Analysis of Synchronisations in Stateful Active Objects

Ludovic Henrio, Cosimo Laneve, Vincenzo Mastandrea

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 effects and behavioural types for deriving synchronisation patterns of stateful active objects and verifying the absence of deadlocks in this context. 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 [12,4,16].

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 [15] and have been adopted in several languages and libraries, such as `Akka` [18], an actor library for `Java` and `Scala` [10], or in `ABS` [12], 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

* This work was partially funded by the ANR project ANR-11-LABX-0031-01

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 [8,6] 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.

```
1  Int n
2  addToStore(Int x){
3    count = n + 1;
4    n = this.store(x,count);
5    return count }
6  store(Int x, Int y){
7    /* storing x */ return y }
```

```
8  //MAIN
9  { Store = new Act(0);
10   x = Store.addToStore(1);
11   x = x + 1; // needed to
                  avoid conflicts
12   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 11) 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 11 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* [7,13], which are conjunctions and disjunctions of object dependencies and method invocations. Like in [6], 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 [6] 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 [6], which detects circularities in the graph of dependencies, highlighting potential *deadlocks* caused by erroneous synchronisation patterns. 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. Due to space limitation, this paper only contains the crucial points of the formalisation; technical details and proofs can be found in [11].

2 The active object model `gASP`

Syntax. The language `gASP` has types T that may be either `Int` or a class `Act`. Extending this work to several classes is not problematic. We use x, y, k, $\cdots$ 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:

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

Expressions with side effects include asynchronous method calls $v.\mathtt{m}(\overline{v})$, where v is the invoked object and $\overline{v}$ are the arguments of the invocation. Operations taking place on different active objects occur in parallel, while operations in the same active object are sequential. Terms z also include $\mathtt{new\ Act}(\overline{v})$ 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.

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

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

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

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

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

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

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

Fig. 1: Evaluation function and semantics of gASP (excerpt) - full version in [11].

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, g, ...

The runtime syntax of gASP is:

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

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: $dom(\ell)$ returns 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 $T\,\mathtt{m}(\overline{T\ x})\ \{\, s \,\}$ then: $\mathrm{bind}(\alpha, \mathtt{m}, \overline{w}, f) = p$ where p is a process in the following shape $\{\, [\, \texttt{destiny} \mapsto f, \texttt{this} \mapsto \alpha, \overline{x} \mapsto \overline{w}\,] \mid s \,\}$, where 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 1 shows the essential rules of the semantics, all the rules can be found in [6,11]. Rule Update replaces the future reference by its value, it can be triggered at any time when a future value is known. The new value may be also a future. 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 synchronisations: if e is an arithmetic expression, the

operands must be evaluated to integers, and, if an operand is a future, the rule can only be applied *after this future has been evaluated and updated*. 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 ensures the strong encapsulation of objects: an active object can only assign its own fields. The initial configuration of a `gASP` program with main body $\{s\}$ is: $main(\,[\,\overline{x} \mapsto \overline{0}\,], \{\,[\,\texttt{destiny} \mapsto f_{main}, \texttt{this} \mapsto main\,]\,|\,s\,\}, \varnothing)$ 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 will 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);

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

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 means 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 [8].

Deadlocks. 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

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

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 $\ell(\texttt{destiny}) = f$.

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* $0 \leq i < n$,

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 all reachable configurations are deadlock free.

Queue with non deterministic effects. 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 method invocations only create request *queues with deterministic effects*.

Definition 2. *An active object $\alpha(a, p, \overline{q})$ has a queue with deterministic effects if when a process in $\overline{q}$ write on a field all the other process in the queue do not perform neither read nor writes on the same field.*

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 any reachable configuration has deterministic effects.*

Example. The execution of the program shown in the introduction reaches the following configuration after the first execution of the `addToStore` method (future f), at the point where either the method `store` (future g) or `addToStore` (future h) can be served: $main\big([\, n \mapsto 0\,], \varnothing, \varnothing\big)\ \ f(1)\quad g(\bot)\quad h(\bot)\ \ \alpha\big([\, n \mapsto g\,], \varnothing, \{\textit{body-of-}\texttt{store}\}, \{\textit{body-of-}\texttt{addToStore}\}\big)$.

From this point, if α serves the invocation of `addToStore` we reach a deadlock because the execution of `addToStore` needs to know the value of the field `n` (to execute Line 4) but the method `store` can only be served after the termination of the current method. 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 Behavioural Type System

In this section we define a type system that associates abstract descriptions, called *behavioural types* to `gASP`programs. This association is done by recording several information: (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 synchronisation 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 not possible to infer at static time which variables contain a future. Consequently, we consider all stored values as futures and some of corresponding values will be already synchronised when created. It is therefore important to distinguish *future names* that are identifiers and *future types* 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 stateful, deadlocks might be caused by accesses 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 noticing 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. Types are either basic types, future types or behavioural types. They are defined as follows:

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

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 placeholder for the object that will synchronise with the invocation, the environment Γ and the effects E record the state changes performed by the method, 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 "check-marked" future types, noted $\mathbb{f}^{\checkmark}$, to represent a future value that has been already synchronised. We use $\mathbb{f}^{[\checkmark]}$ to range over both future types and "check-marked" future types.

Behavioural types include $\mathtt{0}$, 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. Behavioural types also include *synchronisation commitments* f_κ, whose meaning 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 synchronise 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 $\mathtt{0}$ being the identity. The operator "&" has precedence over "+".

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

The image of an environment Γ is noted $\mathrm{im}(\Gamma)$; the restriction of Γ to a set S of names is noted $\Gamma|_S$; the difference operation the difference operation $\Gamma \setminus x$ defined as $\Gamma|_{\mathrm{dom}(\Gamma)\setminus x}$. The following functions on Γ are also used:

- $\mathit{names}(\Gamma) = \mathrm{dom}(\Gamma) \cup \{\alpha \mid \alpha[\overline{x:f}]^{[\checkmark]} \in \mathrm{im}(\Gamma)\}$;
- $\mathit{obj}(\overline{f})$ (resp. $\mathit{int}(\overline{f'})$) is a subset of $\overline{f}$ such that for each $f' \in \mathit{obj}(\overline{f})$ (resp. $f' \in \mathit{int}(\overline{f})$) we have $\Gamma(f') = \alpha[\cdots]$ (resp. $\Gamma(f') \neq \alpha[\cdots]$) for some α;
- $\mathit{Fut}(\Gamma)$ 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 respectively not-check-marked or check-marked;
- $\mathit{unsync}(\Gamma) = \&_{f\in \mathit{aFut}(\Gamma)}\, f_\star$ is the parallel behaviour of the method invocations which are not-yet-synchronised;
- $\Gamma[f^{\checkmark}]$ returns the environment $\Gamma[f \mapsto \mathbb{f}^{\checkmark}]$ when $\Gamma(f)$ is either $\mathbb{f}$ or $\mathbb{f}^{\checkmark}$;

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

$$\mathit{Effects}(\Gamma) = \bigsqcup\{E \mid \Gamma(f) = \lambda X.\mathtt{m}(\overline{g}, X, \Gamma_{\mathtt{m}}, E)\} \quad (2^*)$$

$$x^{\mathtt{h}} \# y^{\mathtt{h}'} = \begin{cases} \mathit{true} & \text{if } x \neq y \text{ or } (x = y \text{ and } \mathtt{h}' = \mathtt{r} = \mathtt{h}) \\ \mathit{false} & \text{otherwise} \end{cases} \quad (3)$$

$$\mathit{instanceof}(E, \sigma)(f) = \begin{cases} \bigsqcup\limits_{g \in \sigma^{-1}(f)} E(g) & \text{if } \forall f_1, f_2 \in \sigma^{-1}(f).f_1 \neq f_2 \Rightarrow E(f_1) \# E(f_2) \\ \mathit{undefined} & \text{otherwise} \end{cases} \quad (4^{**})$$

Fig. 2: Auxiliary functions for effects - full version in [11].

- $\Gamma(f.x) = \begin{cases} g & \text{if } \Gamma(f) = \alpha[\cdots, x{:}g, \cdots] \\ \mathit{undefined} & \text{otherwise} \end{cases}$
- $\Gamma[f.x \mapsto g]$ 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 $\mathit{aFut}(\Gamma_1) \cup \mathit{aFut}(\Gamma_2)$.

Effects. Effects are functions, noted $E, A, \cdots$, that map future names to a set of field names labelled either with `r` (read) or with `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 `m` writes on the field x of the object that is the value of f and reads on the field y. Let `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). In the example in the introduction, the method `addToStore` has the effect $[g \mapsto [n^{\mathtt{w}}]$ where g represents the current object (`this`). 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 2: *update operation with upper bound*(1); *effects of unsynchronised methods*(2); *compatibility*(3); effect instantiation taking into account effect compatibility(4). We extend the definition of the operation $\sqcup$ and $\#$ from effects to sets of effects iterating them for all the element of the sets pairwise.

Judgements. The judgements used in the type system are:

- $\vdash \mathtt{m}{:}(f, \overline{g}, \Gamma_{\mathtt{m}}, X) \to (E, A)$ instantiates the method signature of `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: $\Gamma_{\mathtt{m}} = (\Gamma|_{f \cup obj(\overline{g})})$, where Γ is the environment at invocation point. E, A are the environments storing the effects of `m`: E stores the effects that happen before `m` is synchronised, A stores the effects of the methods invoked by `m` and not synchronised 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;

---

* We notice that $\Gamma(f)$ is not check-marked

** The usage of *instanceof* is illustrated in the description of T-Method-Sign

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

$$\text{(T-Field)}\quad \frac{\Gamma(this.x) = f \qquad E' = E[this.x \mapsto^{\sqcup} \mathtt{r}]}{\Gamma, E \vdash x : f \ \triangleright\ E'}$$

$$\text{(T-Method-Sign)}\quad \frac{\Gamma(\mathtt{m}) = (\overline{f}, X, \Gamma') \to (E, A) \qquad \sigma \text{ renaming} \qquad \Gamma'' = \sigma(\Gamma') \qquad E' = instanceof(E, \sigma) \qquad A' = instanceof(A, \sigma)}{\vdash \mathtt{m} : (\sigma(\overline{f}), \sigma(X), \Gamma'') \to (E', A')}$$

synchronizations: $\Gamma, E \overset{\oplus}{\vdash}_S \ x : \mathsf{L} \ \triangleright\ \Gamma', E'$

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

$$\text{(T-Sync-Field)}\quad \frac{\Gamma \vdash this : \alpha[\cdots]^{\checkmark} \qquad \Gamma, E \vdash x : f \ \triangleright\ E' \qquad \Gamma \vdash f : g \rightsquigarrow this.x \qquad \Gamma' = \Gamma[f^{\checkmark}]}{\Gamma, E \overset{\oplus}{\vdash}_S \ x : f_\alpha \,\&\, unsync(\Gamma') \triangleright \Gamma', E'}$$

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

$$\text{(T-Expression)}\quad \frac{\Gamma, E \overset{\oplus}{\vdash}_S \ v : \mathsf{L} \ \triangleright\ \Gamma', E' \qquad \Gamma', E' \overset{\oplus}{\vdash}_S \ v' : \mathsf{L}' \ \triangleright\ \Gamma'', E''}{\Gamma, E, A \vdash_S v \oplus v' : \Box, \mathsf{L} + \mathsf{L}' \ \triangleright\ \Gamma', E'', A}$$

$$\text{(T-Invk)}\quad \frac{\Gamma, E \vdash v : f \ \triangleright\ E \qquad \Gamma \vdash f : \beta[\cdots]^{\checkmark} \qquad \Gamma, E \vdash \overline{v} : \overline{f'} \ \triangleright\ E' \qquad \overline{h} = f \cup obj(\overline{f'}) \qquad \vdash \mathtt{m} : (f, \overline{f'}, X, \Gamma|_{\overline{h}}) \to (E_{\mathtt{m}}, A_{\mathtt{m}}) \qquad g \text{ fresh} \qquad \overline{g'} = \overline{f'}[^{\Box}/_{int(sFut(\Gamma))}] \qquad \Gamma_{\mathtt{m}} = (\Gamma|_{\overline{h}})[^{\Box}/_{int(sFut(\Gamma))}] \qquad \Gamma' = \Gamma[g \mapsto \lambda X.\mathtt{m}(f, \overline{g'}, X, \Gamma_{\mathtt{m}}, E_{\mathtt{m}})] \qquad \big(Effects(\Gamma')(h') \# y^{(E_{\mathtt{m}} \sqcup A)(h'.y)}\big)^{h' \in \mathrm{dom}(E_{\mathtt{m}} \uplus A) \ \wedge\ y \in fields(\mathtt{Act})}}{\Gamma, E, A \vdash_S v.\mathtt{m}(\overline{v}) : g,\ g_\star \,\&\, unsync(\Gamma) \ \triangleright\ \Gamma', E', A \sqcup A_{\mathtt{m}}}$$

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

$$\text{(T-Assign-Field-Exp)}\quad \frac{x \in fields(\mathtt{Act}) \qquad \Gamma, E, A \vdash_S z : f, \mathsf{L} \ \triangleright\ \Gamma', E', A' \qquad Effects(\Gamma')(this) \# x^{\mathtt{w}} \qquad A'(this) \# x^{\mathtt{w}}}{\Gamma, E, A \vdash_S x = z : \mathsf{L} \triangleright \Gamma'[this.x \mapsto f], E'[this.x \mapsto^{\sqcup} \mathtt{w}], A'}$$

$$\text{(T-Return-Fut)}\quad \frac{\Gamma, E \vdash v : f \ \triangleright\ E' \qquad \Gamma \vdash f : \mathrm{f} \qquad \Gamma(\mathrm{future}) = X \qquad \mathsf{L} = unsync(\Gamma \setminus f)}{\Gamma, E, A \vdash_S \mathtt{return}\ v : f_X \,\&\, \mathsf{L} \triangleright \Gamma, E', A}$$

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

$$\text{(T-Method)}\quad \frac{\Gamma(\mathtt{m}) = (this, \overline{f}, X, \Gamma_{\mathtt{m}}) \to (E, A) \qquad \overline{g} = int(\overline{f} \cup names(\Gamma_{\mathtt{m}})) \qquad \Gamma + \Gamma_{\mathtt{m}} + \overline{x} : \overline{f} + \overline{g} : \Box + \mathrm{future} : X, [this \mapsto \varnothing], \varnothing \vdash_{\mathrm{dom}(\Gamma_{\mathtt{m}})} s : \mathsf{L} \ \triangleright\ \Gamma', E, A' \qquad \overline{w} = flat(this, \overline{f}, \Gamma_{\mathtt{m}}) \qquad \overline{\varkappa} = names(\Gamma') \setminus names(\Gamma_{\mathtt{m}}) \qquad A = A' \sqcup \bigsqcup_{h \in \mathrm{dom}(\Gamma')} \left\{ \left(E_{\mathtt{m}'}|_{\{this, \overline{f}\}} \right) \mid \Gamma'(h) = \lambda Y.\mathtt{m}'(\overline{f}, Y, \Gamma_{\mathtt{m}'}, E_{\mathtt{m}'}) \right\}}{\Gamma \vdash \mathtt{m}\ (\overline{T\ x})\{s\} : (\overline{w}, X) \to (\nu\, \overline{\varkappa})(\Gamma'|_{\overline{\varkappa}} \cdot \Gamma'|_{obj(\overline{f})} \cdot \mathsf{L} \,\&\, (X, \alpha))}$$

Fig. 3: Typing rules -full version in [11].

- $\Gamma, E \overset{\oplus}{\vdash}_S \ e : \mathsf{L} \ \triangleright\ \Gamma', E'$ for typing synchronisations, where S is the set of arguments of the current method, L is the behavioural 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. We assume that every environment Γ is such that $\Gamma(\Box) = \Box^{\checkmark}$ and $\Gamma(this) = \alpha[\cdots]$, where α is the active object running the current method. The typing rules are shown in Figure 3 and the most rellevant ones are discussed.

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 synchro-

nised invocations in Γ and those in A (that will not be synchronised). 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 two parameters have the same value thanks to 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.

In `gASP`, synchronisations 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 synchronised (check-marked) 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 f bound to x is synchronised – henceforth its result is check-marked in the environment. Correspondingly, the futures that are synchronised by f, namely those that are recorded in the effect $E_{\mathtt{m}}$, are synchronised 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 synchronising actor name α. This type will allow us to compute the dependencies of the parallel methods during the analysis.

In the example of the introduction, Line 11 triggers a synchronisation with the first execution of `addToStore`. As a consequence of the application of the rules (T-Expression) and (T-Sync-Invk), `n` now points to a not-yet-known future of the form $f \rightsquigarrow g.n$; this future will be mapped during analysis to the first invocation to `store`.

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 invocation does not introduce any dependency. The substitution on second line replaces synchronised futures by $\square$ to prevent additional synchronisations on these futures.

The behavioural type of statements is a sum of types that are parallel composition of synchronisation dependencies and unsynchronised behaviours. 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 placeholder for the active object synchronising the method currently analysed. The rest of the behavioural type collects the unsynchronised behaviour.

In (T-METHOD), the premises verify the consistency of the typing of $\mathtt{m}$ in the environment with the typing of its body. In particular, the asynchronous effects of $\mathtt{m}$ must be the sum of the asynchronous ones in its body, i.e. A', plus the effects of the invocations that have not been synchronised. We notice that the behavioural type of the method has arguments that are structureless: object are removed and replaced by their flattened version, where the fields are removed and the corresponding values are lifted as arguments, this operation is fulfilled by 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 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 environment Γ' records the updates to the arguments $\overline{f}$ performed by the method, and L is the behavioural type of the body of the method. To make the rule TR-METHOD easier to read we let Γ and Γ' contain more information than we require in the behavioural type analysis, this is the reason why will be used a simplified form of this environment. Instead of Γ will be used Θ which does not define a mapping between future names and object types and future results do not present information about effects $(\lambda X.\mathtt{m}(\overline{g}, X, \Gamma''))$, and Γ' will be renamed as Φ.

Finally, a *behavioural type program* is a pair $\big(\mathcal{L},\ \Theta \cdot \mathsf{L}\big)$, where $\mathcal{L}$ maps *method names* $\mathtt{m}$ to *method behaviours* $(\overline{w}, X) \to (\nu\, \overline{\varkappa})(\Theta' \cdot \Phi \cdot \mathsf{L}')$, $\overline{w}, X$ are the *formal parameters* of $\mathtt{m}$, Θ', Φ and L are the same as above. The last two elements , namely Θ and L, are the *environment* and the *type* of the main body.

The fact that $\Gamma \vdash \{\overline{\mathtt{Int}\ x}, \overline{M}\}\{s\}$ implies that any configuration reached evaluating the program has deterministic effects.

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:
$\Theta = [\ f \mapsto \lambda X.\mathtt{addToStore}(g, \square, X, [\ g \mapsto \alpha[n : \square]^\checkmark\ ], [\ g \mapsto [n^{\mathtt{w}}]\ ]\ ),\ g' \mapsto f \rightsquigarrow g.n,$
$f' \mapsto \lambda X.\mathtt{addToStore}(g, \square, X, [\ g \mapsto \alpha[n : g']^\checkmark\ ], [\ g \mapsto [n^{\mathtt{w}}]\ ]\ )\ ]$.

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 $(g \mapsto \alpha[n : \square]^\checkmark)$, 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 $[\ g \mapsto [n^{\mathtt{w}}]\ ]$.

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

$$\text{BT-FIELD}\quad \frac{\begin{array}{c}\Theta(f) = f' \rightsquigarrow g.x \quad \Theta(f') = \lambda X.\mathtt{m}(\overline{f}, X, \Gamma) \\ \mathcal{L}(\mathtt{m}) = (\overline{w}, Y) \to (\nu\,\overline{\varkappa})(\Theta' \cdot \Phi \cdot \mathsf{L}) \\ \Phi' = \Phi[^{\overline{\varkappa'}}/_{\overline{\varkappa}}][^{\mathit{flat}(\overline{f},\Gamma)}/_{\overline{w}}] \quad \Phi'(g.x) = h\end{array}}{\Theta \cdot \mathcal{C}[\,f_\kappa\,] \to \Theta \cdot \mathcal{C}[\,h_\kappa\,]}$$

Fig. 4: Behavioural type reduction rules

As stated above, $\mathcal{L}$ stores the behavioural type for each method of the program, then we have an entry for `addToStore` and `store`.
$\mathcal{L}(\mathtt{addToStore}) = (\beta, \mathit{this}, g, f, X) \to (\nu\, f')(\Theta_{\mathtt{add}} \cdot \Phi_{\mathtt{add}} \cdot \mathsf{L}_{\mathtt{add}})$ where
$\mathsf{L}_{\mathtt{add}} = (\, g_\alpha + f'_\star + f'_X \,) \,\&(X, \beta) \qquad \Phi_{\mathtt{add}} = [\, \mathit{this} \mapsto \beta[n : f'] \,]$
$\Theta_{\mathtt{add}} = [\; f' \mapsto \lambda X.\mathtt{store}(\mathit{this}, f, \Box, X, [\; \mathit{this} \mapsto \beta[n : g]^{\checkmark} \;], \varnothing) \,]$

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

4 Behavioural type soundness and analysis

The type system defined in Section 3 can be extended to configurations, see [11]: the judgment we use is $\Gamma \vdash cn : \mathsf{K}$ where K is a parallel composition of $\Theta \cdot \mathsf{L}$, one for each configuration element. 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 of cn and the type 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 K' with the circularities of the evaluation of K.

The evaluation of a behavioural types is defined by a transition relation between types $\Theta \cdot \mathsf{L}$ that follows the rules in Figure 4 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 Θ is augmented with the names defined in this body. Note that Θ'' is well-defined because $\mathrm{dom}(\Theta) \cap \mathrm{dom}(\Theta'[^{\overline{\varkappa'}}/_{\overline{\varkappa}}][^{\mathit{flat}(\overline{f},\Gamma)}/_{\overline{w}}]) = \varnothing$ and $(\mathit{flat}(\overline{f}, \Gamma) \cup \overline{w}) \cap \overline{\varkappa'} = \varnothing$. The behavioural type L' is defined by a classical substitution. The substitution $[^{\mathit{flat}(\overline{f},\Gamma)}/_{\overline{w}}]$ replaces active object and future names in $\overline{w}$. This substitution can generate terms of the form $\Box_\alpha$, those terms can safely be replaced by $\mathtt{0}$. 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 Φ in the method of f' and infers h, the future written in the accessed field.

Definition 3. *Let $\mathsf{L} \equiv_d \mathsf{L}'$ whenever L and L' are equal up-to commutativity and associativity of "&" and "+", identity of $\mathtt{0}$ 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 $\mathsf{L} \equiv_d \mathcal{C}[\,(\alpha_1, \alpha_2) \,\&\, \cdots \,\&\, (\alpha_n, \alpha_1)\,]$.*

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

Below we write $\Gamma \vdash cn : \Theta \cdot \mathsf{L}$ to say that the configuration cn has type $\Theta \cdot \mathsf{L}$ in the environment Γ. This judgment requires an extension of the type system in Figures 3 to configurations (see [11]). 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 $\Gamma \vdash P : (\mathcal{L}, \Theta \cdot \mathsf{L})$, then:*

1. *$\Gamma \vdash cn : \Theta \cdot \mathsf{L}$ where cn is the initial configuration;*
2. *if $cn \rightarrow^* cn'$ then there are Γ', Θ' and L' such that $\Gamma' \vdash cn' : \Theta' \cdot \mathsf{L}'$ and if $\Theta' \cdot \mathsf{L}'$ has a circularity then also $\Theta \cdot \mathsf{L}$ has a circularity.*
3. *if $\Theta \cdot \mathsf{L}$ 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 [13,7], which has been adapted to the types of this paper in [6].

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 $\Theta(f') = \lambda X.\mathtt{addToStore}(\cdots)$, we take the behavioural type $\mathsf{L}_{\mathtt{add}}$, build the substitution $[{}^{h}/_{f'}][{}^{\star}/_{X}][{}^{\alpha,g,g',\Box}/_{\beta,this,g,f}]$ that instantiates the parameters adequately, and obtain the behaviour: $(g'_\alpha + h_\star + h_X) \,\&\, (\star, \alpha)$, additionally $\Theta' = \Theta + \Theta'_{\mathtt{add}}$ where $\Theta'_{\mathtt{add}}$ is obtained from $\Theta_{\mathtt{add}}$ applying the same substitution. Finally we can apply BT-Fun and obtain the reduction $\Theta \cdot (f_\star + f_{main} + f'_\star) \rightarrow \Theta' \cdot (f_\star + f_{main} + (g'_\alpha + h_\star + h_X) \,\&\, (\star, \alpha))$.

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: $\Theta' \cdot (\cdots + (g'_\alpha + h_\star + h_X) \,\&\, (\star, \alpha)) \rightarrow$

$\Theta' \cdot (\cdots + (h'_\alpha + h_\star + h_X) \,\&(\star, \alpha))$. 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: $\Theta(f) = \lambda X.\texttt{addToStore}(g, \Box, X, [\, g \mapsto \alpha[n : \Box]^{\checkmark}\,]])$ with the substitution $[^{h}/_{f'}][^{\alpha,g,\Box,\Box}/_{\beta,this,g,f}]$.

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 [14]. 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 [5], 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 [5], 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, [5] 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, [17] 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 [2,9] 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 [8] (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. Antonio Flores-Montoya, Elvira Albert, and Samir Genaim. May-happen-in-parallel based deadlock analysis for concurrent objects. In *Proc. FORTE/F-MOODS 2013*. Springer, 2013.
6. Elena Giachino, Ludovic Henrio, Cosimo Laneve, and Vincenzo Mastandrea. Actors may synchronize, safely! In *Proceedings of PPDP 2016*. ACM, 2016.
7. Elena Giachino, Naoki Kobayashi, and Cosimo Laneve. Deadlock analysis of unbounded process networks. In *Proceedings of CONCUR 2014*. Springer, 2014.
8. Elena Giachino, Cosimo Laneve, and Michael Lienhardt. A framework for deadlock detection in core ABS. *Software and Systems Modeling*, 15(4):1013–1048, 2016.
9. 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.
10. P. Haller and M. Odersky. Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2-3):202–220, 2009.
11. Ludovic Henrio, Cosimo Laneve, and Vincenzo Mastandrea. Analysis of synchronisation patterns in stateful active objects. Research report, I3S ; Inria - Sophia antipolis, 2017. URL: `https://hal.archives-ouvertes.fr/hal-01542595`.
12. 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.
13. Naoki Kobayashi and Cosimo Laneve. Deadlock analysis of unbounded process networks. *Inf. Comput.*, 252:48–70, 2017.
14. John M. Lucassen and David K. Gifford. Polymorphic effect systems. In *Proc. of POPL 1988*, pages 47–57. ACM Press, 1988.
15. Ben Morris. *The Symbian OS Architecture Sourcebook: Design and Evolution of a Mobile Phone OS*. Wiley, 2007.
16. Joachim Niehren, Jan Schwinghammer, and Gert Smolka. A Concurrent Lambda Calculus with Futures. *Theoretical Computer Science*, 364(3):338–356, nov 2006.
17. Marjan Sirjani. Rebeca: Theory, applications, and tools. In *FMCO*, 2006.
18. Derek Wyatt. *Akka Concurrency*. Artima, 2013.