deadlock detection in core ABS. *Software and Systems Modeling*, 15(4):1013–1048, 2016.
10. Anastasia Gkolfi, Crystal Chang Din, Einar Broch Johnsen, Martin Steffen, and Ingrid Chieh Yu. Translating active objects into colored petri nets for communication analysis. In *Proc. FSEN 2017*, Lecture Notes in Computer Science. Springer.
11. P. Haller and M. Odersky. Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2-3):202–220, 2009.
12. Ludovic Henrio, Cosimo Laneve, and Vincenzo Mastandrea. Analysis of synchronisations in stateful active objects (full paper). `https://bitbucket.org/VMastandrea/effects-full-pdf/raw/HEAD/Effects-FULL.pdf`.
13. Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. ABS: A core language for abstract behavioral specification. In *Proceedings of FMCO 2010*, volume 6957 of *LNCS*, pages 142–164. Springer, 2010.
14. Naoki Kobayashi and Cosimo Laneve. Deadlock analysis of unbounded process networks. *Inf. Comput.*, 252:48–70, 2017.
15. John M. Lucassen and David K. Gifford. Polymorphic effect systems. In *Proc. of POPL 1988*, pages 47–57. ACM Press, 1988.
16. Ben Morris. *The Symbian OS Architecture Sourcebook: Design and Evolution of a Mobile Phone OS.* Wiley, 2007.
17. Joachim Niehren, Jan Schwinghammer, and Gert Smolka. A Concurrent Lambda Calculus with Futures. *Theoretical Computer Science*, 364(3):338–356, nov 2006.
18. Marjan Sirjani. Rebeca: Theory, applications, and tools. In *FMCO*, 2006.
19. Derek Wyatt. *Akka Concurrency.* Artima, 2013.

A Flattening, circularities and fixpoint definition of the interpretation function.

In this section we report the definitions from [8] slightly adapted to our current model. Let $\mathtt{R}$ be a set whose elements are either pairs (κ, β), where κ ranges over actor, future names, and variables X or terms f_κ. We observe that, if the set of names is finite, then every set $\mathtt{R}$ built with such names is finite as well. In addition, the collection of all sets $\mathtt{R}$ is also finite. We use $\mathcal{R}, \mathcal{R}', \cdots$ to range over sets of relations $\{\mathtt{R}_1, \cdots, \mathtt{R}_m\}$. Let

- $\mathtt{R}^+$ be the *transitive closure* of $\mathtt{R}$ (namely $\mathtt{R}^+$ is the least relation such that $\mathtt{R} \subseteq \mathtt{R}^+$ and such that $(\kappa, \alpha), (\alpha, \beta) \in \mathtt{R}^+$ implies $(\kappa, \beta) \in \mathtt{R}^+$).
- $\{\mathtt{R}_1, \cdots, \mathtt{R}_m\} \Subset \{\mathtt{R}'_1, \cdots, \mathtt{R}'_n\}$ if and only if, for all $\mathtt{R}_i$, there is $\mathtt{R}'_j$ such that $\mathtt{R}_i \subseteq \mathtt{R}'^{+}_j$.
- $(\alpha_0, \alpha_1), \cdots, (\alpha_{n-1}, \alpha_n) \Subset \{\mathtt{R}_1, \cdots, \mathtt{R}_m\}$ if and only if there is $\mathtt{R}_i$ such that $(\alpha_0, \alpha_1), \cdots, (\alpha_{n-1}, \alpha_n) \in \mathtt{R}_i$.
- $\{\mathtt{R}_1, \cdots, \mathtt{R}_m\} \,\&\, \{\mathtt{R}'_1, \cdots, \mathtt{R}'_n\} \overset{def}{=} \{\mathtt{R}_i \cup \mathtt{R}'_j \mid 1 \le i \le m \text{ and } 1 \le j \le n\}$.

Definition 5. *A set* $\mathtt{R}$ has a circularity *if* $(\alpha, \alpha) \in \mathtt{R}^+$ *for some* α. *A set of elements* $\mathtt{R}$, *noted* $\mathcal{R}$, has a circularity *if there is* $\mathtt{R} \in \mathcal{R}$ *that has a circularity.*

Behavioral types define sets $\mathcal{R}$. This is displayed by the following function. Let $\mathcal{L}$ be a set of method definitions and let $I(\cdot)$, called *flattening*, be a function either on future environments and behavioral types or on method names that (i) maps a method name $\mathtt{m}$ defined in $\mathcal{L}$ to elements $\mathcal{R}$ and (ii) is defined on behavioral types as follows

$$\begin{array}{rll}
I(\Theta \cdot 0) = & \{\varnothing\} & \\
I(\Theta \cdot (\kappa, \beta)) = & \{\{(\kappa, \beta)\}\} & \\
I(\Theta \cdot f_\kappa) = & I(\mathtt{m})[^{flat(\overline{f}, \Gamma), \kappa}/_{\overline{w}, X}] & \text{if } \Theta(f) = \lambda X.\mathtt{m}(\overline{f}, X, \Gamma) \text{ and } \mathcal{L}(\mathtt{m}) = (\overline{w}, X) \to (\nu\, \overline{\varkappa})(\Theta_\mathtt{m} \cdot \Phi_\mathtt{m} \cdot \mathsf{L}_\mathtt{m}) \\
I(\Theta \cdot f_\kappa) = & I(\mathtt{m}')[^{flat(\overline{g'}, \Gamma'), \kappa}/_{\overline{w}, X}] & \text{if } \Theta(f) = f' \rightsquigarrow g.x \ \wedge\ \Theta(f') = \lambda X.\mathtt{m}(\overline{f}, X, \Gamma) \ \wedge \\
& & \mathcal{L}(\mathtt{m}) = (\overline{w}, X) \to (\nu\, \overline{\varkappa})(\Theta_\mathtt{m} \cdot \Phi_\mathtt{m} \cdot \mathsf{L}_\mathtt{m}) \ \wedge\ \Phi_\mathtt{m}[^{flat(\overline{f}, \Gamma)}/_{\overline{w}}](g.x) = h \ \wedge \\
& & \Theta(h) = \lambda X.\mathtt{m}'(\overline{g'}, Y, \Gamma') \ \wedge\ \mathcal{L}(\mathtt{m}') = (\overline{w'}, X') \to (\nu\, \overline{\varkappa'})(\Theta_{\mathtt{m}'} \cdot \Phi_{\mathtt{m}'} \cdot \mathsf{L}_{\mathtt{m}'}) \\
I(\Theta \cdot f_\kappa) = & \{\{f_\kappa\}\} & \text{if } f \notin \text{dom}(\Theta) \\
I(\Theta \cdot \mathsf{L} \,\&\, \mathsf{L}') = & I(\Theta \cdot \mathsf{L}) \,\&\, I(\Theta \cdot \mathsf{L}') & \\
I(\Theta \cdot \mathsf{L} + \mathsf{L}') = & I(\Theta \cdot \mathsf{L}) \cup I(\Theta \cdot \mathsf{L}') &
\end{array}$$

Note that $I(\Theta \cdot \mathsf{L})$ is unique up to a renaming of names that do not occur free in L. Let $I^\perp$ be the map such that, for every $\mathtt{m}$, $I^\perp(\mathtt{m}) = \{\varnothing\}$.

Definition 6. *A state* $\Theta \cdot \mathsf{L}$ has a circularity *if* $I^\perp(\Theta \cdot \mathsf{L})$ *has a circularity. A behavioral type program* $\big(\mathcal{L}, \Theta \cdot \mathsf{L}\big)$ has a circularity *if there exist* Θ' *and* L' *such that* $\Theta \cdot \mathsf{L} \to^* \Theta' \cdot \mathsf{L}'$ *and* $\Theta' \cdot \mathsf{L}'$ *has a circularity.*

The basic item of our algorithm is the computation of methods' interpretation. This computation is performed by means of a standard fixpoint technique that is detailed below.

Let $\big(\mathcal{L}, \Theta \cdot \mathsf{L}\big)$ be a program such that pairwise different method definitions in $\mathcal{L}$ have disjoint formal parameters. Let A be the set of (i) formal parameters of definitions in $\mathcal{L}$, of (ii) free names in $\Theta \cdot \mathsf{L}$ and (iii) containing a special fresh name $\varkappa$. Since A is finite, then every set $\mathtt{R}_A$ built with names in A is finite and similarly for $\mathcal{R}_A$. In particular, the sets $\mathcal{R}_A$ are ordered by the $\subseteq$ relation and form a *finite* lattice [5].

Definition 7. *Let* $\mathcal{L} = \{\mathtt{m}_i \mapsto (\overline{w}_i, X_i) \to (\nu\, \overline{\varkappa}_i)(\Theta_i \cdot \Phi_i \cdot \mathsf{L}_i) \mid i \in \{1..k\}\}$. *The family of flattening functions* $I^{(k)}$ *is defined as follows*

$$\begin{array}{ll}
I^{(0)}(\mathtt{m}_i) & = \{\varnothing\} \\
I^{(k+1)}(\mathtt{m}_i) & = \{\, \mathtt{proj}_{\overline{w}_i, X_i}(R^+) \mid R \in I^{(k)}(\Theta_i \cdot \mathsf{L}_i)\}
\end{array}$$

where $\mathtt{proj}_{\overline{w}, X}(R) \overset{def}{=} \{(\beta, \gamma) \mid\ (\beta, \gamma) \in R \text{ and } \beta, \gamma \in \overline{w}, X\} \bigcup \{(\varkappa, \varkappa) \mid\ (\delta, \delta) \in R \text{ and } \delta \notin \overline{w}, X\}$.

We notice that $I^{(0)}$ is the function $I^\perp$ previously shown. Since, for every k, $I^{(k)}(\mathtt{m}_i)$ ranges over a finite lattice, by the fixpoint theory [5], there exists m such that $I^{(m)}$ is a fixpoint, namely $I^{(m)} \approx I^{(m+1)}$ where $\approx$ is the equivalence relation induced by $\Subset$. In the following, we let I, called the *interpretation function* (of a behavioral type), be the least fixpoint $I^{(m)}$.

The following theorem states the correctness and completeness of our algorithm. Similarly to [8], there is a relation between the circularities of the set $I^{(k)}(\Theta \cdot \mathsf{L})$ and, whenever $\Theta \cdot \mathsf{L} \to \Theta' \cdot \mathsf{L}'$, between the circularities of $I^{(k)}(\Theta \cdot \mathsf{L})$ and of $I^{(k)}(\Theta' \cdot \mathsf{L}')$.

Theorem 2. *A behavioural type program* $\big(\mathcal{L}, \Theta \cdot \mathsf{L}\big)$ *has a circularity if and only if* $I_{\mathcal{L}}(\Theta \cdot \mathsf{L})$ *has a circularity.*

The proof of the theorem is very similar to the corresponding one in [8].

B Behavioural type soundness (Deadlock)

To demonstrate the correctness of the type system and the analysis we have separated the part of the type system concerning to the deadlock analysis (appendix B) and the part related to the effect analysis (appendix C). In this section we will focus only on the deadlock analysis aspect starting from the hypothesis that the analysed program has only deterministic effects (see Definition 3). The correctness of our system guarantees that, if the deadlock-freedom of a behavioral type program associated to a `gASP` program with deterministic effects is assessed, then also the corresponding `gASP` program is guaranteed to be deadlock-free. In other words we are proving that if the analysis shows that no deadlock is present in the behavioral type of the original program, then none of its executions can lead to a deadlock. To this end, we prove that if there is no circularity in the type of a runtime configuration then this configuration exhibits no deadlock, and that if a configuration reduces to a configuration with a circularity then the original configuration already had a circularity. This ensures that if no circularity is found in the behavioral type of a `gASP` program then there is no deadlock in the original program. We state again the Theorem **??** as follow.

Theorem 3. *Let P be a* `gASP` *program with deterministic effects (see Definition 3) and cn be a configuration of its operational semantics, with behavioral type $\Theta \cdot \mathsf{L}$.*
1. *If $\Theta \cdot \mathsf{L}$ has no circularity then cn is deadlock-free;*
2. *if $cn \to cn'$ and the behavioral type $\Theta' \cdot \mathsf{L}'$ of cn' has a circularity, then a circularity is already present in $\Theta \cdot \mathsf{L}$, the behavioral type of cn;*

The theorem relies on Theorem 4 (subject reduction) and on a crucial property of the later stage relation [7, Theorem 5.2].

In order to prove the points *1* and *2* of Theorem 3 we need an extension of the typing to runtime configurations (Section B.1); additionally to prove the point *2* we also need the definition of a *later-stage* relation between behavioral types (Section B.3).

B.1 Runtime Type System for Deadlock detection

In order to infer the behavioral types for runtime configuration we define a runtime type system. To this aim we extend the syntax of behavioral types and define *extended futures* F and *behavioral type for configuration* K as follows:

$$\begin{array}{rcll}
\mathbb{b} & ::= & \square \mid f \mid \alpha[\overline{x:f}] & \text{basic type}\\
\mathbb{f} & ::= & \mathbb{b} \mid \lambda X.\mathtt{m}(f, \overline{g}, X, \Gamma, E) \mid f \rightsquigarrow g.x & \text{future type}\\
F & ::= & f \mid {}^{s}f & \text{extended futures}\\
\kappa & ::= & \star \mid \alpha \mid X & \text{synchronizers}\\
\mathsf{L} & ::= & 0 \mid (\kappa, \alpha) \mid f_{\kappa} \mid \mathsf{L} + \mathsf{L} \mid \mathsf{L} \,\&\, \mathsf{L} & \text{behavioral type}\\
\mathsf{K} & ::= & \mathsf{L} \mid (\nu\, \overline{\varkappa})(\Theta \cdot \mathsf{L}) \mid \mathsf{K} \,\&\, \mathsf{K} & \text{behavioral type for configuration}
\end{array}$$

As regards F, they are introduced for distinguishing two kinds of future names: i) f that has been used in the type system as a static time representation of a future, but it is now used as its runtime representation; ii) ${}^{s}f$ now replaces f in its role of static time future (it is typically used to reference a future that is not created yet).

This type system is a simpler version of the one given in section 3 where we are focusing only on the deadlock analysis part leaving out the aspects related with the effects. This is the reason why the typing judgements are simpler than the corresponding one in the type system, the principle differences are:

1) In the future type $\lambda X.\mathtt{m}(f, \overline{g}, X, \Gamma_{\mathtt{m}}, E)$ we have that E now is a set used to collect two kind of information: the future names of the parameters synchronized by the method `m` (this set of future names is a subset of the domain of $\Gamma_{\mathtt{m}}$) and the fields of the arguments modified by `m`, represented by elements like $g.x$ (g is the argument and x is the field of g in which `m` has stored a future).

2) the $rt\_unsync(\cdot)$ function on environments Δ is similar to $rt\_unsync(\cdot)$ in Section 3, except that it now grabs all ${}^{s}f$ and all futures f. More precisely we define $Fut_R(\Delta)$, $aFut_R(\Delta)$, and $rtunsync\Delta$ to be the functions

$$Fut_R(\Delta) \overset{def}{=} \{F \mid F \in \mathrm{dom}(\Delta)\} \qquad aFut_R(\Delta) \overset{def}{=} \{F \in Fut_R(\Delta) \mid \Delta(F) = \Delta(F)^{\times}\}$$

$$rt\_unsync(\Delta) \overset{def}{=} \&_{F \in aFut_R(\Delta)} F_{\star},$$

where $Fut_R(\Gamma)$ collects all the (static and runtime) futures names in $\mathrm{dom}(\Delta)$, $aFut_R(\Delta)$ is the subset of $Fut_R(\Gamma)$ that contains future names F (static and runtime) such that $\Delta(F)$ is not "checkmarked" (*i.e.* the set of not-yet-synchronized futures); and $rt\_unsync(\Delta)$ performs the parallel composition of the behavioral types of the not-yet-synchronized method invocations.

Runtime Type System for Deadlock detection (typing rules)

configuration and processes: $\Delta \vdash cn : \mathsf{L}$ and $\Delta \vdash p : (\nu\,\overline{\varkappa})(\Theta \cdot \mathsf{L})$

(TR-Future-Undef)

$$\dfrac{\Delta \vdash f : \lambda X.\mathtt{m}(\overline{g}, X, \Delta_{\mathtt{m}}, E_{\mathtt{m}})}{\Delta \vdash f(\bot) : \mathsf{0}}$$

(TR-Future-Eval)

$$\dfrac{\Delta \vdash f : \lambda X.\mathtt{m}(\overline{g}, X, \Delta_{\mathtt{m}}, E_{\mathtt{m}})^{\checkmark} \qquad \Delta \vdash w : f}{\Delta \vdash f(w) : \mathsf{0}}$$

(TR-Actor)

$$\dfrac{\begin{array}{c}\Delta(\alpha) = \alpha[\overline{y : f}] \qquad \Delta \vdash \overline{v} : \overline{f} \qquad \Delta' = \Delta + this : \alpha[\overline{y : f}] \\ \Delta' \vdash p : \mathsf{K}_0 \qquad \forall i \in 1..n.\ \Delta' \vdash q_i : \mathsf{K}_i\end{array}}{\Delta \vdash \alpha(\{\overline{y \mapsto v}\}, p, \{q_1, \cdots, q_n\}) : \mathop{\&}\limits_{i=0}^{n} \mathsf{K}_i}$$

(TR-Parallel)

$$\dfrac{\Delta \vdash cn_1 : \mathsf{K}_1 \qquad \Delta \vdash cn_2 : \mathsf{K}_2}{\Delta \vdash cn_1\ cn_2 : \mathsf{K}_1 \,\&\, \mathsf{K}_2}$$

(TR-Process)

$$\dfrac{\begin{array}{c}\Delta \vdash f : \lambda X.\mathtt{m}(this, \overline{g}, X, \Delta_{\mathtt{m}}, E_{\mathtt{m}}) \qquad \Delta \vdash \overline{v} : \overline{g} \qquad \Delta' = \Delta + \Delta_{\mathtt{m}} + \mathrm{destiny} : f + x : \overline{g} + \mathrm{future} : X \\ \Delta', \varnothing \vdash_{\{this, \overline{g}\}} s : \mathsf{L} \ \triangleright\ \Delta'', E' \qquad \overline{\varkappa} = names(\Delta'') \setminus names(\Delta')\end{array}}{\Delta \vdash \{destiny \mapsto f, \overline{x \mapsto v} \mid s\} : (\nu\,\overline{\varkappa})(\Delta''|_{Fut_R(\Delta'')} \cdot \mathsf{L})}$$

values, variables and method names: $\Delta \vdash x : \mathbb{b}$ and $\vdash \mathtt{m} : (\overline{f}, X, \Gamma') \to (E, A)$

(TR-Val)

$$\dfrac{v \quad integer\text{-}value \text{ or } \mathtt{null}}{\Delta \vdash v : \square}$$

(TR-Var)

$$\dfrac{\Delta(x) = F}{\Delta \vdash x : F}$$

(TR-Fut)

$$\dfrac{\Delta(F) = \mathbb{f}^{[\checkmark]}}{\Delta \vdash F : \mathbb{f}^{[\checkmark]}}$$

(TR-Field)

$$\dfrac{\Delta(this.x) = F}{\Delta \vdash x : F}$$

(TR-Method-Sign)

$$\dfrac{\Delta(\mathtt{m}) = (\overline{F}, X, \Gamma) \to (E) \qquad \sigma \text{ renaming}}{\vdash m : (\sigma(F), \sigma(\overline{g}), \sigma(X), \Gamma \circ \sigma) \to (E \circ \sigma)}$$

synchronizations: $\Gamma, E \ {}^{\oplus}\!\vdash_S \ e : \mathsf{L} \ \triangleright\ \Gamma', E'$

(TR-Synchronized)

$$\dfrac{\Delta, E \vdash v : F \qquad \Delta \vdash F : \mathbb{f}^{\checkmark}}{\Delta, E \ {}^{\oplus}\!\vdash_S \ v : \mathsf{0} \ \triangleright\ \Delta, E}$$

(TR-Sync-Invk)

$$\dfrac{\begin{array}{c}\Delta \vdash this : \alpha[\cdots]^{\checkmark} \qquad \Delta \vdash x : F \\ \Delta \vdash F : \lambda X.\mathtt{m}(\overline{F'}, X, \Gamma_{\mathtt{m}}, E_{\mathtt{m}}) \qquad \Delta' = \Delta[F^{\checkmark}][H^{\checkmark}]^{H \in \mathrm{dom}(E_{\mathtt{m}})} \\ \Delta'' = \Delta'([G.y \mapsto G'][G' \mapsto F \rightsquigarrow G.y])^{G.y \in E_{\mathtt{m}},\ g' \text{ fresh}}\end{array}}{\Delta, E \ {}^{\oplus}\!\vdash_S \ x : F_{\alpha} \,\&\, rt\_unsync(\Delta'') \ \triangleright\ \Delta'', E \cup E_{\mathtt{m}}|_S}$$

(TR-Sync-Field)

$$\dfrac{\begin{array}{c}\Delta \vdash this : \alpha[\cdots] \qquad \Delta \vdash x : F \\ \Delta \vdash F : G \rightsquigarrow this.x \qquad \Delta' = \Delta[F^{\checkmark}]\end{array}}{\Delta, E \ {}^{\oplus}\!\vdash_S \ x : F_{\alpha} \,\&\, rt\_unsync(\Delta') \ \triangleright\ \Delta', E}$$

(TR-Sync-Param)

$$\dfrac{\begin{array}{c}\Delta \vdash this : \alpha[\cdots] \qquad \Delta \vdash x : F \\ \Delta \vdash F : \mathbb{f} \qquad F \in S \qquad \Delta' = \Delta[F^{\checkmark}]\end{array}}{\Delta, E \ {}^{\oplus}\!\vdash_S \ x : F_{\alpha} \,\&\, rt\_unsync(\Delta') \ \triangleright\ \Delta', E \cup \{F\}}$$

expressions with side effects: $\Delta, E \vdash_S z : f,\ \mathsf{L}\ \triangleright\ \Delta', E'$

(TR-Future)
$$\frac{f \in \mathrm{dom}(\Delta)}{\Delta, E \vdash_S f : f,\ \mathtt{0}\ \triangleright\ \Delta, E}$$

(TR-Actor-Name)
$$\frac{F \in \mathrm{dom}(\Delta) \qquad \Delta \vdash F : \alpha[\cdots]^{\checkmark}}{\Delta, E \vdash_S \alpha : F,\ \mathtt{0}\ \triangleright\ \Delta, E}$$

(TR-Atom)
$$\frac{\Delta \vdash v : F}{\Delta, E \vdash_S v : F,\ \mathtt{0}\ \triangleright\ \Delta, E}$$

(TR-Expression)
$$\frac{\Delta, E \ {}^{\oplus}\!\vdash_S\ v : \mathsf{L}\ \triangleright\ \Delta', E' \qquad \Delta', E' \ {}^{\oplus}\!\vdash_S\ v' : \mathsf{L}'\ \triangleright\ \Delta'', E''}{\Delta, E \vdash_S v \oplus v' : {}^{[s]}\Box,\ \mathsf{L} + \mathsf{L}'\ \triangleright\ \Delta'', E''}$$

(TR-New)
$$\frac{\Delta \vdash \overline{v} : \overline{G} \qquad \beta, F \text{ fresh} \qquad \overline{x} = \mathit{fields}(\mathtt{Act})}{\Delta, E \vdash_S \mathtt{new}\ \mathtt{Act}(\overline{v}) : F,\ \mathtt{0}\ \triangleright\ \Delta[f \mapsto \beta[\overline{x : G}]^{\checkmark}], E}$$

(TR-Invk)
$$\frac{\begin{array}{c} \Delta \vdash v : F \qquad \Delta \vdash F : \beta[\cdots]^{\checkmark} \qquad \Delta \vdash \overline{v} : \overline{F'} \qquad \overline{h} = f \cup \mathit{obj}(\overline{f'}) \\ \vdash \mathtt{m} : (F, \overline{F'}, X, \Delta|_{\overline{h}}) \to (E_{\mathtt{m}}) \qquad {}^{s}g \text{ fresh} \qquad \overline{G'} = \overline{F'}[{}^{\Box}/_{int(sFut(\Gamma))}] \qquad \Delta_{\mathtt{m}} = (\Delta|_{\overline{h}})[{}^{\Box}/_{int(sFut(\Gamma))}] \\ \Delta' = \Delta[{}^{s}g \mapsto \lambda X.\mathtt{m}(F, \overline{G'}, X, \Delta_{\mathtt{m}}, E_{\mathtt{m}})] \end{array}}{\Delta, E \vdash_S v.\mathtt{m}(\overline{v}) : {}^{s}g,\ {}^{s}g_{\star} \mathbin{\&} \mathit{rt\_unsync}(\Delta)\ \triangleright\ \Delta', E}$$

statements $\Delta, E \vdash_S s : \mathsf{L}\ \triangleright\ \Delta', E'$

(TR-Assign-Var-Exp)
$$\frac{\begin{array}{c} x \notin \mathit{fields}(\mathtt{Act}) \\ \Delta, E \vdash_S z : F,\ \mathsf{L}\ \triangleright\ \Delta', E' \end{array}}{\Delta, E \vdash_S x = z : \mathsf{L}\ \triangleright\ \Delta[x \mapsto F], E'}$$

(TR-Assign-Field-Exp)
$$\frac{\begin{array}{c} x \in \mathit{fields}(\mathtt{Act}) \\ \Delta, E \vdash_S z : F,\ \mathsf{L}\ \triangleright\ \Delta', E' \end{array}}{\Delta, E \vdash_S x = z : \mathsf{L}\ \triangleright\ \Delta[\mathit{this}.x \mapsto F], E' \cup \{\mathit{this}.x\}}$$

(TR-Assign-Var-Fut)
$$\frac{x \notin \mathit{fields}(\mathtt{Act}) \qquad \Delta \vdash x : F}{\Delta, E \vdash_S x = f : \mathtt{0}\ \triangleright\ \Delta[x \mapsto f], E}$$

(TR-Assign-Field-Fut)
$$\frac{x \in \mathit{fields}(\mathtt{Act}) \qquad \Delta \vdash x : F}{\Delta, E \vdash_S x = f : \mathtt{0}\ \triangleright\ \Delta[\mathit{this}.x \mapsto f], E}$$

(TR-Skip)
$$\Delta, E \vdash_S \mathtt{skip} : \mathtt{0}\ \triangleright\ \Delta, E$$

(TR-Seq)
$$\frac{\Delta, E \vdash_S s_1 : \mathsf{L}_1\ \triangleright\ \Delta_1, E_1 \qquad \Delta_1, E_1 \vdash_S s_2 : \mathsf{L}_2\ \triangleright\ \Delta_2, E_2}{\Delta, E \vdash_S s_1 ; s_2 : \mathsf{L}_1 + \mathsf{L}_2\ \triangleright\ \Delta_2, E_2}$$

(TR-Return-Fut)
$$\frac{\begin{array}{c} \Delta \vdash v : f \qquad \Delta \vdash f : \mathbb{f} \qquad \Delta(\text{future}) = X \\ \Delta(\text{destiny}) = f' \qquad \Delta \vdash f' : \lambda X.\mathtt{m}(\overline{g}, X, \Gamma_{\mathtt{m}}, E_{\mathtt{m}}) \end{array}}{\Delta, E \vdash_S \mathtt{return}\ v : f_X \mathbin{\&} \mathit{rt\_unsync}(\Delta \setminus f)\ \triangleright\ \Delta, E}$$

(TR-Return-Val)
$$\frac{\begin{array}{c} \Delta \vdash v : f \qquad \Delta \vdash f : \mathbb{f}^{\checkmark} \\ \Delta(\text{destiny}) = f' \qquad \Delta \vdash f' : \lambda X.\mathtt{m}(\overline{g}, X, \Gamma_{\mathtt{m}}, E_{\mathtt{m}}) \end{array}}{\Delta \vdash_S \mathtt{return}\ v : \mathtt{0}\ \triangleright\ \Delta}$$

(TR-If)
$$\frac{\begin{array}{c} \Delta, E \vdash_S e : {}^{[s]}\Box,\ \mathsf{L}\ \triangleright\ \Delta', E' \qquad \Delta', E' \vdash_S s_1 : \mathsf{L}_1\ \triangleright\ \Delta_1, E_1 \qquad \Delta', E' \vdash_S s_2 : \mathsf{L}_2\ \triangleright\ \Delta_2, E_2 \\ \Delta_1 =_{\mathtt{unsync}} \Delta_2 \end{array}}{\Delta, E \vdash_S \mathtt{if}\ e\ \{\, s_1 \,\}\ \mathtt{else}\ \{\, s_2 \,\} : \mathsf{L} + \mathsf{L}_1 + \mathsf{L}_2\ \triangleright\ \Delta_1 + \Delta_2, E_1 \cup E_2}$$

B.2 Proof of Theorem 3.1

Since we have a type system for configurations we can now prove the first statement of the Theorem 3.

Lemma 1. *Let suppose $\Delta \vdash cn : \mathsf{K}$ and let D be the set of dependencies of cn. Then, we have $D \subset I_{\mathcal{L}}(\mathsf{K})$.*

Proof. By Definition 2, if cn has a dependency (α, β), then there exist $cn' = \alpha(a, \{\ell \mid C[f]\}, \overline{q})\ \beta(a', p', \overline{q'}) \in cn$ such that $f \in \mathit{destinies}(p', \overline{q'})$. By runtime typing rules TR-Actor, TR-Process, TR-Seq and TR-Synch-*, the behavioural type of cn' is $(\nu\, \overline{\varkappa})\big(\Theta \cdot (f_\alpha + \mathsf{L}_s) \mathbin{\&} (X, \alpha)\big) \mathbin{\&} (\mathop{\&}_{i=1}^{n} \mathsf{K}_i)$.

Having that:

- by rule TR-Invk $\Theta(f) = \lambda X.\mathtt{m}(g, \overline{g'}, X, \Delta_{\mathtt{m}}, E_{\mathtt{m}})$;
- $\Delta_{\mathtt{m}}(g) = \beta[\cdots]^{\checkmark}$;
- $\Delta(\mathtt{m}) = (\nu\, \overline{\varkappa})\big(\Theta \cdot \mathsf{L} \mathbin{\&} (X, \beta)\big)$;

we can infere that during the computation of $I_{\mathcal{L}}(\mathsf{K})$ the rule BT-Red will replace f_α with the behavioural type of the body of $\mathtt{m}$ where the X will be instantiated with α ($\Delta(\mathtt{m})[{}^{\alpha}/_{X}]$). This substitution will generate the pair (α, β) $\Box$

B.3 Later stage relation

As we said before, we need to define a *later-stage* relation between behavioral types (denoted $\succeq_\Delta$), which is a syntactic relationship between behavioral types. We can simplify the basic laws of the later-stage relation saying that a method invocation is larger than the instantiation of its method behavior, and a sum type is larger than each element of the sum. The later-stage relation is the least congruence with respect to runtime behavioral type that contains the rules in Figure 7.

LS-Empty
$$(\nu\,\overline{\varkappa})(\Theta \cdot \mathsf{0}) =_\Delta \mathsf{0}$$

LS-Delete
$$\mathsf{0} \mathbin{\&} \mathsf{K} \succeq_\Delta \mathsf{K}$$

LS-Global
$$\frac{\mathsf{K}_1 \succeq_\Delta \mathsf{K}_1'}{\mathsf{K}_1 \mathbin{\&} \mathsf{K} \succeq_\Delta \mathsf{K}_1' \mathbin{\&} \mathsf{K}}$$

LS-Behavior
$$\frac{\mathsf{K} = (\nu\,\overline{\varkappa})(\Theta \cdot \mathsf{L}) \qquad \mathsf{K}' = (\nu\,\overline{\varkappa'})(\Theta \cdot \mathsf{L}') \qquad \mathsf{L} \succeq_\Delta \mathsf{L}'}{\mathsf{K} \succeq_\Delta \mathsf{K}'}$$

LS-Null
$$\mathsf{L} \succeq_\Delta \mathsf{0}$$

LS-Plus
$$\mathsf{L}_1 + \mathsf{L}_2 \succeq_\Delta \mathsf{L}_i$$

LS-Parallel
$$\frac{\mathsf{L}_1 \succeq \mathsf{L}_1'}{\mathsf{L}_1 \mathbin{\&} \mathsf{L} \succeq_\Delta \mathsf{L}_1' \mathbin{\&} \mathsf{L}}$$

LS-Invk
$$\frac{\begin{array}{c}\Theta(f) = \lambda X.\mathtt{m}(\overline{f}, X, \Gamma) \\ \Delta(\mathtt{m}) = (\overline{w}, X') \rightarrow (\nu\,\overline{\varkappa})(\Theta_{\mathtt{m}} \cdot \mathsf{L}_{\mathtt{m}}) \qquad \overline{\varkappa'} \text{ fresh} \\ \mathsf{L}' = \mathsf{L}_{\mathtt{m}}[^{\overline{\varkappa'}}/_{\overline{\varkappa}}][^{\kappa}/_{X}][^{\mathit{flat}(\overline{f},\Gamma)}/_{\overline{w}}] \qquad \Theta' = \Theta_{\mathtt{m}}[^{\overline{\varkappa'}}/_{\overline{\varkappa}}][^{\mathit{flat}(\overline{f},\Gamma)}/_{w}]\end{array}}{(\nu\,\overline{\varphi})(\Theta \cdot \mathcal{C}[\,f_\kappa\,]) \succeq_\Delta (\nu\,\overline{\varphi})(\Theta \cdot \mathcal{C}[\,\mathsf{0}\,]) \mathbin{\&} (\nu\,\overline{\varkappa'})(\Theta' \cdot \mathsf{L}')}$$

Fig. 7: Later-stage relation rules

B.4 Subject Reduction

Since we have defined both type system for runtime configuration and later stage relation, we can state the Subject Reduction theorem. The subject reduction theorem expresses that if a runtime configuration cn is well typed and $cn \rightarrow cn'$ then cn' is well typed. We cannot demonstrate a statement guaranteeing the equality of types of cn and cn', because our types are behavioural. the type of cn $(\Theta \cdot \mathsf{L})$, and the type of cn', $(\Theta' \cdot \mathsf{L}')$.

Theorem 4 (Subject Reduction). *Let $\Delta \vdash_R cn : K$ and $cn \rightarrow cn'$. Then there exist Δ', K', and an injective renaming of actor and future names $\imath$ such that*
- $\Delta' \vdash_R cn' : K'$ *and*
- $\imath(K) \succeq_\Delta K'$

Proof of Theorem 4 (Subject Reduction) The proof is a case analysis on the reduction rule used in $cn \rightarrow cn'$.

Case: Serve

Serve
$$\alpha(a, \varnothing, \overline{q} \cup \{p\}) \rightarrow \alpha(a, p, \overline{q})$$

Proof. Let Δ, K_p, $\mathsf{K}_1 \cdots \mathsf{K}_n$ exist, by rule TR-Actor we obtain that $\Delta \vdash \alpha(a, \varnothing, \overline{q} \cup \{p\}) \ : \ \mathsf{K}_p \mathbin{\&} (\mathop{\&}\limits_{i=1}^{n} \mathsf{K}_i)$. With the same Δ we can type the configuration $\alpha(a, p, \overline{q})$ by applying the rule TR-Actor and we gain that $\Delta \vdash \alpha(a, p, \overline{q}) \ : \ \mathsf{K}_p \mathbin{\&} (\mathop{\&}\limits_{i=1}^{n} \mathsf{K}_i)$.

It is trivial to demonstrate that the relation $\mathsf{K}_p \mathbin{\&} (\mathop{\&}\limits_{i=1}^{n} \mathsf{K}_i) \succeq \Delta \mathsf{K}_p \mathbin{\&} (\mathop{\&}\limits_{i=1}^{n} \mathsf{K}_i)$ holds.

□

Case: Return

$$\textsc{Return}\quad \frac{[\![v]\!]_{a+\ell} = w \qquad \ell(\texttt{destiny}) = f}{\begin{array}{c}\alpha(a, \{\ell \mid \texttt{return } v\}, \overline{q})\ f(\bot)\\ \to \alpha(a, \varnothing, \overline{q})\ f(w)\end{array}}$$

Let $\varDelta$ and K exists, such that $\varDelta \vdash \alpha(a, \{\ell \mid \texttt{return } v;\}, \overline{q})\ f(\bot) : \mathsf{K}$, then there exist $\varDelta'$ such that $\varDelta' \vdash \alpha(a, \varnothing, \overline{q})\ f(w) : \mathsf{K}'$ and $\mathsf{K} \succeq_{\varDelta} \mathsf{K}'$

Proof. We can distinguish two cases:

1) v is a value or a synchronized future ($\varDelta(v) = f \wedge \varDelta(f) = \mathbb{f}^{\checkmark}$).

   By rules TR-Actor andTR-Process we obtain that

   – there exists $\varDelta''$ that extends $\varDelta$ (like in the application of TR-Actor and TR-Process) such that $\varDelta''(v) = f$ and $\varDelta''(f) = \mathbb{f}^{\checkmark}$;

   – there exists a set of future names S, as in the application of TR-Process, such that $S \subseteq \mathrm{dom}(\varDelta'')$;

   – there exists a set collecting effects E;

   – and by applying TR-Return-Val we infere that $\varDelta'', E \vdash_S \texttt{return } v : \mathsf{0} \ \triangleright\ \varDelta'', E$.

   It follows from the hypothesis and rules TR-Parallel, TR-Actor, TR-Process and TR-Future-Undef that there exist $\overline{\varkappa}$, $\varTheta$, $\mathsf{K}_1, \cdots, \mathsf{K}_n$ such that:
   $\varDelta \vdash \alpha(a, \{\ell \mid \texttt{return } v;\}, \overline{q})\ f(\bot) :\ (\nu\,\overline{\varkappa})\big(\varTheta \cdot \mathsf{0}\big) \mathbin{\&} (\mathop{\&}\limits_{i=1}^{n} \mathsf{K}_i)$.
   Let us choose $\varDelta' = \varDelta[f^{\checkmark}][w \mapsto \varDelta(f)^{\checkmark}]$, by applying the rules TR-Parallel, TR-Actor and TR-Future-Eval we gain that $\varDelta' \vdash \alpha(a, \varnothing, \overline{q})\ f(w)\ :\ (\mathop{\&}\limits_{i=1}^{n} \mathsf{K}_i)$.
   Therefore by the rules LS-Empty and LS-Delte it is trivial to prove that the following relation holds $(\nu\,\overline{\varkappa})\big(\varTheta \cdot \mathsf{0}\big) \mathbin{\&} (\mathop{\&}\limits_{i=1}^{n} \mathsf{K}_i) \succeq_{\varDelta} (\mathop{\&}\limits_{i=1}^{n} \mathsf{K}_i)$.

2) v is an unsynchronized future ($\varDelta(v) = f \wedge \varDelta(f) = \mathbb{f}$).

   By rules TR-Actor and TR-Process we gain that

   – there exists $\varDelta''$ that extends $\varDelta$ with $\varDelta''(v) = f$ and $\varDelta''(f) = \mathbb{f}$;

   – there exist a set of future names S such that $S \subseteq \mathrm{dom}(\varDelta'')$;

   – there exists a set collecting effects E;

   – and finally by TR-Return-Fut we infere $\varDelta'' \vdash_S \texttt{return } v : f_X \mathbin{\&} \mathit{unsync}(\varDelta'' \setminus f)\ \triangleright\ \varDelta''$.

   Considering the previous hypothesis and by the rules TR-Parallel, TR-Actor and TR-Future-Undef we can state that there exist $\overline{\varkappa}$, $\varTheta$, $\mathsf{K}_1, \cdots, \mathsf{K}_n$ such that:
   $\varDelta \vdash \alpha(a, \{\ell \mid \texttt{return } v;\}, \overline{q})\ f(\bot) :\ (\nu\,\overline{\varkappa})\big(\varTheta \cdot f_X \mathbin{\&} \mathit{rt\_unsync}(\varDelta'' \setminus f)\big) \mathbin{\&} (\mathop{\&}\limits_{i=1}^{n} \mathsf{K}_i)$.
   Let us choose $\varDelta' = \varDelta[f^{\checkmark}][w \mapsto \varDelta(f)^{\checkmark}]$, by applying the rules TR-Parallel, TR-Actor and TR-Future-Eval we can infere that: $\varDelta' \vdash \alpha(a, \varnothing, \overline{q})\ f(w)\ :\ (\mathop{\&}\limits_{i=1}^{n} \mathsf{K}_i)$.
   Therefore by the rules LS-Behavior, LS-Empty, and LS-Delete we can prove that the relation $(\nu\,\overline{\varkappa})\big(\varTheta \cdot f_X \mathbin{\&} \mathit{unsync}(\varDelta'' \setminus f)\big) \mathbin{\&} (\mathop{\&}\limits_{i=1}^{n} \mathsf{K}_i) \succeq_{\varDelta} (\mathop{\&}\limits_{i=1}^{n} \mathsf{K}_i)$ holds.

□

Case: Update

$$\textsc{Update}\quad \frac{\begin{array}{c}(a+\ell)(x) = f\\ (a+\ell)[x \mapsto w] = a' + \ell'\end{array}}{\begin{array}{c}\alpha(a, \{\ell \mid s\}, \overline{q})\ f(w)\\ \to \alpha(a', \{\ell' \mid s\}, \overline{q})\ f(w)\end{array}}$$

Let $\varDelta$ and K exist, such that $\varDelta \vdash \alpha(a, \{\ell \mid s\}, \overline{q})\, f(w) : \mathsf{K}$, then there exist $\varDelta'$ such that $\varDelta' \vdash \alpha(a', \{\ell' \mid s\}, \overline{q})\ f(w) : \mathsf{K}'$ and $\mathsf{K} \succeq_{\varDelta} \mathsf{K}'$

Proof. By rules TR-Actor, TR-Process and TR-Future-Eval we have that there exists Δ'' that extends Δ like in the application of TR-Actor and TR-Process and the following hypothesis hold:
- $\Delta(f) = \lambda X.\mathtt{m}(\overline{g}, X, \Delta_{\mathtt{m}}, E_{\mathtt{m}})^{\checkmark}$ and $\Delta(w) = f$;
- there exist a set of future names S such that $S = \{\overline{g}\}$;
- there exists a set collecting effects E;
- $\Delta'', E \vdash_S s : \mathsf{L} \;\triangleright\; \Delta''', E'$.

It follows from the hypothesis and by TR-Parallel, TR-Actor, TR-Process and TR-Future-Eval that there exist $\overline{\varkappa}$, Θ, L, $\mathsf{K}_1, \cdots, \mathsf{K}_n$ such that: $\Delta \vdash \alpha(a, \{\ell \mid s\}, \overline{q})\ f(w) :\ (\nu\,\overline{\varkappa})\big(\Theta \cdot \mathsf{L}\big) \mathbin{\&} (\underset{i=1}{\overset{n}{\&}} \mathsf{K}_i)$.
Let $\Delta' = \Delta[x \mapsto \Delta(w)]$ we have that $\Delta' \vdash (a' + \ell')(x) : \Delta'(x)$, now applying the rules TR-Parallel, TR-Actor, TR-Process and TR-Future-Eval we can conclude that $\Delta' \vdash \alpha(a', \{\ell' \mid s\}, \overline{q})\ f(w)\ :$ $(\nu\,\overline{\varkappa})\big(\Theta \cdot \mathsf{L}\big) \mathbin{\&} (\underset{i=1}{\overset{n}{\&}} \mathsf{K}_i)$.
It is trivial to proof that $(\nu\,\overline{\varkappa})\big(\Theta \cdot \mathsf{L}\big) \mathbin{\&} (\underset{i=1}{\overset{n}{\&}} \mathsf{K}_i) \succeq_\Delta (\nu\,\overline{\varkappa})\big(\Theta \cdot \mathsf{L}\big) \mathbin{\&} (\underset{i=1}{\overset{n}{\&}} \mathsf{K}_i)$.

□

Case: Assign

$$\frac{\overset{\text{Assign}}{\llbracket e \rrbracket_{a+\ell} = w \qquad (a + \ell)[x \mapsto w] = a' + \ell'}}{\alpha(a, \{\ell \mid x = e\ ;\ s\}, \overline{q}) \to \alpha(a', \{\ell' \mid s\}, \overline{q})}$$

Let Δ and K exist where $\Delta \vdash \alpha(a, \{\ell \mid x = e; s\}, \overline{q}) : \mathsf{K}$, then there exist Δ' such that $\Delta' \vdash \alpha(a', \{\ell' \mid s\}, \overline{q}) : \mathsf{K}'$ and $\mathsf{K} \succeq_\Delta \mathsf{K}'$

Proof. We can distinguish two cases:
1) x is a local variable ($x \notin \mathit{fields}(\mathtt{Act})$)

   By rules TR-Actor and TR-Process
   - there exists Δ_2 that extend Δ as in the application of TR-Actor and TR-Process;
   - there exist a set of future names S such that $S \subseteq \mathrm{dom}(\Delta'')$ like defined in TR-Process;
   - there exists a set collecting effects E;
   - by rule TR-Assign-Var-Exp we gain $\Delta_2, E \vdash_S x = e : \mathsf{L}_e \;\triangleright\; \Delta_4, E'$ and $\Delta_4 = \Delta_3[x \mapsto f]$ (L_e, Δ_3, E' and f came from the typing of the expression e. The possible shape of e generates two subcases, ones in which e is a a value or a variable and another in which e is an arithmetic expression. We can say that by rule TR-Atom or TR-Expression, which are the rules that are applied for the first and second case respectively, we have that $\Delta_2, E \vdash_S e : f,\ \mathsf{L}_e \;\triangleright\; \Delta_3, E'$. We do not handle in the detail this two cases because this has no relevant impact in the proof, and we let f, L_e, Δ_3 and E' be the be the future name, the behavioural type, the update of Δ_2 and the update of E that come from the application of the proper rule.)
   - TR-Seq we obtain $\Delta_2, E \vdash_S x = e\ ;\ s : \mathsf{L}_e + \mathsf{L}_s \;\triangleright\; \Delta_2', E'$ with Δ_2' be the update of Δ_4 that is obtained from typing s.

   Considering the previous hypothesis and by the rules TR-Parallel, TR-Actor, TR-Process and TR-Seq we can state that there exist $\overline{\varkappa}$, Θ, $\mathsf{K}_1, \cdots, \mathsf{K}_n$ such that:
   $\Delta \vdash \alpha(a, \{\ell \mid x = e; s\}, \overline{q}) :\ (\nu\,\overline{\varkappa})\big(\Theta \cdot \mathsf{L}_e + \mathsf{L}_s\big) \mathbin{\&} (\underset{i=1}{\overset{n}{\&}} \mathsf{K}_i)$.
   Let us chose $\Delta' = \Delta_4$ by rules TR-Parallel, TR-Actor and TR-Process we obtain that:
   $\Delta' \vdash \alpha(a, \{\ell \mid x = e; s\}, \overline{q}) :\ (\nu\,\overline{\varkappa})\big(\Theta \cdot \mathsf{L}_s\big) \mathbin{\&} (\underset{i=1}{\overset{n}{\&}} \mathsf{K}_i)$.
   Now we can demonstrate that by LS-Plus, we have $\mathsf{L}_e + \mathsf{L}_s \succeq_\Delta \mathsf{L}_s$ which allows us to say that $(\nu\,\overline{\varkappa})\big(\Theta \cdot \mathsf{L}_e + \mathsf{L}_s\big) \mathbin{\&} (\underset{i=1}{\overset{n}{\&}} \mathsf{K}_i) \succeq_\Delta (\nu\,\overline{\varkappa})\big(\Theta \cdot \mathsf{L}_s\big) \mathbin{\&} (\underset{i=1}{\overset{n}{\&}} \mathsf{K}_i)$.
2) x is a field (x $\in \mathit{fields}(\mathtt{Act})$)

   By rules TR-Actor and TR-Process

- there exists Δ_2 that extend Δ (like in the application of TR-Actor and TR-Process);
- there exist a set of future names S such that $S \subseteq \mathrm{dom}(\Delta'')$ as in the application of TR-Process;
- there exists a set collecting effects E;
- by TR-Assign-Field-Exp we obtain $\Delta_2, E \vdash_S x = e : \mathsf{L}_e \ \triangleright\ \Delta_4, E''$ with $\Delta_4 = \Delta_3[x \mapsto f]$ and and $E'' = E' \cup \{\mathit{this}.x\}$ (L_e, Δ_3, f and E' are as in the previous case.)
- by TR-Seqwe gain $\Delta_2, E \vdash_S x = e; s : \mathsf{L}_e + \mathsf{L}_s \ \triangleright\ \Delta_2', E''$ with Δ_2' be the update of Δ_4 that comes from typing s.

As in the previous case, let us chose $\Delta' = \Delta_4$ by rules TR-Parallel, TR-Actor and TR-Process we obtain that:
$\Delta' \vdash \alpha(a, \{\ell \mid x = e; s\}, \overline{q}) :\ (\nu\,\overline{\varkappa})\big(\Theta \cdot \mathsf{L}_s\big) \,\&\, (\overset{n}{\underset{i=1}{\&}} \mathsf{K}_i)$.
Now we can demonstrate that by LS-Plus, we have $\mathsf{L}_e + \mathsf{L}_s \succeq_\Delta \mathsf{L}_s$ which allows us to say that $(\nu\,\overline{\varkappa})\big(\Theta \cdot \mathsf{L}_e + \mathsf{L}_s\big) \,\&\, (\overset{n}{\underset{i=1}{\&}} \mathsf{K}_i) \succeq_\Delta (\nu\,\overline{\varkappa})\big(\Theta \cdot \mathsf{L}_s\big) \,\&\, (\overset{n}{\underset{i=1}{\&}} \mathsf{K}_i)$.

□

Case: New

$$\textsc{New}\ \frac{[\![\overline{v}]\!]_{a+\ell} = \overline{w} \qquad \beta \text{ fresh} \qquad \overline{y} = \mathit{fields}(\mathtt{Act})}{\begin{array}{c}\alpha(a, \{\ell \mid x = \mathtt{new\ Act}(\overline{v})\ ;\ s\}, \overline{q}) \\ \to \alpha(a, \{\ell \mid x = \beta\ ;\ s\}, \overline{q})\ \ \beta([\overline{y} \mapsto \overline{w}], \varnothing, \varnothing)\end{array}}$$

Let Δ and K such that $\Delta \vdash \alpha(a, \{\ell \mid x = \mathtt{new\ Act}(\overline{v}); s\}, \overline{q}) : \mathsf{K}$, then there exist Δ' such that $\Delta' \vdash \alpha(a, \{\ell \mid x = \beta; s\}, \overline{q})\ \beta([\overline{y} \mapsto \overline{w}], \varnothing, \varnothing) : \mathsf{K}'$ and $\mathsf{K} \succeq_\Delta \mathsf{K}'$

Because of the restriction of the language we have that x could not be a field.

Proof. By rules TR-Actor, TR-Process and TR-assign-Var-Exp
- there exists Δ'' that extends Δ as defined in the application of TR-Actor and TR-Process;
- there exist a set of future names S such that $S \subseteq \mathrm{dom}(\Delta'')$ like in the application of TR-Process;
- there exists a set collecting effects E;
- by rule TR-New we gain $\Delta'', E \vdash_S \mathtt{new\ Act}(\overline{v}) : f,\ 0 \ \triangleright\ \Delta''[f \mapsto \beta[\overline{a\!:\!g}]^{\checkmark}], E$ with f fresh.

Considering the previous hypothesis and by the rules TR-Parallel, TR-Actor, TR-Process, TR-Seq and TR-Assign-Var-Exp we can state that there exist $\overline{\varkappa}$, Θ, $\mathsf{K}_1, \cdots, \mathsf{K}_n$ such that:
$\Delta \vdash \alpha(a, \{\ell | x = v.\mathtt{m}(\overline{v}); s\}, \overline{q}) : (\nu\,\overline{\varkappa})\big(\Theta \cdot\ (0 + \mathsf{L}_s) \,\&(X, \alpha)\big) \,\&\, (\overset{n}{\underset{i=1}{\&}} \mathsf{K}_i)$
Let $\Delta' = \Delta[x \mapsto h][h \mapsto \gamma[\overline{y\!:\!g}]^{\checkmark}]$ and $\imath(h) = f$, $\imath(\beta) = \gamma$, where $\imath$ is an injective function on future names and actor names, by TR-Parallel, TR-Actor, TR-Process, TR-assign-Var-Exp and TR-Actor-Name we have that:
$\Delta' \vdash \alpha(a, \{\ell \mid x = \beta; s\}, \overline{q})\ \beta([\overline{y} \mapsto \overline{w}], \varnothing, \varnothing) : (\nu\,\overline{\varkappa})\big(\Theta \cdot\ (0 + \mathsf{L}_s) \,\&(X, \alpha)\big) \,\&\, (\overset{n}{\underset{i=1}{\&}} \mathsf{K}_i) \,\&\, 0$.
It is trivial to verify that $(\nu\,\overline{\varkappa})\big(\Theta \cdot\ (0{+}\mathsf{L}_s) \,\&(X, \alpha)\big) \,\&\, (\overset{n}{\underset{i=1}{\&}} \mathsf{K}_i) \succeq_\Delta (\nu\,\overline{\varkappa})\big(\Theta \cdot\ (0 + \mathsf{L}_s) \,\&(X, \alpha)\big) \,\&\, (\overset{n}{\underset{i=1}{\&}} \mathsf{K}_i) \,\&\, 0$.

□

Case: Invk

$$\textsc{Invk}\ \frac{\begin{array}{ccc}[\![v]\!]_{a+\ell} = \beta & [\![\overline{v}]\!]_{a+\ell} = \overline{w} & \beta \neq \alpha \\ f \text{ fresh} & \mathrm{bind}(\beta, m, \overline{w}, f) = p' & \end{array}}{\begin{array}{c}\alpha(a, \{\ell \mid x = v.\mathtt{m}(\overline{v})\ ;\ s\}, \overline{q})\ \beta(a', p, \overline{q'}) \\ \to \alpha(a, \{\ell \mid x = f\ ;\ s\}, \overline{q})\ \beta(a', p, \overline{q'} \cup \{p'\})\ f(\bot)\end{array}}$$

Let Δ and K exist where $\Delta \vdash \alpha(a, \{\ell \mid x = v.\mathtt{m}(\overline{v}); s\}, \overline{q})\ \beta(a', p, \overline{q'}) : \mathsf{K}$, then there exist Δ' such that $\Delta' \vdash \alpha(a, \{\ell \mid x = f; s\}, \overline{q})\ \beta(a', p, \overline{q'} \cup \{p'\})\ f(\bot) : \mathsf{K}'$ and $\mathsf{K} \succeq_\Delta \mathsf{K}'$.

Because of the restriction of the language we have that v can not be the result of a method invocation then the access of v could not perform a synchronization.

Proof. By applying the rules TR-Parallel, TR-Actor, TR-Process andTR-Seq we have:

- there exist Δ'' that extend Δ (like in the application of TR-Actor and TR-Process) such that $\Delta''(\overline{v}) = \overline{g}$
- there exist a set of future names S such that $S \subseteq \mathrm{dom}(\Delta'')$ as defined in rule TR-Process;
- there exists a set collecting effects E;
- by TR-Invk we can infere that $\Delta'', E \vdash_S v.\mathtt{m}(\overline{v}) : {}^s f \ , \ {}^s f_\star \,\&\, \mathit{rt\_unsync}(\Delta'') \ \triangleright \ \Delta''', E$ such that $\Delta''' = \Delta''[f \mapsto \lambda X'.\mathtt{m}(g, \overline{g'}, X', \Delta_{\mathtt{m}}, E_{\mathtt{m}})]$ where ${}^s f$ is fresh.

Considering the previous hypothesis and by the rules TR-Parallel, TR-Actor, TR-Process, TR-Seq and TR-Invk it follows that exist Θ, $\overline{\varkappa}$, L_s, X and $\mathsf{K}_1, \cdots, \mathsf{K}_n$ such that $\Delta \vdash \alpha(a, \{\ell | x = v.\mathtt{m}(\overline{v}); s\}, \overline{q}) : (\nu\, \overline{\varkappa}, {}^s f)\big(\Theta \cdot \ ({}^s f_\star \,\&\, \mathit{rt\_unsync}(\Delta'') + \mathsf{L}_s) \,\&(X, \alpha)\big) \,\&\, (\overset{n}{\underset{i=1}{\&}} \mathsf{K}_i)$. By applying the rule TR-Actor there also exist $\mathsf{K}'_1, \cdots \mathsf{K}'_n$ such that $\Delta \vdash \beta(a', p', q') : \mathsf{K}'_p \,\&\, (\overset{n}{\underset{i=1}{\&}} \mathsf{K}'_i)$. It is trivial to notice that by TR-Parallel Δ types $\alpha(a, \{\ell \mid x = v.\mathtt{m}(\overline{v}); s\}, \overline{q})\ \beta(a', p, \overline{q'})$ in the parallel composition of the types of the two configurations.
Let us chose Δ' such that $\Delta' = \Delta[f \mapsto \lambda X'.\mathtt{m}(g, \overline{g'}, X', \Delta_{\mathtt{m}}, E_{\mathtt{m}})]$ such that and $\imath({}^s f) = f$ where $\imath$ is an injective function on future names, by rules TR-Parallel, TR-Actor, TR-Process, TR-Seq and TR-Fut we finally obtain that:

- $\Delta' \vdash \alpha(a, \{\ell \mid x = f; s\}, \overline{q}) : (\nu\, \overline{\varphi}, f)\big(\Theta \cdot (0 + \mathsf{L}_s) \,\&(X, \alpha)\big) \,\&\, (\overset{n}{\underset{i=1}{\&}} \mathsf{K}_i)$
- $\Delta' \vdash \beta(a', p', \overline{q'} \cup \{p''\})\ f(\bot) : \mathsf{K}'_p \,\&\, (\overset{n}{\underset{i=1}{\&}} \mathsf{K}'_i) \,\&\, \mathsf{K}_{p''}$ where $\mathsf{K}_{p''}$ is the behavioral type of the method $\mathtt{m}$ instantiated with $g, \overline{g'}, X', \Delta_{\mathtt{m}}$.

Also in this case it is trivial to see that Δ' types $\alpha(a, \{\ell \mid x = f; s\}, \overline{q})\ \beta(a', p', \overline{q'} \cup \{p''\})\ f(\bot)$ in the parallel composition of the types associated to the two element of the configuration.

Moreover, by LS-Invk we can conclude that $(\nu\, \overline{\varkappa}, f)\big(\Theta \cdot \ (f^s_\star \,\&\, \mathit{rt\_unsync}(\Delta'') + \mathsf{L}_s) \,\&(X, \alpha)\big) \succeq_\Delta (\nu\, \overline{\varphi}, f)\big(\Theta \cdot (0 + \mathsf{L}_s) \,\&(X, \alpha)\big) \,\&\, \mathsf{K}_{p''}$.

□

C Behavioural type soundness (Effects)

Theorem 5. *Let P be a* $\texttt{gASP}$ *and cn be a configuration of its operational semantics, and let Γ exists, we have that $\Gamma \vdash cn \Rightarrow cn$ has a queue with deterministic effects.*

Theorem 6. *Let $\Gamma \vdash cn$ and $cn \rightarrow cn'$. Then there exist Γ' such that $\Gamma' \vdash cn'$.*

Lemma 2. *Let $\Gamma \vdash cn \triangleright E$ and $cn \rightarrow cn'$. There there exist Γ', E', and an injective renaming of future and actor names $\imath$ such that:*
- $\Gamma \vdash cn' \triangleright E'$
- $E' \subseteq \imath(E)$

The proof of Theorem 6 and Lemma 2 is a case analysis on the reduction rule used in $cn \rightarrow cn'$.

Runtime Type System for Effect analysis (typing rules)

In order to study the effect for runtime configuration we define a runtime type system. This type system is a simpler version of the one given in section 3 where we are focusing only on the effect analysis part leaving out all the aspects related with deadlock. This is the reason why the typing judgments are simpler then the corresponding one previously shown.

Note: it is relevant to notice that the type system is monotonic for effects, which means that each rule only add new effects and there are no rules that can remove effects from E or A.

configuration and processes: $\Delta \vdash cn : \mathsf{L}$ and $\Delta \vdash p : (\nu\, \overline{\varkappa})(\Theta \cdot \mathsf{L})$

(TR-Future-Undef)
$$\frac{\Delta \vdash f : \lambda X.\mathtt{m}(\overline{g}, \Delta_{\mathtt{m}}, E_{\mathtt{m}})}{\Delta \vdash f(\bot)}$$

(TR-Future-Eval)
$$\frac{\Delta \vdash f : \lambda X.\mathtt{m}(\overline{g}, \Delta_{\mathtt{m}}, E_{\mathtt{m}})^{\checkmark} \qquad \Delta \vdash w : f}{\Delta \vdash f(w)}$$

(TR-Actor)
$$\frac{\begin{array}{c}\Delta(\alpha) = \alpha[\overline{y : f}] \qquad \Delta \vdash \overline{v} : \overline{f} \qquad \Delta' = \Delta + \mathit{this} : \alpha[\overline{y : f}] \\ \Delta', [\mathit{this} \mapsto \varnothing], \varnothing \vdash p \ \triangleright\ E_0, A_0 \\ \forall i \in 1..n.\ \Delta', [\mathit{this} \mapsto \varnothing], \varnothing \vdash q_i \ \triangleright\ E_i, A_i\end{array}}{\Delta \vdash \alpha(\{\overline{y \mapsto v}\}, p, \{q_1, \cdots, q_n\}) \ \triangleright\ \bigsqcup_{0 \leq k \leq n} (E_k \sqcup A_k)}$$

(TR-Parallel)
$$\frac{\Delta \vdash cn_1 \ \triangleright\ E_1 \qquad \Delta \vdash cn_2 \ \triangleright\ E_2}{\Delta \vdash cn_1\ cn_2 \ \triangleright\ E_1 \sqcup E_2}$$

(TR-Process)
$$\frac{\begin{array}{c}\Delta \vdash f : \lambda X.\mathtt{m}(\mathit{this}, \overline{g}, \Delta_{\mathtt{m}}, E_{\mathtt{m}}) \qquad \Delta \vdash \overline{v} : \overline{g} \qquad \overline{g'} = \mathit{int}(\overline{g}) \\ \Delta + \Delta_{\mathtt{m}} + x : \overline{g}, E, A \vdash_{\{\mathit{this}, \overline{g}\}} s : \mathsf{L} \ \triangleright\ \Delta', E', A' \\ A'' = A' \sqcup \bigsqcup_{h \in \mathrm{dom}(\Gamma')} \left\{ \left(E_{\mathtt{m}'}|_{\{\mathit{this}, \overline{g}\}} \right) \mid \Delta'(h) = E_{\mathtt{m}'} \right\}\end{array}}{\Delta, E, A \vdash_{\{\mathit{this}, \overline{g}\}} \{\mathit{destiny} \mapsto f, \overline{x \mapsto v} \mid s\} \ \triangleright\ E', A''}$$

values and method names: $\Delta \vdash x : \mathbb{b}$ and $\Delta \vdash m : (\overline{f}, \Delta') \rightarrow (E, A)$

(TR-Val-Int)
$$\frac{v \quad \textit{integer-value} \text{ or } \mathtt{null}}{\Delta, E, A \vdash v : \square \ \triangleright\ E}$$

(TR-Var)
$$\frac{\Delta(x) = f}{\Delta, E, A \vdash x : f \ \triangleright\ E}$$

(TR-Field)
$$\frac{\begin{array}{c}\Delta(\mathit{this}) = \alpha[x : f, \cdots]^{\checkmark} \\ E' = E[\alpha.x \mapsto^{\sqcup} \mathtt{r}]\end{array}}{\Delta, E \vdash x : f \ \triangleright\ E'}$$

(TR-Var)
$$\frac{\Delta(f) = E}{\Delta \vdash f : E}$$

(TR-Method-Sign)
$$\frac{\Delta(\mathtt{m}) = (\overline{f}) \rightarrow (E, A) \qquad \sigma \text{ renaming}}{\Delta \vdash m : (\sigma(\overline{f}), \Delta'') \rightarrow (E \circ \sigma, A \circ \sigma)}$$

synchronizations: $\Delta, E, A \ {}^{\oplus}{\vdash}_S \ e \ \triangleright\ \Delta', E', A'$

(TR-Sync)
$$\frac{\begin{array}{c}\Delta, E, A \vdash x : f \ \triangleright\ E' \\ \Delta \vdash f : E'' \qquad \Delta' = \Delta \setminus \{f\}\end{array}}{\Delta, E \ {}^{\oplus}{\vdash}_S \ x \ \triangleright\ \Delta', E' \sqcup E''|_S}$$

(TR-Synchronized)
$$\frac{\Delta, E \vdash e : f \ \triangleright\ E' \qquad f \notin \mathrm{dom}(\Delta)}{\Delta, E \ {}^{\oplus}{\vdash}_S \ v \ \triangleright\ \Delta, E'}$$

expressions with side effects: $\mathit{\Delta}, E, A \vdash_S z\!:\!f \;\triangleright\; \mathit{\Delta}', E', A'$

(TR-Atom)

$$\frac{\mathit{\Delta}, E, A \vdash v\!:\!f \;\triangleright\; E'}{\mathit{\Delta}, E, A \vdash_S v\!:\!f \;\triangleright\; \mathit{\Delta}, E', A}$$

(TR-Expression)

$$\frac{\mathit{\Delta}, E \;{}^{\oplus}\!\!\vdash\; v \;\triangleright\; \mathit{\Delta}', E' \qquad \mathit{\Delta}', E' \;{}^{\oplus}\!\!\vdash\; v' \;\triangleright\; \mathit{\Delta}'', E''}{\mathit{\Delta}, E, A \vdash_S v \oplus v'\!:\!\square \;\triangleright\; \mathit{\Delta}'', E'', A}$$

(TR-New)

$$\frac{\mathit{\Delta}, E, A \vdash \overline{v}\!:\!\overline{g} \;\triangleright\; E' \qquad f \text{ fresh}}{\mathit{\Delta}, E, A \vdash_S \texttt{new Act}(\overline{v})\!:\!f \;\triangleright\; \mathit{\Delta}, E', A}$$

(TR-Invk)

$$\frac{\begin{array}{c}\mathit{\Delta}, E, A \vdash v\!:\!f \;\triangleright\; E \\ \mathit{\Delta}, E, A \vdash \overline{v}\!:\!\overline{f'} \;\triangleright\; E' \qquad \mathit{\Delta} \vdash \texttt{m}\!:\!(f, \overline{f'}) \to (E_{\texttt{m}}, A_{\texttt{m}}) \\ g \text{ fresh} \qquad \mathit{\Delta}' = \mathit{\Delta}[g \mapsto E_{\texttt{m}}] \\ \big(\mathit{Effects}(\mathit{\Delta}')(\beta) \;\#\; y^{(E_{\texttt{m}} \sqcup A)(\beta.y)}\big)^{\beta \in \mathrm{dom}(E_{\texttt{m}} \uplus A) \,\wedge\, y \in \mathit{fields}(\texttt{Act})}\end{array}}{\mathit{\Delta}, E, A \vdash_S v.\texttt{m}(\overline{v})\!:\!g \;\triangleright\; \mathit{\Delta}', E', A \sqcup A_{\texttt{m}}|_S}$$

statements $\mathit{\Gamma}, E, A \vdash_S s \;\triangleright\; \mathit{\Gamma}', E', A$

(TR-Assign-Var-Exp)

$$\frac{\begin{array}{c} x \notin \mathit{fields}(\texttt{Act}) \\ \mathit{\Delta}, E, A \vdash z\!:\!f \;\triangleright\; \mathit{\Delta}', E', A'\end{array}}{\mathit{\Delta}, E, A \vdash_S x = z \;\triangleright\; \mathit{\Delta}'[x \mapsto f], E', A'}$$

(TR-Assign-Field-Exp)

$$\frac{\begin{array}{c} x \in \mathit{fields}(\texttt{Act}) \qquad \mathit{\Delta} \vdash \mathit{this}\!:\!\alpha[\cdots]^{\checkmark} \\ \mathit{\Delta}, E, A \vdash z\!:\!f \;\triangleright\; \mathit{\Delta}', E', A' \\ \mathit{Effects}(\mathit{\Delta}')(\alpha) \;\#\; x^{\texttt{W}} \qquad A'(\alpha) \;\#\; x^{\texttt{W}}\end{array}}{\mathit{\Delta}, E, A \vdash_S x = z \;\triangleright\; \mathit{\Delta}'[\mathit{this}.x \mapsto f], E'[\alpha.x \mapsto^{\sqcup} \texttt{W}], A'}$$

(TR-Skip)

$$\mathit{\Delta}, E, A \vdash_S \texttt{skip}\!:\!0 \;\triangleright\; \mathit{\Delta}, E, A$$

(TR-Seq)

$$\frac{\begin{array}{c}\mathit{\Delta}, E, A \vdash s_1 \;\triangleright\; \mathit{\Delta}_1, E_1, A_1 \\ \mathit{\Delta}_1, E_1, A_1 \vdash s_2 \;\triangleright\; \mathit{\Delta}_2, E_2, A_2\end{array}}{\mathit{\Delta}, E, A \vdash_S s_1; s_2 \;\triangleright\; \mathit{\Delta}_2, E_2, A_2}$$

(TR-Return)

$$\frac{\mathit{\Delta}, E, A \vdash v\!:\!f \;\triangleright\; E'}{\mathit{\Delta}, E, A \vdash_S \texttt{return}\ v \;\triangleright\; \mathit{\Delta}, E', A}$$

(TR-If)

$$\frac{\begin{array}{c}\mathit{\Delta}, E, A \vdash e\!:\!f \;\triangleright\; \mathit{\Delta}', E', A' \qquad \mathit{\Delta}', E', A' \vdash s_1 \;\triangleright\; \mathit{\Delta}_1, E_1, A_1 \qquad \mathit{\Delta}', E', A' \vdash s_2 \;\triangleright\; \mathit{\Delta}_2, E_2, A_2 \\ \mathit{\Delta}_1 =_{\texttt{unsync}} \mathit{\Delta}_2\end{array}}{\mathit{\Delta}, E, A \vdash_S \texttt{if}\ e\ \{\, s_1 \,\}\ \texttt{else}\ \{\, s_2 \,\} \;\triangleright\; \mathit{\Delta}_1 + \mathit{\Delta}_2, E_1 \sqcup E_2, A_1 \sqcup A_2}$$

C.1 Proof of Theorem 6 and Lemma 2

Case: Serve

Serve

$\alpha(a, \varnothing, \overline{q} \cup \{p\}) \to \alpha(a, p, \overline{q})$

Let $\mathit{\Delta}$ and E exists, such that $\mathit{\Delta} \vdash \alpha(a, \varnothing, \overline{q} \cup \{p\}) \triangleright E$, then there exist $\mathit{\Delta}'$ such that $\mathit{\Delta}' \vdash \alpha(a, p, \overline{q}) \triangleright E''$ and $\imath(E'') \subseteq E$ where $\imath$ is an injective renaming of future and actor names.

Proof. By rules TR-Actor and TR-Process we obtain that:

- $\forall q \in \overline{q}.\, \mathit{\Delta}, E_q, \varnothing \vdash q \;\triangleright\; E'_q, A_q$
- $\mathit{\Delta}, E_p, \varnothing \vdash p \;\triangleright\; E'_p, A_p$
- $\mathit{\Delta} \vdash \alpha(a, \varnothing, \overline{q} \cup \{p\}) \;\triangleright\; \bigsqcup_{q \in \overline{q}} (E'_q \sqcup A_q) \sqcup E'_p \sqcup A_p.$

With the same $\mathit{\Delta}$ we can type $\alpha(a, p, \overline{q})$ obtaining that $\mathit{\Delta} \vdash \alpha(a, p, \overline{q}) \;\triangleright\; \bigsqcup_{q \in \overline{q}} (E'_q \sqcup A_q) \sqcup E'_p \sqcup A_p.$

It is trivial to verify that $\bigsqcup_{q \in \overline{q}} (E'_q \sqcup A_q) \sqcup E'_p \sqcup A_p \subseteq \bigsqcup_{q \in \overline{q}} (E'_q \sqcup A_q) \sqcup E'_p \sqcup A_p$

□

Case: Return

Return

$$\frac{\llbracket v \rrbracket_{a+\ell} = w \qquad \ell(\texttt{destiny}) = f}{\begin{array}{l}\alpha(a, \{\ell \mid \texttt{return}\ v\}, \overline{q})\ f(\bot) \\ \qquad \to \alpha(a, \varnothing, \overline{q})\ f(w)\end{array}}$$

Let Δ and E exists, such that $\Delta \vdash \alpha(a, \{\ell \mid \texttt{return } v\}, \overline{q})\ f(\bot) \ \triangleright\ E$, then there exist Δ' such that $\Delta' \vdash \alpha(a, \varnothing, \overline{q})\ f(w) \ \triangleright\ E'$ and $E' \subseteq E$.

Proof. By rules TR-Actor, TR-Process and TR-Return there exist Δ_1 that extend Δ like in the application of TR-Process such that $\Delta_1, [\mathit{this} \mapsto \varnothing], \varnothing \vdash \texttt{return } v \ \triangleright\ \Delta_1, E_1, \varnothing$, where $E_1 = [\mathit{this} \mapsto \varnothing] \sqcup [\alpha.v \mapsto^{\sqcup} \texttt{r}]$ if $v \in \mathit{fields}(\texttt{Act})$ or $E_1 = [\mathit{this} \mapsto \varnothing]$ if $v \notin \mathit{fields}(\texttt{Act})$.
Let us chose $\Delta' = \Delta_1[f \mapsto \Delta_1(f)^{\checkmark}][w \mapsto \Delta_1(f)]$ by applying the rule TR-Parallel, TR-Actor, TR-Process and TR-Future-Eval we can type the target configuration and we gain that $\Delta' \vdash \alpha(a, \varnothing, \overline{q})\ f(w) \ \triangleright\ E'$.

We can conclude that $E_s = \imath(E')$ and then $\imath(E') \subseteq E_2$ where $\imath$ is an injective function on future names and actor names.

□

Proof. By rules TR-Actor and TR-Process we have that:
- $\Delta \vdash \alpha(a, \{\ell \mid \texttt{return } v;\}, \overline{q})\ f(\bot) \ \triangleright\ \{\alpha \mapsto []\}$
- $\Delta \vdash p \ \triangleright\ E_p, A_p$
- $\Delta \vdash \alpha(a, \varnothing, \overline{q} \cup \{p\}) \ \triangleright\ E_q \sqcup A_q \sqcup E_p \sqcup A_p$.

With the same Δ we can type $\alpha(a, p, \overline{q})$ having that $\Delta' \vdash \alpha(a, p, \overline{q}) \ \triangleright\ E_p \sqcup A_p \sqcup E_q \sqcup A_q$. □

Case: Update

Update
$$\frac{\begin{array}{c}(a+\ell)(x) = f \\ (a+\ell)[x \mapsto w] = a' + \ell'\end{array}}{\begin{array}{c}\alpha(a, \{\ell \mid s\}, \overline{q})\ f(w) \\ \to \alpha(a', \{\ell' \mid s\}, \overline{q})\ f(w)\end{array}}$$

Let Δ and E exist, such that $\Delta \vdash \alpha(a, \{\ell \mid s\}, \overline{q})\ f(w) \ \triangleright\ E$, then there exist $\Delta' = \Delta[x \mapsto \Delta(w)]$ that we can use to type the target configuration $\Delta' \vdash \alpha(a', \{\ell' \mid s\}, \overline{q})\ f(w) \ \triangleright\ E'$, and in particular we have $\Delta' \vdash (a' + \ell')(x) : \Delta'(x)$.

Proof. It is trivial to notice that $E' = E$. □

Case: Assign

Assign
$$\frac{[\![e]\!]_{a+\ell} = w \qquad (a+\ell)[x \mapsto w] = a' + \ell'}{\alpha(a, \{\ell \mid x = e\ ;\ s\}, \overline{q}) \to \alpha(a', \{\ell' \mid s\}, \overline{q})}$$

Let Δ and E exist where $\Delta \vdash \alpha(a, \{\ell \mid x = e; s\}, \overline{q}) \ \triangleright\ E$, then there exist Δ' such that $\Delta' \vdash \alpha(a', \{\ell' \mid s\}, \overline{q}) \ \triangleright\ E'$ and $E' \subseteq E$.

Proof. To assert $\Delta' \vdash \alpha(a', \{\ell' \mid s\}, \overline{q}) \ \triangleright\ E'$ we need to apply TR-Actor, TR-Process to the source configuration and find Δ_4, E_2, A such that $\Delta_1, [\text{this} \mapsto \varnothing], \varnothing \vdash_S x = e\ ;\ s \ \triangleright\ \Delta_4, E_2, A$, with Δ_1 extends Δ and S is a set containing the future name of the parameters like in the application of TR-Process.

We can distinguish two cases:

1) x is a local variable ($x \notin \mathit{fields}(\texttt{Act})$)

   By applying the rule TR-Assign-Var-Exp we have that $\Delta_1, [\text{this} \mapsto \varnothing], \varnothing \vdash_S x = e \ \triangleright\ \Delta_3, E_1, \varnothing$ such that:
   - $\Delta_3 = \Delta_2[x \mapsto f]$, where f is the type of the evaluation of the expression e and Δ_2 is the update of Δ_1, that are obtained by applying TR-Expression to type e ($\Delta_1, [\text{this} \mapsto \varnothing], \varnothing \vdash_S e : f \ \triangleright\ \Delta_2, E_1, \varnothing$);
   - E_1 is the update of $[\text{this} \mapsto \varnothing]$ which contain all the effects obtained by typing e.

Finally by TR-Seq we obtain that $\Delta_1, [\text{this} \mapsto \varnothing], \varnothing \vdash_S x = e\ ;\ s \ \triangleright\ \Delta_4, E_1 \sqcup E_s, A$ such that Δ_4 is the update of Δ_3 and E_s is the set of effects that are added to E_1 obtained by typing s ($\Delta_3, E_1, \varnothing \vdash_S s \triangleright \Delta_4, E_1 \sqcup E_s, A$). Finally we can state that $\Delta \vdash \alpha(a, \{\ell \mid x = e; s\}, \overline{q}) \triangleright E_1 \sqcup E_s \sqcup A \sqcup E_{\overline{q}}$, where $E_{\overline{q}}$ is the set containing the effect of the queue of process to be executed.

Let us chose $\Delta' = \Delta_3$ we can type the target configuration, in particular we have $\Delta' \vdash \ell'(x) : \Delta'(x)$. We can type the target configuration applying the rules TR-Actor and TR-Process and we gain that $\Delta' \vdash \alpha(a, \{\ell' \mid s\}, \overline{q}) \ \triangleright\ E'$ where E' is the set of effects obtained by typing s ($\Delta', \varnothing, \varnothing \vdash_S s \ \triangleright\ \Delta_4, E_s, A$) and by typing $\overline{q}$. We can conclude that $E_s \sqcup A \sqcup E_{\overline{q}} = \imath(E')$ and then $\imath(E') \subseteq E_1 \sqcup E_s \sqcup A \sqcup E_{\overline{q}}$, where $\imath$ is an injective function on future names and actor names.

2) x is a field ($x \in \mathit{fields}(\mathtt{Act})$)

This case is similar to the previous one, but instead of apply the rule TR-Assign-Var-Exp we apply the rule TR-Assign-Field-Exp that we give us $\Delta_3 = \Delta_2[\mathit{this}.x \mapsto f]$ and $E_1[\alpha.x \mapsto^{\sqcup} \mathtt{w}]$.

Let chose $\Delta' = \Delta_4$ we can type the target configuration, in particular we have $\Delta' \vdash a'(x) : \Delta'(x)$. Typing the target configuration by rules TR-Actor and TR-Process we have that $\Delta' \vdash \alpha(a', \{\ell \mid s\}, \overline{q}) \triangleright E'$ where E' is the set of effects obtained by typing s. We can conclude that $E_s \sqcup A \sqcup E_{\overline{q}} = \imath(E')$ and then $\imath(E') \subseteq E_1 \sqcup E_s \sqcup A \sqcup E_{\overline{q}}$, where $\imath$ is an injective function on future names and actor names.

□

Case: New

$$\dfrac{\llbracket \overline{v} \rrbracket_{a+\ell} = \overline{w} \qquad \beta \ \text{fresh} \qquad \overline{y} = \mathit{fields}(\mathtt{Act})}{\begin{array}{c}\alpha(a, \{\ell \mid x = \mathtt{new\ Act}(\overline{v})\ ;\ s\}, \overline{q}) \\ \to \alpha(a, \{\ell \mid x = \beta\ ;\ s\}, \overline{q}) \ \ \beta([\overline{y} \mapsto \overline{w}], \varnothing, \varnothing)\end{array}}\ \text{New}$$

Let Δ and E exist, such that $\Delta \vdash \alpha(a, \{\ell \mid x = \mathtt{new\ Act}(\overline{v})\ ;\ s\}, \overline{q}) \ \triangleright\ E$, then there exist Δ' and E' such that $\Delta' \vdash \alpha(a, \{\ell \mid x = \beta; s\}, \overline{q}) \ \beta([\overline{y} \mapsto \overline{w}], \varnothing, \varnothing) \ \triangleright\ E'$ and $E' \subseteq E$.

x is not a field because of the restriction of the language.

Proof. By rules TR-Actor, TR-Process and TR-New there exist Δ_1 and S, that are, like in the application of TR-Process, the extension of Δ and the set containing the future name of the parameters respectively, such that $\Delta_1, [\mathit{this} \mapsto \varnothing], \varnothing \vdash_S \mathtt{new\ Act}(\overline{v}) : f \ \triangleright\ \Delta_2, E_1, \varnothing$, where $\Delta_2 = \Delta_1[f \mapsto \beta[\overline{a{:}g}]^{\checkmark}]$ and $E_1 = [\mathit{this} \mapsto \varnothing] \sqcup [\alpha.v \mapsto^{\sqcup} \mathtt{r}]^{v \in \overline{v}}$.
Finally by applying the rules TR-Assign-Var-Exp and TR-Seq we obtain that $\Delta_2, E_1, \varnothing \vdash_S x = \mathtt{new\ Act}(\overline{v})\ ;\ s \ \triangleright\ \Delta_3, E_2, A$ where Δ_3 and E_2 are the updates of Δ_2 and E_1 that we gain typing s. We want to underline that by construction $E_2 = E_1 \sqcup E_s$ where we call E_s the set of effects that are added to E_1 obtained by typing s ($\Delta_2[x \mapsto f], E_1, \varnothing \vdash_S s \ \triangleright\ \Delta_3, E_1 \sqcup E_s, A$).
Let us chose $\Delta' = \Delta_3[\beta \mapsto \Delta_3(f)]$ we can type the target configuration, in particular we have $\Delta' \vdash \overline{w} : \Delta'(f.y)$. We can type the target configuration applying the rules TR-Parallel, TR-Actor and TR-Process and we gain that $\Delta' \vdash \alpha(a, \{\ell' \mid s\}, \overline{q}) \ \triangleright\ E'$ where E' is the set of effects obtained by typing s ($\Delta', \varnothing, \varnothing \vdash_S s \ \triangleright\ \Delta_4, E_s, A$) and $\Delta' \vdash \beta([\overline{y} \mapsto \overline{w}], \varnothing, \varnothing) \ \triangleright\ \varnothing$.
We can conclude that $E_s = \imath(E')$ and then $\imath(E') \subseteq E_2$ where $\imath$ is an injective function on future names and actor names.

□

Case: Invk

$$\dfrac{\begin{array}{c}\llbracket v \rrbracket_{a+\ell} = \beta \qquad \llbracket \overline{v} \rrbracket_{a+\ell} = \overline{w} \qquad \beta \neq \alpha \\ f \ \text{fresh} \qquad \mathrm{bind}(\beta, m, \overline{w}, f) = p'\end{array}}{\begin{array}{c}\alpha(a, \{\ell \mid x = v.\mathtt{m}(\overline{v})\ ;\ s\}, \overline{q}) \ \beta(a', p, \overline{q'}) \\ \to \alpha(a, \{\ell \mid x = f\ ;\ s\}, \overline{q}) \ \beta(a', p', \overline{q'} \cup \{p'\}) \ f(\bot)\end{array}}\ \text{Invk}$$

Let Δ and E exist where $\Delta \vdash \alpha(a, \{\ell \mid x = v.\mathtt{m}(\overline{v})\ ;\ s\}, \overline{q}) \ \beta(a', p, \overline{q'}) \ \triangleright\ E$, then there exist Δ' and E' such that $\Delta' \vdash \alpha(a, \{\ell \mid x = f\ ;\ s\}, \overline{q}) \ \beta(a', p', \overline{q'} \cup \{p'\}) \ f(\bot) \ \triangleright\ E'$ and $E' \subseteq E$.

Proof. By rules TR-ACTOR, TR-PROCESS we have that:

- $\Delta \vdash \{\ell \mid x = v.\mathtt{m}(\overline{v}) \ ; \ s\} \ \triangleright \ E_p, A_p$
- $\Delta \vdash \overline{q} \ \triangleright \ E_{\overline{q}}, A_{\overline{q}}$
- $\Delta \vdash \alpha(a, \{\ell \mid x = v.\mathtt{m}(\overline{v}) \ ; \ s\}, \overline{q}) \ \triangleright \ E_p \sqcup A_p \sqcup E_{\overline{q}} \sqcup A_{\overline{q}}$
- $\Delta \vdash p \ \triangleright \ E_p, A_p$
- $\Delta \vdash \overline{q'} \ \triangleright \ E_{q'}, A_{q'}$
- $\Delta \vdash \beta(a', p', \overline{q'}) \ \triangleright \ E_{p'} \sqcup A_{p'} \sqcup E_{\overline{q'}} \sqcup A_{\overline{q'}}$.

By rule TR-SEQ and TR-INVK we have that there exist Δ_2 that extend Δ, E_1, A_1 and S with $S \subseteq \Delta_2$, such that:

- $\Delta_2, E_1 \vdash v\!:\!g \ \triangleright \ E_2$
- $\Delta_2, E_2 \vdash \overline{v}\!:\!\overline{g} \ \triangleright \ E_3$
- $\Delta \vdash \mathtt{m}\!:\!(g, \overline{g'}) \to (E_{\mathtt{m}}, A_{\mathtt{m}})$
- $\Delta_2, E_1, A_1 \vdash x = v.\mathtt{m}(\overline{v}) \ \triangleright \ \Delta_4, E_2, A_1 \sqcup A_{\mathtt{m}}|_S$

Now we can distinguish two cases:

- Case 1: s contains a synchronization of the method invocation $v.\mathtt{m}(\overline{v})$, then by rule TR-SYNCH we have that $\Delta_2, E_1, A_1 \vdash x = v.\mathtt{m}(\overline{v}) \ ; \ s \ \triangleright \ \Delta_3, E_1 \sqcup E_{\mathtt{m}}|_S \sqcup E_s, A_1 \sqcup A_{\mathtt{m}}|_S \sqcup A_s$, where E_s and A_s are the effects of the other statement in s. At the end we have that $\Delta \vdash \alpha(a, \{\ell \mid x = v.\mathtt{m}(\overline{v}) \ ; \ s\}, \overline{q}) \ \triangleright \ E_1 \sqcup E_{\mathtt{m}}|_S \sqcup E'_s \sqcup A_1 \sqcup A_{\mathtt{m}}|_S \sqcup A_s \sqcup E_{\overline{q}} \sqcup A_{\overline{q}}$, where E'_s contain the same effects of E_s plus all the effects of the method invoked in s and not synchronized.
- Case 2: s does not contain a synchronization of the method invocation $v.\mathtt{m}(\overline{v})$, then we have that $\Delta_2, E_1, A_1 \vdash x = v.\mathtt{m}(\overline{v}) \ ; \ s \ \triangleright \ \Delta_3, E_1 \sqcup E_s, A_1 \sqcup A_{\mathtt{m}}|_S \sqcup A_s$ (E_s and A_s are the same of the case 1). At the end we have that $\Delta \vdash \alpha(a, \{\ell \mid x = v.\mathtt{m}(\overline{v}) \ ; \ s\}, \overline{q}) \ \triangleright \ E_1 \sqcup E''_s \sqcup A_1 \sqcup A_{\mathtt{m}}|_S \sqcup A_s \sqcup E_{\overline{q}} \sqcup A_{\overline{q}}$, where this time $E''_s = E_{\mathtt{m}}|S \sqcup E'_s$ because the method $\mathtt{m}$ is one of the not synchronized method at the end of the execution.

It si easy to notice that the effects of $\alpha(a, \{\ell \mid x = v.\mathtt{m}(\overline{v}) \ ; \ s\}, \overline{q})$ in both cases are equivalent.

Let $\Delta' = \Delta[f \mapsto E_{\mathtt{m}}]$ we have that by rules TR-ACTOR and TR-PROCESS $\Delta' \vdash \alpha(a, \{\ell \mid x = f \ ; \ s\}, \overline{q}) \ \beta(a', p', \overline{q'} \cup \{p'\}) \ f(\bot) \ \triangleright \ E$