Analysis of synchronisation patterns in active objects based on behavioural types

Vicenzo Mastandrea



École doctorale STIC

Unité de recherche: Inria, équipe Focus / I3S, équipe Scale

Thèse de doctorat

Présentée en vue de l'obtention du grade de

Docteur en Science

Mention Informatique

de

UNIVERSITÉ CÔTE D'AZUR

par

Vincenzo Mastandrea

Analysis of synchronisation patterns in active objects based on behavioural types

Dirigée par Ludovic Henrio et Cosimo Laneve

Soutenue le 15 Décembre 2017

Devant le jury composé de :

Ferruccio	Damiani	Associate Professor, University of Turin	Rapporteur
Robert	de Simone	Research Director, Inria Aoste	Examinateur
Ludovic	Henrio	Research Director, CNRS, I3S Scale	Directeur de thèse
Florian	Kammüller	Senior Lecturer, Middlesex University	Examinateur
Cosimo	Laneve	Professor, University of Bologna, Inria Focus	Co-directeur de thèse
Tobias	Wrigstad	Associate Professeur, University of Uppsala	Rapporteur

Abstract [1]

The active object concept is a powerful computational model for defining distributed and concurrent systems. This model has recently gained prominence, largely thanks to its simplicity and its abstraction level. Thanks to this characteristics, the Active object model help the programmer to implement large scale systems, with an higher degree of parallelism and better performance.

The literature presents many models based on active objects and each one is characterised by a different level of transparency for the programmer.

In this work we study an Active object model with no explicit future type and with wait-by-necessity synchronisations, a lightweight technique that synchronises invocations when the corresponding values are strictly needed. Implicit future type and wait-by-necessity synchronisations entails the highest level of transparency that an Active object model can provide. Although high concurrency combined with a high level of transparency leads to good performances, they also make the system more prone to problems such as deadlocks.This is the reason that led us to study deadlock analysis in this active objects model.

The development of our deadlock analysis is divided in two main works, which are presented in an incremental order of complexity. Each of these works faces a different difficulty of the proposed active object language. In the first work we focus on the implicit synchronisation on the availability of some value (where the producer of the value might be decided at runtime), whereas previous work allowed only explicit synchronisation on the termination of a well-identified request. This way we are able to analyse the data-flow synchronisation inherent to languages that feature wait-by-necessity.

In the second work we present 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

[1]This work was partly funded by the French Government (National Research Agency, ANR) through the "Investments for the Future" Program reference ANR-11-LABX-0031-01.

fields, thus allowing us to compute behavioural types that express synchronisation patterns in a precise way.

For both analysis we provide a type-system and a solver inferring the type of a program so that deadlocks can be identified statically. As a consequence we can automatically verify the absence of deadlocks in active object based programs with wait-by-necessity synchronisations and stateful active objects.

Résumé [2]

Le concept d'objet actif est un modèle de calcul puissant utilisé pour définir des systèmes distribués et concurrents. Ce modèle a récemment gagné en importance, en grande partie grâce à sa simplicité et à son niveau d'abstraction. Grâce à ces caractéristiques, le modèle d'objet actif aide le programmeur à implémenter des grands systèmes, avec un haut degré de parallélisme et de meilleures performances.

La littérature présente de nombreux modèles basés sur des objets actifs et chacun est caractérisé par un différent niveau de transparence pour le programmeur.

Dans ce travail, nous étudions un modèle d'objet actif sans type futur explicite et avec 'attente par nécessité', une technique qui déclenche uns synchronisation sur la valeur retournée par une invocation lorsque celle ci est strictement nécessaires. Les futurs implicites et l'attente par nécessité impliquent le plus haut niveau de transparence qu'un modèle d'objet actif peut fournir. Bien que la concurrence élevée combinée à un haut niveau de transparence conduise à de bonnes performances, elles rendent le système plus propice à des problèmes comme les deadlocks. C'est la raison qui nous a conduit à étudier l'analyse de deadlocks dans ce modèle d'objets actifs.

Le développement de notre analyse de les deadloks est divisé en deux travaux principaux, qui sont présentés dans un ordre de complexité croissant. Chacun de ces travaux est confronté à une difficulté différente de la langue d'objet active proposée.

Dans le premier travail, nous nous concentrons sur la synchronisation implicite sur la disponibilité d'une certaine valeur (où le producteur de la valeur pourrait être décidé au moment de l'exécution), alors que les travaux précédents n'autorisaient que la synchronisation explicite lors de la fin d'une requête bien identifiée. De cette façon, nous pouvons analyser la synchronisation des flux de données inhérente aux langues qui permettent une attente par nécessité.

Dans le deuxième travail, nous présentons une technique d'analyse statique basée sur des effets et des types comportementaux pour dériver des modèles de

[2]This work was partly funded by the French Government (National Research Agency, ANR) through the "Investments for the Future" Program reference ANR-11-LABX-0031-01.

synchronisation d'objets actifs et confirmant l'absence de deadlock dans ce contexte. Cela est difficile parce que les objets actifs utilisent des futurs pour référencer les résultats d'invocations asynchrones en attente et que ces futurs peuvent être stockés dans des champs d'objet, passées comme paramètre de méthode ou retournés par des méthodes. Notre système d'effets trace l'accès aux champs d'objet, ce qui nous permet de calculer des types comportementaux qui expriment des modèles de synchronisation de manière précise.

Pour les deux analyses, nous fournissons un système de type et un solveur dont ils déduisent le type du programme afin que les deadlocks puissent être identifiés de manière statique. En conséquence, nous pouvons vérifier automatiquement l'absence de blocages dans des programmes basés sur des objets actifs avec des synchronisations d'attente par nécessité et des objets actifs dotés d'un état interne.

Contents

List of Figures

List of Listings

Chapter 1

Introduction

Contents

1.1 Motivation

The evolution of the Internet infrastructure, the development of technologies that have enormously increased connection bandwidth, and the exponential increase of the number of mobile devices has opened the doors to new technologies and paradigms. Nowadays computations are less and less centralised and increasingly distributed.

Two paradigms are becoming increasingly prominent, having a strong impact on the industry and on research, they are: cloud computing Mell and Grance [2011] and Internet of Things paradigm (IoT) Al-Fuqaha et al. [2015]. Nowadays industry rush to launch new cloud or IoT based products while research lab and university try to develop technique that could make the implementation of these complex distributed systems easier and scalable.

One of the open question that the research tries to solve is to find a computational model that best suits the requirements of those distributed systems. Thanks to its features, more and more often, the Active object model has been chosen as computational model for distributed system.

The Active object model provides an alternative to the conventional concurrency model that relies on synchronisation of shared mutable state using locks. The approach proposed by the Active object model, which is based on non-blocking interactions via asynchronous method calls, perfectly fits the requests in the programming of complex distributed platforms.

For instance, by design IoT systems involves a large amount of devices that by interacting with each other and executing minimal tasks change their internal state, within the constraints of cost and power. In the same way, the Active object model with its lightweight design is able to scale really well without consuming too many computing resources, by breaking down business logic into minimal tasks executed by each active objects.

Another element that relates active objects and IoT is that, during the development of complex distributed systems, it is almost impossible collect a huge number of devices to run the system, simulation are used to test these system. In fact, the Active object model is very suitable to develop simulation of huge distributed systems thanks to the fact that each active object has the possibility to create new active objects with a programmable supervision strategies. Spawning new active object makes this model also very suitable for simulating device managers and hierarchical groups of devices.

Additionally, the features provided by the Active object model make it a good choice also for cloud applications. The property of scalability guaranteed by the Active object model is precisely one of the main reason that has prompted Microsoft to base its framework Orleans on it. In fact, as stated by Microsoft, Orleans has as main goal the building of distributed and high-scale computing applications, without the need to learn and apply complex concurrency or other scaling patterns.

There are also other many reasons that make the Active object model suitable for implementing complex distributed system, such as the loose-coupling and the strict isolation of the active objects, stemming from the fact that each task executed by an active object can only modify the state of the active object that is

running it.

Though simplifying the development of complex distributed system is a notable problem, verifying that these systems do not present well-known concurrency bugs is also very important. Indeed problems such as deadlocks, race conditions, and data races are not rare in distributed programs.

The Active object model allows formalisation and verification thanks to the fact that the communication pattern is well-defined as well as the accesses to objects and variables. The characteristics of the Active object model facilitates the development of tools for static analysis and validation of programs, that can be used by programmers to avoid and detect concurrency bugs.

The possibility to develop efficient analysis with the other features already mentioned before, brought the Active object model to be used in some important europeans project, such as the Envisage[1] project, in the area of cloud computing.

For all these reasons, the active object approach can be considered a very good model to develop complex distributed systems due to the the loose-coupling, the strict isolation, the absence of shared memory, and the liability to formalisation and verification.

Therefore a lot of different models and languages based on active objects have been developed during the years. These models and languages generally differ in how futures (special objects returned by an asynchronous invocations, which are used to handle synchronisations) are treated and how synchronisation patterns can be built.

Another important aspect that differentiates these models is how much they can hide to the programmer all the aspects that concern concurrency and distribution. The transparency plays an important rules in the definition of an Active object model, especially because the active objects want to be a well-integrated abstraction for concurrency and distribution that leaves the programmers focused on the functional requirements. However, the majority of the Active object models provide explicit future definition and special operations to handle synchronisations, which allows the programmer to check whether the method has finished and at the same time retrieves the method result. The information provided by the programmers facilitate the development of verification tools, that can also demonstrate a

[1]EU project FP7-ICT-610582 ENVISAGE: Engineering Virtualized Services.

good level of accuracy in the analysis of synchronisation patterns. However, with this approach, programmers must know how to deal with synchronisations, and may be tempted to add too much synchronisation points to simplify the reasoning on the program.

1.2 Objectives

We have seen that the development of good verification tools for the analysis of synchronisation patterns is important as much as the transparency of the model, especially because in distributed systems problems like deadlocks, data races, and race condition represent insidious and recurring threats.

With this thesis we want to help the programmers that use Active object model to implement distributed systems, by providing a software analysis technique that is able analyse synchronisation patterns in order to detect the presence of deadlocks and to analyse effects. Furthermore, wanting to help more the programmer, with our analysis we want to target an Active object model with a very high level of transparency. The active model we want to target is a model in which futures are not explicitly identified, then programmers do not have to make distinction between values and future, and where also synchronisations are implicit and performed only on the availability of the result of a method invocation.

Many different static and dynamic analysis techniques have been used to analyse deadlocks and more generally synchronisation patterns, such as: abstract interpretation, model checking, symbolic execution, data-flow analysis. Unfortunately, as we will see in Section 2.8, most of these technique lack in the analysis of deadlock for systems with mutual recursion and dynamic resource creation. Moreover, a strong commitment from the programmer is required to annotate programs in order to process those analysis.

In this thesis we also want to provide an analysis that can to be performed with very little human interaction, or better in a fully automatic way, which does not have impact on the performance of the system, and can easily scale.

Another important feature that a program analysis should have is the possibility to be easily extended, improved, and even adapted to analyse different models. Io order to provide a good adaptability and maintainability of the analysis, we want

to develop an approach that is modular and that also allows several techniques to be combined.

Finally, by taking into account all the difficulties that we can face during the development of such analysis, we† close the thesis with a reflection on the essential features and the best synchronisation strategies that an Active object model should have, in order to identify the good mix between transparency and verification aspects that an active object should have.

1.3 Contributions

The global contribution of this thesis is provide a static analysis technique for the verification of the absence of deadlocks and an analysis of effects in active object programs with transparent futures. The development of this static analysis technique gives us also the possibility to better understand how different synchronisation approaches can impact the static verification. The main contribution of this thesis can be summarised in three main points described below. More details on the content of the chapters are available in Section 1.4.

Deadlock analysis technique for Active object model. The first contribution propose a static analysis technique based on behavioural types in which we removed the possibility of having stateful active object, in order to focus on how to handle the two main characteristics of the model: the absence of explicit future types and implicit synchronisations. As we will see in more details in Chapter 4, the combination of these two characteristics of the model can bring to the necessity of producing a set of dependencies between active objects that can be unbounded in case of recursive methods. We provide:

- a **behavioural type system** that associates behavioural types to program methods;
- a **behavioural type analysis** that is able to translate behavioural types into a potential unbounded graph of "waiting for" dependencies;
- an **adaptation of the analysis of circularities** proposed by Kobayashi and Laneve [2017], that, taking as input the behavioural type sys-

tem and the behavioural type analysis, is able to detect deadlocks;

- the **proof of the correctness** of the deadlock analysis proposed.

Effect analysis for active-object programs. The second main contribution of this thesis is an effect analysis technique based on behavioural type systems that is able to detect the presence of race conditions that can introduce nondeterminism in the execution of an active-object program. Also in this case the **proof of the correctness** of the effect system has been provided.

Deadlock analysis technique handling stateful active object The third main contribution of this thesis is the extension of the static analysis technique proposed in Chapter 4, which takes into account the information provided by the effect analysis and is able to detect deadlocks in an active-object program with stateful active objects. In this contribution we provided: a **behavioural type system**, a **behavioural type analysis**, **adaptation of the analysis of circularities**, and the **proofs of the correctness** of the deadlock analysis proposed.

1.4 Document contents

The presented thesis is structured in five chapters summarised below.

Chapter 2 introduces the context in which the thesis. It is structured in two parts: the first one focus on the Active object model, while the second part focus on program analysis. In the first sections of this chapter we present an overview of the main aspects of the Active object model, a detailed classification of the active object language, and a presentation of the most important active object frameworks. The part related to the Active object model ends with a section dedicated to the use of futures in the Active object model that shows how futures are defined and the approach used to synchronise of them. The second part of this chapter presents the problem that comes out implementing a distributed system and describes the principle program analysis techniques used to detect or face these problems. The chapter ends with the definition of the problems tackled and the objectives of this thesis and a whole section which describes related works.

Chapter 3 present ground-ASP (`gASP`), the language analysed in this thesis through a brief and informal introduction of the characteristics of the language, and then with a section that formalise the semantics of the language and in which will be stated a definition of deadlock for `gASP`. The explanation of all the key points in this chapter are correlated by simple example that want to guide the reader in the comprehension.

Chapter 4 introduces the first contribution of this thesis [Giachino et al., 2016]. In this chapter we present a deadlock detection technique for `gASP` programs. The technique proposed faces the problems that arise from analysing an Active object model language with *implicit future* types and *wait-by-necessity* synchronisation. The deadlock analysis proposed is based on abstract descriptions called *behavioural types*, which are associated to programs by a type system. The purpose of the type system is to collect dependencies between actors and between futures and actors. The technique also provides a solver that, taking in input the behavioural type of a `gASP` program, is able to identify circular dependencies between active objects that indicate the possible presence of deadlocks. In this chapter the proof of the correctness of the analysis proposed is also provided.

Chapter 5 introduces the second contribution of this thesis [Henrio et al., 2017]. This chapter extends the analysis presented in the previous one to handle stateful active object. The presence of stateful active object requires us to analyse synchronisation patterns where the synchronisation of a method can occur in a different context from whence the method has been invoked. For that purpose the type system has been extended with the analysis of method effects. Our effect system traces the access to object fields (e.g. read and write access), and the storage of futures inside those fields, allowing us to compute behavioural types that express synchronisation patterns in a precise way. As in the previous chapter the behavioural types are thereafter analysed by a solver, also extended to treat stateful active object, which discovers potential deadlocks.

Chapter 6 closes the thesis summarising the main points of our work and presenting how restrictions can be removed, the possible extensions of this work, and how it can be applied for different purposes.

Chapter 2

Background and Related Works

Contents

This chapter presents the technical background this work is based on. We start by an overview of the Active object model, continuing then with an investigation of the different existing Active object models and languages. During this investigation are pointed out their different features and semantic aspects. After a short summary of the most relevant problems related with the implementation of concurrent and distributed programs, we will move to an overview of the program analysis technique used to prevent or detect the discussed problems. Then we will end the chapter with a brief explanation of our technique and an investigation of similar works.

2.1 Active Object Programming Model

The active object programming model integrates the basic Actor model [Hewitt et al., 1973] [Agha, 1986] with object oriented concepts, whereas the keyword active object was introduced by Lavender and Schmidt [1996], whereas the idea was already introduced in ABCL [Yonezawa et al., 1986].

The actor model is based on concurrent entities (actors) which interact only via asynchronous message passing and react to message reception. An actor is composed by a single thread that processes its messages sequentially and a mailbox in which the messages are collected, although the message delivery is guaranteed, messages can arrive in any order. The state of an actor can be affected only by itself during the processing of a message, which implies that actors can not have side effects on other actors' state. We call this property *state encapsulation*. State encapsulation combined with the absence of multi-threading ensures the absence of *data races*.

The term data race indicates a critical race condition caused by concurrent reads and writes of a shared memory location that leads to an undefined behaviour. Though Actor model prevents any form of direct data-race, race-condition still exists, typically upon message handling or communication.

Actors execute concurrently and communicate asynchronously without transfer of control, which implies that when a message is sent to an actor the sender

continues its execution concurrently, without knowing when the message will be processed. Whenever a computation depends on the result of a message, the programmer must specify a *callback* mechanism where the invoker sends its identity and the invoked actor sends a result message to the invoker.

All the features mentioned above bring us to the consideration that Actor model enforces decoupling which makes this model one of the best choices for parallel and distributed programming. However, callbacks introduce an inversion of control that makes the reasoning on the program difficult. The results of invocations are received at a later stage, when the actor might be in a different state, which makes harder to assess program correctness.

The problem of inversion of control that the callback mechanism introduces is not present in the Active Objects model. This model combines the Actor model with the object-oriented paradigm unifying the notion of Actor and object by giving to each actor an object type and replacing message passing by asynchronous method invocations. Contrarily to the messages of actors, the method invocations of active objects return a result. For each asynchronous method invocation a place-holder is created, in order to store the incoming result, permitting then at the invoker to continue its execution. This place-holder is named *future*. A complete disquisition about future will be presented in Section 2.3.

As we have seen for actors, active objects have a thread associated to them as well. An activity contains one active object, several encapsulated objects named *passive objects* and a queue that collets all the *requests* that are going to be served by the activity.

A method invocation on an active object is named *request*. It is created and added to the *request queue* of the callee, meanwhile the caller continues its execution. The majority of the active object languages implements asynchronous communication between active objects with a First In First Out (FIFO) point-to-point strategy. Other languages enforce the causal order of the messages to preserve the semantics of two sequential method calls. Although these policies could make this models less general, they ensure more determinism yielding active objects to be less prone to race conditions than actors.

There is not a strict separation between Actors and Active objects, they share the same objectives and mostly the same structure, as discussed by Boer et al.

[2017]. For instance, Scala and Akka mix the characteristics of the two models: Scala use method invocations to send messages and Akka allows actors to synchronise on futures.

We can conclude saying that Actors and Active objects enforce decoupling stemming from the fact that each processes running on an activity can have only direct access to the object that are part of that activity. Strict isolation of concurrent entities makes them desirable to implement distributed systems, that is why Active objects have been playing a significant role in service-oriented programming and mobile computing [Crespo et al., 2013] [Göransson, 2014].

2.2 Classification of Active Object Languages

Numerous adaptations of the original Active object model have been proposed in the literature. The difference between these models is mostly based on: how this models associate objects to threads; how requests are served; and, as will be presented later, how futures are defined and the approach used to synchronise on them.

In this section, we present a classification of the principal models and languages derived from Active objects, which is based on the first two points.

2.2.1 Objects in Active objects models

One of the principal characteristics that differentiates the various Active object models is the relation between the concept of active object and the classical concept of object in object oriented programming. The existing active object languages can be divided in three different categories.

Uniform Object Model

In this model all objects are active objects which communicate through requests. The fact that there are only active objects implies that all the objects have their own execution thread and request queue. The characteristics of the uniform object model makes it very convenient when the aims of the model are a good formalisation and an easier reasoning. That is the main reason that led Creol [Johnsen

et al., 2003, 2006] [Johnsen and Owe, 2007] to use this model, allowing it to have a compositional proof theory [de Boer et al., 2007].

On the contrary the uniform object model lacks in scalability when put into practice, the performance of the application becomes quickly poor when executed on commodity computers.

One of the ways to mitigate the lack of scalability is to implement the model by associating each object to a *virtual thread*. Virtual threads are threads that can be manipulated at the application level, but that might not exist at the operating system level. Although this solution could solve the lack in scalability, virtual thread are sometimes difficult to implement and the management of all the additional data structure attached to each active object can also be problematic. Despite the scalability issue the uniform object model still remains one of the best model if the target is the facility to formalise and reason on it.

Non Uniform Object Model

In the non uniform object model, objects may be either active or passive, and are organised into activity. Activities contain one active object and several passive objects. Passive objects are part of the state of the active object, which is accessible only locally and cannot be the target by asynchronous method invocations.

The calculus ASP, introduced by Caromel et al. [2004], is an example of a non uniform object model, which limits the accessibility of a passive object to a single active object preventing concurrent state modification. This model, compared to the previous one, decreases the number of active objects, solving the lack of scalability (in terms of the number of threads) of the uniform object model. As we could expect this feature of the language makes the model become more complex than the previous one, making this model tougher to formalise and reason about.

Object Group Model

The object group model, where ABS [Johnsen et al., 2011b] is one of the main representatives, is a model inspired from the notion of *group membership* [Birman and Joseph, 1987] in which groups are assigned only once. Activities, which in ABS are called Concurrent Object Groups (COGs), have the same structure of

the non uniform object model, but in this case methods can be directly invoked on the object of the group from outside the activity. As we can expect this model is in the middle between the previous two. Conserving the facility to be formalised, object group model is able to reach a better scalability than the uniform model, but not as high as the scalability of the non uniform object model because of the structure necessary to make all objects accessible (this is only true in a distributed setting).

2.2.2 Scheduling Models

After the analysis of how Active-object model deals with objects, in this subsection we want to show how these models treat another important concept: the threads. More precisely we are going to present a small classification about how requests interleaving is handled by the scheduler in the various Active object models.

Mono-threaded Scheduling

The mono-threaded scheduling is the approach proposed in the original Actor and Active object models, and in ASP. It specifies that an activity contains only a single thread that serves all the request without possible interleaving. This model does not present any feature that can stop the execution of a method, before its completion. The absence of interleaving gives to this model two fundamental properties: the absence of data races (race condition can still be possible); and local determinism. It should be noted that only local determinism is guaranteed. The order in which the requests are served is still unpredictable, then the global behaviour of a system is still nondeterministic. The absence of interleaving makes the model more prone to deadlocks. However, local determinism is a good property in order to develop a more precise static analysis technique, and it was one of key reasons that leads us to face the problem of deadlock detection of the ASP model.

Cooperative Scheduling

The cooperative scheduling model, introduced by Creol, takes its inspiration from coroutines, which have been presented the first time by Conway [1963] and then

implemented in Simula 67 [Dahl et al., 1968]. As coroutines can have multiple entry points for suspending and resuming execution. Cooperative scheduling, contrarily to mono-threaded scheduling, is based on the idea that a request should not lock the thread of an active object until its termination, leaving the possibility to suspend and resume request execution.

Therefore cooperative scheduling represents a solution to the need for a controlled preemption of requests, providing the possibility to release the thread of an active object during the execution of a method invocation. With this approach interleaving between request executions can be handled implicitly or explicitly. Languages and models like Creol and AmbientTalk, interrupt the execution of the current thread while waiting for the result of another method invocation.

Other models like ABS, Encore, and JCobox provide explicit mechanism to release a running request before completion, in order to let another request progress. The release of a request may be done either based on specific conditions or in an unconditional way. Though requests might interleave and activities have only a single thread running at a time, data races are avoided. Letting the programmer place release points, if associated with a high expertise can decrease the possibility of generating deadlocks, otherwise it can lead to a system implementation with unpredictable behaviours caused by the large number of data race conditions. This depends on the fact that, as we have said for mono-threaded scheduling, the interleaving of request execution has impact on the determinism of the system behaviour. Cooperative scheduling does not guarantee neither local nor global determinism, making program analysis tougher and less precise.

Multi-thread Scheduling

Multi-threaded scheduling model supports the parallel execution of several threads inside an active object. This approach can be divided in two categories: single request parallelism and multi request execution. Languages as Encore belong to the first category, where data-level parallelism is allowed inside a request. Requests may not execute in parallel, but a single request processing can be parallelised and splitted in several threads. The second category, which includes programming languages like MultiASP [Henrio and Kammüller, 2015], allows the contemporary

execution of multiple requests.

Unlike the previous two scheduling models, in multi-threaded scheduling data races can occur if the requests manipulate concurrently the same active object field. The first category leads to data races only within a single request, the second one allows data races only within a single activity, because activities are still isolated from each other.

2.3 Futures in Active object model

Describing Active objects is not possible without mentioning the relevant role played by futures. How futures are defined and the approach used to synchronise on them has a significant impact on the features of the model. In the following we discuss the concept of future and how it has evolved during the years; the role of futures in Active object models and how the various models differ in the accessing of future and synchronisation patterns.

Future concept

In programming languages, a future is an entity representing the result of a parallel computation. A future is thus a place-holder for a value being computed, and we generally say that a future is *resolved* when the associated value is computed. In some languages, a distinction is made between the future resolution and its *update*, i.e. the replacement of a reference to a future by the computed value. A future is a programming abstraction that has been introduced by Baker Jr. and Hewitt [1977]. It has then been used in programming language like MultiLisp [Halstead, 1985]. A future is not only a place-holder, but also it provides naturally some form of synchronisation that allows some instructions to be executed when a given result is computed. The synchronisation mechanism provided by futures is closely related with the flow of data in the programs.

In MultiLisp, the future construct creates a thread and returns a future. The created future can be manipulated by operations like assignment, which do not need a real value, whereas the program would *automatically block* when trying to use a future for an operation that would require the future value (e.g. an arithmetic

expression). In MultiLisp, futures are *implicit* in the sense that they do not have a specific type and that there is no specific instruction for accessing a future, however they are created explicitely. The program only blocks when the value computed by another thread is necessary. Typed futures appeared with ABCL/f [Taura et al., 1994] in order to represent the result of asynchronous method invocations, i.e. methods invocations that are performed in parallel with the code that triggered the method.

To our knowledge, the first work on formalisation by semantic rules of Futures was presented by Flanagan and Felleisen [1999, 1995] and was intended at program optimisation. This work focuses on the futures of MultiLisp, which are explicitly created. The authors "compile" a program with futures into a low-level program that does explicit touch operations for resolving the future, and then optimise the number of necessary touch operations.

In a similar vein, $\lambda(\mathit{fut})$ is a concurrent lambda calculus with futures with cells and handles. Niehren, Schwinghammer, and Smolka [2006] defined a semantics for this calculus, and two type systems. Futures in $\lambda(\mathit{fut})$ are explicitly created, similarly to Multilisp. Alice ML, developed by Niehren et al. [2007], can be considered as an implementation of $\lambda(\mathit{fut})$.

Futures for actors and active objects

ABCL/f paved the way for the appearance of active object languages like Eiffel//, which then inspired ProActive, and in parallel, Creol.

ProActive is a Java library; in ProActive [Caromel et al., 2006] futures are implicit like in MultiLisp, in the Java library they are implemented with proxies that hide the future and provide an object interface similar to a standard object except that any access requiring the object's value (e.g. a method invocation) may trigger a blocking synchronisation. The ASP calculus [Caromel et al., 2004] formalises the Java library, and in particular allowed the proof of results of partial confluence.

Creol is an active object language with a non-blocking wait on future: the only way to access a future is the *await* statement that enables *cooperative multi-threading* based on future availability: the execution of the current method is

interrupted when reaching the *await* statement, and only continued when the future is resolved. After that the current method has been unscheduled other request can be served. The *await* statement creates interleaving between requests that avoids blocking the execute of the current actor.

de Boer, Clarke, and Johnsen [2007] provided the first richer version of future manipulation through cooperative scheduling and a compositional proof theory for the language. The future manipulation primitives of this language will be then used, almost unchanged, in JCobox [Schäfer and Poetzsch-Heffter, 2010], ABS [Johnsen et al., 2011a], and Encore [Brandauer et al., 2015] three more recent active object languages. In those different works, futures are now explicitly typed with a parametric type of the form *fut*<*T*> and they can be accessed in various ways, which will be discussed below.

At the same time, AmbientTalk was developed by Dedecker et al. [2006]. It features a quite different semantics for future access. AmbientTalk is based on the E Programming Language [Miller et al., 2005] which implements a communicating event-loop actor model. Futures in AmbientTalk are fully asynchronous: calls on futures trigger an asynchronous invocation that will be executed when the future is available. A special construct *when-becomes-catch* is used to define the continuations to be executed when the future is resolved.

In a more industrial settings, futures were introduced in Java in 2004 and used in one of the standard library for concurrent programming. A parametric type is used for future variables which are explicitly retrieved by a *get* primitive [Goetz et al., 2006]. Somehow, these simple futures are explicitly created and have a locking access. Akka [Haller and Odersky, 2009, Wyatt, 2013] is a scalable library for implementing actors on top of Java and Scala. In Akka, futures are massively used, either in actor messages that return a value or systematically in the messages of the *typed actors*[1]. Akka also use a parametric type for representing futures. Futures can also be created explicitly, a bit similarly to the MultiLisp *future* construct that creates a new thread.

In the last year also Microsoft has appeared on the scene of active object with Orleans [Bykov et al., 2011, Bernstein and Bykov, 2016]. Orleans is an actor framework developed at Microsoft research and used in several Microsoft products,

[1] Akka's typed actors are some kind of active objects.

including online games relying on a cloud infrastructure. Its strength is the runtime support for creation, execution, and state management of actors. Orleans relies on a non-uniform Active object model with copies of transmitted passive objects, like ASP. The semantics of futures is based on continuations and relies on an *await* statement similar to that of ABS and Encore, however, there is no primitive for cooperative scheduling. Consequently, the programmer has to take care of possible modifications of the object state between the *await* and the future access. This semantics for future access is similar to the way futures are handled in general in Akka.

The different forms of future access

Originally, futures were designed as synchronisation entities, the first way to use futures is to block a thread trying to use a future. When future are created transparently for the programmer and the synchronisation occurs only when the future value is strictly needed, this is called *wait-by-necessity*. In this setting futures can be transmitted between entities/functions without synchronisation. Futures that can be manipulated as any standard object of the languages are sometimes called *first-class futures*. In MultiLisp and ASP, future synchronisation is transparent and automatic.

Other languages provide future manipulation primitives, starting from touch/peek in ABCL/f, the advent of typed futures allowed the definition of richer and better checked future manipulation libraries.

In some languages like Creol or AmbientTalk, futures can only be accessed asynchronously, i.e. the constructs for manipulating a future only allows the programmer to register some piece of code that will be executed when the future is resolved. In Creol, the principle is to interrupt the execution of the current thread while waiting for the future. In AmbientTalk, such asynchronous continuation can also be expressed in-line but additionally future access can trigger an asynchronous method invocation that will be scheduled asynchronously after the current method is finished.

More recently, languages feature at the same time different future access primitives. JCobox, Encore, and ABS allows the programmer to choose between a

cooperative scheduling access using *await* like in Creol, or a strict synchronisation preventing thread interleaving, using a *get* primitive. Interestingly, Encore also provides a way to trigger asynchronous method invocation on future resolution called *future chaining*, similarly to AmbientTalk.

Akka has a distinguished policy concerning future access. While blocking future access is possible in Akka (using *Await.result* or *Await.ready*, not to be confused with the await of ABS!), Akka advocates not to use blocking access. Instead, asynchronous future accesses should be preferred according to the documentation, like in AmbientTalk, by triggering a method invocation upon future availability. Akka future manipulation comes with several advance features such as ordering of asynchronous future accesses, failure handling (a future can be determined as a success or a failure, i.e. an exception), response timeout, etc.

Complex synchronisation patterns with futures

It is worth mentioning that futures are also used to implement some more complex synchronisation patterns. For example, Encore can use futures to coordinate parallel computations [Fernandez-Reyes et al., 2016] featuring operators to gather futures or perform computation pipelining. In ASP and ProActive, group of futures can be created to represent results of group communications enabling SPMD computation with active objects [Baduel et al., 2005]. Akka also provides methods for pipelining, iterating, and folding several future computations.

2.4 Active object languages and frameworks

In the following section are presented some of the most popular active object programming languages. For each of them we show the main features, the ecosystem (i.e., related tools or backends), and the reasons why they have become so relevant.

2.4.1 ABS

The Abstract Behavioral Specification language (ABS), introduced by Johnsen et al. [2011b], is an object-oriented modelling language based on active objects

that has been created with the purpose of modelling and verifying distributed applications. ABS is inspired by Creol and JCobox from where it took some of its main features. The object group model of ABS, which derives from JCobox, is based on the notion of Concurrent Object Group (COG). The objects of an application are allocated into several COGs. A COG is composed by a request queue and a set of threads. These threads have been created as a result of asynchronous method calls to any of the objects that the COG owns. In ABS both asynchronous method calls and futures are explicit. Futures are explicitly typed with a parametric type of the form `Fut<T>`. Despite the contemporary presence of more than one thread in a single COG, only one thread is executing (*active*) at a time. ABS also provide a cooperative scheduling like Creol and JCobox. ABS features an `await` language construct, that releases the thread if the specified future is unresolved or a defined condition does not hold and a `get` construct, that blocks the thread execution until the specified future is resolved, but the thread is still held by the process. There is also the construct `suspend` that unconditionally unschedules the thread.

Since ABS main purpose is the verification of distributed applications it is accompanied by a significant number of verification tools[2]. Below are presented the most relevant tool divided by the analysed problem:

Deadlock analysis: two tools are available to statically study deadlock freedom of ABS program: DSA and SACO. DSA is a deadlock analyser, designed by Giachino et al. [2015], which is based on two modules: an behavioural inference system and an algorithm capable to identify circularities in an unbounded dependency graph. SACO, introduced by Milanova et al. [2005], performs a points-to-point analysis which identifies the set of objects and tasks created along any execution and construct a dependency graph. A precise may-happen-in-parallel (MHP) has been presented by Albert et al. [2012], and is also used to reduce the number of false positive in the identification of deadlocks.

Resource analysis: two other tools are available for resource analysis of ABS programs. SRA computes upper bounds of virtual machine usages in a dia-

[2]ABS related tools can be found at: `http://abs-models.org/`

lect of ABS, called vml, with explicit acquire and release operations of virtual machines. This tool is based on a type system associating programs with behavioural types that record relevant information for resource usage (creations, releases, and concurrent operations) and a translation function that takes behavioural types and return cost equations. It is integrated with the solver CoFloCo that, given the cost equations, produces the result.
The second tool, SACO, as in the case of deadlock, performs a points-to and a may-happen-in-parallel analysis to compute an upper bounds of virtual machine usages in an ABS program. In this approach, computer resources are abstracted away through a model with quantifiable measures. This analysis has also been generalised for concurrent object-based programs [Albert et al., 2014].

Program verifier: KeY-ABS [Din et al., 2015a,b], based on the KeY reasoning framework Beckert et al. [2007], allows the specification and verification of general, user-defined properties on ABS programs.

ABS backends for Java, Haskell [Bezirgiannis and Boer, 2016], and ProActive are available. To improve the performance of the previous Java backend based on Java 6 a new backend based on Java 8 [Serbanescu et al., 2016] is currently under development. The implementation of this new backend has to face a relevant problem: in fact, unlike Haskel, the JVM 8 does not support thread continuation, then it is not able to fully preserve the semantic of ABS. There is also a ProActive backend for ABS that specially targets distributed High Performance Computing (HPC). This backend is fully implemented and, moreover, the correctness of the translation is formally proven [Boer et al., 2017].

2.4.2 ASP and ProActive

ASP [Caromel et al., 2004] is a programming language based on the Active object model. ASP has proven properties of determinism, and particularly fits the formalisation of object mobility, groups, and componentised objects [Caromel and Henrio, 2005]. ASP follows a non uniform Active object model with high transparency: active and passive objects are almost always manipulated in the program

through the same syntactic constructs. Contrarily to Creol, JCobox, and ABS, ASP does not provide a cooperative scheduling model. In ASP once a request starts to be executed, it runs up to completion without ever releasing the thread. Unlike ABS, in ASP future are not explicitly typed, being therefore completely transparent to the programmer. In practice, futures are dynamically created upon asynchronous remote method calls.

Synchronisation is also handled automatically. ASP features implicit synchronisation of futures (called wait-by-necessity). With wait-by-necessity the program execution is only blocked when a value to be returned by a method is needed to evaluate a term. ASP has first class futures, which means that futures can be sent as arguments of method invocations, returned by methods, or stored in object fields without being synchronised. Indeed, as futures are transparent, they are also transparently passed between activities. When the future is resolved, its value is automatically updated for all activities it has been passed to. Several future update strategies have been explored in ProActive for this purpose by Henrio, Khan, Ranaldo, and Zimeo [2011].

ProActive [Baduel et al., 2006] is the implementation of ASP in Java. The Active object model [Lavender and Schmidt, 1996] and the distributed object model of Java (Java Remote Method Invocation (RMI)) [Wollrath et al., 1996] were released the same year. ProActive has combined those two models and has implemented the semantics of ASP in a Java library that offers full support for distributed execution. As the Active object model of ASP is transparent, ProActive active objects are bounded as much as possible with the same syntactic constructs as regular Java objects.

One aspect of ProActive is also dedicated to components [Baude et al., 2015]. ProActive active objects form a programming model that is suitable for component-based composition of distributed applications through the Grid Component Model (GCM). The Vercors platform, developed by Henrio, Kulankhina, and Madelaine [2016], enables the design, specification and verification of ProActive components through an Eclipse plugin, similarly to the verification abilities of ABS.

ProActive is intended for distribution, it forms a complete middleware that supports application deployment on distributed infrastructures such as clusters, grids, and clouds. The ProActive middleware has proven to be scalable and suitable for

distributed HPC [Amedro et al., 2010]. ProActive also implements MultiASP, the active object programming language that extends ASP with the support of multiactive objects. Unlike ASP, MultiASP feature a multi-thread scheduling models (Section 2.2.2).

2.4.3 Encore

Encore [Brandauer et al., 2015] is an active object-based parallel language inspired from Joëlle [Clarke et al., 2008]. Encore is a programming language, that relies on the Active object model mixed with other parallel patterns, as Single Instruction Multiple Data (SIMD), based on a non uniform object model as ASP and a cooperative scheduling model as ABS. Encore presents a syntax similar to ABS, to distinguish between synchronous and asynchronous calls, but as in ASP, method calls on active object are asynchronous and method calls on passive object are synchronous. More precisely in Encore we speak about active and passive classes, such that an object belonging to an active class is an active object and one belonging to a passive class is a passive object. Additionally futures are typed dynamically, but contrarily to ASP their value must be explicitly retrieved via a `get` construct. One of the main differences with the programming languages presented before is that Encore provides two forms of asynchronous computation: asynchronous method calls and asynchronous parallel constructs inside a request. Internal parallelism can be explicit through `async` blocks, or it can be implicit through *parallel combinators*, an abstraction that spawns SIMD tasks and joins them automatically. It is relevant to mention that in Encore all parallel constructs are unified with the use of futures for handling asynchrony. In Encore, active objects encapsulate passive objects, in terms of *ownership*, but unlike ProActive, passive objects can be shared by reference across the activity boundaries. In order to prevent data race caused by concurrent modifications on passive object, Encore provide a capability system [Castegren and Wrigstad, 2016]. The programmer can assign to a passive object a *capability* type. Capability types define both the accessible interface of the object and the level of accessibility of this interface.

Encore extends the cooperative scheduling model of ABS with providing a chaining operator, that adds a callback to a future, in order to execute it when

the future is resolved.

An Encore program is compiled through a source-to-source compiler, written in Haskell, that produces C code complying to the C11 standard, which gives to the programmer the possibility to apply to a compiled Encore program all the existing tools for C programs.

2.4.4 Scala and Akka actors

Scala Actors [Haller and Odersky, 2009] was one of the first object oriented languages not originally based on actor, which at a later time provided a library implementing the actor model. We can say that Scala Actors is one of the reasons why actor model is so pervasive in distributed settings, as much in education as in industry [Haller, 2012]. Starting with Scala 2.11.0, the Scala Actors library is deprecated, leaving room to Akka Inc. [2017]. Akka is a platform developed in Scala and Java for building scalable event-driven fault-tolerant systems on the Java Virtual Machine (JVM). Akka improves the performance of distributed actors on the JVM due to a better implementation of serialisation. Akka actors are implemented on top of dispatchers, which makes the deployment more transparent to the user obtaining better performance and scalability. Dispatchers manage thread pools backed by a blocking queue and implement lightweight event-driven threads that can scale up to millions, even on commodity hardware.

Both Scala and Akka actors support the use of futures for the convenience of the programmer, although this makes a significant difference with respect to the original actor model. The rich type system of Scala, the powerful object syntax, and the composability of higher-order functions make actor definitions and lifecycle management expressive yet succinct.

Akka actors communicate by asynchronous message passing. Scala's infix operator notation, type inference capabilities, and pattern matching on algebraic data types help build nice abstractions, as finite state machines, based on message processing.

As we mention in Section 2.1, callback-oriented event-based programming is not very intuitive because of nonlinear control flow in the programming model. However Scala supports *delimited continuations*, a powerful control flow abstrac-

tion that lets developers program in a direct style even with inversion of control. Delimited continuation are supported by Scala as a compiler plugin, which is used by Akka to implement expressive APIs for data-flow concurrency.

Akka also features Typed Actors, which are the Akka implementation of the Active Objects pattern. Exactly as considered in the Active object pattern, in Typed Actors in Akka method invocations are asynchronous instead of synchronous that has been the default way since Smalltalk came out.

Typed Actors consist of a public interface and an implementation. As with normal Actors you have a public interface instance that will delegate method calls asynchronously to a private instance of the implementation. The advantage of Typed Actors is that they provide a static contract, and it is not needed to define the messages. Unfortunately, in Akka, Typed Actors do not provide the same features provided by the normal Actors (i.e become/unbecome).

In addition Akka also has a number of other components that make concurrent and distributed computing simpler, such as: remote actors, that can be transparently distributed across multiple JVMs; agent-based computing, similar to Clojure [Hickey]; an integrated implementation of software transactional memory [Lesani and Lain, 2013]; a transparent and adaptive load-balancing, cluster rebalancing and a fault-tolerant platform.

The possibility to implement complex distributed system, allowed by the Scala expressiveness, combined with a strong interoperability with Java, makes Akka very suitable for the implementation of huge distributed industrial system.

2.5 Deadlocks and Data race condition

Parallel and distributed programs are difficult to write and a high expertise is needed to deal with the thread synchronisation. Wrong synchronisation or the incorrect placement of release points can lead to nondeterministic behaviour, data races, race condition or deadlocks, as can be seen from the works of Savage et al. [1997], Netzer and Miller [1991], Flanagan and Freund [2009], and Havender [1968].

Trying to detect or predict these two problems is not trivial, since they may not occur during every execution, though they may have catastrophic effects for the overall functionality of the system. In 1994, a deadlock flaw in the Automated

Baggage System of airport of Denver was one of the causes of losses for more than 100 million dollars, and data race even leaded to the loss of human lives, as reported by Leveson and Turner [1993], and also caused massive material losses [Ass].

2.5.1 Deadlocks

Deadlock is a common problem in multiprocessing systems, parallel computing, and distributed systems. A deadlock occurs when there is a circular dependency between the threads. Every thread that holds at least one resource is waiting for the release of some resources held by another thread. The term *resource* can be used for devices, processors, storage media, programs, and data.

A deadlock situation can arise if and only if the following four conditions, which are known as Coffman conditions [Coffman et al., 1971], hold simultaneously in a system:

- at any given instant of time a resource can be held by only one task (**mutual exclusion** condition);
- tasks hold at least one resource while waiting for additional resources (**waiting for** condition);
- resources can not be removed from the task that is holding them, they can only be released voluntarily by the task (**no preemption** condition);
- there exist chain of tasks such that each process must be waiting for at least one resource which is being held by the next task in the chain (**circular wait** condition).

In the Active object field the resources that may contribute to a possible deadlock are the threads that are contained inside an activity, while the tasks are not the activities but the method invocations that each activity have to serve. The occurrence of a deadlock might depend on a variety of conditions which differ depending on the characteristics of the language that we are considering. The circumstances that may lead the system in a deadlock can vary considerably, depending of the scheduling model or the kind of future and future synchronisation approach the chosen language provides.

Figure 2.1 and Figure 2.2 illustrate how deadlock can be reached in an Active object model with uniform object and mono-threaded scheduling, like ASP. The

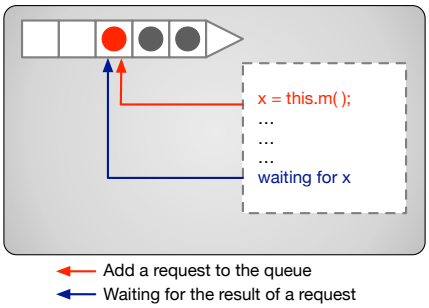

Figure 2.1 – Deadlock involving one active object.

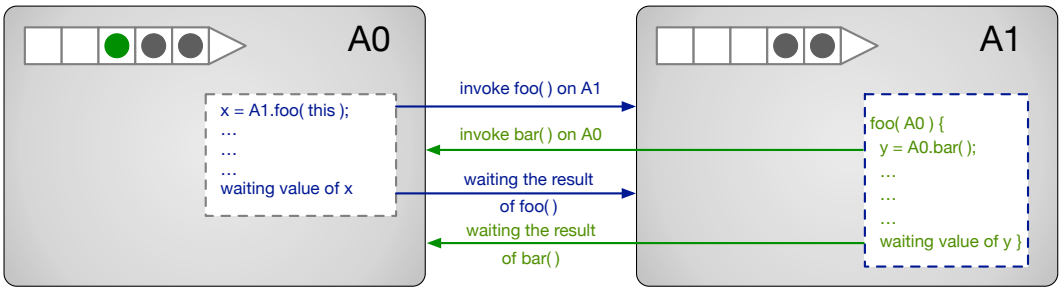

Figure 2.2 – Deadlock involving two active objects.

first example represents a *self deadlock*. In this example we can see the running thread invoking a method `foo` on the same active object. Then, the request related to the method invocation is added to the request queue. Going ahead in the execution the current process wants to use the result of the invocation of `foo` previously done. As we are considering an Active object model without interleaving of request execution, we have that the current process stops until the future related to the method invocation of `foo` is resolved. It is trivial to notice that because `foo` was invoked on the same active object it will never be executed. To execute `foo` the current process should end and it can not until `foo` is not computed.

A similar case involving two active objects is shown in the second example. In this case we have an active object A0, which performs a method invocation (`foo`) on the active object A1, and then it stops the execution waiting for the result of the invocation just done. The method `foo`, processed in A1, starts and it does a method call (`bar`) on the active object A0. This method call is inserted in the

request queue of A0 that is still executing the original request. Also in this case, we can easily notice that the execution of the program leads to a deadlock because the method running in the active object A0 is waiting for the result of the method `foo`, and `foo` is waiting for the result of the method `bar`, and we know that `bar` can not be executed until A0 completes the execution of the running request.

2.5.2 Data race and race condition

A race condition occurs when a program doesn't work as it is supposed to because of an unexpected ordering of events that produces contention over the same resource. A data race occurs when two or more threads access a shared memory location, at least one of the two accesses is a write, and the relative ordering of this accesses is not enforced by a synchronisation pattern. As we have said many times data races and race conditions are some of the worst concurrency bugs, because programs can lead to a nondeterministic behaviour. In a small system unpredictable behaviours may not be relevant, but if we move to critical systems the impact that data races may have could be tragic.

As we have seen before, the majority of the models presented still admit race condition, but most of them prevent data race. Eliminating all data races or race condition in an industrial systems, result to be not possible or inconvenient because of the associated costs or the performance drop.

As reported by Kasikci et al. [2012], Microsoft left purposely unfixed 23 data races in Internet Explorer and Windows Vista due to performance implications [Narayanasamy et al., 2007] and similarly, several races have been left unfixed in the Windows kernel, due to fixing those races did not justify the associated costs [Erickson et al., 2010].

For Kasikci et al. [2012] another very relevant reason due to data races have been left unfixed is that 76%–90% data races do not affect program correctness [Narayanasamy et al., 2007, Erickson et al., 2010, Engler and Ashcraft, 2003, Voung et al., 2007].

The nondeterminism introduced by race condition and data races, having impact on the behaviour of a system, makes static analysis more complex. In case of static deadlock analysis, race condition and data race hinder a precise iden-

tification of the synchronisation patters. The huge number of all the possible synchronisation patterns generated by nondeterminism increase significantly the execution time of the analysis. This time can be reduced by heuristics or by modelling the analysis taking into account some specific characteristics of the studied language or of the implementation of its backend.

2.6 Program analysis

When complex software systems, are designed, the time and effort spent on testing and verification is larger then the time spent on construction. Many different techniques are used to reduce and ease the verification efforts while increasing the coverage. Program analysis is a process which automatically analyses programs behaviour to ensure or detect specific property such as correctness, safety and liveness. Program analysis focuses on two major areas: *program optimisation* and *program correctness*, while the former focuses on improving the performance of programs reducing the resource usage, the latter focuses on ensuring that the program works as intended.

Program analysis can be performed in two main ways: without executing the program (static program analysis) or at runtime (dynamic program analysis). There are as well hybrid techniques that combine static and dynamic program analysis.

Dynamic program analysis

Dynamic analysis is the analysis of computer programs that is performed by executing programs on a real or virtual environment. As dynamic analysis cannot guarantee the full coverage of the source code, to be really effective the target program must be executed with sufficient test inputs, being sure to produce an adequate number of possible behaviours. The usage of software testing measures, such as code coverage, ensures that an adequate portion of the possible behaviours has been tested. Minimising the effect that instrumentation and temporal properties may have on the execution is a non trivial aspect of this technique.

An exhaustive dynamic program analysis, which might run during the exe-

cution of the application, may negatively impact the performance of the system, though an inadequate testing can lead to catastrophic failures similar to the maiden flight of the Ariane 5 rocket launcher, where run time errors resulted in the destruction of the vehicle, as reported by Dowson [1997].

Static program analysis

The term static analysis generally identifies automatic verification, whereas not fully automatic static analysis are usually referred in the literature as program understanding, program comprehension, or code review. It is usually performed on the source code or on some form of the intermediate code generated by the compilers.

Unlike dynamic program analysis the static approach, ideally, aims at guarantees about what the programs may do for all possible inputs, and these guarantees have to be provided automatically. This possibility to reason about the whole set of possible inputs ensures, theoretically, a certain degree of quality before the deployment stage.

Aiming to analyse all the possible program execution, static analysis may take a considerable amount of time or may lead to an exponential problem even for not very big programs. Unlike dynamic analysis the static one takes place during compilation time, making this technique not as time sensitive as the dynamic, however some level of abstraction allows to scale the entire set of possible program executions into a smaller and better analysable model. A really precise technique which is not able to analyse real programs is useless as well as a technique that analyse all the possible programs with a low accuracy, to have a good static analysis a trade-off between precision and abstraction is needed.

Anyway, static analysis is not a technique that is able to say yes every time programs have some property and say no every time they do not. Indeed static analysis is computationally undecidable. Below we present a simple proof that explains why static analysis is undecidable and a solution to this problem.

Undecidability of static analysis. Rice's theorem [Rice, 1953] informally states that all interesting questions about the behaviour of programs (written in Turing-complete programming languages) are undecidable.

We present below and adaptation of the proof proposed by Møller and Schwartzbach.

Let us assume that for all programs P and for all of its possible inputs X_P there is an analyser $A(P, X_P)$ which checks if P is 'safe', in the meaning that for example has no bug, such that:

$$A(P, X_P) = true \iff P(X_P) \textit{ is safe.}$$

Suppose we have a function *not_safe*() which creates an unsafe program, we could use A to decide the halting problem by using as input the following program where TM(j) simulates the j'th Turing machine on empty input:

```
Test() { if(TM(j)) not_safe(); }
```

We can see that the program is 'safe' only if the j'th Turing machine does not halt on empty input. If the hypothetical analyser A exists, then we have a decision procedure for the halting problem, which is known to be impossible.

The undecidability of the static analysis seems to be a discouraging result that could not be solved, but the solution is to produce approximative answers that are still precise enough to fulfil our goal. Most often, such approximations are conservative (or safe), meaning that all errors lean to the same side, which is determined by our intended application.

Consider again the problem of determining if a program is 'safe', the analyser A may answer *true* if the program is definitely 'safe' and must answer 'maybe' if the program may be 'not safe'. The trivial solution is of course to answer *maybe* all the time, so we are facing the challenge of answering *true* as often as possible while obtaining a reasonable analysis performance.

2.6.1 Static analysis techniques

Static analysis approach can be based on one formal technique or on a combination of them. In the following we briefly summarise some of the most used formal methodologies for static program verification.

Abstract interpretation

Abstract interpretation is a program analysis technique formalised by Patrick and Radhia Cousot [1977, 1979]. Abstract interpretation introduces two main concepts:*concrete semantics* and *abstract semantics* of a program. The concrete semantics is a mathematical model which formalises the set of all its possible executions in all possible execution environments, while abstract semantics is a superset of the concrete semantics which covers all possible concrete cases. Abstract interpretation is a technique that considering an *abstract semantics*, requires to prove that if the abstract semantics verifies some properties then also the concrete semantics verifies the same properties.

Model checking

Model checking is a verification technique that, given a system model, is able to explore all possible scenarios in a systematic manner showing that the model satisfies a certain property. The limitation of this technique is that it is able to verify only systems that are finite state or have finite state abstractions. The main challenge is to examine the largest possible amount of states. As it is pointed out by Baier and Katoen [2008], model checkers can currently handle state spaces (the set of the possible states in which the system can be) of about 10^8 to 10^9 states with explicit state-space enumeration, while for specific problems, the dimension of state spaces can be incremented by clever algorithms and tailored data structures handling a number of spaces which goes from 10^{20} up to even 10^{476} as shown by Staunstrup et al. [2000]. Model checking is, potentially, able to discover even the most subtle errors that often remain undetected using emulation, simulation and testing. Techniques based on model checking have been also used in the Active object model environment. In particular, Sirjani [2007] uses the characteristics of actor languages to limit, by partial order reduction, modular verification and abstraction techniques, the state space of the model to check.

Ameur-Boulifa et al. [2017] provide an parametrised model of an active object application that is abstracted into a finite model afterwards.

Symbolic execution

Symbolic execution that could be considered as a specific case of abstract interpretation is a technique which tries to fill the gap between static and dynamic analysis [King, 1976]. This approach analyses programs using a symbolic interpreter to determine what inputs cause each part of a program to execute. The symbolic interpreter follows the execution of the program taking as inputs symbolic values rather than actual values as in normal execution. The drawback of this technique is that symbolically executing all the possible program paths does not scale to large programs. The number of feasible program paths grows exponentially with the program size and can even be infinite in case of programs with unbounded iterations. Solutions to the path explosion problem generally use path-finding heuristics to increase the code coverage [Ma et al., 2011], reduce execution time by parallelising independent paths [Staats and Păsăreanu, 2010], or by merging similar paths [Kuznetsov et al., 2012].

As we have already evidenced, one of the strengths of symbolic execution is that this approach analyses programs reasoning path-by-path rather than reasoning input-by-input like dynamic analysis and other testing paradigms. However, in case of programs for which few inputs take the same path the advantage with testing each input separately is not so marked. Symbolic execution is also leaking when programs interact with their environment by performing system calls, receiving signals, etc. In fact, consistency problems may arise when execution reaches components that are not under control of the symbolic execution tool (e.g., kernel or libraries).

Deductive verification

Deductive verification is another program analysis approach, which consists of generating from the program and its specifications a collection of proof obligations, for which the truth has to be demonstrate by either interactive theorem provers, such as HOL, Isabelle or Coq, automatic theorem provers, or satisfiability modulo theories solvers. The approach of integrating theorem proving with program analysis requires theorem provers in the form of satisfiability checkers for various theories and combination of theories. This approach has two main disadvantage:

the former is that, to facilitate the generation of proof obligation, both the program and its specifications have to be annotated with assertions such as invariants. Some deductive verification tools, like KeY [Ahrendt et al., 2016], use other program analysis technique, such as symbolic execution, to generate more assertions. The later, and probably the biggest, problem of this technique is that it typically requires the user to understand in detail why the system works correctly, and to convey this information to the verification system, either in the form of a sequence of theorems to be proven or in the form of specifications of system components.

Data-flow analysis

Data-flow analysis is a fixpoint based program verification technique, presented the first time by Kildall [1973], which is widely used by compilers to optimise program code. In this approach Control Flow Graphs (CFG) [Allen, 1970] are used. CFG are the representation, using graph notation, of all paths that might be traversed during a program execution. A simple way to perform data-flow analysis of programs is to set up data-flow equations for each node of the CFG and solve them by repeatedly calculating the output from the input locally at each node until the whole system reaches a fixpoint. When finally the whole system has reached a stable state, the verification is performed on each node of the CFG.

Behavioural type systems

Static analysis based on behavioural types systems [Ancona et al., 2016] provide a strong mechanism for reasoning on the possible executions of a program. The behavioural type system usually extend standard types by adding dynamic information that are strictly related to the features of the program targeted by the analysis. As this technique has been chosen to tackle static analysis of deadlock and data race condition in this manuscript, it will be presented more precisely in the section below.

2.6.2 Behavioural type analysis

The analysis based on a behavioural type is a modular technique that can be considered to be composed by four modules. The first module consist of a tra-

ditional static checker. This checker is responsible for verifying if the program is syntactically correct and has no obvious semantic errors.

The second and one of the most relevant module consists of a behavioural type system. The main goal of the behavioural type system is to obtain what we call a *behavioural type program*. The behavioural type program is a representation of the program that is more suitable for the upcoming verification processes and in which all the necessary and most relevant information, strictly related to the analysed problem, are underlined. The success or the failure of the typing process has no meaning related to the targeted problem, a failure during the typing process only implies the impossibility to perform the verification analysis (i.e because of the violation of some restriction). In a program each method can be typed individually, the typing process of a method has no impact on the behavioural types of the other methods. It is possible to consider that once a behaviour is associated to a method it never changes and it will be reused during the analysis of each program in which that method is used.

The third module of a behavioural type system analysis. Taking into account the whole behavioural type program, it derives an abstract representation of the program. The abstract representation is strongly related with the problem targeted by the analysis, for instance this representation can be an object dependency graph in case the targeted problem is the analysis of deadlock, or a set of cost equations in case we want to perform a resource analysis. Unlike the previous module, this one have to be applied each time the program is modified (i.e a method is replaced, or the main function has been modified).

The last module verifies the property that we are considering over the abstracted representation of the program. The verification process is only tied to the abstract program representation easing the use of third-party, and possibly more general, analysers.

One of the big advantage of this approach is clearly the modularity. Where the type system is generally strictly related with the language that is considered, the verification process depends on the abstract representation of the program. This property allow us to tackle the analysis of different programming languages by developing new behavioural type systems that produce the same abstract representation, allowing us to do not develop new verification modules.

The modularity of this technique is not limited to the ease of handle in a simple way different problems and languages, but it extends to the possibility of proving the correctness separately for each module.

The correctness of the typing process and its abstract representation is totally not related with the correctness of the analysis verification, and is usually proven by means of a subject reduction technique [Curry and Feys, 1958]. The correctness of the analysis strongly depend on the analysis technique and in case of tird-party analyser are used the correctness proofs may be already provided.

2.7 Problems tackled, objectives, and contributions

This work presents the theoretical foundations of a static analysis technique based on behavioural type systems. The targeted is the analysis of synchronisation patterns in Active object models.

One of the main goals of this thesis is to present the strength of the behavioural type system based approach for the static analysis of concurrent and distributed programs.

We chose to apply our technique to an Active object model language because of the notable role that this model is having in the concurrent and distributed environment and of the fact that in our opinion this role is going to be more prominent in the next years.

To accomplish our objective, we focus on the detection of deadlocks that, as we have seen before, is a problem with high relevance and application in the newest developments distributed systems environment.

The development of our deadlock analysis is divided in two main works, which are presented in an incremental order of complexity. Each of these works face different features of the proposed active object language. The first work analyses how to trace synchronisation patterns in a language in which futures can be passed around as parameters or returned from methods. The second work analyses synchronisation patterns in stateful objects. As we will see in the continuation the latter deadlock analysis requires an additional effect analysis.

In both of the discussed works are presented:

a behavioural type systems which is able to associate behavioural types from the program code;

a behavioural type analysis which is able to translate a behavioural type program into an abstract representation of the program more suitable for the upcoming verification processes (dependency graph);

an analysis of circularity which is the analysis that takes in input a dependency graph and is able to detect circular dependencies that lead to a possible deadlock;

the proof of correctness of the behavioural type systems and the analysis, to demonstrate the correctness of the behavioural type systems in characterising the underlying programs and producing correct program abstractions without losing relevant information for the analysis.

Even if the contribution of this work is mainly the static analysis of deadlocks in active object language, the orchestration of these type systems as well as the proof of correctness behind them are also a remarkable part of the results obtained in this work. The design of these solutions takes into account the typical problems of static analysis of concurrent and distributed programs. The strategy used for solving these problems is also a relevant contributions of this work and could be generalised to other kind of static analyses.

2.8 Related works on behavioural types and deadlock analysis

The behavioural type model has been introduced by Giachino and Laneve [2013, 2014] for detecting deadlocks in a concurrent object-oriented language; the decision algorithm for the circularity of behavioural types has been also defined by Giachino et al. [2014]. This technique improves the previous deadlock-freedom analysis of Kobayashi [2006] in a significative way, as demonstrated in by Giachino et al. [2014] and Kobayashi and Laneve [2017].

In the literature there exists a wide range of works that statically analyse

deadlock based on types or directly on behavioural types. Abadi et al. [2006], Boyapati et al. [2002], Flanagan and Qadeer [2003] and Vasconcelos et al. [2009], in their works, define a type system that computes a partial order of the locks in a program and a 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.

Likewise, it has been done by Visser et al. [2003], the tool Java PathFinder for every method is computed a tree of lock orders, then it searches for mismatches between such orderings. Instead, our technique results to be more flexible because, during the inference of the behavioural types, any ordering of locks is computed. A computation that may acquire two locks in different order at different moments results to be not correct with other technique, contrary to ours.

Other proposals, like the one of Flanagan et al. [2002], require that the program is annotated specifying loop invariants, pre and post conditions, in order to be able to detect deadlocks. As the annotations are explicitly provided by the programmer, these techniques are called partially automatic. On the contrary our technique, that is fully automatic, directly infers behavioural types from the code without any additional information.

A well-known deadlock analyser for pi-calculus developed by Igarashi and Kobayashi [2001] [1998, 2006, 2007] is TyPiCal. TyPiCal uses a technique that, without the limitation of imposing pre-defined order of channel names, is able to derive inter-channel dependency information. This technique is also able to deal with recursive behaviours and new channels creation. However, since TyPiCal is totally based on an inference system, recursive behaviours are not completely handled, in fact the analysis returns false positives when a networks with arbitrary numbers of nodes is recursively created.

Type-based deadlock analysis has also been studied by Puntigam and Peter [2001]. In this contribution, types define the states of objects and can express acceptability of messages. The exchange of messages modifies the state of the objects. In this context, a deadlock is avoided by setting an ordering on types. Unlike our approach, the one of Puntigam and Peter [2001] uses a deadlock prevention technique, rather than detection, and no inference system for types is provided. Another approach for deadlock prevention is defined by West et al. [2010]. This

technique is applied to SCOOP, a concurrent language based on Eiffel developed by Morandi, Bauer, and Meyer [2010]. In the work of West et al. [2010], as for the one of Flanagan et al. [2002], the programmer has to annotate methods with preconditions that identify the used processors[3] and the order in which processors have to take locks, while in our approach this information is automatically inferred.

Other different analysis technique are used to tackle the deadlock detection, for instance, model checking is often used to verify stateful distributed systems. Sirjani [2007] presents an actor language, Rebeca, which uses the inherent characteristics of actor model to introduce compositional verification, symmetry, abstraction, and partial order reduction techniques in order to limit the model to check. Ameur-Boulifa et al. [2017] provide a 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. An alternative model checking technique is proposed by Bouajjani and Emmi [2012] for multi-threaded asynchronous communication languages with futures (as ABS). This technique addresses infinite-state programs that admit thread creation but still they do not allow dynamic resource creation.

The impossibility to handle infinite thread creation or dynamic resource creation is common also to other techniques. Gkolfi et al. translate active objects into Petri-nets and model-check the generated net, in case of infinite state, they would still need an infinite set of colours for the tokens. For Carlsson and Millroth [1997] circular dependencies among processes are detected as erroneous configurations, but dynamic creation of names is not treated.

The problem of verifying deadlocks in infinite state models has been studied in other contributions. For example, de Boer et al. [2013] compare a number of unfolding algorithms for Petri Nets with techniques for safely cutting potentially infinite unfolding. Also in this work, dynamic resource creation is not addressed and it is not clear how to scale the technique.

A design pattern methodology which allows the development of data race-free and deadlock-free programs in CoJava has been presented by Kerfoot et al. [2009]. CoJava is a subset of Java for which in well typed programs are eliminated data race and data-based deadlock by a type-based techniques. For Kerfoot et al. [2009] the

[3]A processors is SCOOP is the analogue of activity in ASP or cogs in ABS

sharing of mutable data between threads is prevented by a typechecker, while and the use of ownership, which imposes a hierarchy on active objects, and promises to prevent deadlock from arising. Active objects unrelated through ownership must communicate through the use of promise objects that act as receptacles for return values. The active object waiting for a response may wait for a time period, if no result comes within this period a timeout event occurs. Instead, active objects related by ownership need not use the promise object approach. An owner may safely block waiting for responses from the objects it owns. Also in this case, contrary to our technique, the analysis is not fully automatic and needs additional information to be performed, these information must be provided by the programmer. In the work of Kerfoot et al. [2009] the correctness of the type-based approaches in removing data races have been demonstrated, no guarantee of deadlock freedom is provided by the system.

The approach that is the closest to our works is the one of Albert, Flores-Montoya, and Genaim [2012]. In this work the authors generate a finite graph of program points by integrating an effect and point-to analysis for computing aliases with an analysis returning points that may run in parallel. In the model presented by Albert et al. [2012], futures are passed (by-value) between methods only as parameters or return values, the possibility of storing futures in object field is treated as a possible extension and not formalised. Furthermore this aspect is not considered combined to the possibility of having infinite recursion. However, the work of Albert et al. [2012], as most of the works previously investigated, analyses finite abstraction of the computational models of the language. In our case, the behavioural type model associated to the program handles unbounded states.

Non static technique have also been used. In these techniques the main approach is to analyse only real scenarios. For instance, the analysis of deadlock proposed by Bensalem and Havelund [2006] tags the locked objects with the label identifying the threads acquiring the locks, allowing to deal with re-entrance problems in a very simple way. One of the main difference with our work is in the detection of the parallel execution. While our technique, detecting the parallel states at static time, obviously introduces some over-approximation, the one presented by Bensalem and Havelund [2006], beeing an runtime analysis, can tag each segment of the program, that is reached by the execution flow, specifying

exactly the order of lock acquisitions. It is worth to mention that Bensalem and Havelund [2006] also use an hybrid strategy which detects deadlocks that might occur in case of different scheduler choices. The drawback of the technique proposed by Bensalem and Havelund [2006], as for all the non static techniques, is that it is able to analyse exhaustively only programs that have a finite set of relevant inputs.

As Chapter 5 uses a dedicated effect analysis for active objects, we also analyse contributions that study effects or that integrate effects in the analysis of deadlocks. Regarding effect systems, up-to our knowledge, the first paper proposing effect systems for analysing data races of concurrent systems dates back to the late 80's [Lucassen and Gifford, 1988]. In the work of Lucassen and Gifford [1988] is presented a technique that uses a type and an effect system to statically infer types and effects of each expression in programs written in FX-87 [Hammel and Gifford, 1988]. The effect system permits concurrency analysis in the presence of first-class function values, and it is also able to mask effects on first-class, user-defined, heap-allocated data structures. 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.

Pun [2013], in her PhD. thesis, proposes an analysis which captures the lock interaction of processes using a behavioural type and effect system for a concurrent extension of ML. Similarly to our work, the the technique proposed by Pun [2013] is mostly divided in two steps: a type and effect inference algorithm, followed by an analysis to verify deadlock freedom. However, this analysis approximates infinite state space it imposes to define an upper bound on the number of locks, and a limit on the recursion depth for non-tail-recursive function calls. Thanks to these two restrictions the behaviour is over-approximated by arbitrary chaotic behaviour that can be exhaustively checked for deadlocks. However, this analysis approximates infinite behaviours with a chaotic behaviour that nondeterministically acquires and releases locks, thus becoming imprecise. The second approach statically analyses programs by reducing the problem of deadlock checking to data race checking. This analysis, as the previous one, is based on a type and effect system which formalises the data-flow of lock usages, over-approximating how often different locks are being held. In this approach additional variables are used to signal a

race at control points that potentially are involved in a deadlock. The limitations of the work of Pun [2013] come out from the fact that races, despite deadlocks, are binary global concurrency error, is solved by adding additional locks. These locks prevent that parts of the program, already considered in a deadlock cycle, give raise to a race, indicating falsely or prematurely a deadlock by a race.

Chapter 3

Ground-ASP a core language for ASP

Contents

In this chapter we present `ground-ASP` (`gASP`) a simple Active object model, that we target by our analysis. As `gASP` is a core language for ASP [Caromel et al., 2004] it presents some limitations compared to ASP, these limitations and more generally the differences between `gASP` and the full language ASP are discussed in Section 3.1. This chapter is composed by other four main section: Section 3.2 presents the syntax of the language and informally introduces the semantics; Section 3.3 analyses in detail the features of `gASP` through a comparison with ABS; Section 3.4 presents the operational semantics of `gASP` which is explained thanks

to a small example in Section 3.5. The chapter ends with a summary of what has been seen

3.1 A brief introduction of ground ASP (gASP)

In this section we present the features of `gASP`, detailing how the language associates objects to threads, the scheduling mechanism, how futures are defined, and the the approach used to synchronise on futures.

In `gASP` state encapsulation is guaranteed: an active object can only access to the state of its own fields during the processing of a method. Active objects can not have side effects on other active object fields. Contrary to ASP, `gASP` uses a uniform object model. All the objects created are consider as active, then each of them has an associated execution thread. Active objects communicate only through asynchronous method invocation. Unlike ASP where direct invocation on the same object are considered as synchronous method invocation, in `gASP` invocations on the same active object can only be asynchronous. The way to do asynchronous call on the same object, in ASP, is via a method invocation done on its own remote reference (which has been received somehow).

When, a method is called on a `gASP` object, a request is added on the set of requests of that object. We want underline that we are speaking about a set and not about a queue, because unlike ASP or most of the active object languages, requests are not served in any specific order. When an active object serves a new request, it picks any of the requests in the request set. We decided to stick to a more liberal organisation for ready processes, because FIFOs seem too constraining in a distributed system setting, where the dispatch of invocations is nondeterministic. Furthermore, in this way we are able to develop an analysis without restriction related to the scheduling order, indeed if our technique asserts that a program is deadlock free then it will also be deadlock free when a different policy will be applied to ready processes.

`gASP` uses a mono-threaded scheduling approach: as we have seen in Section 2.2.2 it specifies that an activity contains only a single thread that serves all the request without possible interleaving. The absence of interleaving makes `gASP` free from data races and it guarantees local determinism. As in ASP, global

determinism is not guaranteed and race condition can still be possible; while in ASP the non-determinism is restricted to the communication ordering, in `gASP` also request service is non deterministic.

Another relevant point to analyse is how `gASP` handles futures, especially future creation and synchronisation. The model `gASP` provides *first-class futures*: futures can be sent as parameter of method calls, returned from method invocations and stored in object fields without be necessarily synchronised.

In `gASP` everything related with futures tends to be as much transparent as possible. Futures are not explicitly typed, this allows the programmer not to make a distinction between futures and values. The absence of an explicit future type allows us to make totally transparent another aspect related with futures: the synchronisation. Unlike ABS, Akka, Encore, and other active object languages that provide explicit construct for synchronising future (i.e `get`), synchronisation in `gASP` is fully transparent. We call the synchronisation mechanism *wait-by-necessity*. 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.

Wait-by-necessity is not the only consequence of not having explicit future types, the other relevant consequences will be analysed in the next section through a comparison with an Active object model which provides explicit future types.

3.2 The gASP language

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` is defined in Figure 3.1. While the statement in `gASP` are standard it is worth to mention that the semicolon is used as statement separator and not as statement termination. Expressions with side effects include asynchronous method call $v.\mathtt{m}(\overline{v})$, where v is

$$
\begin{array}{rcll}
P & ::= & C\ \{\, s \,\} & \text{program} \\
C & ::= & D\ \overline{M} & \text{class} \\
M & ::= & T\ \texttt{m}\ (\overline{T\ y})\ \{\, s \,\} & \text{method definition} \\
D & ::= & T\ x \ \mid\ D\ \texttt{;}\ D & \text{variable declaration} \\
T & ::= & \texttt{Act} \ \mid\ \texttt{Int} & \text{type} \\
s & ::= & \texttt{skip} \ \mid\ x = z \ \mid\ \texttt{if}\ e\ \{\, s \,\}\ \texttt{else}\ \{\, s \,\} \ \mid\ s\ \texttt{;}\ s \ \mid\ \texttt{return}\ v & \text{statement} \\
z & ::= & e \ \mid\ v.\texttt{m}(\overline{v}) \ \mid\ \texttt{new Act}(\overline{v}) & \text{expression with side effects} \\
e & ::= & v \ \mid\ v \oplus v & \text{expression} \\
v & ::= & x \ \mid\ \textit{integer-values} & \text{atom}
\end{array}
$$

Figure 3.1 – The language gASP.

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. Expression with side effects z also include $\texttt{new Act}(\overline{v})$ that creates a new active object whose fields contain the values related to $\overline{v}$. A (pure) expression e may be an atom v (i.e., a variable or a value) 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.

Some of the key concepts of gASP, introduced in Section 3.1, comes out analysing the syntax. For instance the transparency of the future is found in the fact that variables can be typed only as Int or Act. It is also possible to see that every time an object is created, using new Act, an active object is created. The possibility to create only active objects and the fact that there is only one way to invoke methods implies that active objects can only communicate by asynchronous call. In gASP when a method is called ($x = v.\texttt{m}(\overline{v})$) a new future is created and stored in the variable targeted by the assignment, the corresponding request is added to the queue of pending requests of the active object on which the method is invoked.

More consideration comes out from the fact that in the syntax there is no statement related to future synchronisation. In fact gASP uses the wait-by-necessity mechanism and does not need a specific statements to synchronise on futures. The future synchronisation is hidden inside the evaluation of the expressions, this is

the reason why the evaluation of an expression can stop the execution of the method until all futures in the expression are resolved. In gASP when a method is finished the method result is associated to the future, indeed the method result can also be a future. At any moment if the result of the method call is available it can be replaced with the method result in all the variable that store this future. The update can be done at any moment since when the result of the method call is available to the moment in which it is needed to evaluate an expression. However a future is considered as resolved only when a value (an integer or an active object) is associated to the future, for this reason waiting until a future is resolved is not equivalent to waiting for the completion of the method associated to that future. As we will see in the following two sections the update mechanism strongly characterises the semantic of gASP.

3.3 Explicit future type and wait-by-necessity

In this section the main features of gASP will be explained more in detailed through a comparison with another active object language. We chose ABS as flagship for language with explicit future type and explicit synchronisation, for different reasons: it is a simple language with the aims of modelling and verification; it carries a strong formalisation; and ABS has been targeted by several analysis based on behavioural types. ABS, which has been presented in detail in Section 2.4.1, is an Active object model based on an object group model and cooperative scheduling, with explicit future type with of the form Fut<T> and an explicit synchronisation construct (get).

The feature of gASP on which we want to focus are: (i) implicit future type and (ii) wait-by-necessity synchronisation.

Let us start the comparison analysing the first of the two point. The implicit future type of gASP will be compared with the explicit parametric types of ABS.

Explicit and implicit future type

The typing approach is strongly related with expressiveness. For example the absence of explicit future types gives in the possibility to write recursive functions

a) gASP

```
1  Int fact(Int n, Int r){
2    if (n == 1) return r;
3    else { x = new Act();
4           r = r*n;
5           y = x.fact(n-1,r);
6           return y }}
7
8  //MAIN
9  { x = new Act();
10   y = x.fact(3,1);
11   z = y + 1 }
```

b) ABS

```
1  Int fact_nc(Int n){
2    Fut<Int> x ;
3    Int m ; Act z ;
4    if (n==1) { return 1 ; }
5    else { z = new cog Act();
6           x = z!fact_nc(n-1);
7           m = x.get;
8           return n*m; }}
9
10 //MAIN
11 { x = new cog Act();
12   y = x!fact(3);
13   z = y.get;
14   k = z + 1;
15 }
```

Figure 3.2 – Factorial in gASP and ABS.

which return futures.

To illustrate this point let us try to implement a method which is able to compute the factorial of n in both languages: gASP and ABS.

The implementations presented in Figure 3.2 are quite similar. In both cases for each invocation of the method `fact` a new active object is created and the factorial of $n-1$ is invoked on it. However, while in gASP the method directly returns the future of the invocation of fact(n-1), the ABS program has to return an integer. There are two reasons why the program written in ABS could not return directly the future contrarily to the gASP implementation: a problem of consistency of types for the two branches of the if, and a problem of type definition. Let us start discussing the first and simpler one. The gASP type system allows the programmer to write a program, as the one proposed in Figure 3.2.a, which returns a value in the branch true of the if (line 2, `return 1`) and a future of an integer in the branch false (line 6, `return y`, where `y` stores the future of the invocation done in line 5). The program written in this way can be typed because integer values and futures of integer are both typed as `Int`, which is the type defined in the signature of the method `fact`. On the contrary, a language with explicit future type, as ABS,

types an integer value as `Int` and a future of an integer as `Fut<Int>`, the two types would not fit the type defined in the signature.

The second problem is related to the question "which should be the type of a recursive call that returns a future in ABS?" To think about how to answer to this question, we should remember that, with explicit future types, a variable which stores the result of a method invocation has type `Fut<Int>` if the method returns an integer, and it has type `Fut<Fut<Int>` if the method returns a `Fut<Int>`. Considering this, we have that, to write the type of a recursive method that returns the future of the recursive invocation we have to know, a priori, the number of recursive calls done during the execution. For instance, in `x = fact(1)`, the variable `x` should have type `Fut<Int>`; in `x = fact(2)`, `x` should have type `Fut<Fut<int>>`; in `x = fact(3)`, `x` should have type `Fact<Fact<Fact<Int>>>` and so on. It is clear that if we want to compute the factorial of n, without knowing n a priori, it would not be possible define the type of the method.

Though, at first glance, this difference between the two languages may appear not so important, in the setting of distributed systems the behavioural differences between the two program presented in Figure 3.2 are significant. To understand better the behavioural differences we drive the discussion through the proposed example.

We should focus on lines 5 and 6 of the `gASP` program, and the lines 6-8 of the ABS program. As the `gASP` program, after invoking `fact(n-1)`, directly returns the result of the invocation, without performing any kind of synchronisation, there is no guarantee that the value has been actually computed. Then, when line 6 is executed, what is really returned is not the result of the invocation, but the future related to this result invocation. After the return statement, the method ends and the thread of the active object becomes available to serve other requests. In the `gASP` version, the only process which performs synchronisation is the one executing the main function, which is waiting for the result of all the recursive invocations of the method factorial.

The behaviour of the ABS program is totally different. As we can see at line 6 the method performs a synchronisation, then what is returned is the actual value of the method invocation just done. This synchronisation is mandatory, for the reasons explained above. The presence of this synchronisation implies that, in

a) gASP

```
1  Int fact(Int n, Int r){
2    if (n == 1) return r;
3    else { r = r*n;
4           n = n-1;
5           y = this.fact(n,r);
6           return y; }}
7
8  //MAIN
9  { x = new Act();
10   y = x.fact(3,1);
11   z = y + 1 }
```

b) ABS

```
1  Int fact_nc(Int n){
2    Fut<Int> x ;
3    Int m ; Act z ;
4    if (n==1) { return 1 ; }
5    else { x = this!fact_nc(n-1);
6           m = x.get;
7           return n*m; }}
8
9  //MAIN
10 { x = new cog Act();
11   y = x!fact(3);
12   z = y.get;
13   k = z + 1;
14 }
```

Figure 3.3 – Factorial with one active object in gASP and ABS.

our example, the active object computing `fact(3)` have to wait for the result of `fact(2)`, and similarly `fact(2)` waits for the result of `fact(1)`. Then, unlike the gASP version, the thread computing `fact(m)`, lock the active object thread until `fact(m-1)` has been computed.

As we have just seen the gASP program keeps the lock of only one active object until the result of `fact(n)` has been computed, while the ABS program keeps the lock of n active objects that will be released in order from `fact(1)` to `fact(n)`. The fact that many active objects are waiting for a result may increase considerably the chance that a deadlock occurs.

We present in Figure 3.3 a program similar to the one of Figure 3.2. This time, instead of creating a new active object at each recursive call and invoke `fact(n-1)` on it, all the recursive invocation are invoked on the same object (in line 5 of both programs, the method is invoked on `this`). In gASP this modification has no significant impact, because, as we have already said, the recursive execution of the method factorial do not lock the active object execution thread, then all the recursive calls can be served, and the factorial can be computed. However, the ABS version of the program leads to a deadlock. As the invocation of `fact(n)` locks the active thread until `fact(n-1)` is computed, `fact(n-1)` will never be

a) gASP

```
1  Int bar(Int x, Int y){
2    return x * y; }
3
4  Int foo(Int n, Int m){
5    x = this.bar(n,m);
6    return x; }
7
8  //MAIN
9  { x = new Act();
10   y = x.foo(3,1);
11   z = y + 1 }
```

b) ABS

```
1  Int bar(Int x, Int y){
2    return x * y; }
3
4  Fut<Int> foo(Int n, Int m){
5    Act x = this.bar(n,m);
6    return x; }
7
8  //MAIN
9  { Act x = new Act();
10   Fut<Fut<Int>> y = x.foo(3,1);
11   Fut<Int> w = y.get;
12   Int z = w.get;
13   k = z + 1; }
```

Figure 3.4 – Synchronisation example for gASP and ABS

computed, because it needs the execution thread locked by its caller. To be able to implement in ABS a not deadlocked factorial, which use only one active object, the programmer has to use cooperative scheduling and the possibility to explicitly unschedule the running process (the programmer have to use an `await` before the `get` of line 6).

We can easily conclude that, although with explicit future type and explicit synchronisation, it is still possible to write program with a behaviour similar to the one of gASP, the possibility to easily return future decreases the possibility to lead in a deadlock without requiring a strong expertise to the programmer.

Wait-by-necessity and explicit synchronisation

The second main difference between gASP and languages as ABS is the synchronisation on future. The wait-by-necessity approach used in gASP does not differ from the explicit approach only because it carries an high level of transparency, but it also leads to a totally different synchronisation semantic.

Let us discuss this difference through an example shown in Figure 3.4. In this example we implement two methods: `foo` and `bar`. The method `bar` takes two parameters as input and returns the product. The method `foo` only calls the

method `bar` and returns the result without synchronising on it. The main function creates an active object, then the method `foo` is invoked passing two integers, and then the method adds 1 to the result of `foo`. The main differences between the two programs are the type of the method `foo` and the type of the variable `y`, which have been explained before, and how the two languages handle the synchronisation on the result of `foo`. The `gASP` program at line 11 needs the value of `y` to evaluate the expression. As the method `foo` returns a future instead of a value (at line 6 the invocation of `bar` is returned without being synchronised), the execution of the main function is stoped at line 11 waiting the end of both methods `foo` and `bar`. The ABS program, instead, needs two instructions to get the value computed by `bar`. As we can see on line 11 of the ABS program, the first `get` instruction returns a future, this future is related to the invocation of `bar` returned by `foo`. Then the execution of the main program is stopped at line 11 until only the method `foo` ends. The second synchronisation, at line 12, which returns the value computed by `bar`, stops the execution of the main function until the method `bar` is finished. Only at this point the expression at line 13 can be computed. The difference of transparency that there exists between the two approaches is evident, but it also true that the transparency decrease the control that the programmer has defining synchronisation patterns. In `gASP`, as we have seen, the synchronisation is hidden in the evaluation of an expression and the synchronisation of the two methods `foo` and `bar` could not be done in two different steps like in ABS. Even though it is true that with explicit synchronisation the programmer has a greater control on the behaviour of the program, it is also true that if it does not come with a strong expertise, it could introduce several problems. In fact the possibility to add instructions between two `get` (i.e we can have some instruction between the line 11 and 12 of the ABS programs), like method invocation, could be really dangerous if it is not done very carefully.

The drawback of the transparent approach compared to the explicit one, is that it is more difficult to analyse statically. The difficulty lies in the fact that it is not trivial to keep track of all the methods that have to be synchronised when a variable is evaluated. While, in languages as ABS, each `get` expresses a single waiting condition with only one active object thread, a synchronisation in `gASP` stops the execution of the program until the evaluation of different methods,

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

Figure 3.5 – Runtime syntax of gASP.

probably running in different active objects, are finished (as in the example shown, where the main function was waiting for both `foo` and `bar`). The complexity in keeping track of the list of the methods to synchronise can be even higher in case of recursive functions, like the factorial presented in Figure 3.2, in which this list may be infinite. Chapter 4 will illustrate how to design a static analysis that is able to face this problem.

Now that the syntax of gASP has been defined and the main features have been explained informally, in the next section the formal semantics of gASP will be introduced.

3.4 Syntax and semantics of gASP

The semantics of gASP is defined in the following sections. The definition of deadlocked program according to the operational semantics will be defined in Section 3.4.2. The chapter will end with an example which will show the execution of a gASP program in terms of transitions between configurations.

3.4.1 Semantics of gASP

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 shown in Figure 3.5.

Configurations, denoted *cn*, are non empty sets of active objects and futures. The elements of a configuration are denoted by the juxtaposition *cn cn'* – therefore we identify configurations that are equal up-to associativity and commutativity. 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 $f(\cdot)$

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

Figure 3.6 – The evaluation function

represents a *future* which may be an actual value (called *future value*) or $\perp$ 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 in the operational semantics:

- $dom(\ell)$ return the domain of ℓ;
- $fields(\texttt{Act})$ is the list of fields of `Act`;
- $\ell[x \mapsto v]$ is the standard map update;
- the memory $a + \ell$ is defined as $(a + \ell)(x) \;=\; \begin{cases} \ell(x) & \text{if } x \in \mathrm{dom}(\ell) \\ a(x) & \text{if } x \in \mathrm{dom}(a) \end{cases}$
  we also let $(a + \ell)[x \mapsto w] = a' + \ell'$ be such that $a' = a$ and $l' = \ell[x \mapsto w]$, if $x \in \mathrm{dom}(\ell)$ or $x \notin \mathrm{dom}(l) \cup \mathrm{dom}(a)$, or $a' = a[x \mapsto w]$ and $\ell' = \ell$, if $x \in \mathrm{dom}(a)$;
- $[\![e]\!]_{a+\ell}$ returns the value of e by computing the expression and retrieving the value of the identifiers that is stored in $a + \ell$;
- $[\![\overline{e}]\!]_{a+\ell}$ returns the tuple of values of $\overline{e}$.
- $\mathrm{bind}(\alpha, m, \overline{w}, f)$, where the method m is defined by $T\ \mathtt{m}(\overline{T\ x})\ \{\, s \,\}$, returns the following process $\{\, [\, \texttt{destiny} \mapsto f, \texttt{this} \mapsto \alpha, \overline{x} \mapsto \overline{w}\,] \mid s \,\}$. We observe that the special field `destiny` in the local memory ℓ records the name of the future corresponding to the method invocation.

The operational semantics of `gASP` is defined by a transition relation between configurations and it is shown in Figure 3.7. Most of the rules are standard and the most relevant are described in detail below.

SERVE : it schedules a new process to be executed. As it has been already stated in `gASP` the processes ready to be executed are organised into a set and SERVE picks a ready process.

$$\textsc{Serve}\quad \alpha(a, \varnothing, \overline{q} \cup \{p\}) \to \alpha(a, p, \overline{q}) \qquad \textsc{Context}\quad \frac{cn \to cn'}{cn\ cn'' \to cn'\ cn''}$$

$$\textsc{Return}\quad \frac{[\![v]\!]_{a+\ell} = w \qquad \ell(\texttt{destiny}) = f}{\alpha(a, \{\ell \mid \texttt{return}\ v\}, \overline{q})\ f(\bot) \to \alpha(a, \varnothing, \overline{q})\ f(w)} \qquad \textsc{Update}\quad \frac{(a+\ell)(x) = f \qquad (a+\ell)[x \mapsto w] = a' + \ell'}{\alpha(a, \{\ell \mid s\}, \overline{q})\ f(w) \to \alpha(a', \{\ell' \mid s\}, \overline{q})\ f(w)}$$

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

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

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

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

Figure 3.7 – Operational semantics of gASP.

Update : it performs the update of a future when the corresponding value has been computed. The future value w may be an integer, an active object, or a future name. It is important to notice that this rule is not triggered by a specific statement, then it can be applied in any moment when the future value of f has been computed.

Assign : this rule shows the gASP semantic for the assignment of a value or a name (active object or future name) to a local variable or a field (*cf.* definition of $a+\ell$). The relevant point here is the evaluation function $[\![e]\!]_{a+\ell} = w$ because it may require synchronisations. In fact, according to the definition of $[\![e]\!]_{a+\ell}$

in Figure 3.6, 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*. However, if e is a future name f or a variable which stores a future name, this future is assigned without any synchronisation. Then an alias has been created.

NEW : this rule presents the creation of a new active object. The created object has a fresh name as identifier, an empty set of process to be executed and there is no process running on it. The relevant point here is that the evaluation function $[\![\overline{v}]\!]_{a+\ell} = \overline{w}$ never implies a synchronisation. In case one of the parameter is a future, it is passed as it is and it is not synchronised. It also important to notice that all the field of the object created have to be initialised through parameters: $\overline{v}$ and $\overline{y}$ have the same cardinality.

INVK : it shows the asynchronous method call done on a different active object. $(w = [\![\overline{v}]\!]_{a+\ell})$. We can notice that two main actions are performed during the application of this rule. The former is the creation of a future element $f(\bot)$ which is added to the configuration. The future name f is fresh and the future value of f is not defined because obviously the method is not already computed. The latter is the addition of the process p', which is obtained by the function $\text{bind}(\beta, m, \overline{w}, f)$, to the set of processes to be executed of the active object targeted by the method invocation. It is important to notice that the evaluation of $[\![v]\!]_{a+\ell}$ must return an object name. If, instead, it returns a future then the rule cannot be applied and a synchronisation occurs. On the contrary, the evaluation function $[\![\overline{v}]\!]_{a+\ell} = \overline{w}$ never implies a synchronisation, then future can be passed as parameter.

The rule INVK-SELF is a particular case of INVK in which the caller and the callee turn out to be the same active object. Then, the two rules are similar except that the process p' is added to the set of waiting processes of the active object that is performing the method call.

IF-TRUE / IF-FALSE : this rules present the semantic of `gASP` for the if statement. Though the rules are trivial, we want to underline that the evaluation of the condition $([\![e]\!]_{a+\ell})$ must result in a integer which may trigger a synchronisation.

Note that this semantics naturally ensures the strong encapsulation featured by

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:

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

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

3.4.2 Deadlocks and queues with deterministic effects

As we have previously seen, in gASP, when computing an expression, if one of the operand 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 $\ell(\texttt{destiny}) = f$.

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.

Definition 2 (Deadlock-free program)**.** *A program is deadlock-free if, denoting cn its initial configuration, for every cn′ s.t.* $cn \rightarrow^* cn'$, *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 to demonstrate the correctness of the type system.

Definition 3. *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 3.4.1. *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. Below we formalise the notion of programs with *queue with non deterministic effects*, which are 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.

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

Definition 4 (**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 $x^{\mathtt{w}} \in q_j$.*

Definition 5. *A configuration cn has* deterministic effects *if every active object of this configuration has a* queue with deterministic effects.

Definition 6. *A* gASP *program* has deterministic effects *if $cn \rightarrow^* cn'$, cn'* has deterministic effects*, where cn is the initial configuration for this program.*

3.5 A complete example: dining philosophers in gASP

In this section we present a complete example detailing the execution of a program and showing how a deadlocked configuration can be reached. The program presented in Listing 3.1 and 3.2 implements the dining philosophers problem. For the sake of simplicity we present a version with only two philosophers in order to better focus on the key steps.

To make the example easier to follow, it is better to have a clear idea about which active objects refer to *forks* and which refer to *philosophers*. We have implemented, then, the diner philosophers problem in a version of `gASP` which allows the definition of more than one class. As we said previously, extending the language in order to have several classes is not problematic.

```
1  class Philosopher{
2    Int behave(Fork fR, Fork fL)
3    { fut = fR.grab(fL);
4      aux = fut + 0;
5      c = this.behave(fR, fL) }
6  }
7
8  class Fork{
9    Int grab(Fork fL)
10   { fut = fL.grab_second();
11     aux = fut + 0;
12     return aux }
13
14   Int grab_second() { return 0 }
15 }
```

Listing 3.1 – Dining philosophers in `gASP` - Philosopher and Fork

```
16 // MAIN //
17 { fork1 = new Fork() ;
18   fork2 = new Fork() ;
19   p1 = new Philosopher();
20   p2 = new Philosopher();
21   fut1 = p1.behave(fork1, fork2);
```

```
22    fut2 = p2.behave(fork2, fork1) }
```

Listing 3.2 – Dining philosophers in gASP - Main function

In the following we detail the key steps of the evaluation of the program. Considering that the initial configuration is:

$$main(\varnothing, \{\, [\texttt{destiny} \mapsto f_{main}, \texttt{this} \mapsto main\,] \mid \texttt{fork1} = \texttt{newFork}()\ ;\ \texttt{fork2} = \texttt{newFork}()\ ;\ \cdots\}, \varnothing)$$

let us start detailing the execution of the lines of code between 17 and 20

```
17  fork1 = new Fork() ;
18  fork2 = new Fork() ;
19  p1 = new Philosopher();
20  p2 = new Philosopher();
```

The execution of this code requires the application of the rules New and Assign four times. Applying these rules, four active objects are created: the active objects γ and δ represent the two philosophers, whereas α and β represent the two forks. The configuration becomes:

$$main(\varnothing, \{\, [\ \texttt{destiny} \mapsto f_{main},\ \texttt{this} \mapsto main,\ \texttt{fork1} \mapsto \alpha,\ \texttt{fork2} \mapsto \beta,\ \texttt{p1} \mapsto \gamma,\ \texttt{p2} \mapsto \delta\ ] \mid \cdots\}, \varnothing)$$
$$\alpha(\varnothing,\varnothing,\varnothing) \quad \beta(\varnothing,\varnothing,\varnothing) \quad \gamma(\varnothing,\varnothing,\varnothing) \quad \delta(\varnothing,\varnothing,\varnothing)$$

In lines 21 and 22 the method `behave` is invoked twice.

```
21  fut1 = p1.behave(fork1, fork2);
22  fut2 = p2.behave(fork2, fork1);
```

This method encodes the behaviour of a philosopher that wants to grab the two forks and then start eating. The two invocations of `behave` is performed respectively on γ and δ. Each method invocation requires the application of the rule Invk. As defined in Figure 3.7 every time the rule Invk is applied a new future created. The future elements $f_0(\bot)$ and $f_1(\bot)$ are added to the configuration. Through the application of the rule Assign, f_0 and f_1 are stored in `fut1` and `fut2` respectively. The two method invocations are immediately served (rule Serve), this is possible because both active objects γ and δ have no running process. The main function can terminate, but the because the two invocations of `behave` are running the program is not considered as ended.

$main(\varnothing, \varnothing, \varnothing)$

$\gamma(\varnothing, \{\ [\texttt{destiny} \mapsto f_0,\ \texttt{this} \mapsto \gamma,\ \texttt{fR} \mapsto \alpha,\ \texttt{fL} \mapsto \beta\ ] \mid \texttt{body} - \texttt{of} - \texttt{behave}\ \}, \varnothing)$

$\delta(\varnothing, \{\ [\texttt{destiny} \mapsto f_1,\ \texttt{this} \mapsto \delta,\ \texttt{fR} \mapsto \beta,\ \texttt{fL} \mapsto \alpha\ ] \mid \texttt{body} - \texttt{of} - \texttt{behave}\ \}, \varnothing)$

$\alpha(\varnothing, \varnothing, \varnothing) \qquad \beta(\varnothing, \varnothing, \varnothing) \qquad f_0(\bot) \qquad f_1(\bot)$

The execution of the two invocations of `behave` can run in parallel in the two active objects γ and δ.

```
2  Int behave(Fork fR, Fork fL)
3  { fut = fR.grab(fL);
4    aux = fut + 0;
5    c = this.behave(fR, fL) }
```

This method defines how the two philosophers grab the fork on their right[1]. The fork represented by the active object α has been grabbed by the philosopher represented by γ and the fork β has been grabbed by δ. As before, the requests related to the two methods `grab` can immediately be served because the active objects α and β have no running process. The expression presented in line 4, which is a way to enforce synchronisation, can not be computed in both active object because the its evaluation requires the result of the method `grab`. The process running in active object γ and δ are locked until future f_2 and f_3 are resolved. Applying the rules Invk, Assign, and Serve for both active objects, we reach the following configuration:

$\gamma(\varnothing, \{\ [\texttt{destiny} \mapsto f_0,\ \texttt{this} \mapsto \gamma,\ \texttt{fR} \mapsto \alpha,\ \texttt{fL} \mapsto \beta,\ \texttt{fut} \mapsto f_2\ ] \mid \texttt{aux} = \texttt{fut} + 0\ ;\ \cdots\}, \varnothing) \quad f_0(\bot)$

$\delta(\varnothing, \{\ [\texttt{destiny} \mapsto f_1,\ \texttt{this} \mapsto \delta,\ \texttt{fR} \mapsto \beta,\ \texttt{fL} \mapsto \alpha,\ \texttt{fut} \mapsto f_3\ ] \mid \texttt{aux} = \texttt{fut} + 0\ ;\ \cdots\}, \varnothing) \quad f_1(\bot)$

$\alpha(\varnothing, \{\ [\ \texttt{destiny} \mapsto f_2,\ \texttt{this} \mapsto \alpha,\ \texttt{fL} \mapsto \beta\ ] \mid \texttt{body} - \texttt{of} - \texttt{grab}\ \}, \varnothing) \qquad f_2(\bot)$

$\beta(\varnothing, \{\ [\ \texttt{destiny} \mapsto f_3,\ \texttt{this} \mapsto \beta,\ \texttt{fL} \mapsto \alpha\ ] \mid \texttt{body} - \texttt{of} - \texttt{grab}\ \}, \varnothing) \qquad f_3(\bot)$

As we can see in the lines of code from 9 to 12, the method `grab` defines how philosophers grab the fork on their left.

```
9  Int grab(Fork fL)
10 { fut = fL.grab_second();
11   aux = fut + 0;
12   return aux }
```

[1]We detail here the execution that leads to a deadlock, of course another scheduling is also possible.

At this point the active object α invokes the method `grab_second` on β and β does the same on α. Both requests are put in the queue of the corresponding active object, but cannot be served because both active objects α and β have already a running process. We reach then the following deadlocked configuration:

$\gamma(\varnothing, \{\, [\texttt{destiny} \mapsto f_0,\ \texttt{this} \mapsto \gamma,\ \texttt{fR} \mapsto \alpha,\ \texttt{fL} \mapsto \beta,\ \texttt{fut} \mapsto f_2\,] \mid \texttt{aux} = \texttt{fut} + \texttt{0 ;}\ \cdots\}, \varnothing) \qquad f_0(\bot)$

$\delta(\varnothing, \{\, [\texttt{destiny} \mapsto f_1,\ \texttt{this} \mapsto \delta,\ \texttt{fR} \mapsto \beta,\ \texttt{fL} \mapsto \alpha,\ \texttt{fut} \mapsto f_3\,] \mid \texttt{aux} = \texttt{fut} + \texttt{0 ;}\ \cdots\}, \varnothing) \qquad f_1(\bot)$

$\alpha(\varnothing, \{\, [\,\texttt{destiny} \mapsto f_2,\ \texttt{this} \mapsto \alpha,\ \texttt{fL} \mapsto \beta,\ \texttt{fut} \mapsto f_4\,] \mid \texttt{aux} = \texttt{fut} + \texttt{0 ;}\ \cdots\}, \{\, [\texttt{destiny} \mapsto\ f_5] \mid$
$\texttt{body} - \texttt{of} - \texttt{grab\_second}\}) \qquad f_2(\bot) \qquad f_5(\bot)$

$\beta(\varnothing, \{\, [\,\texttt{destiny} \mapsto f_3,\ \texttt{this} \mapsto \beta,\ \texttt{fL} \mapsto \alpha,\ \texttt{fut} \mapsto f_5\,] \mid \texttt{aux} = \texttt{fut} + \texttt{0 ;}\ \cdots\}, \{\, [\texttt{destiny} \mapsto\ f_4] \mid$
$\texttt{body} - \texttt{of} - \texttt{grab\_second}\}) \qquad f_3(\bot) \qquad f_4(\bot)$

It is trivial to see that no other rule can be applied, the computation is deadlocked because the active objects γ and δ wait until f_2 and f_3, respectively, are not resolved. The circular dependency that cause the deadlock concerns α and β. The active object α and β are locked until the future f_4 and f_5, respectively, are resolved, but for resolving these two futures we need the termination of the process in α and β. As we can notice this last configuration reached reflect exactly the definition of deadlocked configuration that we state before (see Definition 1).

The execution presented is only one of the possible execution of the system, in which we show how the deadlock can be reached with the minimum number of steps. There are also other possible executions that allow the philosophers to eat an think an unpredictable amount of times, before leading to a deadlock. This unpredictable number of safe interleaving between the two philosophers can be so high that the deadlock could never be reached. This is the reason why dynamic analysis may classify programs as deadlock free though they still can potentially lead to deadlocked configurations.

3.6 Conclusions

In this chapter we presented an Active object model called `ground-ASP`, which is a core language for ASP [Caromel et al., 2004]. As stated in Section 3.1, `gASP` is based on uniform object model and uses a mono-thread scheduling approach. Section 3.3 points out the main features of `gASP`, which are: the absence of explicit

type for futures and the synchronisation mechanism called *wait-by-necessity*. This section also brings out the problems related to the static analysis of a model with those features. Those problems can be summarised in 3 main points:

Unbounded set of dependencies. The possibility of returning future mixed with the absence of an explicit future type brings to a possible unbounded nesting of futures. As we have seen in the example of the factorial function, presented in Figure 3.2, the main function was the only responsible for the synchronisation of all the recursive calls. Then the unique synchronisation performed by the main function should be associated not to a single waiting dependencies, but to a set of waiting dependencies, in order to state that active object executing the main is waiting for the result of all the recursive function that running in different active objects. As will be presented more in detail in the next chapter, the problem of generating statically this set is that we do not know a priori the number of the dependencies that we have to generate (i.e. for the factorial problem, this number depends on a parameter) and in case of unbounded recursive calls the number of dependencies can be unbounded.

Tracing of future. Even though `gASP` does not have explicit future types, it still creates a future when a method in invoked. These futures can be passed as parameters and stored in object field. The tracing of these futures is a relevant aspect in the analysis of synchronisation patterns.

Nondeterminism. Since `gASP` is stateful, it is possible to store futures in object fields and pass these objects 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.

In this chapter we have also formally defined some important concepts, which will be used in the rest of the thesis, such as: deadlocked configuration for `gASP`, deadlock-freedom programs, and the concepts of queue and program with deterministic effects.

In the next chapter we will present a static analysis technique that is able to consider all the possible executions of a program, and to identify as potentially deadlocked all the programs that have, at least, one possible execution that leads

to a deadlocked configuration.

Chapter 4

Behavioural type analysis

Contents

In this chapter we present a static analysis of synchronisation patterns for an Active object model with *wait-by-necessity* and implicit future type. The chapter presents our first work, in which we focus on how to trace synchronisation patterns in a language in which futures can be transparently passed around as parameters or returned from methods. In the analysis proposed in this chapter, we also want to solve the first of the three main problems discussed in Section 3.6, which refers to the identification of the set of dependencies, potentially unbounded, that can be triggered by a single synchronisation.

In order to focus on these aspects we restrict the analysis only on programs in which active objects are not stateful. The discussion about the analysis of `gASP` programs with stateful active object will be delayed to Chapter 5 that will present our second main work.

To make the type system and the behavioural type analysis not too heavy, other small limitations have been imposed to `gASP` programs. These restrictions are presented in section 4.1.

The deadlock detection technique we are going to present uses abstract descriptions, called *behavioural types*. The behavioural type, their syntax and the the rest of the notations that will be used are presented in Section 4.2. Those behavioural types are associated to programs by a type system, which has been defined in Section 4.3. The purpose of the type system is to collect dependencies between active objects and between futures and active objects. At each point of the program, the behavioural type gathers information on local synchronisations and on active objects potentially running in parallel. We perform such an analysis for each method body, gathering the behavioural information at each point of the program.

The chapter continues with the presentation of the behavioural type analysis, Section 4.4, which takes in input the behavioural type of a program and identifies the presence of deadlocks.

To help the the reader, typing process and the behavioural type analysis will come with a small example that shows these two processes applied to the program of Figure 3.2.a.

To guarantee the soundness of our behavioural type system, in Section 4.5 we informally explain how the soundness of the type system is proven, while the full proof is delayed to Appendix A.1.

This chapter will mostly focus on the behavioural type system and the behavioural type analysis that are our main contribution. However, the behavioural type analysis presented may not terminate because of the recursive nature of the behavioural types. Then to make our analysis complete, we need a fixed-point technique that is able to find circularities in a finite time. In Section 4.6 will be briefly present an adaptation of the fixpoint analysis that was designed by Giachino, Kobayashi, and Laneve [2014].

The chapter ends with an example in which the entire analysis is applied to the dining philosophers problem, already presented in section 3.5, and a final section that summarise what has been presented in the chapter and explaining how some of the limitations imposed can be removed.

4.1 Restriction

We focus here on a sublanguage of gASP differing from the full language in the following aspects: fields only contain synchronised integers, i.e., (*i*) neither futures (*ii*) nor active objects; and (*iii*) all futures created in a method must be either returned or synchronised.

Regarding (*i*) and (*ii*), they allow us to avoid any analysis of the content of fields and keep the types for active objects simpler.

Regarding item (*iii*), this enforces that, *once the future for a method has been synchronised, all the futures directly or indirectly created by that method are synchronised too.* Notice that if a future is returned by the current method, then it will be synchronised by whomever will synchronise on the current method result. This prevents from having computation running in parallel without any mean to synchronise on it.

These restrictions simplify the presentation of the analysis and allow us to focus on the original aspects related to transparent futures and to the type system itself. They are enforced by the type system of Section 4.3. We discuss how to relax these restrictions in Section 4.9.

4.2 Notations

A *behavioural type program* is a pair $\big(\mathcal{L}, \Theta \cdot \mathsf{L}\big)$, where $\mathcal{L}$ is a *finite set* of *method behaviours* $\mathtt{m}(\alpha, \overline{\mathtt{x}}, X) = (\nu\,\overline{\varphi})(\Theta_{\mathtt{m}} \cdot \mathsf{L}_{\mathtt{m}})$, with $\alpha, \overline{\mathtt{x}}, X$ being the *formal parameters* of $\mathtt{m}$, $\Theta_{\mathtt{m}}$ the *future environment* of $\mathtt{m}$, $\mathsf{L}_{\mathtt{m}}$ the behavioural type of the *body* of $\mathtt{m}$, and Θ and L are the *main future environment* and the *main behavioural type*, respectively. The binder $(\nu\overline{\varphi})$ binds the occurrences of $\overline{\varphi}$ in $\Theta_{\mathtt{m}}$ and $\mathsf{L}_{\mathtt{m}}$, with φ ranging over future or active object names.

$$
\begin{array}{rcll}
\mathbb{r} & ::= & \square \mid \alpha & \text{basic type} \\
\mathbb{x} & ::= & \mathbb{r} \mid \mathbb{r}_f & \text{extended type} \\
\kappa & ::= & \star \mid \alpha \mid X & \text{synchronisers} \\
\mathsf{L} & ::= & \mathtt{0} \mid (\kappa, \alpha) \mid f_\kappa \mid \mathsf{L} + \mathsf{L} \mid \mathsf{L} \,\&\, \mathsf{L} & \text{behavioural type} \\
\lambda X.\mathtt{m}(\alpha, \overline{\mathbb{x}}, X)^{[\checkmark]} & ::= & \lambda X.\mathtt{m}(\alpha, \overline{\mathbb{x}}, X) \mid \lambda X.\mathtt{m}(\alpha, \overline{\mathbb{x}}, X)^{\checkmark} & \text{future behavior} \\
 & \mid & \lambda X.\mathtt{m}(\alpha, \overline{\mathbb{x}}, X)^{\rightarrow} &
\end{array}
$$

Figure 4.1 – Syntax of behavioural types.

A *future environment* Θ maps future names to future behaviours (without synchronisation information) $\lambda X.\mathtt{m}(\alpha, \overline{\mathbb{x}}, X)$. In the method behaviour, the formal parameter α corresponds to the identity of the object on which the method is called (the `this`), while X, called *handle*, is a place-holder for the active object that will synchronise with the method. In practice several active objects can synchronise with the same future, but only one at a time; X will thus be instantiated by a single active object at each point of the analysis. $\overline{\mathbb{x}}$ are the types of the method parameters.

The syntax of behavioural types L is defined in Figure 4.1. The basic types $\mathbb{r}$ are used for values: they may be either $\square$, to model integers, or any active object name α. The extended type $\mathbb{x}$ is the type of variables, and it may be a value type $\mathbb{r}$ or a *not-yet-synchronised type* $\mathbb{r}_f$ (in order to retrieve the value $\mathbb{r}$ it is necessary to synchronise the future f). The behavioural type $\mathtt{0}$ enforces no dependency, (κ, α) enforces the dependency between κ and α meaning that, if κ is instantiated by an active object β, then β will need α to be available in order to proceed its execution. The term f_κ may represent different behaviours depending on the value of κ: $f_\star$ represents an unsynchronised future f, which is a pointer in the future environment to the corresponding method invocation; f_α represents the synchronisation of the active object α with the future f; f_X represents the return of a future f by the method associated to the handler X. The type $\mathsf{L} \,\&\, \mathsf{L}'$ is the *parallel composition* of L and of L': it is the behaviour of two methods running in parallel and not necessarily synchronised. The sum $\mathsf{L} + \mathsf{L}'$ composes the dependencies of L and L'

independently: it is the composition of two behaviours that cannot occur at the same time, either because one occurs before the other or because they are exclusive. The behaviour of a future (stored in the environment) is tagged ✓ if the future is synchronised and → if the future is returned by the return statement; else it has no tag. Whenever parentheses are omitted, the operation "&" has precedence over "+". We will shorten $\mathsf{L}_1 \,\&\, \cdots \,\&\, \mathsf{L}_n$ into $\&_{i\in\{1..n\}}\, \mathsf{L}_i$. In the syntax of L, the operations "&" and "+" are associative, commutative with $\mathtt{0}$ being the identity on &, and behavioural types are equal up-to alpha renaming of bound names.

The judgments of the type system have a typing context Γ mapping variables to extended types $\mathbb{x}$, future names to future behaviour $\lambda X.\mathtt{m}(\alpha, \overline{\mathbb{x}}, X)^{[\checkmark]}$, and method names to their signatures of the form $(\alpha, \overline{\mathbb{x}}, X) \to \mathbb{r}$, where $\alpha, \overline{\mathbb{x}}, X$ are the formal parameters of the method behaviour and $\mathbb{r}$ is the type of the returned value, respectively. Judgments have the following form:

- $\Gamma \vdash \mathtt{m} : (\alpha, \overline{\mathbb{x}}) \to \mathbb{r}$ for instantiating the method signature of $\mathtt{m}$.
- $\Gamma \vdash v : \mathbb{x}$ for typing variables and values.
- $\Gamma \vdash_S z : \mathbb{x}, \mathsf{L} \triangleright \Gamma'$ for typing expressions with side effects, where S is the set of parameters of the body of the method currently typed, L is the behavioural type of the expression, and Γ' is the environment obtained by updating Γ to reflect possible future creations.
- $\Gamma \vdash_S s : \mathsf{L} \triangleright \Gamma'$ for typing statements, where S and L are as before, and Γ' is obtained by updating Γ to reflect possible variable assignment.

Since Γ is a function, we use the standard predicates $x \in \mathrm{dom}(\Gamma)$ or $x \notin \mathrm{dom}(\Gamma)$. We define some additional auxiliary functions on Γ in Figure 4.2 and 4.3.

Equation 4.1 defines the update of the typing environment Γ, modifying the type associated to the variable x. The equation 4.2 updates Γ so that every variable that was previously mapped to $\mathbb{x}$ is now mapped to $\mathbb{x}'$.

The following functions on Γ are also used: $\Gamma(f)^{\checkmark}$ (Eq. 4.3) corresponds to $\Gamma(f)$ check-marked; $\Gamma(f)^{\rightarrow}$ is defined similarly to $\Gamma(f)^{\checkmark}$; $\Gamma(f)^{\times}$ (Eq. 4.4) corresponds to $\Gamma(f)$ such that any kind of marks has been removed; the restriction of Γ to a set S of names (Eq. 4.5) is noted $\Gamma|_S$; and the difference operation is defined in equation 4.6.

The definition of the sum and the merge of two environments are defined in the equation 4.7 and 4.8. The former equation simply returns an environment with is

$$\Gamma[x \mapsto \mathbb{z}](y) = \begin{cases} \mathbb{z} & \text{if } y = x \\ \Gamma(y) & \text{otherwise} \end{cases} \tag{4.1}$$

$$\Gamma[x \mapsto \mathbb{x}']^{\Gamma(x)=\mathbb{x}} \stackrel{def}{=} \Gamma[x_1 \mapsto \mathbb{x}] \cdots [x_n \mapsto \mathbb{x}] \text{ if } \{y \,|\, \Gamma(y) = \mathbb{x}\} = \{x_1, \cdots, x_n\} \tag{4.2}$$

$$\Gamma(f)^{\checkmark} = \begin{cases} \lambda X.\mathtt{m}(\alpha, \overline{\mathbb{x}}, X)^{\checkmark} & \text{if } \Gamma(f) = \lambda X.\mathtt{m}(\alpha, \overline{\mathbb{x}}, X)^{[\checkmark]} \\ \textit{undefined} & \text{if } f \notin \mathrm{dom}(\Gamma) \end{cases} \tag{4.3}$$

$$\Gamma(f)^{\times} = \begin{cases} \lambda X.\mathtt{m}(\alpha, \overline{\mathbb{x}}, X) & \text{if } \Gamma(f) = \lambda X.\mathtt{m}(\alpha, \overline{\mathbb{x}}, X)^{[\checkmark]} \\ \textit{undefined} & \text{if } f \notin \mathrm{dom}(\Gamma) \end{cases} \tag{4.4}$$

$$\Gamma|_S(x) = \begin{cases} \Gamma(x) & \text{if } x \in S \\ \textit{undefined} & \text{otherwise} \end{cases} \tag{4.5}$$

$$(\Gamma \setminus x)(y) = \begin{cases} \Gamma(y) & \text{if } x \neq y \\ \textit{undefined} & \text{if } x = y \end{cases} \tag{4.6}$$

$$(\Gamma + \Gamma')(x) = \begin{cases} \Gamma(x) & \text{if } x \in \mathit{dom}(\Gamma) \\ \Gamma'(x) & \text{if } x \in \mathit{dom}(\Gamma') \end{cases} \tag{4.7}$$

Note: $\Gamma + \Gamma'$ is defined only if $\forall\, x \in \mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Gamma').\, \Gamma(x) = \Gamma'(x)$

$$\mathtt{Merge}(\Gamma_1, \Gamma_2)(x) = \begin{cases} \mathbb{r}_f & \text{if } \Gamma_i(x) = \mathbb{r}_f \wedge \Gamma_j(x) = \mathbb{r}, \\ & i, j \in \{1, 2\} \\ \mathbb{x} & \text{if } \Gamma_1(x) = \Gamma_2(x) = \mathbb{x} \\ \textit{undefined} & \text{otherwise} \end{cases} \tag{4.8}$$

Figure 4.2 – Auxiliary definitions

$$\texttt{Fut}(\Gamma) \stackrel{def}{=} \{f \mid f \in \mathrm{dom}(\Gamma)\} \tag{4.9}$$

$$\texttt{AFut}(\Gamma) \stackrel{def}{=} \{f \in \texttt{Fut}(\Gamma) \mid \Gamma(f) = \Gamma(f)^{\times}\} \tag{4.10}$$

$$\mathit{unsync}(\Gamma) \stackrel{def}{=} \mathop{\&}_{f \in \texttt{AFut}(\Gamma)} f_{\star}, \tag{4.11}$$

Figure 4.3 – Auxiliary definitions

the union of the two environment, but this operation is defined if and only if all the element that the two environment have in common ($\forall\, x \in \mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Gamma')$) are mapped to the same element ($\Gamma(x) = \Gamma(x')$). The later equation, defining the merge, returns an environment in which all the variable mapped by the two environment on the exactly same type are still mapped on that type; the variables mapped to the same type that are synchronised in one environment and not in the other are considered as not synchronised; and all the other cases are not defined.

Figure 4.3 presents the auxiliary functions: $\texttt{Fut}(\Gamma)$ collects all the futures stored in Γ, $\texttt{AFut}(\Gamma)$ collects all the futures that are not tagged with a $\checkmark$ or $\rightarrow$, i.e. not-yet-synchronised futures, and $\mathit{unsync}(\Gamma)$ performs the parallel composition of the behavioural types of not-yet-synchronised method invocations, it collects the unsynchronised future of all the methods running in parallel.

4.3 Typing Rules

The typing rules are presented in Figure 4.4, 4.5, and 4.6. The most significant ones are discussed below. In general, a statement has a behaviour that is a sum of behaviours. Each term of the sum is a parallel composition of synchronisation dependencies and unsynchronised behaviours. We propagate this way the set of methods running in parallel as a set of not-yet-synchronised futures all along the type analysis (see the role played by $\mathit{unsync}(\Gamma')$ in rules (T-Sync), (T-Invk), (T-Return)). The statements that create no synchronisation at all (i.e. that do not access a future, nor call a method, nor return from a method) have behaviour $\texttt{0}$ and their unsynchronised behaviour is the same as the one of the previous statement.

expressions and addresses
- judgements used: $\Gamma \vdash v : \mathbb{x}$ and $\Gamma \vdash \mathtt{m} : (\alpha, \overline{\mathbb{x}}, X) \to \mathbb{r}$

$$\frac{\Gamma(x) = \mathbb{x}}{\Gamma \vdash x : \mathbb{x}}\ (\text{T-Var}) \qquad \frac{v \quad \textit{integer-value} \text{ or } \mathtt{null}}{\Gamma \vdash v : \square}\ (\text{T-Val})$$

$$\frac{\Gamma(\mathtt{m}) = (\alpha, \overline{\mathbb{x}}, X) \to \mathbb{r} \qquad \sigma \text{ renaming} \qquad \sigma(\square) = \square \qquad \mathbb{r} \notin \{\square, \alpha, \overline{\mathbb{x}}\} \implies \sigma(\mathbb{r}) \text{ fresh}}{\Gamma \vdash \mathtt{m} : (\sigma(\alpha), \sigma(\overline{\mathbb{x}}), \sigma(X)) \to \sigma(\mathbb{r})}\ (\text{T-Method-Sign})$$

expressions with side effects
- judgement used: $\Gamma \vdash_S e : \mathbb{x}, \mathsf{L} \triangleright \Gamma'$

$$\frac{\Gamma \vdash v : \mathbb{r}_f \qquad \Gamma(\mathtt{this}) = \alpha \qquad \Gamma' = (\Gamma[y \mapsto \mathbb{r}]^{\Gamma(y)=\mathbb{r}_f})[f \mapsto \Gamma(f)^{\checkmark}]}{\Gamma \vdash_S v : \mathbb{r},\ f_\alpha \,\&\, \mathit{unsync}(\Gamma') \triangleright \Gamma'}\ (\text{T-Sync}) \qquad \frac{\Gamma \vdash v : \mathbb{r}}{\Gamma \vdash_S v : \mathbb{r}, \mathtt{0} \triangleright \Gamma}\ (\text{T-Sync-Val})$$

$$\frac{\Gamma \vdash_S e : \square, \mathsf{L}_1 \triangleright \Gamma' \qquad \Gamma' \vdash_S e' : \square, \mathsf{L}_2 \triangleright \Gamma''}{\Gamma \vdash_S e \oplus e' : \square, \mathsf{L}_1 + \mathsf{L}_2 \triangleright \Gamma''}\ (\text{T-Expression}) \qquad \frac{\alpha \text{ fresh} \qquad \Gamma \vdash \overline{v} : \overline{\square}}{\Gamma \vdash_S \mathtt{new}\ \mathtt{Act}(\overline{v}) : \alpha, \mathtt{0} \triangleright \Gamma}\ (\text{T-New})$$

$$\frac{\Gamma \vdash_S v : \alpha, \mathsf{L} \triangleright \Gamma' \qquad \Gamma' \vdash \overline{v} : \overline{\mathbb{z}} \qquad \Gamma' \vdash \mathtt{m} : (\alpha, \overline{\mathbb{x}}, X) \to \mathbb{r} \qquad f \text{ fresh} \qquad \Gamma'' = \Gamma'[f \mapsto \lambda X.\mathtt{m}(\alpha, \overline{\mathbb{z}}, X)]}{\Gamma \vdash_S v.\mathtt{m}(\overline{v}) : \mathbb{r}_f, \mathsf{L} + f_\star \,\&\, \mathit{unsync}(\Gamma') \triangleright \Gamma''}\ (\text{T-Invk})$$

Figure 4.4 – Typing rules for expressions, addresses and expressions with side-effects.

We omit the unsynchronised part in this case as no deadlock can be created at those steps.

Figure 4.4 presents the typing rules for expressions, addresses and expressions with side effects. While (TR-Var) and (TR-Val) are trivial, the rule (T-Method-Sign) deserves some explanations. The rule (T-Method-Sign) instantiates a method signature according to the invocation parameters, what is important to notice is that if the returned type is neither an integer, nor the object

executing the method, nor an object passed as parameter, then the object returned has a fresh name.

Rule (T-Sync) types the synchronisation of a not-yet-synchronised value v, namely when v has type $\mathbb{r}_f$. The rule dereferences the future f by assigning to v the type $\mathbb{r}$. The corresponding behavioural type is $f_\alpha \,\&\, unsync(\Gamma')$, where α is the synchronising active object, and the not-yet-synchronised method invocations are added in parallel. The environment Γ is updated in order to record the synchronisation in the possible aliases of v ($\Gamma[y \mapsto \mathbb{r}]^{\Gamma(y)=\mathbb{r}_f}$) and to record that f has been computed, by adding the the tag $\checkmark$ to the type of f ($f \mapsto \Gamma(f)^{\checkmark}$). Notice that in case v is already associated to a value type, namely $\mathbb{r}$, no synchronisation and no environment update is performed, as defined in rule (T-Sync-Val).

Rule (T-Invk) types a method invocation. It creates a new future and stores it in Γ. The receiving active object v must have a type α, meaning that the value cannot be an unresolved future, while the parameters $\overline{v}$ may have future types. The resulting behavioural type is the sum of the behavioural types resulting from the possible synchronisation on v and the not-yet-synchronised method invocations – i.e., the new one coputing f and those that are running in parallel, represented by $unsync(\Gamma')$.

Rule (T-Expression) types the arithmetic expressions, the behavioural type assigned is the sum of the behavioural type of the sub expressions. This rule is applied until the sub expressions are terms v that will be typed using the rules (T-Synch) and (T-Synch-Val). The typing of the creation of a new active object is handled by rule (T-New). A fresh name is created and assigned to this new active object. The rule also checks that all the fields of this new object are synchronised integers as required by the restriction (*ii*) presented in Section 4.1.

Figure 4.5 presents the typing rules for statements. Rules (T-Assign-Field), (T-Assign-Val), and (T-Assign-Exp) type the assignment. The first one constrains fields to be assigned to (already synchronised) integers; the second one types the assignment of values to local variables, without performing any synchronisation, thus supporting the aliasing of future variables; the third rule types the assignment of an expression $e \oplus e'$ a method invocation or an active object creation to a local variable. In the last case, a synchronisation might be required.

Rule (T-Seq), given the types L_1 and L_2 of the two subsequent statements,

statements

- judgements used: $\Gamma \vdash_S s : \mathsf{L} \triangleright \Gamma'$

$$\frac{x \in \mathit{fields}(\texttt{Act}) \qquad \Gamma \vdash v : \square}{\Gamma \vdash_S x = v : \mathsf{0} \triangleright \Gamma}\ (\text{T-Assign-Field}) \qquad \frac{x \notin \mathit{fields}(\texttt{Act}) \qquad \Gamma \vdash v : \mathbb{x}}{\Gamma \vdash_S x = v : \mathsf{0} \triangleright \Gamma[x \mapsto \mathbb{x}]}\ (\text{T-Assign-Val})$$

$$\frac{\begin{array}{c} x \notin \mathit{fields}(\texttt{Act}) \\ z \text{ is not a value } v \qquad \Gamma \vdash_S z : \mathbb{x}, \mathsf{L} \triangleright \Gamma' \end{array}}{\Gamma \vdash_S x = z : \mathsf{L} \triangleright \Gamma'[x \mapsto \mathbb{x}]}\ (\text{T-Assign-Exp}) \qquad \frac{\begin{array}{c} \Gamma \vdash_S s_1 : \mathsf{L}_1 \triangleright \Gamma_1 \\ \Gamma_1 \vdash_S s_2 : \mathsf{L}_2 \triangleright \Gamma_2 \end{array}}{\Gamma \vdash_S s_1; s_2 : \mathsf{L}_1 + \mathsf{L}_2 \triangleright \Gamma_2}\ (\text{T-Seq})$$

$$\frac{\begin{array}{c} \Gamma(\texttt{destiny}) = \mathbb{r} \qquad \Gamma(\texttt{future}) = X \qquad \Gamma \vdash v : \mathbb{r}'_{f'} \\ \mathbb{r}' \in S \vee \mathbb{r} \in S \Longrightarrow \mathbb{r} = \mathbb{r}' \qquad \Gamma' = \Gamma[f' \mapsto \Gamma(f')^{\rightarrow}] \end{array}}{\Gamma \vdash_S \texttt{return}\ v : f'_X \,\&\, \mathit{unsync}(\Gamma') \triangleright \Gamma'}\ (\text{T-Return}) \qquad \frac{\begin{array}{c} \Gamma(\texttt{destiny}) = \mathbb{r} \qquad \Gamma \vdash v : \mathbb{r}' \\ \mathbb{r}' \in S \vee \mathbb{r} \in S \Longrightarrow \mathbb{r} = \mathbb{r}' \end{array}}{\Gamma \vdash_S \texttt{return}\ v : \mathsf{0} \triangleright \Gamma}\ (\text{T-Return-Val})$$

$$\Gamma \vdash_S \texttt{skip} : \mathsf{0} \triangleright \Gamma\ (\text{T-Skip}) \qquad \frac{\begin{array}{c} \Gamma \vdash_S e : \square, \mathsf{L} \triangleright \Gamma' \qquad \Gamma' \vdash_S s_1 : \mathsf{L}_1 \triangleright \Gamma_1 \qquad \Gamma' \vdash_S s_2 : \mathsf{L}_2 \triangleright \Gamma_2 \\ (\texttt{AFut}(\Gamma_1) \cup \texttt{AFut}(\Gamma_2)) \setminus \texttt{AFut}(\Gamma') = \varnothing \\ \Gamma'' = \texttt{Merge}(\Gamma_1, \Gamma_2) \cup \Gamma_1|_{\texttt{Fut}(\Gamma_1)\setminus\texttt{Fut}(\Gamma')} \cup \Gamma_2|_{\texttt{Fut}(\Gamma_2)\setminus\texttt{Fut}(\Gamma')} \end{array}}{\Gamma \vdash_S \texttt{if}\ e\ \{\, s_1 \,\}\ \texttt{else}\ \{\, s_2 \,\} : \mathsf{L} + \mathsf{L}_1 + \mathsf{L}_2 \triangleright \Gamma''}\ (\text{T-If})$$

Figure 4.5 – Typing rules for statements.

correctly associates a new composite type to the sequential composition of the statements. In rule (T-Return) we check that the return type value corresponds to the value type of `destiny` unless it is a fresh name (not belonging to the formal parameters of the method). In Γ, `future` contains the handler variable X. The corresponding future is tagged with a $\rightarrow$ in the resulting environment (because it is returned to the caller). In case v is already associated to a value type, namely $\mathbb{r}$, no synchronisation and no environment update is performed as defined in rule (T-Return-Val).

In rule (T-If) the behavioural type of a conditional statement is the sum of the behavioural type resulting from the typing of the expression e and the behavioural types of the two branches. The rule can be applied provided that the futures created in the branches are either returned (by a *return* statement) or syn-

(T-Method)

$$\frac{\begin{array}{c}\Gamma(\mathtt{m}) = (\alpha, \overline{\mathbb{x}}, X) \to \mathbb{r} \qquad \overline{y} = \mathit{fields}(\mathtt{Act}) \\ \Gamma + \mathtt{this} : \alpha + \overline{y} : \overline{\square} + \overline{x : \mathbb{x}} + \mathtt{destiny} : \mathbb{r} + \mathtt{future} : X \vdash_{\{\alpha, \overline{\mathbb{x}}\}} s : \mathsf{L} \triangleright \Gamma' \\ \mathtt{AFut}(\Gamma') = \varnothing \qquad \overline{\varphi} = \mathit{var}(\mathsf{L}) \setminus \{\alpha, \overline{\mathbb{x}}\} \\ \Theta_{\mathtt{m}} = [f \mapsto \Gamma'(f)^{\times}]^{f \in \mathtt{Fut}(\Gamma')} \qquad \mathsf{L}_{\mathtt{m}} = \mathsf{L} \,\&(X, \alpha)\end{array}}{\Gamma \vdash \mathtt{m}\ (\overline{T\ x})\{s\} : \mathtt{m}(\alpha, \overline{\mathbb{x}}, X) = (\nu\, \overline{\varphi})(\Theta_{\mathtt{m}} \cdot \mathsf{L}_{\mathtt{m}})}$$

(T-Program)

$$\frac{\begin{array}{c}\left(\Gamma \vdash \mathtt{m}\ (\overline{T\ x})\{s\} : \mathcal{L}(\mathtt{m})\right)^{(\mathtt{m}\,(\overline{T\ x})\{s\}) \in \overline{M}} \\ \Gamma + \mathtt{this} : \mathit{main} \vdash_{\varnothing} s : \mathsf{L} \triangleright \Gamma' \qquad \Theta = [f \mapsto \Gamma'(f)^{\times}]^{f \in \mathtt{Fut}(\Gamma')}\end{array}}{\Gamma \vdash \{\overline{\mathtt{Int}\ x}, \overline{M}\}\ \{s\} : (\mathcal{L}, \Theta \cdot \mathsf{L})}$$

Figure 4.6 – Typing rules for methods and programs.

chronised (constraint on line 2), variables are mapped to the same futures in both branches (line 3). The environment is updated with the changes that occur in the two branches, when they are equal, and the futures created in one of the branches, when they are synchronised or returned ($\Gamma_1|_{\mathtt{Fut}(\Gamma_1)\setminus\mathtt{Fut}(\Gamma')} \cup \Gamma_2|_{\mathtt{Fut}(\Gamma_2)\setminus\mathtt{Fut}(\Gamma')}$). Variables that are modified in different ways by the two branches are restricted to be almost the same behavioural type. The only allowed difference is if one future is synchronised in one of the branch and not in the other; in this case the merge retains the unsynchronised behaviour.

Finally Figure 4.6 presents the typing rules for methods and program. In rule (T-Method), the behavioural type of the method body is extended with a parallel pair (X, α) which will be instantiated by the active object performing a synchronisation on this method. This rule also checks if restriction (*iii*) of section 4.1 is respected, the rule constrains method bodies to synchronise all the futures created in the body or return them ($\mathtt{AFut}(\Gamma') = \varnothing$). The environment $\Theta_{\mathtt{m}}$ is defined by comprehension, collecting all the futures created in the method without synchronisation information ($\Theta_{\mathtt{m}} = [f \mapsto \Gamma'(f)^{\times}]^{f \in \mathtt{Fut}(\Gamma')}$).

Example 4.3.1. *Let us discuss the typing of the factorial method shown in Figure 3.2.a in Section 3.1.*
We begin by typing of the main body.

The typing environment at the beginning only maps method names to method signatures. Having only the `fact` *method, the typing environment result to be:*

$$\Gamma = [\ \texttt{fact} \mapsto (\ \alpha_{\texttt{fact}},\ \Box_{f_{\texttt{fact}}},\ \Box_{g_{\texttt{fact}}},\ X\ ) \to \Box\ ].$$

To apply the rule (T-Program) *we need to verify that the typing environment* Γ *extended with* `this` : *main types the body of the main function with the behavioural type* L*:*

$$\Gamma + \texttt{this} : \mathit{main} \vdash_{\varnothing} \ \texttt{x} = \texttt{newAct}()\ ;\ \texttt{y} = \texttt{x.fact}(3,1)\ ;\ \texttt{z} = \texttt{y} + 1 : L \ \triangleright\ \Gamma''.$$

The rule (T-Seq) *state that the behavioural type of a sequence of statements is the sum of the behavioural of each statement, that can be individually typed.*

To type the statement `x = new Act()` *with the typing environment* $\Gamma + \texttt{this} : \mathit{main}$ *we apply the rules* (T-Assign-Exp) *and* (T-New)*, obtaining that:*

$$\Gamma + \texttt{this} : \mathit{main} \vdash_{\varnothing} \texttt{x = new Act()} : 0 \ \triangleright\ \Gamma + \texttt{this} : \mathit{main} + x : \alpha.$$

We can see that the type of a `new` *statement is an empty behaviour, and that the application of these two rules only has impact on the typing environment that has been extended relating the variable* x *with the type of the object just created.*

To type statement `y = x.fact(3,1);`*, that is a method invocation, we have to apply the rules* (T-Assign-Exp) *and* (T-Invk)*, and we have that:*

$$\Gamma + \texttt{this} : \mathit{main} + x : \alpha \vdash_{\varnothing} \texttt{y = x.fact(3,1)} : f''_{\star} \ \triangleright\ \Gamma'$$

where $\Gamma' = \Gamma + \texttt{this} : \mathit{main} + x : \alpha + y : \Box_{f''} + f' : \lambda X.\texttt{fact}(\alpha, \Box, \Box, X)$.

Finally to type the arithmetic expression we have to use the rules (T-Assign-Exp),(T-Expression)*, and then to type* `0` *we will use the rule* (T-Val)*, while to type* `y`*, because in* `y` *is stored the future* f'' *(*$\Gamma(y) = \Box_{f''}$*) we use the rule* (T-Synch)*, and we get:*

$$\Gamma' \vdash_{\varnothing} \texttt{z = y + 0;} : f''_{main} \ \triangleright\ \Gamma''$$

where $\Gamma'' = \Gamma + \texttt{this} : main + x : \alpha + y : \Box_{f''} + z : \Box + \ f'' : \lambda X.\texttt{fact}(\alpha, \Box, \Box, X)^{\checkmark}$.

Now that we have the type of the body of the main function, by rule (T-PROGRAM) *we can state that this function has behavioural type* $(\mathsf{L} \cdot \Theta)$ *where:*

$$\begin{aligned} \mathsf{L} &= f''_{\star} \ + \ f''_{main} \\ \Theta &= \{ \ f'' \mapsto \lambda X.\texttt{fact}(\alpha, \Box, \Box, X) \ \}. \end{aligned}$$

At this point we want to type the method `fact`*. To accomplish this goal we need to apply the rule* (T-METHOD)*. This rule requires that we type the body of method* `fact`*, (we abbreviate the body of fact with* $s_{\texttt{fact}}$*). To type* $s_{\texttt{fact}}$ *we use the typing environment* Γ *extended as follow:*

$$\Gamma' = \Gamma + \texttt{this} : \alpha + \texttt{n} : \Box_{f_{\texttt{fact}}} + \texttt{r} : \Box_{g_{\texttt{fact}}} + \texttt{destiny} : \Box + \texttt{future} : X.$$

The typing of $s_{\texttt{fact}}$ *will be done similarly to the body of the main function, obtaining as final result that:*

$$\Gamma' \vdash_{\{\alpha, \Box_{f_{\texttt{fact}}}, \Box_{g_{\texttt{fact}}}\}} s_{\texttt{fact}} : \mathsf{L}_{\texttt{fact}} \ \triangleright \ \Gamma''$$

where

$$\begin{aligned} \Theta_{\texttt{fact}} &= \{ \ f' \mapsto \lambda X. \ \texttt{fact}(\beta, \Box, \Box, X) \ \} \\ \mathsf{L}_{\texttt{fact}} &= (\ f_{\alpha} \ + g_X \ + \ g_{\alpha} \ + f'_{\star} \ + \ f'_X \ ) \ \& \ (X, \alpha) \end{aligned}$$

The type $\mathsf{L}_{\texttt{fact}}$ *is a sum of five types (we omit the terms that are* `0`*). The first term* f_{α} *refers to the synchronisation of the parameter* `n`*. The second term* g_X *is the type of the return of a potential future in the true branch of the* `if` *(*`r` *is a parameter, then a potential future, which has not been synchronised before being returned, then at that point it is still considered as a potential future). The term* g_{α} *represents the synchronisation of the parameter* `r` *of a possible future that can be performed during the evaluation of the expression at line 4. The fourth term* $f'_{\star}$ *is the type associated to the method invocation done at line 5; in this behavioural type we can see that the future* f' *created by this method invocation is related to the method invocation* $(\Theta_{\texttt{fact}}(f') = \lambda X. \ \texttt{fact}(\beta, \Box, \Box, X))$*. Finally we have that the term* f'_X *is the behavioural type associated to the return of the future* f' *that is done at line 6. The sum of these five terms is in conjunction with the pair* (X, α)

which represents the possible synchronisation made by a synchroniser X that want to synchronise an invocation of the method `fact`.

Now that we have both the behavioural type of the main function and the one of the method `fact` *we can state that the behavioural type of the program is:*

$$\big(\ \mathtt{fact}(\alpha, \square_f, \square_g, X) = (\nu\,\beta, f')(\Theta_{\mathtt{fact}} \cdot \mathsf{L}_{\mathtt{fact}}),\ \Theta \cdot \mathsf{L}\ \big).$$

4.4 Behavioural type Analysis

The behavioural types presented above are actually terms of a recursive model that expresses dependencies and features recursion and dynamic name creation. This model turns out to be an extension of the so called deadLock Analysis Model (LAM), defined by Giachino et al. [2014], and in this section we correspondingly extend the theory.

The operational semantics of a behavioural type program $\big(\mathcal{L}, \Theta \cdot \mathsf{L}\big)$ is a transition system where *states* are pairs of a *future environment* Θ (mapping future names to method invocation instances) and a behavioural type. The definition of transitions requires some notation. *Contexts*, noted $\mathcal{C}[\,]$, are defined by the syntax

$$\mathcal{C}[\,] \quad ::= \quad [\,] \qquad | \qquad \mathsf{L} \,\&\, \mathcal{C}[\,] \qquad | \qquad \mathsf{L} + \mathcal{C}[\,]$$

As usual, $\mathcal{C}[\mathsf{L}]$ is the type where the hole of $\mathcal{C}[\,]$ is replaced by L. The *environment update of futures* (with $f' \notin \mathrm{dom}(\Theta)$) and *extended type substitution* are respectively defined as follows:

$$(\Theta[f'/f])(g) \stackrel{def}{=} \begin{cases} \Theta(f) & \text{if } g = f' \\ \Theta(g) & \text{if } g \neq f \end{cases} \qquad \mathsf{L}[\mathbb{r}/\mathbb{r}'_f] = \mathsf{L}[\mathsf{0}/f_\kappa]$$

The last substitution replaces all occurrences of a future synchronisation by a null behaviour. It is used when a (synchronised) value $\mathbb{r}$, i.e. a value that is not a future, is passed to a method. Indeed, to have the most generic signature, in method behaviours (stored in Θ), all formal parameters are assumed to be potentially a non synchronised future $\mathbb{r}'_f$.

The *transition relation* is the least one satisfying the rule

BT-RED

$$\frac{\begin{array}{c}\Theta(f) = \lambda X.\mathtt{m}(\alpha, \overline{\mathbb{x}}, X) \\ \mathtt{m}(\alpha', \overline{\mathbb{x}'}, X) = (\nu\, \overline{\varphi})(\Theta_{\mathtt{m}} \cdot \mathsf{L}_{\mathtt{m}}) \in \mathcal{L} \qquad \kappa \neq X \qquad \overline{\varphi'} \text{ fresh} \\ \mathsf{L}' = \mathsf{L}_{\mathtt{m}}[\overline{\varphi'}/\overline{\varphi}][\alpha, \overline{\mathbb{x}}, \kappa/\alpha', \overline{\mathbb{x}'}, X] \qquad \Theta' = \Theta \cup \Theta_{\mathtt{m}}[\overline{\varphi'}/\overline{\varphi}][\alpha, \overline{\mathbb{x}}/\alpha', \overline{\mathbb{x}'}]\end{array}}{\Theta \cdot \mathcal{C}[\, f_\kappa \,] \to \Theta' \cdot \mathcal{C}[\, \mathsf{L}' \,]}$$

The initial state of $(\mathcal{L}, \Theta \cdot \mathsf{L})$ is $\Theta \cdot \mathsf{L}$. We write $\to^*$ for the reflexive and transitive closure of $\to$.

A behavioural type L is evaluated by successively replacing each future dereference with the corresponding type instance. Name creation is handled by replacing bound names of method bodies with fresh names. Notice that when $\kappa = X$ the rule does not apply meaning that the behaviour of the return of a future (f_X) is not evaluated until that future is synchronised by some active object (f_β).

Example 4.4.1. *Let us consider the behavioural type program of Figure 3.2.a in section 3.1:*

$$(\mathtt{fact}(\alpha, \square_f, \square_g, X) = (\nu\, \beta, f')(\Theta_{\mathtt{fact}} \cdot \mathsf{L}_{\mathtt{fact}}), \Theta \cdot \mathsf{L})$$

where

$$\begin{array}{lcl} \Theta_{\mathtt{fact}} & = & \{f' \mapsto \lambda X.\ \mathtt{fact}(\beta, \square, \square, X)\} \\ \mathsf{L}_{\mathtt{fact}} & = & \Big(\big(f_\alpha\ + g_X\ +\ g_\alpha\ + f'_\star\ +\ f'_X\big)\ \&\ (X, \alpha)\Big) \\ \Theta & = & \{f'' \mapsto \lambda X.\mathtt{fact}(\alpha, \square, \square, X)\} \\ \mathsf{L} & = & f''_\star + f''_{main} \end{array}$$

Starting from the behavioural type of the main function $\Theta \cdot f''_\star + f''_{main}$ *, by rule* (BT-RED) *we reduce the term* $f''_\star$ *obtaining that:*

$$\Theta \cdot \mathsf{L} \ \to\ \Theta' \cdot \ \big((g'_\star\ +\ g'_\star)\ \&\ (\star, \alpha)\big) + f''_{main}$$

then we continue with the reduction of the therm f''_{main}

$$\begin{array}{lll} \Theta \cdot \mathsf{L} & \to \Theta' \cdot \big((g'_\star\ +\ g'_\star)\ \&\ (\star, \alpha)\big) + f''_{main} \\ & \to \Theta'' \cdot \big((g'_\star\ +\ g'_\star)\ \&\ (\star, \alpha)\big) + \big((g''_{main}\ +\ g''_{main})\ \&\ (main, \alpha)\big) \\ & \to \cdots \end{array}$$

and so on, where

$$\begin{aligned}\Theta' &= \Theta \cup \{g' \mapsto \lambda X.\ \mathtt{fact}(\beta', \square, \square, X)\} \\ \Theta'' &= \Theta' \cup \{g'' \mapsto \lambda X.\ \mathtt{fact}(\beta'', \square, \square, X)\}.\end{aligned}$$

The term g''_{main} will be replaced by the instantiated behavioural type of the method, producing the pair $(main, \beta'')$, reflecting the fact that the active object main is synchronising also the nested recursive calls.

Notice that reduction of the behavioural types of the program never ends: the type grows in the number of "+"-terms, which in turn become larger and larger as the evaluation progresses. In principle, we do not know whether the former will produce a deadlock at some further evaluation step. Since the computation is infinite, we do not know when to stop looking for a deadlock. The fixpoint technique of Section 4.6 offers us a finite way to discriminate between deadlocked and deadlock-free programs.

4.5 Type soundness

The correctness of the type system in Section 4.3 is demonstrated by means of 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, in our case we cannot demonstrate a statement guaranteeing standard type-preservation (the equality of types of cn and cn') because our types are behavioural. However, it is critical for the correctness of the analysis that there is a relation between the type of cn, let it be $\Theta \cdot \mathsf{L}$, and the type of cn', let it be $\Theta' \cdot \mathsf{L}'$. Therefore, a subject reduction for the type system of Section 4.3 requires the extension of the typing to configurations; and the definition of a *later-stage* relation between behavioural types.

Runtime type system

In order to state the subject reduction theorem, we first have to extend the type syntax of Figure 4.1 to be able to type the runtime configurations. To this aim we

$$
\begin{array}{rcll}
\mathbb{r} & ::= & \square \mid \alpha & \text{basic type} \\
\mathbb{x} & ::= & \mathbb{r}_F \mid \mathbb{r} & \text{extended type} \\
F & ::= & f \mid f^s & \text{future type} \\
\kappa & ::= & \star \mid \alpha \mid X & \text{synchronisers} \\
\mathsf{L} & ::= & \mathsf{0} \mid (\kappa, \alpha) \mid F_\kappa \mid \mathsf{L} + \mathsf{L} \mid \mathsf{L} \,\&\, \mathsf{L} & \text{behavioural type} \\
\mathsf{K} & ::= & (\nu\, \overline{\varphi})(\Theta \cdot \mathsf{L}) \mid \mathsf{K} \,\&\, \mathsf{K} & \text{behavioural type for configuration}
\end{array}
$$

Figure 4.7 – Syntax of behavioural types at runtime.

extend the syntax of behavioural type in Figure 4.7 introducing *extended futures* F and *behavioural type for configuration* K.

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) f^s now replacing f in its role of static time future (it is typically used to reference a future that is not created yet).

The typing judgments are identical to the corresponding ones used in the type system, except for some minor differences:

1) the typing environment, now maps method names to a pair of elements (i.e $\Delta(\mathtt{m}) = ((\alpha, \overline{\mathbb{x}}, X) \to \mathbb{r}, \mathsf{K})$) that are respectively the method signature and its behavioural type at runtime. It is called Δ;
2) the $rt\_unsync(\cdot)$ function on environments Δ is similar to $unsync(\cdot)$ in Section 4.3, except that it now grabs all f^s and all futures f that were created by the current thread f . More precisely we define $\mathtt{Fut}_R(\Delta)$, $\mathtt{AFut}_R(\Delta)$, and $rt\_unsync(\Delta)$ as in Figure 4.8.

$\mathtt{Fut}_R(\Gamma)$ collects all the (static and runtime) futures stored in Δ, $\mathtt{AFut}_R(\Delta)$ collects all the (static and runtime) futures that are not tagged with a $\checkmark$ or $\to$, and $rt\_unsync(\Delta)$ performs the parallel composition of the behavioural types of such not-yet-synchronised method invocations.

The typing rules for the runtime configuration are given in Figures 4.9, 4.10 and 4.11. Except for a few rules (in particular, those in Figures 4.9 which type

$$\mathtt{Fut}_R(\Delta) \stackrel{def}{=} \{F \mid F \in \mathrm{dom}(\Delta)\} \tag{4.12}$$

$$\mathtt{AFut}_R(\Delta) \stackrel{def}{=} \{F \in \mathtt{Fut}_R(\Delta) \mid \Delta(F) = \Delta(F)^{\times}\} \tag{4.13}$$

$$rt\_unsync(\Delta) \stackrel{def}{=} \mathop{\&}_{F \in \mathtt{AFut}_R(\Delta)} F_\star \tag{4.14}$$

Figure 4.8 – Auxiliary definitions

the runtime element of a configuration), all the typing rules have a corresponding one in the type system defined in Section 4.3.

In the type system at runtime, the rule (TR-Parallel) types an entire configuration with the parallel composition of the runtime behavioural type, in which each term corresponds to one element of the configuration. The rule (TR-ActObj) types an active object with the parallel composition of the runtime behavioural type of running process and behavioural types of all the process in the queue of pending processes. Rule (TR-Process) results to be very similar to the rule (T-Method) presented in Figure 4.6. As we have already said most of the rule are similar to the one of the static type system, and only differ for the fact that they handle both static and runtime future, while the rule (TR-Invk) needs some more details. Rule (TR-Invk) creates a static fresh future, at this point we are sure that this future can only be a static future whatever it is moment in which this statement is typed the method invocation has not been performed, then the runtime future has not been created. As it is possible to see in the proof of the subject reduction for the case of Invk while the typing environment that types cn contains a static future, the environment that types cn' (in cn' we have that the method has been invoked) has to contain the corresponding runtime version of the static future created typing cn.

$$\frac{\Delta \vdash cn_1 : \mathsf{K}_1 \qquad \Delta \vdash cn_2 : \mathsf{K}_2}{\Delta \vdash cn_1\ cn_2 : \mathsf{K}_1 \,\&\, \mathsf{K}_2}\ (\text{TR-Parallel}) \qquad \frac{-}{\Delta \vdash f(w) : \mathtt{0}}\ (\text{TR-Future}) \qquad \frac{\Delta \vdash^{\alpha} p : \mathsf{K}_1 \qquad \Delta \vdash^{\alpha} \overline{q} : \mathop{\&}_{i=2}^{n} \mathsf{K}_i}{\Delta \vdash \alpha(a, p, \overline{q}) : \mathop{\&}_{i=1}^{n} \mathsf{K}_i}\ (\text{TR-ActObj})$$

$$\frac{\begin{array}{c}\Delta \vdash \mathtt{m} : (\alpha, \overline{\mathbb{x}}, X) \to \mathbb{r} \qquad \Delta(f) = \lambda X.\mathtt{m}(\alpha, \overline{\mathbb{x}}, X) \\ \Delta + \overline{x : \mathbb{x}} + \mathtt{destiny} : \mathbb{r} + \mathtt{future} : X \vdash^{\alpha}_{\{\alpha,\overline{\mathbb{x}}\}} s : \mathsf{L} \triangleright \Delta' \\ \mathtt{AFut}_R(\Delta') = \varnothing \qquad \overline{\varphi} = var(\mathsf{L}) \setminus \{\alpha, \overline{\mathbb{x}}\} \\ \Theta_{\mathtt{m}} = [f \mapsto \Delta'(f)^{\times}]^{f \in \mathtt{Fut}_R(\Delta')} \qquad \mathsf{L}_{\mathtt{m}} = \mathsf{L} \,\&\, (X, \alpha)\end{array}}{\Delta \vdash^{\alpha} \{destiny \mapsto f, \overline{x \mapsto v} \mid s\} : (\nu\, \overline{\varphi})(\Theta_{\mathtt{m}} \cdot \mathsf{L}_{\mathtt{m}})}\ (\text{TR-Process})$$

Figure 4.9 – Typing rules for runtime configurations.

Later-stage

The *later-stage* relation $\succeq_{\Delta}$ is a syntactic relationship between behavioural types. Formally, the later-stage relation is the least congruence with respect to runtime behavioural type that contains the rules in Figure 4.12. We can simplify the basic laws of the later-stage relation saying that a method invocation is larger than the instantiation of its method behaviour (LS-Invk), and a sum type is larger than each element of the sum.

We are now ready to state the Subject Reduction theorem.

Theorem 4.5.1 (Subject Reduction)**.** *Let $\Delta \vdash_R cn : \mathsf{K}$ and $cn \to cn'$. Then there exist Δ', K', and an injective renaming of active object names i such that*

- *$\Delta' \vdash_R cn' : \mathsf{K}'$ and*
- *$i(\mathsf{K}) \succeq_{\Delta} \mathsf{K}'$*

The proof is a case analysis on the reduction rule used in $cn \to cn'$, and can be found in Appendix A.1. To better understanding the proofs it is worth to notice that the theorem 4.5.1 states that the runtime environment Δ' exists, then to type cn' we have to define the runtime typing environment that contains all the information needed to type cn'. As it will be possible to see in the proofs, the runtime typing environment that we use to type the target configuration (cn')

expressions and addresses

- judgments used: $\Delta \vdash w : \mathbb{x}$ and $\Delta \vdash m : (\alpha, \overline{\mathbb{x}}, X) \to \mathbb{r}$

(TR-Var)
$$\frac{\Delta(x) = \mathbb{x}}{\Delta \vdash^{\alpha} x : \mathbb{x}}$$

(TR-Val)
$$\frac{v \quad \textit{integer-value} \text{ or } \texttt{null}}{\Delta \vdash^{\alpha} v : \mathbb{r}}$$

(TR-Act)
$$\frac{-}{\Delta \vdash^{\alpha} \beta : \beta}$$

(TR-Fut)
$$\frac{\Delta(f) = \lambda X.\mathtt{m}(\beta, \overline{\mathbb{z}}, X) \qquad \Delta \vdash \mathtt{m} : (\beta, \overline{\mathbb{x}}, X) \to \mathbb{r}}{\Delta \vdash^{\alpha} f : \mathbb{r}_f}$$

(TR-Method-Sign)
$$\frac{\begin{array}{c}\Delta(\mathtt{m}) = (\nu\, \overline{\varphi})(\Theta_{\mathtt{m}} \cdot \mathsf{L}_{\mathtt{m}}) \\ \sigma \text{ renaming} \qquad \sigma(\square) = \square \qquad \mathbb{r} \notin \{\square, \alpha, \overline{\mathbb{x}}\} \implies \sigma(\mathbb{r}) \text{ fresh}\end{array}}{\Delta \vdash m : (\sigma(\alpha), \sigma(\overline{\mathbb{x}}), X) \to \sigma(\mathbb{r})}$$

expressions with side effects

judgement used - $\Delta \vdash^{\alpha}_{R} e : \mathbb{x} \ , \ \mathsf{L} \ \triangleright \ \Delta'$

(TR-Sync)
$$\frac{\begin{array}{c}\Delta \vdash v : \mathbb{r}_F \qquad \Delta(\mathtt{this}) = \alpha \\ \Delta' = (\Delta[y \mapsto \mathbb{r}]^{\Delta(y) = \mathbb{r}_F})[F \mapsto \Delta(F)^{\checkmark}]\end{array}}{\Delta \vdash^{\alpha}_{R} v : \mathbb{r} \ , \ F_{\alpha} \,\&\, rt\_unsync(\Delta') \ \triangleright \ \Delta'}$$

(TR-Sync-Val)
$$\frac{\Delta \vdash^{\alpha} v : \mathbb{r}}{\Delta \vdash^{\alpha}_{R} v : \mathbb{r} \ , \ \mathtt{0} \ \triangleright \ \Delta}$$

(TR-Expression)
$$\frac{\begin{array}{c}\Delta \vdash^{\alpha}_{R} e : \square \ , \ \mathsf{L}_1 \ \triangleright \ \Delta' \\ \Delta' \vdash^{\alpha}_{R} e' : \square \ , \ \mathsf{L}_2 \ \triangleright \ \Delta''\end{array}}{\Delta \vdash^{\alpha}_{R} e \oplus e' : \square \ , \ \mathsf{L}_1 + \mathsf{L}_2 \ \triangleright \ \Delta''}$$

(TR-New)
$$\frac{\alpha \text{ fresh} \qquad \Delta \vdash \overline{v} : \overline{\square}}{\Delta \vdash^{\alpha}_{R} \mathtt{new}\ \mathtt{Act}(\overline{v}) : \alpha \ , \ \mathtt{0} \ \triangleright \ \Delta}$$

(TR-Invk)
$$\frac{\begin{array}{c}\Delta \vdash^{\alpha}_{R} v : \alpha \ , \ \mathsf{L} \ \triangleright \ \Delta' \qquad \Delta' \vdash \overline{v} : \overline{\mathbb{z}} \qquad \Delta' \vdash \mathtt{m} : (\alpha, \overline{\mathbb{x}}, X) \to \mathbb{r} \\ f^s \text{ fresh} \qquad \Delta'' = \Delta'[f^s \mapsto \lambda X.\mathtt{m}(\alpha, \overline{\mathbb{z}}, X)]\end{array}}{\Delta \vdash^{\alpha}_{R} v.\mathtt{m}(\overline{v}) : \mathbb{r}_{f^s} \ , \ \mathsf{L} \ + \ f^s_{\star} \,\&\, rt\_unsync(\Delta') \ \triangleright \ \Delta''}$$

Figure 4.10 – Runtime typing rules for expressions, addresses and expressions with side-effects.

statements

- judgement used: $\Delta \vdash^{\alpha}_{R} s : \mathsf{L} \triangleright \Delta'$

$$\frac{x \in \mathit{fields}(\mathtt{Act}) \setminus \mathrm{dom}(\Delta) \qquad \Delta \vdash^{\alpha} v : \square}{\Delta \vdash^{\alpha}_{R} x = v : \mathtt{0} \triangleright \Delta} \text{(TR-Assign-Field)}$$

$$\frac{x \notin \mathit{fields}(\mathtt{Act}) \setminus \mathrm{dom}(\Delta) \qquad \Delta \vdash^{\alpha} w : \mathbb{x}}{\Delta \vdash^{\alpha}_{R} x = w : \mathtt{0} \triangleright \Delta[x \mapsto \mathbb{x}]} \text{(TR-Assign-Val)}$$

$$\frac{x \notin \mathit{fields}(\mathtt{Act}) \setminus \mathrm{dom}(\Delta) \qquad z \text{ is not a value } v \qquad \Delta \vdash^{\alpha}_{R} z : \mathbb{x} \ , \ \mathsf{L} \triangleright \Delta'}{\Delta \vdash^{\alpha}_{R} x = z : \mathsf{L} \triangleright \Delta'[x \mapsto \mathbb{x}]} \text{(TR-Assign-Exp)}$$

$$\frac{\Delta \vdash^{\alpha}_{R} s_1 : \mathsf{L}_1 \triangleright \Delta_1 \qquad \Delta_1 \vdash^{\alpha}_{R} s_2 : \mathsf{L}_2 \triangleright \Delta_2}{\Delta \vdash^{\alpha}_{R} s_1; s_2 : \mathsf{L}_1 + \mathsf{L}_2 \triangleright \Delta_2} \text{(TR-Seq)}$$

$$\frac{\Delta(\mathtt{destiny}) = \mathbb{r} \qquad \Delta(\mathtt{future}) = X \qquad \Delta \vdash v : \mathbb{r}'_{F'} \qquad \mathbb{r}' \in R \vee \mathbb{r} \in R \Longrightarrow \mathbb{r} = \mathbb{r}' \qquad \Delta' = \Delta[F' \mapsto \Delta(F')^{\rightarrow}]}{\Delta \vdash^{\alpha}_{R} \mathtt{return}\ v : F'_X \,\&\, \mathit{rt\_unsync}(\Delta') \triangleright \Delta'} \text{(TR-Return)}$$

$$\frac{\Delta(\mathtt{destiny}) = \mathbb{r} \qquad \Delta \vdash v : \mathbb{r}' \qquad \mathbb{r} \in R \Longrightarrow \mathbb{r} = \mathbb{r}'}{\Delta \vdash^{\alpha}_{R} \mathtt{return}\ v : \mathtt{0} \triangleright \Delta} \text{(TR-Return-Val)}$$

$$\Delta \vdash^{\alpha}_{R} \mathtt{skip} : \mathtt{0} \triangleright \Delta \text{(TR-Skip)}$$

$$\frac{\Delta \vdash^{\alpha}_{R} e : \square \ , \ \mathsf{L} \triangleright \Delta' \qquad \Delta' \vdash^{\alpha}_{R} s_1 : \mathsf{L}_1 \triangleright \Delta_1 \qquad \Delta' \vdash^{\alpha}_{R} s_2 : \mathsf{L}_2 \triangleright \Delta_2 \qquad (\mathtt{AFut}(\Delta_1) \cup \mathtt{AFut}(\Delta_2)) \setminus \mathtt{AFut}(\Delta') = \varnothing \qquad \Delta'' = \mathtt{Merge}(\Delta_1, \Delta_2) \cup \Delta_1|_{\mathtt{Fut}(\Delta_1) \setminus \mathtt{Fut}(\Delta)} \cup \Delta_2|_{\mathtt{Fut}(\Delta_2) \setminus \mathtt{Fut}(\Delta)}}{\Delta \vdash^{\alpha}_{R} \mathtt{if}\ e\ \{\, s_1 \,\}\ \mathtt{else}\ \{\, s_2 \,\} : \mathsf{L} + \mathsf{L}_1 + \mathsf{L}_2 \triangleright \Delta''} \text{(T-If)}$$

Figure 4.11 – Runtime typing rules for statements.

is the Δ' resulting from the application of the rule (TR-Process) to the source configuration (cn).

4.6 Analysis of circularities

In Section 4.4 we have seen that the behavioural type analysis mostly performs the unfolding of the behavioural types of method invocations. Because of the inherent recursive nature of the behavioural type, and the fact that method calls can create fresh names, we have that the behavioural type analysis may create infinite states.

In this section we present the analysis of the circularity proposed by Giachino,

$$\text{(LS-RuntimeEmpty)}\quad \mathsf{K} \succeq_\Delta \mathsf{0} \qquad \text{(LS-Global)}\quad \frac{\mathsf{K}_1 \succeq_\Delta \mathsf{K}_1'}{\mathsf{K}_1 \,\&\, \mathsf{K} \succeq_\Delta \mathsf{K}_1' \,\&\, \mathsf{K}} \qquad \text{(LS-Behavior)}\quad \frac{\mathsf{K} = (\nu\,\overline{\varphi})(\Theta \cdot \mathsf{L}) \qquad \mathsf{K}' = (\nu\,\overline{\varphi'})(\Theta \cdot \mathsf{L}') \qquad \mathsf{L} \succeq_\Delta \mathsf{L}'}{\mathsf{K} \succeq_\Delta \mathsf{K}'}$$

$$\text{(LS-Empty)}\quad \mathsf{L} \succeq_\Delta \mathsf{0} \qquad \text{(LS-Plus)}\quad \mathsf{L}_1 + \mathsf{L}_2 \succeq_\Delta \mathsf{L}_i \qquad \text{(LS-Parallel)}\quad \frac{\mathsf{L}_1 \succeq \mathsf{L}_1'}{\mathsf{L}_1 \,\&\, \mathsf{L} \succeq_\Delta \mathsf{L}_1' \,\&\, \mathsf{L}}$$

$$\text{(LS-Invk)}\quad \frac{\begin{array}{c}\Delta(f) = \lambda X.\mathtt{m}(\alpha', \overline{\mathbb{z}'}, X) \\ \Delta(m) = \big((\alpha, \overline{\mathbb{x}}, X) \to \mathbb{r}, \mathsf{K}_\mathtt{m}\big) \qquad \Delta \vdash \mathtt{m} : (\alpha', \overline{\mathbb{x}'}, X) \to \mathbb{r}' \\ \overline{\alpha} = fn(\mathsf{K}) \setminus fn(\alpha, \overline{\mathbb{x}}, \mathbb{r}) \qquad \overline{\alpha'} \cup fn(\alpha', \overline{\mathbb{x}'}, \mathbb{r}') = \varnothing\end{array}}{(\nu\,\overline{\varphi})(\Theta \cdot (f_\kappa \,\&\, \mathsf{L}) + \mathsf{L}_s) \succeq_\Delta (\nu\,\overline{\varphi})(\Theta \cdot \mathsf{L}_s) \,\&\, \mathsf{K}_\mathtt{m}[\overline{\alpha'}/\overline{\alpha}][\alpha', \overline{\mathbb{x}'}, \mathbb{r}'/_{\alpha, \overline{\mathbb{x}}, \mathbb{r}}]}$$

Figure 4.12 – The later-stage relation.

Kobayashi, and Laneve [2014] slightly adapted to our current model. Below we introduce some preliminary notion that will be used later in the definition of the analysis, then the analysis of circularities will be defined.

Preliminaries.

Let $\mathtt{R}$ be a set whose elements are either pairs (κ, β), where κ ranges over active object, 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

$(\alpha_0, \alpha_1), \cdots, (\alpha_{n-1}, \alpha_n) \in \mathtt{R}_i$.

– $\{\mathtt{R}_1, \cdots, \mathtt{R}_m\} \,\&\, \{\mathtt{R}'_1, \cdots, \mathtt{R}'_n\} \stackrel{def}{=} \{\mathtt{R}_i \cup \mathtt{R}'_j \mid 1 \leq i \leq m \text{ and } 1 \leq j \leq n\}$.

Definition 7. *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.*

For instance, $\big\{\{(\alpha,\beta),(\beta,\gamma)\},\ \{(\gamma,\beta),(\delta,\beta),(\beta,\gamma)\},\ \{(\eta,\delta)\}\big\}$ has a circularity because of the second element of the set.

Behavioural 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 behavioural 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 behavioural types as follows

$$
\begin{array}{rcll}
I(\Theta \cdot \mathtt{0}) & = & \{\varnothing\} & \\
I(\Theta \cdot (\kappa, \beta)) & = & \{\{(\kappa, \beta)\}\} & \\
I(\Theta \cdot f_\kappa) & = & I(\mathtt{m})[\alpha, \overline{\mathbb{x}}, \kappa/\alpha', \overline{\mathbb{x}'}, X] & \text{if } \Theta(f) = \lambda X.\mathtt{m}(\alpha, \overline{\mathbb{x}}, X) \\
 & & & \text{and } \mathtt{m}(\alpha', \overline{\mathbb{x}'}, X) \text{ is defined in } \mathcal{L} \\
I(\Theta \cdot f_\kappa) & = & \{\{f_\kappa\}\} & \text{if } f \notin \mathrm{dom}(\Theta) \\
I(\Theta \cdot \mathtt{L} \,\&\, \mathtt{L}') & = & I(\Theta \cdot \mathtt{L}) \,\&\, I(\Theta \cdot \mathtt{L}') & \\
I(\Theta \cdot \mathtt{L} + \mathtt{L}') & = & I(\Theta \cdot \mathtt{L}) \cup I(\Theta \cdot \mathtt{L}') &
\end{array}
$$

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

For example, let $\mathcal{L}$ define $\mathtt{m}(\alpha, \beta, \gamma, X)$ and $\mathtt{n}(\alpha, \beta, X)$ and let

$$
\begin{array}{l}
I(\mathtt{m}) = \{\{(\alpha,\beta),(X,\gamma)\}\} \qquad I(\mathtt{n}) = \{\{(\beta,\alpha)\}\} \\
\Theta = \{f \mapsto \lambda X.\mathtt{m}(\alpha,\beta,\gamma,X),\ f' \mapsto \lambda X.\mathtt{n}(\beta,\gamma,X),\ f'' \mapsto \lambda X.\mathtt{m}(\delta,\beta,\gamma,X)\} \\
\mathtt{L} = f_\star \ \&\ f''_\star \ +\ (\alpha,\beta) \ \&\ f'_X \ \&\ f''_\alpha \ +\ (\eta,\delta).
\end{array}
$$

Then

$$
\begin{array}{rl}
I(\Theta \cdot \mathtt{L}) = & \big\{\{(\alpha,\beta),(\star,\gamma),(\delta,\beta),(\star,\gamma)\}, \{(\alpha,\beta),(\gamma,\beta),(\delta,\beta),(\alpha,\gamma)\},\ \{(\eta,\delta)\}\big\} \\
I^\perp(\Theta \cdot \mathtt{L}) = & \big\{\varnothing, \{(\alpha,\beta)\}, \{(\eta,\delta)\}\big\}\ .
\end{array}
$$

Definition 8. *A state* $\Theta \cdot \mathsf{L}$ has a circularity *if* $I^{\perp}(\Theta \cdot \mathsf{L})$ *has a circularity. A behavioural type program* $\big(\mathcal{L}, \Theta \cdot \mathsf{L}\big)$ has a circularity *if there exist* Θ' *and* L' *such that* $\Theta \cdot \mathsf{L} \rightarrow^{*} \Theta' \cdot \mathsf{L}'$ *and* $\Theta' \cdot \mathsf{L}'$ *has a circularity.*

Analysis of circularities.

This analysis of circularities 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) which also contains 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 [Davey and Priestley, 2002].

Definition 9. *Let* $\mathcal{L} = \{\mathtt{m}_i(\alpha_i, \overline{\mathbb{x}}_i, X_i) = (\nu\, \overline{\varphi}_i)(\Theta_i \cdot \mathsf{L}_i) \mid i \in \{1..k\}\}$. *The family of flattening functions* $I^{(k)}$ *is defined as follows*

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

where

$$\begin{array}{rcl} \mathtt{proj}_{\alpha, \overline{\mathbb{x}}, X}(R) \stackrel{def}{=} & \{(\beta, \gamma) \mid\ (\beta, \gamma) \in R \text{ and } \beta, \gamma \in \alpha, \overline{\mathbb{x}}, X\} \bigcup \\ & \{(\varkappa, \varkappa) \mid\ (\delta, \delta) \in R \text{ and } \delta \notin \alpha, \overline{\mathbb{x}}, X\}. \end{array}$$

We notice that $I^{(0)}$ is the function $I^{\perp}$ presented above. Since, for every k, $I^{(k)}(\mathtt{m}_i)$ ranges over a finite lattice, by the fixpoint theory [Davey and Priestley, 2002], 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 behavioural type), be the least fixpoint $I^{(m)}$.

We can summarise the analyse of circularities as reported below. Starting with a behavioural type program $\big(\mathcal{L}, \Theta \cdot \mathsf{L}\big)$, and $k = 0$, proceed with the following steps:

1) compute $I^{(k+1)}(\Theta \cdot \mathsf{L})$: for every $\mathtt{m}_i$ in $\mathcal{L}$ compute $I^{(k+1)}(\mathtt{m}_i)$ as follow:

1.a) replace bound names with fresh names;

1.b) replace f_κ with $I^{(k)}(\mathtt{m})[\alpha, \overline{\mathbb{x}}, \kappa/\alpha', \overline{\mathbb{x}'}, X]$ where $\Theta(f) = \lambda X.\mathtt{m}(\alpha, \overline{\mathbb{x}}, X)$ and $\mathtt{m}(\alpha', \overline{\mathbb{x}'}, X)$ is defined in $\mathcal{L}$;

1.c) let $I^{(k+1)}(\mathtt{m}_i)$ be the transitive closure of the behavioural type resulting from (1.b) in which fresh names are projected out;

2) if $I^{(k+1)}(\Theta \cdot \mathsf{L}) \neq I^{(k)}(\Theta \cdot \mathsf{L})$ then $k = k + 1$ and go to (1), else exit.

Soundness of the analysis of circularities.

The following theorem states the correctness and completeness of the analysis of circularities. Similarly to the work of Giachino et al. [2014], 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 4.6.1. *Let $\left(\mathcal{L}, \Theta \cdot \mathsf{L}\right)$ be the behavioural type of a* gASP *program. Then $I_{\mathcal{L}}(\mathsf{L})$ has a circularity if and only if $I_{\mathcal{L}}(\Theta \cdot \mathsf{L})$ has a circularity.*

The proof can be found in Appendix A.2.

4.7 Properties of deadlock

With our analysis we want guarantee that, if the deadlock-freedom of a behavioural type program associated to a gASP program is assessed, then also the corresponding gASP program is guaranteed to be deadlock-free. In other words we want to prove that if the analysis shows that no deadlock is present in the behavioural 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 behavioural type of a gASP program then there is no deadlock in the original program.

Theorem 4.7.1. *Let P be a* gASP *program and cn be a configuration of its operational semantics, with behavioural type $\Theta \cdot \mathsf{L}$.*

1. *If* $\Theta \cdot L$ *has no circularity then cn is deadlock-free;*
2. *if* $cn \to cn'$ *and the behavioural type* $\Theta' \cdot L'$ *of* cn' *has a circularity, then a circularity is already present in* $\Theta \cdot L$*, the behavioural type of cn;*

The theorem is proven by relying on Theorem 4.5.1 (subject reduction) and on a crucial property of the later stage relation:

Theorem 4.7.2. *If* $\Theta \cdot L \succeq \Theta' \cdot L'$*, then* $I_{\mathcal{L}}(\Theta' \cdot L') \Subset I_{\mathcal{L}}(\Theta \cdot L)$.

The proof of Theorem 4.7.2, as well as the proofs of the foregoing theorems, is very similar to the corresponding one of Giachino et al. [2014].

4.8 Deadlock analysis for dining philosophers

In this section will be presented an example in which our analysis is applied to a non trivial problem. To this aim we chose to target the dining philosophers problem, which is presented in Listing 3.1.

Describing the application of our deadlock analysis on the dining philosopher problem, first the program will be typed by our behavioural type system, then we show the behavioural type analysis applied to the behavioural types obtained by the typing phase, and finally we present how the analysis of circularities is performed.

Behavioural typing

We start by showing the behavioural typing process. As we have said before the method of our program are typed one by one, we start typing the method `behave`.

```
2  Int behave(Fork fR, Fork fL)
3  { fut = fR.grab(fL);
4    aux = fut + 0;
5    c = this.behave(fR, fL) }
```

$$\texttt{behave}(\gamma, \alpha_b, \beta_{b'}, X) = (\nu\, g, g')\big(\Theta_{\texttt{behave}} \cdot b_\gamma + g_\star \ \textit{with}\ \Theta_{\texttt{behave}}(g) = \lambda X.\texttt{grab}(\alpha, \beta_{b'}, X) + g_\gamma + g'_\star \ \textit{with}\ \Theta_{\texttt{behave}}(g') = \lambda X.\texttt{behave}(\gamma, \alpha, \beta_{b'}, X) \ \&\ (X, \gamma)\big)$$

The behavioural type associated to the method `behave` is the following one:

$$\texttt{behave}(\gamma, \alpha_b, \beta_{b'}, X) = (\nu\, g, g')(\Theta_{\texttt{behave}} \cdot (b_\gamma + g_\star + g_\gamma + g'_\star) \,\&\, (X, \gamma))$$

in which

$$\Theta_{\texttt{behave}} = \{g \mapsto \lambda X.\texttt{grab}(\alpha, \beta_{b'}, X),\ g' \mapsto \lambda X.\texttt{behave}(\gamma, \alpha, \beta_{b'}, X)\}.$$

The behavioural type of a method is composed of two parts, the type of the method signature and the type of the body. The part that refers to the signature in this case is $\texttt{behave}(\gamma, \alpha_b, \beta_{b'}, X)$, in which we can see that to the current active object is given the name γ and the two forks `fR` and `fL` are typed as α_b and $\beta_{b'}$ respectively. The type of the body is the sum of four terms in conjunction with the pair (X, γ) which represents the potential synchronisation made by a synchroniser X on the current method. By applying the rule (T-Invk) we get the first two terms $b_\gamma + g_\star$. The first term indicates that we are synchronising the variable `fR`, because we need to know its value to be able to invoke the method `grab` on this active object. It is relevant to notice that this is a possible synchronisation, because at runtime that parameter can be instantiated with either a future or a value, we consider the case in which it is a future and we synchronise on it on the first access. During the analysis of circularities, when we know the instantiation of the parameters for our program, in case the this parameter is instantiated with a value the term, b_γ will be replaced by `0` and no synchronisation will be performed. The second term $g_\star$ corresponds to the invocation of the method `grab`, in fact g is a fresh future name created for this method invocation (see that g is bounded by $(\nu\, g, g')$) that in Θ is assigned to the behavioural type of the invocation of `grab` $\Theta(g) = \lambda X.\texttt{grab}(\alpha, \beta_{b'}, X)$. Looking at the behavioural type of the invocation of line 3 we can infer that this method will be called on the active object α, which is the active object stored in `fR`, and the parameter `fL` will be passed without any previous synchronisation (see that the type of `fL` is g').

The third term of the behavioural type of the body, g_γ, refers to line 4. This behaviour expresses that the synchronisation of the method invocation related to the future g will be performed by the active object γ which is the current active

object. The last term refers to the method invocation of `behave` of the current active object, that as usual creates a new future name g' assigned in Θ to a behaviour of the method invocation $\Theta(g') = \lambda X.\texttt{behave}(\gamma, \alpha, \beta_{b'}, X)$.

Now that we have obtained the type of the method `behave`, we move to type the method `grab`.

```
9  Int grab(Fork fL)
10 { fut = fL.grab_second();
11   aux = fut + 0;
12   return aux }
```

$$\begin{aligned}&\texttt{grab}(\alpha, \beta_{d'}, X) = (\nu\, d)\big(\Theta_{\texttt{grab}} \cdot \\ &\quad d'_\alpha + d_\star \ \textit{with}\ \Theta_{\texttt{grab}}(d) = \lambda X.\texttt{grab\_second}(\beta, X) \\ &+\ d_\alpha \\ &+\ 0 \\ &\&\ (X, \alpha)\big)\end{aligned}$$

In the same way the method `grab` is typed obtaining that:

$$\texttt{grab}(\alpha, \beta_{d'}, X) = (\nu\, d)(\ \Theta_{\texttt{grab}} \cdot \big(\, d'_\alpha + d_\star + d_\alpha) \,\&\, (X, \alpha)\ \big)$$

in which

$$\Theta_{\texttt{grab}} = \{\ d \mapsto \lambda X.\texttt{grab\_second}(\beta, X)\ \}.$$

In this case the first two terms of the behavioural type $d'_\alpha + d_\star$ refer to the method invocation in line 10. The first term is the possible synchronisation of `fL` while the second is the invocation of `grab_second` for which the fresh future name d is created and associated to the behaviour of the invocation $(\Theta_{\texttt{grab}}(d) = \lambda X.\texttt{grab\_second}(\beta, X))$. The last term d_α is the type of the synchronisation of the invocation just done (line 11).

The typing process of the method `grab_second`, reported below, is trivial because this method only returns a value.

```
17 Int grab_second() { return 0 }
```

The behavioural type of `grab_second` turns out to be the following one:

$$\texttt{grab\_second}(\alpha, X) = \big(\ \varnothing \cdot\ (X, \alpha)\ \big)$$

Now that we have defined the type of both the methods `behave`, `grab` and `grab_second` we want also to identify the behavioural type of the main functions.

```
17   // MAIN //
18  { fork1 = new Fork() ;                0
19    fork2 = new Fork() ;              + 0
20    p1 = new Philosopher();           + 0
21    p2 = new Philosopher();           + 0
22    fut1 = p1.behave(fork1, fork2);   + f⋆
23    fut2 = p2.behave(fork2, fork1) }  + f′⋆ & f⋆
```

The main function is typed by the rule (T-Program) obtaining the following behavioural type:

$$\Theta \cdot f_\star + f'_\star \ \& \ f_\star$$

in which

$$\Theta = \{f \mapsto \lambda X.\texttt{behave}(\gamma, \alpha, \beta, X),\ \ f \mapsto \lambda X.\texttt{behave}(\delta, \beta, \alpha, X)\}$$

While the statements in the lines between 18 to 21 are typed with an empty behaviour by the rule (T-New), the statements of line 22 and 23 are typed by the rule (T-Invk). It is relevant to notice that the behaviour of the second invocation is the parallel composition of two terms $f'_\star$ and $f_\star$. While the first term refers to the second invocation of `behave` of line 23, the second term is the result of $unsync(\Gamma)$ in the rule (T-Invk). This behaviour tells us that the invocation related to the future f' is potentially running in parallel with the method invocation related to f, which implies that all the active dependencies between active objects created by one method invocation can contribute to generate a circular dependency with the dependencies of the other method invocation.

Behavioural type analysis

Figure 4.13 shows the reduction of the behavioural type resulting from the application of the rule BT-red presented in section 4.4. Starting from the behavioural type of the main function $\Theta \cdot \mathsf{L}$, at each reduction step, applying BT-red, we replace one term of the behavioural type that refers to the invocation of a method by the behavioural type of the body of that method adequately

$$
\begin{aligned}
&\Theta \cdot (f_\star + f'_\star \ \& \ f_\star) \\
&\to \Theta_1 \cdot (\cdots + (g_\star + g_\gamma + g'_\star)\,\&(\star,\gamma) \ \& \ f'_\star + \cdots) \\
&\to \Theta_2 \cdot (\cdots + (g_\star + g_\gamma + g'_\star)\,\&(\star,\gamma) \ \& \ (g''_\star + g''_\delta + g'''_\star)\,\&(\star,\delta) + \cdots) \\
&\to \Theta_3 \cdot (\cdots + (g_\star + (d_\star + d_\alpha) \ \& \ (\gamma,\alpha) + g'_\star) \ \& \ (\star,\gamma) \ \& \ (g''_\star + g''_\delta + g'''_\star) \ \& \ (\star,\delta) + \cdots) \\
&\to \Theta_4 \cdot (\cdots + (g_\star + (d_\star + d_\alpha) \ \& \ (\gamma,\alpha) + g'_\star) \ \& \ (\star,\gamma) \ \& \ (g''_\star + (d'_\star + d'_\beta)\,\&(\delta,\beta) + \\
&g'''_\star)\,\&(\star,\delta) + \cdots) \\
&\to \Theta_4 \cdot (\cdots + (g_\star + (d_\star + (\alpha,\beta)) \ \& \ (\gamma,\alpha) + g'_\star) \ \& \ (\star,\gamma) \ \& \ (g''_\star + (d'_\star + d'_\beta) \\
&\& \ (\delta,\beta) + g'''_\star) \ \& \ (\star,\delta) + \cdots) \\
&\to \Theta_4 \cdot (\cdots + (g_\star + (d_\star + (\alpha,\beta) \ \& \ (\gamma,\alpha) + g'_\star) \ \& \ (\star,\gamma) \ \& \ (g''_\star + (d'_\star + (\beta,\alpha) \ \& \\
&(\delta,\beta) + g'''_\star)\,\&(\star,\delta) + \cdots) \\
&\to \ \cdots
\end{aligned}
$$

$$
\begin{aligned}
&\Theta = \{f \mapsto \lambda X.\texttt{behave}(\gamma,\alpha,\beta,X), f' \mapsto \lambda X.\texttt{behave}(\delta,\beta,\alpha,X)\} \\
&\Theta_1 = \Theta \cup \{g \mapsto \lambda X.\texttt{grab}(\alpha,\beta,X), g' \mapsto \lambda X.\texttt{behave}(\gamma,\alpha,\beta,X)\} \\
&\Theta_2 = \Theta_1 \cup \{g'' \mapsto \lambda X.\texttt{grab}(\beta,\alpha,X), g''' \mapsto \lambda X.\texttt{behave}(\delta,\beta,\alpha,X)\} \\
&\Theta_3 = \Theta_2 \cup \{d \mapsto \lambda X.\texttt{grab\_second}(\beta,X)\} \\
&\Theta_4 = \Theta_3 \cup \{d' \mapsto \lambda X.\texttt{grab\_second}(\alpha,X)\}
\end{aligned}
$$

Figure 4.13 – Behavioural type analysis.

instantiated. Knowing that $\Theta(f) = \lambda X.\texttt{behave}(\gamma,\alpha,\beta,X)$ and the behavioural type of the method `behave` is $\texttt{behave}(\gamma,\alpha_b,\beta_b,X) = (\nu\, g, g')(\Theta_{\texttt{behave}} \cdot \mathsf{L}_{\texttt{behave}})$, the first reduction replaces f_{main} with $\mathsf{L}_{\texttt{behave}}$ on which we replace the formal parameters with the actual parameters with $\mathsf{L}_{\texttt{behave}}[\gamma,\alpha,\beta,main/_{\gamma,\alpha_b,\beta_{b'},X}] = (g_\star + g_\gamma + g'_\star)\,\&(main,\gamma)$. Similarly, we compute the other steps of reduction. In Figure 4.13 we do not show the complete reduction of the behavioural type, that can be infinite; instead we guide the reduction through some steps that are relevant to reach the significant state shown in the last line. After simplification, we obtain a type in which one term has the following shape $\Theta_4 \cdot (\cdots + (\alpha,\beta)\,\&(\gamma,\alpha)\,\&(main,\gamma)\,\&(\beta,\alpha)\,\&(\delta,\beta)\,\&(\star,\delta) + \cdots)$ in which we can identify a circularity caused by the presence of the pairs (α,β) and (β,α).

Analysis of circularities

As we mentioned before, because the behavioural types are recursive, the behavioural type analysis may not terminate. For this reason we need to use a fixpoint technique that is able to detect circular dependencies in a finite time. Knowing the behavioural type of our program we can evaluate the flattening function for each method by Definition 9.

$$I^{(0)}(\texttt{grab\_second}) = \{\varnothing\},$$
$$I^{(1)}(\texttt{grab\_second}) = \{\{(X,\alpha)\}\}$$

$$I^{(0)}(\texttt{grab}) = \{\varnothing\}$$
$$I^{(1)}(\texttt{grab}) = \{\{(X,\alpha)\}\}$$
$$I^{(2)}(\texttt{grab}) = \{\{d'_\alpha,(X,\alpha)\},\ \{(\star,\beta),(X,\alpha)\}\},\ \{(\alpha,\beta),(X,\alpha)\},$$

$$I^{(0)}(\texttt{behave}) = \{\varnothing\}$$
$$I^{(1)}(\texttt{behave}) = \{\{(X,\gamma)\}\}$$
$$I^{(2)}(\texttt{behave}) = \{\{b_\gamma,(X,\gamma)\},\ \{(\star,\alpha),(X,\gamma)\},\ \{(\gamma,\alpha),(X,\gamma)\},\ \{(\star,\gamma)\}$$
$$I^{(3)}(\texttt{behave}) = \{\{b_\gamma,(X,\gamma)\},\ \{(\star,\alpha),(X,\gamma)\},\ \{(\gamma,\alpha),(X,\gamma)\},\ \{(\star,\gamma)\}$$
$$\{(\alpha,\beta),(X,\gamma)\},\{b'_\beta,(\gamma,\alpha),(X,\gamma)\},$$
$$\{(\gamma,\alpha),(X,\gamma)\},\{(\alpha,\beta),(\gamma,\alpha),(X,\gamma)\}\}$$

Finally we compute $I(\Theta \cdot \mathsf{L})$. We show below only a small relevant fragment of the result.

$$I(\Theta \cdot \mathsf{L}) \ = \ \{\ \cdots,(\star,\gamma),\ (\star,\delta),\ (\gamma,\alpha),\ (\delta,\beta),\ (\alpha,\beta),\ (\beta,\alpha),\ \ \cdots\}$$

In the state proposed we can see the the most relevant dependencies generated by the dining philosopher program. The pairs $(\star,\gamma)$ and $(\star,\delta)$ refer to the two invocations of `behave` done by the main program, respectively on the active object γ and δ. The presence of $\star$ in these dependencies states that the two invocations of `behave` have not been synchronised by the main function. The dependencies (γ,α) and (δ,β) are related to the invocations of `grab` performed by the active object γ on the active object α and by δ on β. We can notice that these two method invocations have been synchronised, then we have that the active object γ is waiting a result from the active object α and δ is waiting a result from β. The last two dependencies (α,β) and (β,α) refer to the invocations of `grab_second`

done by the two forks α and β. We notice here that each fork is waiting a result from the other fork, in fact α is waiting a result from β and β does the same with α. The circularity identified by the presence of these last two pairs shows that a deadlock can occur in our program.

4.9 Conclusion

In this chapter we have studied deadlock detection for `gASP` with two complementary techniques: a type system for extracting behavioural descriptions out of programs and a fixpoint technique for computing dependency models of behavioural descriptions. The work builds on and extends previous work where a similar technique has been used to detect deadlocks in pi-calculus [Giachino et al., 2014, Kobayashi and Laneve, 2017] and in an object-oriented language [Giachino and Laneve, 2013, 2014, Giachino et al., 2015].

The work presented in this chapter highlights the differences between explicit and implicit futures: while explicit futures enable the synchronisation upon the end of the execution of a method, implicit futures trigger synchronisation upon access to some data. The data-flow implicit synchronisation makes programming easier and execution more efficient as the program is only blocked if data is really needed and in an automatic way. However reasoning and finding deadlocks on a program with data-flow synchronisation is difficult for automatic tools. This chapter shows that the analysis of such data-flow synchronisation is possible and that the programming model can be at the same time easier and more efficient to program, while enabling automatic detection of deadlocks.

In particular, the technical contribution of presented in this chapter addresses in main problems. The first problem is directly related with the absence of explicit type for future variable. This characteristic of `gASP` do not allow us to know, at static point, if a parameter of the method that we are typing is a value or a future. To solve this problem, in the typing process of a method we have considered all the parameters as potential futures. As we have seen in Section 4.3, for each parameter we create a future that does not refer to any method invocation. We have delayed the identification about the nature of the parameter (future or value) and the binding of the future name to the method invocation to the behavioural

analysis defined in Section 4.4.

The second problem (discussed in Section 3.6) faced in this chapter, is related to the possible unbounded nesting of futures, which is a direct consequence of the possibility of returning future mixed with the absence of an explicit future type. This nesting of futures, in case of just a single synchronisation, may produce an unbounded set of dependency pairs. To solve this problem, that can not be solved during the typing phase, we moved the generation of the pair of dependencies to the behavioural type analysis. To make this possible we provide the behavioural type of each method with the special pair (X, α), where X is a place holder for the name of the active object that will synchronise the method, while α is the name of the active object running the current method. We also identify both method call and method synchronisation with the name of the future related to the method that we are invoking or synchronising. To this future name is associated a subscript that can be $\star$ in case a of method invocation, or the name of the current active object in case of a method synchronisation. The subscript associated to the method name, will allow us to instantiate the place holder X during the behavioural type analysis. In this way, the dependency pairs will be added only in case of synchronisation, while for method invocation the behavioural type analysis generates pairs as $(\star, \alpha)$.

The behavioural type analysis is a transition system that can generate an infinite number of states; Section 4.6 showed how the fixpoint analysis of circularities presented by Giachino et al. [2014] can be adapted to our current model.

In order to simplify our arguments, in this chapter, we focussed on a sublanguage where (i) fields do not contain futures, (ii) nor active objects, and (iii) futures are either returned or synchronised within a method body.

For allowing futures to be stored in fields, and relaxing restriction (i), we would need to track the identities of those futures, since they could be synchronised by any active object who has (direct or indirect) access to them. Thus, the type of an active object must be extended to a tuple containing also the types of all its field. In this case we have to resort to record types (as done by Giachino et al. [2015]). This would allow us to safely analyse *immutable* fields of any type. For enabling also the field update we would need to track also the effect of each method on the fields of the active objects taken as parameter. Parallel modification of the same field would result in a conflict that we must detect.

All these points will be examined in the next chapter in which we present our deadlock analysis extended to handle stateful active objects.

To relax the restriction (*ii*), the tuple is not sufficient anymore, because each of the field can contain in turn an active object with fields whose content have to be tracked. The management of recursive types is not handled in this thesis. The discussion about possible solutions for this problem is delayed to Chapter 6.

To remove restriction (iii) and allow pending unsynchronised futures at the end of a method execution, the type system would need a minor modification in the method typing rule, in order to collect also the unsynchronised behaviour corresponding to those unsynchronised futures. Then, dealing with the new type system will affect mostly the complexity of the analysis, as already solved in in the work of Giachino et al. [2015].

Chapter 5

Behavioural type analysis for stateful active object

Contents

In this chapter the gASP language is extended with the possibility of having stateful active objects. The possibility of storing futures in object fields makes the analysis of synchronisation patterns more 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

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

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

Figure 5.1 – gASP program with stateful active object

of the corresponding method must finish before the value of the future can be accessed. This chapter extends the static analysis technique described in Chapter 4 with the handling of stateful objects, by tracing the effects of methods on fields, including the storage of futures inside object fields. 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.

To illustrate synchronisation in active objects, consider the example in Figure 5.1. This program creates an active object (line 12) and calls the `addToStore` method asynchronously twice (lines 13 and 15). 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 another 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 programs with non-deterministic effects. These non-deterministic programs could also be analysed 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.

As already discussed in the previous chapter, to deal with method returning futures, we use a place-holder that represents the object that will access a future. Actually, the type system that will be presented below extends the one of Chapter 4 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 $method \rightsquigarrow object.\,field\_name$ for delegations.

The analysis of the behavioural type is performed by the solver defined by Giachino et al. [2016], 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.

This chapter will present first how we restrict `gASP` in order to focus on the main

aspects and make the presentation of the analysis more readable. The chapter will continue by presenting, in Section 5.2, the syntax of the behavioural type, the typing environments, the typing judgments, the effects functions and all the auxiliary functions that will be used in the type system, which will be presented immediately afterwards in Section 5.3. In Section 5.4 the behavioural type analysis will described first formally and then though a small example, then Section 5.5 shows how we prove the correctness of our analysis and finally in Section 5.6 we present how to adapt the analysis of circularities of Section 4.6 to handle delegations.

5.1 Restriction

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);

(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,(i) does not allow to have recursive data structures, and (ii) prevents computations running in parallel without any mean to synchronise on them. As already mentioned in Chapter 4, admitting futures that are never synchronised requires to collect the corresponding behaviours and add them to any possible continuation, like it has been done by Giachino et al. [2015]. How to remove (i) will be discussed in Section D.5.1.

5.2 Notations

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:

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

Basic types $\mathbb{r}$ 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 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 "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 $\mathtt{0}$, 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. 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 $\mathtt{0}$ being the identity for & and +. Behavioural types are equal up-to alpha renaming of bound names. Whenever parentheses are omitted, 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:

- $\mathit{names}(\Gamma) = \mathrm{dom}(\Gamma) \cup \{\alpha \mid \alpha[\overline{x : f}]^{[\checkmark]} \in \mathrm{im}(\Gamma)\}$;
- $\mathit{obj}(\overline{f})$ 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;
- $\mathtt{Fut}(\Gamma)$ is the set of future names in $\mathrm{dom}(\Gamma)$; $\mathtt{AFut}(\Gamma)$ and $\mathit{sFut}(\Gamma)$ are the subset of $\mathtt{Fut}(\Gamma)$ that contain future names f such that $\Gamma(f)$ is not "checkmarked" or checkmarked respectively;
- $\mathit{unsync}(\Gamma) = \&_{f \in \mathtt{AFut}(\Gamma)} f_\star$ is the performs the parallel composition of thebehavioral types of not-yet-synchronized method invocations, it collects the unsynchronized future of all the methods running in parallel;
- $\Gamma[f^\checkmark]$ 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 $\mathtt{AFut}(\Gamma_1) \cup \mathtt{AFut}(\Gamma_2)$.
- 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 `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

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

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

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

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

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

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

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

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

Figure 5.2 – Auxiliary functions for effects.

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 5.2: (1) *update operation with upper bound* ; (2) *merge of effects*;(3) *effects of unsynchronised methods*; (4-5) *compatibility*; (6) *effect instantiation* taking into account effect compatibility.

Judgements. The judgements used in the type system are:

- $\Gamma \vdash: m(f, \overline{g}, \Gamma_{\mathtt{m}}, X) \to (E, A)$ 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: $\Gamma_{\mathtt{m}} = (\Gamma|_{f \cup obj(\overline{g})})$, where Γ is the environment at invocation point. E, A are two environments that 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 the accesses to values and variables that do not require any synchronisation; values and variables are typed with future names, and E' is the update of E;
- $\Gamma \vdash f : \mathbb{f}$ for typing future names with future types;
- $\Gamma, E \overset{\oplus}{\vdash}_S e : \mathsf{L} \ \triangleright \ \Gamma', E'$ for typing expressions that must be synchronised, where S is the set of arguments of the 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.

5.3 Behavioural 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 $\Gamma(\mathit{this}) = \alpha[\cdots]$ 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 5.4. 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 synchronised 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 actual parameters are not linear and deals with them through the use of the *instanceof* function. In the signature, each parameter has a

values, variables and method names

- judgments used: $\Gamma \vdash \mathtt{m} : (\overline{f}, X, \Gamma') \to (E, A)$, $\Gamma \vdash v : \mathbb{r}$, and $\Gamma \vdash f : \mathbb{f}$

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

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

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

$$\text{(T-Synchronised)}\ \frac{\Gamma, E \vdash v : f \ \triangleright\ E' \qquad \Gamma \vdash f : \mathbb{f}^{\checkmark}}{\Gamma, E \overset{\oplus}{\vdash}_S v : \mathtt{0} \ \triangleright\ \Gamma, E'} \qquad \text{(T-Sync-Field)}\ \frac{\Gamma \vdash \textit{this} : \alpha[\cdots]^{\checkmark} \qquad \Gamma, E \vdash x : f \ \triangleright\ E' \qquad \Gamma \vdash f : g \rightsquigarrow \textit{this}.x \qquad \Gamma' = \Gamma[f^{\checkmark}]}{\Gamma, E \overset{\oplus}{\vdash}_S x : f_\alpha \,\&\, \textit{unsync}(\Gamma') \ \triangleright\ \Gamma', E'}$$

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

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

Figure 5.3 – Typing rules for names and synchronisations

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 con-

expressions with side effects

- judgement used: $\Gamma, E, A \vdash_S z : f, \mathsf{L} \triangleright \Gamma', E', A'$

$$\frac{\Gamma, E \vdash v : f \triangleright E'}{\Gamma, E, A \vdash_S v : f, \mathtt{0} \triangleright \Gamma, E', A}\ (\text{T-Atom}) \qquad \frac{\Gamma, E \overset{\oplus}{\vdash}_S v : \mathsf{L} \triangleright \Gamma', E' \quad \Gamma', E' \overset{\oplus}{\vdash}_S v' : \mathsf{L}' \triangleright \Gamma'', E''}{\Gamma, E, A \vdash_S v \oplus v' : \square, \mathsf{L} + \mathsf{L}' \triangleright \Gamma', E'', A}\ (\text{T-Expression})$$

$$\frac{\Gamma, E \vdash \overline{v} : \overline{g} \triangleright E' \qquad \beta, f \text{ fresh} \quad \overline{x} = \mathit{fields}(\mathtt{Act}) \quad \Gamma' = \Gamma[f \mapsto \beta[\overline{x : g}]^{\checkmark}]}{\Gamma, E, A \vdash_S \mathtt{new\ Act}(\overline{v}) : f, \mathtt{0} \triangleright \Gamma', E', A}\ (\text{T-New})$$

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

Figure 5.4 – Typing rules for expressions and expressions with side-effects.

flicts: two operations of write on the same object appeared due to the aliasing created between parameters.

The rules for typing synchronisations are defined at the bottom of Figure 5.4. In gASP, synchronisations are due to the evaluation of expressions e that are not variables. We use the notation $\overset{\oplus}{\vdash}$ 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

statements

- judgement used: $\Gamma, E, A \vdash_S s : \mathsf{L} \triangleright \Gamma', E', A$

$$\text{(T-Assign-Var-Exp)}\quad \frac{x \notin \mathit{fields}(\texttt{Act}) \qquad \Gamma, E, A \vdash_S z : f, \mathsf{L} \triangleright \Gamma', E', A'}{\Gamma, E, A \vdash_S x = z : \mathsf{L} \triangleright \Gamma'[x \mapsto f], E', A'}$$

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

$$\text{(T-Skip)}\quad \Gamma, E, A \vdash_S \texttt{skip} : \mathsf{0} \triangleright \Gamma, E, A \qquad \text{(T-Seq)}\quad \frac{\Gamma, E, A \vdash_S s_1 : \mathsf{L}_1 \triangleright \Gamma_1, E_1, A_1 \qquad \Gamma_1, E_1, A_1 \vdash_S s_2 : \mathsf{L}_2 \triangleright \Gamma_2, E_2, A_2}{\Gamma, E, A \vdash_S s_1; s_2 : \mathsf{L}_1 + \mathsf{L}_2 \triangleright \Gamma_2, E_2, A_2}$$

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

$$\text{(T-Return-Val)}\quad \frac{\Gamma, E \vdash v : f \triangleright E' \qquad \Gamma \vdash f : \mathbb{f}^{\checkmark}}{\Gamma, E, A \vdash_S \texttt{return}\ v : \mathsf{0} \triangleright \Gamma, E', A}$$

$$\text{(T-If)}\quad \frac{\Gamma, E, A \vdash_S e : \square, \mathsf{L} \triangleright \Gamma', E', A' \qquad \Gamma', E', A' \vdash_S s_1 : \mathsf{L}_1 \triangleright \Gamma_1, E_1, A_1 \qquad \Gamma', E', A' \vdash_S s_2 : \mathsf{L}_2 \triangleright \Gamma_2, E_2, A_2 \qquad \Gamma_1 =_{\texttt{unsync}} \Gamma_2}{\Gamma, E, A \vdash_S \texttt{if}\ e\ \{\, s_1 \,\}\ \texttt{else}\ \{\, s_2 \,\} : \mathsf{L} + \mathsf{L}_1 + \mathsf{L}_2 \triangleright \Gamma_1 + \Gamma_2, E_1 \sqcup E_2, A_1 \sqcup A_2}$$

Figure 5.5 – Typing rules for expressions, expressions with side-effects and statements.

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 let us compute the dependencies of the parallel methods during the analysis.

Figure 5.5 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

methods and programs

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

(T-Method)

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

(T-Program)

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

Figure 5.6 – Typing rules methods and programs.

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 $\square$ to prevent additional synchronisations on these futures.

The rules for statements are collected in Figure 5.5. 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 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 5.6 type methods and programs. 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 we use a simplified form of this environment. Instead of Γ will be used Θ which is defined as Γ, but does not map future names to object types and future results do not present information about effects $(\lambda X.\mathtt{m}(\overline{g}, X, \Gamma''))$. The environment Γ' will be also renamed as Φ.

Finally, a *behavioural type program* is defined as $\big(\mathcal{L},\ \Theta \cdot \mathsf{L}\big)$, 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 5.3.1. *Let us discuss the typing of the program in Figure 5.1. The behavioural type of this program is of the form:*

$$(\ \mathcal{L}, \Theta \cdot f_\star + f_{main} + f'_\star\ )$$

where:

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

We observe that the behavioural type of the main function performs two invocations of `addToStore`*, which are represented by the terms $f_\star$ and $f'_\star$ respectively. The first invocation, in line 13, is typed by by the rule* (T-Invk) *in the behavioural type $f_\star$ such that:*

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

As we can see from its behaviour, this method 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, in line 15, ($f'_\star$) is performed on the same object but `n` *stores the value written by the first invocation:*

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

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 synchronised (line 14 and behavioural type term f_{main}), 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`*.*

$$\mathcal{L}(\texttt{addToStore}) = (\beta, \mathit{this}, g, f, X) \to (\nu\, f')(\Theta_{\texttt{add}} \cdot \Phi_{\texttt{add}} \cdot L_{\texttt{add}})$$

where

$$\begin{aligned}L_{\texttt{add}} &= (\, g_\alpha + f'_\star + f'_X\,)\,\&(X, \beta) \qquad \Phi_{\texttt{add}} = [\, \mathit{this} \mapsto \beta[n : f']\,]\\ \Theta_{\texttt{add}} &= [\, f' \mapsto \lambda X.\texttt{store}(\mathit{this}, f, \square, X, [\, \mathit{this} \mapsto \beta[n : g]^{\checkmark}\,], \varnothing)\,]\end{aligned}$$

The behavioural type shows that the method `addToStore` *performs three main actions. The first action is the possible synchronisation of the parameter* g*, expressed by* g_α*. The second action is the invocation of the method* `store` *corresponding to future* f'*, which is expressed by the term* $f'_\star$*. The third action, which is the return of the result of the invocation of* `store`*, is expressed by the term* f'_X *stating that the* f' *is returned.*

For the sake of simplicity, we suppose that the method `store` *does not perform any relevant operation. The behavioural type of* `store` *is the following one:*

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

5.4 Behavioural type Analysis

As already stated in Chapter 4, the behavioural types are terms of a recursive model that expresses dependencies and features recursion and dynamic name creation.

The operational semantics of a behavioural type program $\left(\mathcal{L}, \Theta \cdot \mathsf{L}\right)$ is a transition system where *states* are pairs of a *future environment* Θ (mapping future names to method invocation instances) and a behavioural type L. We focus here on the way we address delegation types that are new relatively to the previous chapter. The evaluation of a behavioural types is defined by a transition relation between types $\Theta \cdot \mathsf{L}$ that follows the rules in Figure 5.7 and includes a specific rule for delegation types. We use *type contexts*:

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

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 $(\mathit{flat}(\overline{f}, \Gamma) \cup \overline{w}) \cap \overline{\varkappa'} = \varnothing$ and $\mathrm{dom}(\Theta) \cap \mathrm{dom}(\Theta'[\overline{\varkappa'}/_{\overline{\varkappa}}][\mathit{flat}(\overline{f}, \Gamma)/_{\overline{w}}]) = \varnothing$. The behavioural type L' is defined by a classical substitution. The substitution $[\mathit{flat}(\overline{f}, \Gamma)/_{\overline{w}}]$ replaces active object and

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

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

Figure 5.7 – Behavioural type reduction rules

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.

Example 5.4.1. *We show how a circularity appears when we apply the reduction rule for the program in Figure 5.1. The behavioural type of the program is the one shown in Section 5.3:*
the main function has type $(\ \mathcal{L}, \Theta \cdot f_\star + f_{main} + f'_\star\ )$ *where:*

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

`addToStore` *has behavioural type*

$$\mathcal{L}(\mathtt{addToStore}) = (\beta, \mathit{this}, g, f, X) \to (\nu\, f')(\Theta_{\mathtt{add}} \cdot \Phi_{\mathtt{add}} \cdot \mathsf{L}_{\mathtt{add}})$$

where

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

`store` *has behavioural type*

$$\mathcal{L}(\texttt{store}) = (\Gamma, this, g, f, X) \rightarrow (\varnothing \cdot \varnothing \cdot (X, \gamma))$$

We start from the behavioural 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` *in line 15. The rule* BT-Fun *replaces the behavioural type of method invocation $f'_\star$ with the body of* `addToStore` *properly instantiated and adds to the current future environment all the method invocation performed by* `addToStore` *properly instantiated. To perform* BT-Fun*, first, we have to instantiate the behavioural type $\mathsf{L}_{\texttt{add}}$ accordingly to the method invocation related to* `f'`*, which is:*

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

then we have to build the substitution

$$\mathsf{L}_{\texttt{add}}[^{h}/_{f'}][^{\star}/_{X}][\alpha, g, g', ^{\Box}/_{\beta}, this, g, f]$$

that instantiates $\mathsf{L}_{\texttt{add}}$ replacing the formal parameters with the actual parameter defined $\Theta(f')$, and we obtain the behaviour:

$$(g'_\alpha + h_\star + h_X\,)\,\&(\star, \alpha).$$

Now that the behavioural type that should replace f' has been obtained we have to add to Θ the method invocation performed by `addToStore` *instantiated accordingly to the invocation $\Theta(f')$. We have that*

$$\Theta' = \Theta + \Theta'_{\texttt{add}}$$

where $\Theta'_{\texttt{add}}$ is obtained from $\Theta_{\texttt{add}}$ applying the same substitution applied to $\mathsf{L}_{\texttt{add}}$. 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 synchronisation of the field `n` *(line 4) 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*. 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 (the future stated in the delegation $\Theta'(g') = f \rightsquigarrow g.n$), with the substitution $[h/f'][\alpha, g, \square, \square/\beta, this, g, f]$ and we obtain:*

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

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 behavioural type of* `store` *adequately instantiated and obtain:*

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

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.*

5.5 Type soundness

To demonstrate the correctness of the type system and the analysis we decided to separated the part of the type system concerning to the deadlock analysis, that will be discussed in below, and the part related to the effect analysis, which is discussed in Section 5.5.2.

5.5.1 Deadlock type soundness

In the following we will focus only on the deadlock analysis aspect starting from the hypothesis that the analysed program has only deterministic effects (see Definition 4). The correctness of our system guarantees that, if the analysis accesses the deadlock-freedom of a behavioural type program associated to a `gASP` program with deterministic effects, then the `gASP` program is guaranteed to be deadlock-free. 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 $\Theta \cdot \mathsf{L}$ of cn and the type $\Theta' \cdot \mathsf{L}'$ of cn'. Informally, we are proving that if the analysis shows that no deadlock is present in the behavioural 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. Therefore, a subject reduction for the type system of Section 5.3 requires the extension of the typing to configurations; and a *later-stage* relation between behavioural types.

Runtime type system

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

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

configuration and processes

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

(TR-Future-Undef)
$$\frac{\Delta \vdash f : \lambda X.\mathtt{m}(\overline{g}, X, \Delta_{\mathtt{m}}, E_{\mathtt{m}})}{\Delta \vdash f(\bot) : \mathsf{0}}$$

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

(TR-Actor)
$$\frac{\Delta(\alpha) = \alpha[\overline{y : f}] \qquad \Delta \vdash \overline{v} : \overline{f} \qquad \Delta' = \Delta + this : \alpha[\overline{y : f}] \qquad \Delta' \vdash p : \mathsf{K}_0 \qquad \forall i \in 1..n.\, \Delta' \vdash q_i : \mathsf{K}_i}{\Delta \vdash \alpha(\{\overline{y \mapsto v}\}, p, \{q_1, \cdots, q_n\}) : \overset{n}{\underset{i=0}{\&}} \mathsf{K}_i}$$

(TR-Parallel)
$$\frac{\Delta \vdash cn_1 : \mathsf{K}_1 \qquad \Delta \vdash cn_2 : \mathsf{K}_2}{\Delta \vdash cn_1\ cn_2 : \mathsf{K}_1 \,\&\, \mathsf{K}_2}$$

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

Figure 5.8 – Runtime typing rules for configuration

The main novelty of this type syntax it the extended futures 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).

Later-stage

The *later-stage* relation $\succeq_\Delta$ is a syntactic relationship between behavioural types. Formally, the later-stage relation is the least congruence with respect to runtime behavioural type that contains the rules in Figure 5.9. We can simplify the basic laws of the later-stage relation saying that a method invocation is larger than the instantiation of its method behaviour (LS-Invk), and a sum type is larger than each element of the sum.

$$\text{(LS-RuntimeEmpty)}\quad \mathsf{K} \succeq_\Delta \mathsf{0} \qquad \text{(LS-Global)}\quad \frac{\mathsf{K}_1 \succeq_\Delta \mathsf{K}_1'}{\mathsf{K}_1 \,\&\, \mathsf{K} \succeq_\Delta \mathsf{K}_1' \,\&\, \mathsf{K}} \qquad \text{(LS-Behavior)}\quad \frac{\mathsf{K} = (\nu\,\overline{\varphi})(\Theta \cdot \mathsf{L}) \qquad \mathsf{K}' = (\nu\,\overline{\varphi'})(\Theta \cdot \mathsf{L}') \qquad \mathsf{L} \succeq_\Delta \mathsf{L}'}{\mathsf{K} \succeq_\Delta \mathsf{K}'}$$

$$\text{(LS-Empty)}\quad \mathsf{L} \succeq_\Delta \mathsf{0} \qquad \text{(LS-Plus)}\quad \mathsf{L}_1 + \mathsf{L}_2 \succeq_\Delta \mathsf{L}_i \qquad \text{(LS-Parallel)}\quad \frac{\mathsf{L}_1 \succeq \mathsf{L}_1'}{\mathsf{L}_1 \,\&\, \mathsf{L} \succeq_\Delta \mathsf{L}_1' \,\&\, \mathsf{L}}$$

$$\text{(LS-Invk)}\quad \frac{\begin{array}{c}\Delta(f) = \lambda X.\mathtt{m}(\alpha', \overline{\mathbb{z}'}, X) \\ \Delta(m) = \big((\alpha, \overline{\mathbb{x}}, X) \to \mathbb{r}, \mathsf{K}_\mathtt{m}\big) \qquad \Delta \vdash \mathtt{m} : (\alpha', \overline{\mathbb{x}'}, X) \to \mathbb{r}' \\ \overline{\alpha} = fn(\mathsf{K}) \setminus fn(\alpha, \overline{\mathbb{x}}, \mathbb{r}) \qquad \overline{\alpha'} \cup fn(\alpha', \overline{\mathbb{x}'}, \mathbb{r}') = \varnothing\end{array}}{(\nu\,\overline{\varphi})(\Theta \cdot (f_\kappa \,\&\, \mathsf{L}) + \mathsf{L}_s) \succeq_\Delta (\nu\,\overline{\varphi})(\Theta \cdot \mathsf{L}_s) \,\&\, \mathsf{K}_\mathtt{m}[\overline{\alpha'}/\overline{\alpha}][\alpha', \overline{\mathbb{x}'}, \mathbb{r}'/\alpha, \overline{\mathbb{x}}, \mathbb{r}]}$$

Figure 5.9 – The later-stage relation.

Deadlock type soundness

Since we have defined both type system for runtime configuration and later stage relation, we can formally state the main theorem that guarantees the soundness of the behavioural type system.

Theorem 5.5.1. *Let P be a* gASP *program with deterministic effects (see Definition 4) and cn be a configuration of its operational semantics, with behavioural type K.*

1. *If K has no circularity then cn is deadlock-free;*
2. *if $cn \to cn'$ and the behavioural type K' of cn' has a circularity, then a circularity is already present in K, the behavioural type of cn;*

The proof of this theorem is done proving *(1)* through the Lemma 5.5.2 and proving *(2)* through the proof of Theorem 5.5.3.

Lemma 5.5.2. *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 3, 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-* (Appendix B.1), the behavioural type of cn' is $(\nu\,\overline{\varkappa})\big(\Theta \cdot (f_\alpha + \mathsf{L}_s)\,\&(X, \alpha)\big)\ \&\ (\overset{n}{\underset{i=1}{\&}}\mathsf{K}_i)$.

Having that:

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

we can infer 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)

□

Theorem 5.5.3 (Subject Reduction)**.** *Let $\Delta \vdash_R cn : \mathsf{K}$ and $cn \to cn'$. Then there exist Δ', K', and an injective renaming of actor and future names $\imath$ such that*

- $\Delta' \vdash_R cn' : \mathsf{K}'$ *and*
- $\imath(\mathsf{K}) \succeq_\Delta \mathsf{K}'$

The proof is a case analysis on the reduction rule used in $cn \to cn'$ and can be found in Appendix B.1.4.

5.5.2 Effect type soundness

In this section we want prove the correctness of our effect analysis, which is stated by the following theorem.

Theorem 5.5.4. *Let P be a* gASP *program, if $\Gamma \vdash P$ then P has deterministic effects.*

In order to prove this theorem we need to extend our type system to runtime configurations that is presented in Figure 5.10 - 5.11 and Figure 5.12.

configuration and processes: $\Delta \vdash cn \ \triangleright \ E$ and $\Delta, E, A \vdash p \ \triangleright \ E, A$

$$\frac{\Delta \vdash: f\lambda X.\mathtt{m}(\overline{g}, \Delta_{\mathtt{m}}, E_{\mathtt{m}})}{\Delta \vdash f(\bot)} \ \text{(TR-Future-Undef)} \qquad \frac{\Delta \vdash: f\lambda X.\mathtt{m}(\overline{g}, \Delta_{\mathtt{m}}, E_{\mathtt{m}}) \quad \Delta \vdash: wf}{\Delta \vdash f(w)} \ \text{(TR-Future-Eval)} \qquad \frac{\Delta \vdash cn_1 \qquad \Delta \vdash cn_2}{\Delta \vdash cn_1 \ cn_2} \ \text{(TR-Parallel)}$$

$$\frac{\begin{array}{c}\Delta(\alpha) = \alpha[\overline{y : f}] \qquad \Delta \vdash \overline{v} : \overline{f} \qquad \Delta' = \Delta + this : \alpha[\overline{y : f}] \\ \Delta', \varnothing, \varnothing \vdash p \ \triangleright \ E_0, A_0 \\ \forall i \in 1..n.\, \Delta', [this \mapsto \varnothing], \varnothing \vdash q_i \ \triangleright \ E_i, A_i \qquad (E_k \# E_j)^{k,j \in [1,n] \wedge k \neq j}\end{array}}{\Delta \vdash \alpha(\{\overline{y \mapsto v}\}, p, \{q_1, \cdots, q_n\})} \ \text{(TR-Actor)}$$

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

Figure 5.10 – Typing rules for runtime configurations.

Runtime Type System for Effect analysis (typing rules)

In order to analyse effect for runtime configuration we define a runtime type system. This type system is a simpler version of the one given in section 5.3 where we are focusing only on the effect analysis leaving out all the aspects related with deadlock. This is the reason why the behavioural type syntax and also the typing judgments are simpler then the corresponding shown in Section 5.2.

$\mathtt{r}$	$::=$	$\square \mid f \mid \alpha[\overline{x : f}]$	basic type
$\mathtt{f}$	$::=$	$\mathtt{r} \mid \lambda X.\mathtt{m}(f, \overline{g}, \Gamma, E)$	future type
F	$::=$	$f \mid {}^{s}f$	extended futures

The main differences with the type system presented in Section 5.3 are:

- future types does not present delegation and future results do not contain the place holder X;
- we define *extended futures* F which are introduced for distinguishing two kinds of future names: i) f that has been used in the type system as a

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

$$\frac{v \quad \textit{integer-value} \text{ or } \texttt{null}}{\Delta, E, A \vdash v : \square \triangleright E} \text{(TR-Val-Int)} \qquad \frac{\Delta(x) = f}{\Delta, E, A \vdash x : f \triangleright E} \text{(TR-Var)} \qquad \frac{\Delta(\mathit{this}) = \alpha[x : f, \cdots] \quad E' = E[\alpha.x \mapsto^{\sqcup} \mathtt{r}]}{\Delta, E \vdash x : f \triangleright E'} \text{(TR-Field)}$$

$$\frac{\Delta(f) = \mathbb{f}}{\Delta \vdash f : \mathbb{f}} \text{(TR-Var)} \qquad \frac{\Delta(\mathtt{m}) = (\overline{f}, \Delta_{\mathtt{m}}) \to (E, A) \qquad \sigma \text{ renaming} \quad E' = \mathit{instanceof}(E, \sigma) \qquad A' = \mathit{instanceof}(A, \sigma)}{\Delta \vdash m : (\sigma(\overline{f}), \Delta_{\mathtt{m}} \circ \sigma) \to (E', A')} \text{(TR-Method-Sign)}$$

synchronizations: $\Delta, E, A \overset{\oplus}{\vdash}_S v \triangleright \Delta', E', A'$

$$\frac{\Delta, E, A \vdash x : f \triangleright E' \quad \Delta \vdash f : \lambda X.\mathtt{m}(f, \overline{g}, \Delta_{\mathtt{m}}, E_{\mathtt{m}})}{\Delta, E \overset{\oplus}{\vdash}_S x \triangleright \Delta, E' \sqcup E_{\mathtt{m}}|_S} \text{(TR-Sync-Invk)} \qquad \frac{\Delta, E \vdash v : f \triangleright E' \qquad \Delta \vdash f : \mathbb{f} \quad \mathbb{f} \neq \lambda X.\mathtt{m}(f, \overline{g}, \Delta_{\mathtt{m}}, E_{\mathtt{m}})}{\Delta, E \overset{\oplus}{\vdash}_S v \triangleright \Delta, E'} \text{(TR-Synchronized)}$$

Figure 5.11 – Runtime typing rules for expressions, addresses and synchronisations.

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).

– now judgments only associate types to atoms and expressions and anymore behavioural type to statements.

expressions with side effects: $\Delta, E, A \vdash S : z \triangleright f\Delta', E', A'$

The proof of the Theorem 5.5.4 can be split in two steps. The former involves in the proof that given a program P and a configuration cn, reached during the execution of P, our type system can type cn only if cn has has non deterministic effects, as we formally state in the following lemma.

Lemma 5.5.5. *Let P be a* gASP*, cn be a runtime configuration, and let Δ be such that $\Delta \vdash cn$; then cn has a queue with deterministic effects.*

By Definition 4 we know that a configuration cn has *deterministic effects* if every active object of this configuration has a *queue with deterministic effects*, then to prove lemma 5.5.5 we need to prove that given an active object

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

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

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

(TR-Expression)
$$\frac{\begin{array}{c}\Delta, E \overset{\oplus}{\vdash} \ v \ \triangleright\ \Delta', E' \\ \Delta', E' \overset{\oplus}{\vdash} \ v' \ \triangleright\ \Delta'', E''\end{array}}{\Delta, E, A \vdash_S v \oplus v' : \square \ \triangleright\ \Delta'', E'', A}$$

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

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

statements: $\Gamma, E, A \vdash_S s \ \triangleright\ \Gamma', E', A$

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

(TR-Assign-Field-Exp)
$$\frac{\begin{array}{c}x \in \mathit{fields}(\mathtt{Act}) \qquad \Delta \vdash \mathit{this} : \alpha[\cdots] \\ \Delta, E, A \vdash z : F \ \triangleright\ \Delta', E', A' \\ \mathit{Effects}(\Delta')(\alpha) \# x^{\mathtt{w}} \qquad A'(\alpha) \# x^{\mathtt{w}}\end{array}}{\Delta, E, A \vdash_S x = z \ \triangleright\ \Delta'[\mathit{this}.x \mapsto F], E'[\alpha.x \mapsto^{\sqcup} \mathtt{w}], A'}$$

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

(TR-Seq)
$$\frac{\begin{array}{c}\Delta, E, A \vdash s_1 \ \triangleright\ \Delta_1, E_1, A_1 \\ \Delta_1, E_1, A_1 \vdash s_2 \ \triangleright\ \Delta_2, E_2, A_2\end{array}}{\Delta, E, A \vdash_S s_1; s_2 \ \triangleright\ \Delta_2, E_2, A_2}$$

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

(TR-If)
$$\frac{\begin{array}{c}\Delta, E, A \vdash e : f \ \triangleright\ \Delta', E', A' \\ \Delta', E', A' \vdash s_1 \ \triangleright\ \Delta_1, E_1, A_1 \qquad \Delta', E', A' \vdash s_2 \ \triangleright\ \Delta_2, E_2, A_2 \\ \Delta_1 =_{\mathtt{unsync}} \Delta_2\end{array}}{\Delta, E, A \vdash_S \mathtt{if}\ e\ \{\, s_1 \,\}\ \mathtt{else}\ \{\, s_2 \,\} \ \triangleright\ \Delta_1 + \Delta_2, E_1 \sqcup E_2, A_1 \sqcup A_2}$$

Figure 5.12 – Runtime typing rules for expressions with side-effects and statements.

$\alpha(a, p, \{q_1, \cdots, q_n\})$ if Δ exists such that $\Delta \vdash \alpha(a, p, \{q_1, \cdots, q_n\})$ then $\alpha(a, p, \{q_1, \cdots, q_n\})$ has a queue with deterministic effects.

Proof. Let $\alpha(a, p, \{q_1, \cdots, q_n\})$ be an active object with non deterministic effects, this implies, by Definition 4, that one of the following predicates holds:

1. $x^{\mathtt{w}} \in q_i \wedge x^{\mathtt{w}} \in q_j$ where $x \in dom(a)$ and $i, j \in [1, n]$ with $i \neq j$
2. $x^{\mathtt{w}} \in q_i \wedge x^{\mathtt{r}} \in q_j$ where $x \in dom(a)$ and $i, j \in [1, n]$ with $i \neq j$

Let us also suppose that there exists Δ such that $\Delta \vdash \alpha(a, p, \{q_1, \cdots, q_n\})$.

Case 1: $x^{\mathtt{w}} \in q_i \wedge x^{\mathtt{w}} \in q_j$.

For each pair of $i, j \in [1, n]$ with $i \neq j$, by rule TR-Actor and TR-Process we gain that there exist Δ'_i and Δ'_j that extend Δ such that $\Delta'_i, \varnothing, \varnothing \vdash_{S_i} q_i \ \triangleright \ E_i, A_i$ and $\Delta'_j, \varnothing, \varnothing \vdash_{S_j} q_j \ \triangleright \ E_j, A_j$, and also the rule TR-Actor checks if the effects of q_i and the effects of q_j are compatible ($E_i \# E_j$).
By (1) we have that there exist $x \in dom(a)$ (which is equal to say that $x \in \mathit{fields}(\mathtt{Act})$) such that $x^{\mathtt{w}} \in q_i$ and $x^{\mathtt{w}} \in q_j$ that by definition means that there exist a statement s_i in q_i and a statement s_j in q_j such that $x = z$ for some z. To type the statement $x = z$ in q_i and q_j we have to apply the rule TR-Assign-Field which implies that $E_i(\alpha.x) = \mathtt{w}$ and $E_j(\alpha.x) = \mathtt{w}$ that makes the condition $E_i \# E_j$ false.

Case 2: $x^{\mathtt{w}} \in q_i \wedge x^{\mathtt{r}} \in q_j$.

For each pair of $i, j \in [1, n]$ with $i \neq j$, by rule TR-Actor and TR-Process we gain that there exist Δ'_i and Δ'_j that extend Δ such that $\Delta'_i, \varnothing, \varnothing \vdash_{S_i} q_i \ \triangleright \ E_i, A_i$ and $\Delta'_j, \varnothing, \varnothing \vdash_{S_j} q_j \ \triangleright \ E_j, A_j$, and also the rule TR-Actor checks if the effects of q_i and the effects of q_j are compatible ($E_i \# E_j$).
By (2) we have that there exist $x \in dom(a)$ such that $x^{\mathtt{w}} \in q_i$ and $x^{\mathtt{r}} \in q_j$ that by definition means that there exist a statement s_i in q_i such that $x = z$ for some z and a statement s_j in q_j such that x appear in the left side of an assignment or in the guard of an if statement. To type the statement $x = z$ in q_i we have to apply the rule TR-Assign-Field which implies that $E_i(\alpha.x) = \mathtt{w}$, whereas to type s_j we are going to apply the rule TR-Field which implies that $E_j(\alpha.x) = \mathtt{r}$. that makes the condition $E_i \# E_j$ false.

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

Figure 5.13 – Definition of flattening function.

As we have seen it is not possible that our type system is able to type a configuration if it presents an active object with a queue with non deterministic effects.

□

The later step to prove Theorem 5.5.4 consist in proving that if we type a configuration cn and $cn \to cn'$, than there exist an environment Δ' such that Δ' types cn', as formalised in the following lemma.

Lemma 5.5.6. *Let $\Delta \vdash cn$ and $cn \to cn'$. Then there exist Δ' such that $\Delta' \vdash cn'$.*

The proof of this lemma is a case analysis on the reduction rule used in $cn \to cn'$ and it is presented in Appendix C.1.2

5.6 Analysis of circularities

The analysis of circularities in this case is an adaptation of the one presented in Chapter 4. Let $\mathcal{L}$ be a set of method definitions and let $I(\cdot)$, called *flattening*, be a function either on future environments and behavioural types or on method names that (i) maps a method name $\mathtt{m}$ defined in $\mathcal{L}$ to elements $\mathcal{R}^{(1)}$ and (ii) is defined on behavioural types as follows in Figure 5.13.

The definition of Chapter 4 is extended with the case in which a future refers to a delegation. This case as the one in which the future refers to a method invocation

[1] $\mathcal{R}$ is defined in the Preliminaries of Section 4.6

is defined as $I(\Theta \cdot f_\kappa) = I(\mathtt{m}')[\mathit{flat}(\overline{g'}, \Gamma'), \kappa/\overline{w}, X]$. In case of method invocation we know directly from $\Theta(f)$ the name of the method invoked and the parameter with which this method has been invoked. In case f refers to a delegation we need first to discover for which future this delegation stand for. Paying attention to the condition related to this case we can notice that to infer which is the method invocation represented by the delegation we apply the same steps defined in the rule (BT-Field).

5.7 Conclusion

In this chapter we have defined a technique based on behavioural type systems for analysing the absence of deadlocks in a program with stateful active objects. As we have already seen in this chapter, stateful active objects bring with them two main problems: nondeterminism in the typing of the expression when the program has queues with nondeterministic effects; and the difficulty to relate futures with their method invocations, when futures created in a different context are passed to the current method as parameters. To face these two problem we extended the type system presented in Chapter 4. This extension tracks the effects that each method can have on the fields of an object received as parameters, and introduces delegations.

Solving the problem of nondeterminism without loosing in precision requires an extension of the behavioural type associated to the methods. We have extended the behavioural type of a method adding a function that collects the effect of the method. This function maps the future names of the parameters accessed during the execution of the method to the a set of field names. These field names are labelled either with `r` or `w` to indicate an access in reading or writing respectively. Thanks to this additional information, we are able to type only the programs with deterministic effects. To guarantee the correctness of this effect analysis we also proved that: if our type system is able to define the type of a program, then this program has no active object that has a queue with nondeterministic effects in any of its possible executions.

The impossibility of having queue with nondeterministic effects guarantees that if a method stores a future in the field of an argument, then the next access to

the field should occur after the end of that method. The information about effect, provided by the effect analysis, allows us to infer if the method we are synchronising has stored a future in the field of an argument. However, it is still not possible to determinate the name of the future stored and the method invocation related to this future outside the context in which this future has been created. In the solution proposed in this chapter, every time we synchronise a method that has writing effects on a field of an argument, we type the field of that object with a fresh future and we associate this future to a delegation. The delegation, which has the form $g \rightsquigarrow o.\, x$, indicates that the method related to the future g has stored a future in the field x of the object o. Delegations allow us to move to the analysis of the behaviour, in which we have more global information, the identification of the method invocation that we did not know at static time.

We also present how the behavioural types that are obtained by the type system are analysed, and how to extend the analysis of circularities to identify potential deadlocks in case of stateful active objects.

In this chapter we also presented a novel paradigm in which all variables and values are typed as a futures, and these futures are in turn typed to future types. Future types can be check-marked if they refer to values or synchronised method, and not check-marked if they refer to a potential unsynchronised method invocation. Compared to the approach used in the previous chapter, this one allows us to better deal with implicit futures. For instance, aliasing is handled more easily because all the aliases point on the same future, then in case of a future is synchronised we have to the update the type of the future only once, and not for each alias that store that future.

Chapter 6

Conclusion

Contents

6.1 Summary

In the era in which software are not anymore centralised, and cloud computing and IoT paradigms are leading in the industry and research environment, Actor and Active object model are becoming increasingly prominent.

Despite these two models facilitate the development of complex distributed systems, the implementation of these systems still remains a non trivial task. This is due to the fact that well-known concurrency bugs such as deadlocks, race conditions, and data races are not rare in distributed systems.

With this thesis we help programmers that use the Active object model to implement complex distributed systems, by providing a software analysis technique to analyse synchronisation patterns. The analysis proposed is able, without requir-

ing human interaction, to detect the absence of deadlocks and to analyse effects. Our aim was to provide a program analysis that is modular and combines several different techniques.

The first contribution presented in this thesis proposes a static analysis technique based on behavioural types. The static analysis we developed is composed by: a behavioural type system that associates behavioural types to program methods; a behavioural type analysis that translates behavioural types into a potential unbounded graph of "waiting" dependences; and an analysis of circularities that takes as input the behavioural type program and detects deadlocks in finite time.

The development of this analysis is focused on how to handle the implicit future types and *wait-by-necessity* synchronisations. In fact, the combination of these two features may require to define an unbounded set of dependencies between active objects, if we are synchronising a recursive methods that returns a future.

The first problem is to identify which parameters are futures or values. It has been solved by typing every parameter as a potential future not associated to any method invocation. Then, we have delayed the identification about the nature of the parameter (future or value) and the binding of the future name to the method invocation to the behavioural analysis.

The second problem, related to the nesting of futures that may produce an unbounded set of dependency pairs, has been solved by moving the generation of the dependencies to the behavioural type analysis. We provide the behavioural type of each method with the special pair (X, α), where X is a place holder for the name of the active object that will synchronise the method, while α is the name of the active object running the current method. The place holder X will be instantiated, during the behavioural type analysis, only in case of synchronisations and not for method invocations.

The second contribution of this thesis is the development of an effect analysis, based on behavioural type systems, to detect the presence of race conditions that may introduce nondeterminism in the execution of an active-object program. The behavioural type system associates to each method a function that traces effects. This function maps the future names of the parameters accessed during the execution of the method, to a set of field names. These field names are labelled either with `r` or `w`, indicating respectively an access in reading or writing. Thanks to this

additional information we are able to type only the programs with deterministic effects. The impossibility of having queues with nondeterministic effects guarantees that: if a method stores a future in the field of an argument, then the next access to the field should occur after the end of that method.

The third main contribution of this thesis is an extension of the first contribution in order to analyse active-object program with stateful active objects. This analysis takes into account the information provided by the effect analysis and identifies the absence of deadlocks. The static analysis we developed, as the previous one, is composed by: a behavioural type system; a behavioural type analysis; and an adaptation of the analysis of circularities. The main problem tackled by this analysis is the tracking of futures stored in object fields. In fact, the information provided by the effect analysis only allows us to know when a method already synchronised has stored a future inside a field of an argument. How to know the identity of this future and the method invocation related to it outside the context that invoked this method, was still an open problem. This problem has been solved introducing delegations. The delegations that has the form $g \rightsquigarrow o.\, x$, indicates that the method related to the future g has stored a future in the field x of the object o. Delegations allow us to move the the identification of the method invocation, for a future stored in a field, to the analysis of the behaviour, in which we have more global information.

Overall, we build a proven and thorough analysis for synchronisation patterns for programs implemented using the Active object model. We developed two analysis: a deadlock analysis technique based on behavioural types in which we developed a behavioural type system, a behavioural type analysis, and we have extended the analysis of circularities presented by Giachino et al. [2014], Kobayashi and Laneve [2017] to handle our behavioural types; and an effect analysis also based on behavioural types. We tackle in this thesis the problems related to analyse an Actor model with implicit future types, *wait-by-necessity* synchronisation and stateful active objects. Implicit future types and *wait-by-necessity* synchronisation require the creation of unbounded sets of dependency pairs in case of synchronisations, and also the identification of futures and synchronisation points. The management of stateful active object requires to handled or to detect nondeterminism caused by the nondeterministic order of accesses on fields, and

also it requires the tracking of future names and method invocations which are synchronised in a context different from the one in which they are created.

6.2 Perspectives

The results we presented in this thesis are promising for the Active object model and its verification. The work we have done is a suitable starting point of several research topics, that we will discuss in the next subsections.

6.2.1 Improvement of the analysis

When it comes to future works we immediately think about possible improvements to our analysis. The first way to improve our analysis is certainly related to the removal of some restrictions. Most of the restriction imposed to `gASP` and stated in Section 4.1 and Section 5.1 have been imposed in order to simplify the behavioural type system. How to remove those restrictions is well know for us and it has been discussed in Section 4.9. The only restriction that is present in both of our works, and which has not been investigated in this thesis, is the impossibility of having recursive data types. In fact, in both Section 4.1 and Section 5.1 it is stated that the field of an object have to be of type `Int`, then they can be only integer or a future of an integer. How to manage recursive data type is a non trivial task, in fact it requires to extend our analysis in order to make it able to handle unbounded object records.

Thanks to the modularity of our analysis, we think that the management of unbounded object records can be done by a separated analysis. Such analysis should be able to flatten the structure of the objects, in order to give to our analysis object types in the form of a record, as it is now. The precise technique that can be used to flatten unbounded data structures as not been invested.

The improvement of our analysis can be also investigated outside the field of restrictions. Our analysis, losing a bit in generality, can gain in precision if a precise scheduling algorithm is fixed. Taking into account a precise scheduling approach (i.e FIFO scheduling), gives us much more information related to the real interleaving of processes. As we have seen in this thesis the amount of information

a) Deadlock

```
1  Int foo( ) { return 1 }
2
3  Int bar(Int k)
4  { k = k + 1;
5    return k }
6
7  // MAIN //
8  { x = this.foo();
9    y = this.bar(x) }
```

b) Effect

```
1  Int k
2
3  Int foo( )
4  { k = this.m();
5    return 1 }
6
7  Int bar( )
8  { z = k + 1;
9    return k }
10
11 // MAIN //
12 { x = this.foo();
13   y = this.bar() }
```

Figure 6.1 – False positive in deadlock and effect analysis for FIFO scheduling

that the analysis is able to extract from the program is directly proportional to its precision. In fact, on the current state, both deadlock and effect analysis take into account that two method invocations, even if performed by the same method, can be scheduled in any order. In Figure 6.1 have been presented two example in which our deadlock analysis (Figure 6.1.a) and our effect analysis (Figure 6.1.b) give as result a false positive in case of FIFO scheduling.

Let us to start with the program of Figure 6.1.a. In this program we have a main function that invokes the methods `foo` and `bar`. While `foo` only returns 1, the method `bar` synchronises the future related to the invocation of `foo`, which has been received as parameter. During the execution of this program we have that if the execution of request `bar` precedes the execution of request of request `foo`, a deadlock occurs. This happens because `bar` stops its execution until the value of `x` is available, but this value can not be computed because to execute `foo` the method `bar` must terminate.

The analysis proposed correctly identifies this program as potentially deadlocked, while if we consider a precise scheduling algorithm, like FIFO, this result will be a false positive, because `bar` is never executed before `foo`.

The program of Figure 6.1.b shows a program in which the method `foo` writes a future on the field `k` and `bar` reads the value of `k`. With our analysis, because

the scheduling is nondeterministic, this program is identified as a program with nondeterministic effects, because we can not statically identify the value stored inside the field `k` that depends on the scheduling order. Also in this case, if we consider a scheduling algorithm as FIFO, the result of our analysis is a false positive.

The false positive results come from the fact that, because of the nondeterministic scheduling, the behavioural type of the main functions considers the invocation of `foo` in parallel with the invocation of `bar`, that in case of FIFO scheduling should be considered as in sequence. However, it is important to underline that possible method invocations nested inside `foo` have to be considered as in parallel with the invocation of `bar`. The order in which this request will be served is not guaranteed even if we consider FIFO scheduling.

Considering FIFO scheduling in our analysis only requires minor modifications in the method typing rule. In fact we have to split the behavioural type of each method in two parts. The first part should be related with the behaviour of the body of that method, while the second part should collect the the method invocation performed by the method. This solution allows us to define a more precise parallel composition of the behaviours of the method invocations, especially in case multiple invocations done by a method on the same object.

6.2.2 New paradigm for futures design

The development of an analysis for synchronisation patterns points out the impact that the design and the implementation of futures has on the ease of use for programmers and on the complexity and precision of the analysis.

On one side, explicit futures give more control to the programmer; allow complex operations on futures (i.e. cooperative multithreading or future chaining); and also allow more precise identification of the synchronisation points. On the other side, implicit futures keep the programmer unaware of whether the control flow is operating on regular values or on futures; block the program only when a future is really needed; permit a better code reuse (methods are written in the same way independently of which variables contain a future or a value); and also admit the returning of future for recursive method invocations.

We can summarise saying that, while implicit futures enforce data-flow synchronisation, explicit futures implement control flow synchronisation through a statement that fills the future.

While we were facing all the problems related to the development of such analysis, we were also trying to understand if is possible to mix the two approaches in order to gain the advantages from both of them.

In fact, one of the possible future work will be the definition a new paradigm for the design and implementation of futures.

In the following we will present the features that this new model should have and how this characteristics will give use the advantage of both control-flow and data-flow synchronisation approach.

The first characteristic for this model is the use of **explicit future types**. Explicit futures of this model should be not intended as the future of active object language as ABS, in which future type are defined as parametric types. Futures should be explicitly typed, but this type only indicates that we are referring to a possible future (a variable typed as future can still store a value), but no distinction is made between a future of an integer and a future or a future of an integer. Typing the future in this way, we have that: the programmer has more control on the point of synchronisation and is exposed to the points of synchronisation that occur in his program; and it is easier statically track futures. That are the two main advantage of an approach with explicit future types. Furthermore, this new definition of explicit future types also permits a better code reuse and allows recursive methods that return futures.

The second feature of this new model should be an **explicit data-flow synchronisation** approach, in which a simple synchronisation operation is defined, as in control-flow synchronisation approach, but with an data-flow orientation. This synchronisation operation resolves a future returning a value, even in the case of nested futures (i.e factorial function), exactly as *wait-by-necessity* synchronisation does. Having an explicit synchronisation operation defined in this way gives more control to the programmer and allows the definition of a single synchronisation recursive functions.

6.2.3 Annotations checker for MultiASP

Multiactive objects model [Henrio and Kammüller, 2015] is a multi-threaded extension of the active object programming model. The main principle behind this model is to enable the parallel execution of several requests, where, as we have seen in this thesis, the Active object model only allows the processing of one request at a time. Contrary to the standard Active object model, the Multiactive model, by having shared-memory between threads, can be affected by data races.

To avoid data races, requests that execute in parallel must be acknowledged by the programmer: the programmer must state which requests are compatible regarding data race freedom.

The compatibility of two requests is statically declared (when writing a class), but may depend on dynamic parameters.

ProActive [Baduel et al., 2006] language, that has been presented in Section 2.4.2, implements the multiactive object programming model. ProActive offers to the programmer the possibility to declare compatibility of requests, and thus to have multiactive objects. ProActive allows the programmer to define request compatibility in the following way:

@Group defines a set of requests that have the same compatibility requirements. Methods belonging to the same group are considered as compatible;

@MemberOf can be defined on top of method definitions, and defines that a method belongs to a group.

@Compatible is used to specify the groups that are compatible.

We present below an example of compatibility annotation in ProActive.

```
1 @DefineGroups({
2   @Group(name="group1"),
3   @Group(name="group2")
4   @Group(name="group3")
5 })
6
7 @DefineRules({
8   @Compatible({"group1", "group3"})
9 })
```

Listing 6.1 – Annotation for ProActive

In Listing 6.1 we create three groups and we specify that the methods that belong to the `group1` can run in parallel with the method belonging to `group3`, and that methods that belong to `group2` are not compatible with any other methods.

Considering the compatibility groups and the compatibility rules defined in Listing 6.1, we can annotate the program presented in Figure 6.1.b as follow.

```
1  Int k
2
3  @MemberOf: group1
4  Int foo( )
5  { k = this.m(); return 1 }
6
7  @MemberOf: group2
8  Int bar( )
9  { z = k + 1; return k }
10
11 @MemberOf: group3
12 Int m( ) { return 1 }
13
14 // MAIN //
15 { x = this.foo(); y = this.bar() }
```

Listing 6.2 – Program of Figure 6.1.b with compatibility annotation

If we consider this program as a Multiactive object program, we have that despite the methods `foo`, `bar` and `m` are invoked on the same active object, these method invocations can be executed in parallel. As methods `foo` and `bar` perform concurrent access on the field `k`, this program is affected by data races. We want to underline that, considering the Multiactive object model, this program does not simply lead to a race condition, as for the case of the standard Active object model, but to it leads to a data race. It is important to understand this difference because: while in case of the standard Active object model we are sure that method `bar` is returning a value (`k` has been synchronised in the evaluation of the expression `z = k + 1`), if we consider the Multiactive object model we do not know if `bar` returns a value or the future related to the method invocation of `m` stored inside `k` by `foo`. To avoid data races in this program, we have annotated it as shown in Listing 6.2. In this way we are stating that method `bar` can not run in parallel with any other

methods, while `foo` and `m` being compatible can be executed at the same time.

In Multiactive objects, the execution safety is partially entrusted to the programmer that has to define compatibility annotations. To ease the responsibility of the programmer a tool which implements the effect system proposed in Chapter 5 can be developed to check if the specified annotation are sufficient to avoid data races in ProActive programs. As the effect analysis proposed has been developed for a nondeterministic scheduling approach, it already considers method running on the same object as running in parallel. For this reason, such analysis do not require significant improvements to check compatibility annotations. One of the minor improvement needed is taking into account the FIFO scheduling (used by ProActive) that, as shown in Section D.5.1, has a significant impact on the analysis of effects.

Appendix A

Proofs of Chapter 4

A.1 Subject reduction

Lemma A.1.1. $\Delta \vdash_R e : \mathbb{x}\ ,\ \mathsf{L} \triangleright \Delta'$ *and* $[\![e]\!]_\ell = w$ *for some* ℓ*, imply that* $\Delta' \vdash_R w : \mathbb{x}'\ ,\ \mathtt{0} \triangleright \Delta'$ *where* $\mathbb{x}' = \mathbb{x}$ *or* $\mathbb{x} = \mathbb{r}_f$ *and* $\mathbb{x}' = \mathbb{r}$.

Proof. We can prove it by cases.

Base Cases:

Case b.1: e can be an integer value, an active object name or a variable containing an integer value or an active object name ($e = v$ and $[\![e]\!]_\ell = \alpha$ or $[\![e]\!]_\ell$ is an integer value).
By rules TR-Sync-Val and TR-Val we obtain $\Delta \vdash_R e : \mathbb{r}\ ,\ \mathtt{0} \triangleright \Delta$ and again by applying the rules TR-Sync-Val and TR-Val we conclude that $\Delta \vdash_R [\![e]\!]_\ell : \mathbb{r}\ ,\ \mathtt{0} \triangleright \Delta$.

Case b.2: e is a future variable x where $\Delta(x) = \mathbb{r}_F$.
By applying the rules TR-Sync and TR-Var we obtain that Δ types e such that $\Delta \vdash_{\{\alpha,\mathbb{x}\}} e : \mathbb{r}\ ,\ F_\alpha \,\&\, rt\_unsync(\Delta') \triangleright \Delta'$ where Δ' is the update of Δ such that $\Delta' = (\Delta[y \mapsto \mathbb{r}]^{\Delta(y)=\mathbb{r}_F})[F \mapsto \Delta(F)^{\checkmark}]$. We conclude that using Δ' we type the evaluation of e applying the rules TR-Sync-Val and TR-Val such that $\Delta' \vdash [\![e]\!]_\ell : \mathbb{r}\ ,\ \mathtt{0} \triangleright \Delta'$.

Induction step: e_1 and e_2 are two expressions such that $[\![e_1]\!]_\ell = k_1$ and $[\![e_2]\!]_\ell = k_2$ where k_1 and k_2 are integer values. By induction hypothesis and the rule TR-Expression we have $\Delta \vdash_R e_1 : \square\ ,\ \mathsf{L}_1 \triangleright \Delta'$ implies $\Delta' \vdash_R [\![e_1]\!]_\ell : \square\ ,\ \mathtt{0} \triangleright \Delta'$

and $\Delta' \vdash_R e_2 : \square\ ,\ \mathsf{L}_2\ \triangleright\ \Delta''$ implies $\Delta'' \vdash_R [\![e_2]\!]_\ell : \square\ ,\ \mathtt{0}\ \triangleright\ \Delta''$.
By rules TR-Expression and TR-Val we can infer $\Delta \vdash_R e_1 \oplus e_2 : \square\ ,\ \mathsf{L}_1 + \mathsf{L}_2\ \triangleright\ \Delta''$ and this implies $\Delta'' \vdash_R k_1 \oplus k_2 : \square\ ,\ \mathtt{0}+\mathtt{0}\ \triangleright\ \Delta''$.

□

Theorem 4.5.1. (Subject Reduction) *Let $\Delta \vdash_R cn : \mathsf{K}$ and $cn \to cn'$. Then there exist Δ', K', and an injective renaming of active object names i such that*

- $\Delta' \vdash_R cn' : \mathsf{K}'$ *and*
- $i(\mathsf{K}) \succeq \mathsf{K}'$

Proof. The proof is a case analysis on the reduction rule used in $cn \to cn'$ and we assume that the evaluation of an expression $[\![w]\!]$ always terminates.

Case Serve**.**

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

Bt hypothesis $\Delta \vdash \alpha(a, \varnothing, \overline{q} \cup \{p\}) : \mathsf{K}$ applying the rule TR-ActObj there exist $\mathsf{K}_1, \cdots, \mathsf{K}_n$ and K_p such that $\mathsf{K} = (\overset{n}{\underset{i=1}{\&}} \mathsf{K}_i) \,\&\, \mathsf{K}_p$. With the same Δ we can type $\alpha(a, p, \overline{q})$ obtaining that $\Delta \vdash \alpha(a, p, \overline{q}) : \mathsf{K}_p \,\&\, (\overset{n}{\underset{i=1}{\&}} \mathsf{K}_i)$. By commutativity of $\&$ we can easily prove that $(\overset{n}{\underset{i=1}{\&}} \mathsf{K}_i) \,\&\, \mathsf{K}_p \succeq \mathsf{K}_p \,\&\, (\overset{n}{\underset{i=1}{\&}} \mathsf{K}_i)$.

Case Update**.**

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

Ex hypothesis we know that $\Delta \vdash \alpha(a, \{\ell | s\}, \overline{q})\ f(w) : \mathsf{K}$, by TR-Parallel, TR-ActObj and TR-Process there exist $\mathsf{K}_1, \cdots, \mathsf{K}_n$, L, $\overline{\varphi}$ and Θ such that $\mathsf{K} = (\nu\, \overline{\varphi})(\Theta \,\cdot\, \mathsf{L}) \,\&\, (\overset{n}{\underset{i=1}{\&}} \mathsf{K}_i) \,\&\, \mathtt{0}$. Accordingly with the restrictions described for our type system we have that $x \in \ell$ then $a' = a$ and $\ell' = \ell[x \mapsto w]$. As $\Delta \vdash w : \mathbb{x}$, we can choose $\Delta' = \Delta[x \mapsto \mathbb{x}]$ to type $\alpha(a, \{\ell' | s\}, \overline{q})\ f(w)$. By rules TR-Parallel, TR-ActObj and TR-Process we obtain that $\Delta' \vdash \alpha(a, \{\ell[x \mapsto w] | s\}, \overline{q})\ f(w) : \mathsf{K}$. It's trivial to see that $\mathsf{K} \succeq \mathsf{K}$.

Case New**.**

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

Ex hypothesis we know that $\Delta \vdash \alpha(a, \{\ell | x = \mathtt{new\ Act}(\overline{v}); s\}, \overline{q}) : \mathsf{K}$.
By TR-ActObj, TR-Process, TR-Seq, TR-New and TR-Val there exist $\mathsf{K}_1, \cdots, \mathsf{K}_n$, L, $\overline{\varphi}$ and Θ such that $\mathsf{K} = (\nu\, \overline{\varphi}, \beta)(\Theta \cdot \mathsf{0} + \mathsf{L}) \,\&\, (\overset{n}{\underset{i=1}{\&}} \mathsf{K}_i)$.
With the same Δ, applying the rules TR-Process, TR-ActObj, TR-Process, TR-Seq, TR-Assign-Val and TR-Act, we type the configuration $\alpha(a, \{\ell | x = \beta; s\}, \overline{q})\ \beta([z \mapsto w], \varnothing, \varnothing)$ such that: $\Delta \vdash \alpha(a, \{\ell | x = \beta; s\}, \overline{q})\ \beta([z \mapsto w], \varnothing, \varnothing) : \mathsf{K} \,\&\, \mathsf{0} \,\&\, \mathsf{0}$. It's trivial to see that $\mathsf{K} \succeq \mathsf{K} \,\&\, \mathsf{0} \,\&\, \mathsf{0}$.

Case Return**.**

$$\frac{w = [\![v]\!]_{a+\ell} \qquad \ell(\mathtt{destiny}) = f}{\alpha(a, \{\ell \mid \mathtt{return}\ v;\}, \overline{q})\ f(\bot) \rightarrow \alpha(a, \varnothing, \overline{q})\ f(w)}\ \textsc{Return}$$

By hypothesis we know that $\Delta \vdash \alpha(a, \{\ell \mid \mathtt{return}\ v;\}, \overline{q})\ f(\bot) : \mathsf{K}$. Applying TR-Parallel, TR-ActObj and TR-Process there exist L, $\mathsf{K}_1, \cdots, \mathsf{K}_n$, $\overline{\varphi}$, X and Θ such that $\mathsf{K} = (\nu\, \overline{\varphi})(\Theta \cdot \mathsf{L} \,\&\, (X, \alpha)) \,\&\, (\overset{n}{\underset{i=1}{\&}} \mathsf{K}_i) \,\&\, \mathsf{0}$ we can distinguish two cases:
Case 1: $\Delta(v) = \mathbb{r}$, by rule TR-Return-Val we have $\mathsf{L} = \mathsf{0}$
Case 2: $\Delta(v) = \mathbb{r}_f$ by rule TR-Return we have $\mathsf{L} = f_X \,\&\, \mathit{rt\_unsync}(\Delta \setminus f)$
In both cases, with the same Δ by rules TR-Parallel, TR-ActObj and TR-Process we type $\Delta \vdash \alpha(a, \varnothing, \overline{q})\ f(w) : \mathsf{0} \,\&\, (\overset{n}{\underset{i=1}{\&}} \mathsf{K}_i) \,\&\, \mathsf{0}$.
It's trivial to see that $(\nu\, \overline{\varphi})(\Theta \cdot \mathsf{0} \,\&\, (X, \alpha)) \,\&\, (\overset{n}{\underset{i=1}{\&}} \mathsf{K}_i) \,\&\, \mathsf{0} \succeq \mathsf{0} \,\&\, (\overset{n}{\underset{i=1}{\&}} \mathsf{K}_i) \,\&\, \mathsf{0}$ or $(\nu\, \overline{\varphi})(\Theta \cdot f_X \,\&\, \mathit{rt\_unsync}((\Delta \setminus f)) \,\&\, (X, \alpha)) \,\&\, (\overset{n}{\underset{i=1}{\&}} \mathsf{K}_i) \,\&\, \mathsf{0} \succeq \mathsf{0} \,\&\, (\overset{n}{\underset{i=1}{\&}} \mathsf{K}_i) \,\&\, \mathsf{0}$.

Case IF-TRUE**.**

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

Ex hypothesis we know that $\Delta \vdash \alpha(a, \{\ell \mid \texttt{if}\ v\ \{\, s_1 \,\}\ \texttt{else}\ \{\, s_2 \,\}\ ;\ s\}, \overline{q}) : \mathsf{K}$.
By TR-PARALLEL, TR-ACTOBJ, TR-PROCESS and TR-SEQ there exist L, $\mathsf{L}_s\, \mathsf{K}_1, \cdots, \mathsf{K}_n, \overline{\varphi}, X$ and Θ such that $\mathsf{K} = (\nu\, \overline{\varphi})\big(\Theta\ \cdot\ (\mathsf{L} + \mathsf{L}_s)\, \&(X, \alpha)\big)\ \&\ (\overset{n}{\underset{i=1}{\&}} \mathsf{K}_i)$.
Let $\Delta_S = \overline{x : \mathbb{X}} + \texttt{destiny} : \mathbb{I} + \texttt{future} : X$ by role TR-IF we have that:
$\Delta + \Delta_S \vdash^{\alpha}_{R} e : \mathsf{L}_e\ \triangleright\ \Delta' + \Delta_S$,
$\Delta' + \Delta_S \vdash^{\alpha}_{R} s_1 : \mathsf{L}_1\ \triangleright\ \Delta_1 + \Delta_S$,
$\Delta_1 + \Delta_S \vdash^{\alpha}_{R} s_2 : \mathsf{L}_2\ \triangleright\ \Delta_2 + \Delta_S$ and
$\Delta + \Delta_S \vdash^{\alpha}_{R} \texttt{if}\ e\ \{\, s_1 \,\}\ \texttt{else}\ \{\, s_2 \,\} : \mathsf{L}_e + \mathsf{L}_1 + \mathsf{L}_2\ \triangleright\ \Delta'' + \Delta_S$ where $\Delta'' = \texttt{Merge}(\Delta_1, \Delta_2) \cup \Delta_1|_{\texttt{Fut}(\Delta_1)\setminus\texttt{Fut}(\Delta)}\ \cup\ \Delta_2|_{\texttt{Fut}(\Delta_2)\setminus\texttt{Fut}(\Delta)}$, then $\mathsf{L} = \mathsf{L}_e + \mathsf{L}_1 + \mathsf{L}_2$.
Thanks to the Lemma A.1.1 we can say that $\Delta + \Delta_S \vdash e : \mathbb{X}\ ,\ \mathsf{L}_e\ \triangleright\ \Delta' + \Delta_S$ implies that $\Delta' + \Delta_S \vdash w : \square\ ,\ \mathtt{0}\ \triangleright\ \Delta' + \Delta_S$ where $w = [\![e]\!]_{\ell}$.
Now we can use Δ' and the rules TR-PARALLEL, TR-ACTOBJ, TR-PROCESS and TR-SEQ to say that $\Delta' \vdash \alpha(a, \{\ell | s_1; s\}, \overline{q}) : (\nu\, \overline{\varphi})\big(\Theta\ \cdot\ (\mathsf{L}_1 + \mathsf{L}_s)\, \&(X, \alpha)\big)\ \&\ (\overset{n}{\underset{i=1}{\&}} \mathsf{K}_i)$.
It's trivial to prove by the rules LS-RBEHAVIOR and LS-PLUS that
$(\nu\, \overline{\varphi})\big(\Theta\ \cdot\ (\mathsf{L}_e + \mathsf{L}_1 + \mathsf{L}_2 + \mathsf{L}_s)\, \&(X, \alpha)\big)\ \&\ (\overset{n}{\underset{i=1}{\&}} \mathsf{K}_i) \succeq$
$(\nu\, \overline{\varphi})\big(\Theta\ \cdot\ (\mathsf{L}_1 + \mathsf{L}_s)\, \&(X, \alpha)\big)\ \&\ (\overset{n}{\underset{i=1}{\&}} \mathsf{K}_i)$.

Case INVK**.**

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

By hypothesis we know that $\Delta \vdash \alpha(a, \{\ell | x = v.\mathtt{m}(\overline{v}); s\}, \overline{q})\ \beta(a', p', q') : \mathsf{K}$,

applying TR-Parallel, TR-ActObj, TR-Process and TR-Seq there exist L_s, $\mathsf{K}_1, \cdots, \mathsf{K}_n$, K'_p and $\mathsf{K}'_1, \cdots, \mathsf{K}'_m$, f, $\overline{\varphi}$, X and Θ such that $\mathsf{K} = (\nu\,\overline{\varphi}, f)\big(\Theta \cdot (\mathsf{L} + \mathsf{L}_s)\,\&(X, \alpha)\big)\,\&\,(\overset{n}{\underset{i=1}{\&}}\mathsf{K}_i)\,\&\,\mathsf{K}'_p\,\&\,(\overset{m}{\underset{i=1}{\&}}\mathsf{K}'_i)$.
Let $\Delta_S = \overline{x : \mathbb{X}} + \texttt{destiny} : \mathbb{r} + \texttt{future} : X$ by role TR-Sync we have that

$\Delta + \Delta_S \vdash^{\alpha}_R v : \beta\ ,\ \mathsf{L}'\ \triangleright\ \Delta' + \Delta_S$ and by role TR-Invk we have

$\Delta + \Delta_S \vdash^{\alpha}_R v.\mathtt{m}(\overline{v}) : \mathbb{r}_{f^s}\ ,\ \mathsf{L}'\ +\ f^s_\star\,\&\,rt\_unsync(\Delta')\ \triangleright\ \Delta'' + \Delta_S$ than $\mathsf{L} =$

$\mathsf{L}'\ +\ f^s_\star\,\&\,rt\_unsync(\Delta')$. Using Δ'' by rules TR-Parallel, TR-ActObj, TR-Process, TR-Seq and TR-Fut we finally obtain that $\Delta'' \vdash \alpha(a, \{\ell \mid x = f; s\}, \overline{q})\ \beta(a', p', \overline{q'} \cup \{p''\})\ f(\bot) : \mathsf{K}'$ where $\mathsf{K}' = (\nu\,\overline{\varphi}, f)\big(\Theta \cdot (\mathsf{0} + \mathsf{L}_s)\,\&(X, \alpha)\big)\,\&\,(\overset{n}{\underset{i=1}{\&}}\mathsf{K}_i)\,\&\,\mathsf{K}'_p\,\&\,(\overset{n}{\underset{i=1}{\&}}\mathsf{K}'_i)\,\&\,\mathsf{K}_{p''}$, where $\mathsf{K}_{p''} = (\nu\,\overline{\varphi'})(\Theta_{\mathtt{m}} \cdot \mathsf{L}_{\mathtt{m}})$ is the behavioral type of the method $\mathtt{m}$ instantiated with β and $\overline{\mathbb{X}}$.

Case Assign.

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

Ex hypothesis we know that applying TR-ActObj, TR-Process, TR-Seq and TR-Assign-Exp we have that there exist L_e, L_s, $\mathsf{K}_1, \cdots, \mathsf{K}_n$, $\overline{\varphi}$, X and Θ such that
$\Delta \vdash \alpha(a, \{\ell \mid x = e; s\}, \overline{q}) : (\nu\,\overline{\varphi})\big(\Theta \cdot (\mathsf{L}_e + \mathsf{L}_s)\,\&(X, \alpha)\big)\,\&\,(\overset{n}{\underset{i=1}{\&}}\mathsf{K}_i)$.
Let $\Delta_S = \overline{x : \mathbb{X}} + \texttt{destiny} : \mathbb{r} + \texttt{future} : X$, by hypothesis we can also know that $\Delta + \Delta_S \vdash e : \mathbb{X}\ ,\ \mathsf{L}_e\ \triangleright\ \Delta' + \Delta_S$.
Now thanks to the Lemma A.1.1 we know that $\Delta + \Delta_S \vdash e : \mathbb{X}\ ,\ \mathsf{L}_e \triangleright \Delta' + \Delta_S$ implies $\Delta' + \Delta_S \vdash w : \square\ ,\ \mathsf{0}\ \triangleright\ \Delta' + \Delta_S$ where $w = [\![e]\!]_\ell$, and we can finally conclude that by rules TR-ActObj and TR-Process $\Delta' \vdash \alpha(a, \{\ell' \mid s\}, \overline{q}) : (\nu\,\overline{\varphi})\big(\Theta \cdot \mathsf{L}_s\,\&(X, \alpha)\big)\,\&\,(\overset{n}{\underset{i=1}{\&}}\mathsf{K}_i)$.
It's trivial prove that by rules LS-Global and LS-Plus
$(\nu\,\overline{\varphi})\big(\Theta \cdot (\mathsf{L}_e + \mathsf{L}_s)\,\&(X, \alpha)\big)\,\&\,(\overset{n}{\underset{i=1}{\&}}\mathsf{K}_i)\ \mathit{later}\ (\nu\,\overline{\varphi})\big(\Theta \cdot \mathsf{L}_s\,\&(X, \alpha)\big)\,\&\,(\overset{n}{\underset{i=1}{\&}}\mathsf{K}_i)$.

□

A.2 Analysis of circularities

Proposition A.2.1. *Let* $\mathtt{m}(\alpha, \overline{\mathtt{x}}, X) = (\nu\, \overline{\varphi})(\Theta \cdot \mathsf{L}_{\mathtt{m}}) \in \mathcal{L}$.

(i) for every k, $I^{(k)}(\mathtt{m})$ *is an element in the lattice of* $\mathcal{R}_A$.

(ii) for every k, $I^{(k)}(\mathtt{m}) \Subset I^{(k+1)}(\mathtt{m})$.

Proof. (i) is immediate by definition. To see (ii) , observe that $I(\Theta \cdot \mathsf{L})$ is monotonic on I (i.e., $I(\mathtt{m}) \Subset I'(\mathtt{m})$ for every $\mathtt{m}$ implies $I(\Theta \cdot \mathsf{L}) \Subset I'(\Theta \cdot \mathsf{L})$), which follows by a straightforward structural induction on L. Then, a standard induction on k gives $I^{(k)}(\mathtt{m}) \Subset I^{(k+1)}(\mathtt{m})$. □

Since, for every k, $I^{(k)}(\mathtt{m}_i)$ ranges over a finite lattice, by the fixpoint theory Davey and Priestley [2002], 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 behavioural type), be the least fixpoint $I^{(m)}$.

The following three lemmas are preparatory to the theorem of correctness and completeness of our algorithm Theorem 5.5.1. They establish a relation between the circularities of the approximants $I^{(k)}(\Theta \cdot \mathsf{L})$ - Lemma A.2.3 and, whenever $\Theta \cdot \mathsf{L} \to \Theta \cdot \mathsf{L}'$, between circularities of $I(\Theta \cdot \mathsf{L})$ and $I(\Theta \cdot \mathsf{L}')$ - Lemma A.2.4.

Lemma A.2.2. *Let* $\mathcal{C}$ *be a context,* Θ *be a future environment,* L *be a behavioural type and* $I(\cdot)$ *be a* flattening. *Then we have:*

1) $I(\Theta \cdot \mathcal{C}[\,\mathsf{L}\,])$ *has a circularity if and only if* $\mathcal{C}[\,\mathtt{R}\,]$ *has a circularity, for some* $\mathtt{R} \in I(\Theta \cdot \mathsf{L})$.
2) *Let* $\mathtt{R}$ *be a binary relation on names and* $\overline{a}$ *be the names in both* $\mathcal{C}$ *and* $\mathtt{R}$. *Then* $I(\Theta \cdot \mathcal{C}[\,\mathsf{L}\,])$ *has a circularity if and only if* $I(\Theta \cdot \mathcal{C}[\,\mathtt{proj}_{\overline{a}}(R^+)\,])$ *has a circularity.*

Proof. Both the properties follow almost immediately from the definitions. To see 1, note that it follows by a straightforward induction on $\mathcal{C}$ that $\mathtt{R} \in I(\Theta \cdot \mathcal{C}[\,\mathsf{L}\,])$ if and only if $\mathtt{R} \in I(\Theta \cdot \mathcal{C}[\,\mathtt{R}\,])$ for some $\mathtt{R} \in I(\Theta \cdot \mathsf{L})$. □

Lemma A.2.3. *Let*

$$\big(\mathtt{m}_1(\alpha_1, \overline{\mathtt{x}}_1, X_1) = (\nu\, \overline{\varphi}_1)(\Theta_1 \cdot \mathsf{L}_1),\ \cdots,\ \mathtt{m}_n(\alpha_n, \overline{\mathtt{x}}_n, X_n) = (\nu\, \overline{\varphi}_n)(\Theta_n \cdot \mathsf{L}_n),\, \mathsf{L}\big)$$

be a behavioural type program and let $\Theta \cdot \mathcal{C}[\, f^{i_1}_{\kappa_1}\,]\cdots[f^{i_m}_{\kappa_m}] \to^m \Theta' \cdot \mathcal{C}[\, \mathsf{L}'_{i_1}\,]\cdots[\mathsf{L}'_{i_m}]$ *where* $\mathcal{C}[\,\cdot\,]\cdots[\cdot]$ *is a multiple context without function invocations,* $\Theta(f^{i_j}) = \lambda X'_j.\mathtt{m}_{i_j}(\alpha'_j, \overline{\mathbb{X}'}j, X'_j)$, $\Theta' = \bigcup_{j=1}^{m} \Theta_j[\overline{\varphi'_j}/\overline{\varphi_{i_j}}][\alpha'_j, \overline{\mathbb{X}'}j, \kappa_j/\alpha_{i_j}, \overline{\mathbb{X}}_{i_j}, X_{i_j}] \cup \Theta$ *and* $\mathsf{L}'_{i_j} = \mathsf{L}_{i_j}[\overline{\varphi'_j}/\overline{\varphi_{i_j}}][\alpha'_j, \overline{\mathbb{X}'}j, \kappa_j/\alpha_{i_j}, \overline{\mathbb{X}}_{i_j}, X_{i_j}]$.

Then, the following two properties are equivalent:

(1) $I^{(k+1)}(\Theta \cdot \mathcal{C}[\, f^{i_1}_{\kappa_1}\,]\cdots[f^{i_m}_{\kappa_m}])$ *has a circularity,*

(2) $I^{(k)}(\Theta' \cdot \mathcal{C}[\, \mathsf{L}'_{i_1}\,]\cdots[\mathsf{L}'_{i_m}])$ *has a circularity.*

Proof. To show the implication $1 \Rightarrow 2$, suppose that

$$I^{(k+1)}(\Theta \cdot \mathcal{C}[\, f^{i_1}_{\kappa_1}\,]\cdots[f^{i_m}_{\kappa_m}])$$

has a circularity. By repeated applications of Lemma A.2.2(1), there exists $\mathtt{R}_j \in I^{(k+1)}(\Theta \cdot f^{i_j}_{\kappa_j})$ with $1 \le j \le m$ such that $I^{(k+1)}(\Theta \cdot \mathcal{C}[\,\mathtt{R}_1\,]\cdots[\mathtt{R}_m])$ has a circularity. By the definition of $I^{(k+1)}(\Theta \cdot f^{i_j}_{\kappa_j})$ and $I^{(k+1)}(\mathtt{m}_i)$, where $\Theta(f) = \lambda X.\mathtt{m}_i(\alpha, \overline{\mathbb{X}}, X)$

$$\mathtt{R}_j = \mathtt{proj}_{\overline{a}}(R^+)[\alpha'_j, \overline{\mathbb{X}'}j, \kappa_j/\alpha_{i_j}, \overline{\mathbb{X}}_{i_j}, X_{i_j}]$$

with $\mathtt{R}'_j \in I^{(k)}(\mathtt{L}_{i_j})$. This implies that

$$I^{(k+1)}(\mathcal{C}[\,\mathtt{proj}_{\overline{a'_1}}((\mathtt{R}'_1[\alpha'_1, \overline{\mathbb{X}'}1, \kappa_1/\alpha_1, \overline{\mathbb{X}}_1, X_1])^+)\,]\cdots[\mathtt{proj}_{\overline{a'_m}}((\mathtt{R}'_m[\alpha'_m, \overline{\mathbb{X}'}m, \kappa_m/\alpha_m, \overline{\mathbb{X}}_m, X_m])^+)])$$

also has a circularity. By repeated applications of Lemma A.2.2(2),

$$I^{(k+1)}(\mathcal{C}[\,\mathtt{R}'_1[\alpha'_1, \overline{\mathbb{X}'}1, \kappa_1/\alpha_1, \overline{\mathbb{X}}_1, X_1]\,]\cdots[\mathtt{R}'_m[\alpha'_m, \overline{\mathbb{X}'}m, \kappa_m/\alpha_m, \overline{\mathbb{X}}_m, X_m]])$$

has also a circularity. Since $\mathcal{L}$ contains no function invocations,

$$I^{(k)}(\mathcal{C}[\,\mathtt{R}'_1[\alpha'_1, \overline{\mathbb{X}'}1, \kappa_1/\alpha_1, \overline{\mathbb{X}}_1, X_1]\,]\cdots[\mathtt{R}'_m[\alpha'_m, \overline{\mathbb{X}'}m, \kappa_m/\alpha_m, \overline{\mathbb{X}}_m, X_m]])$$

has a circularity, and by repeated applications of Lemma A.2.2(1), $I^{(k)}(\Theta' \cdot \mathcal{C}[\,\mathtt{L}'_{i_1}\,]\cdots[\mathtt{L}'_{i_m}])$ also has a circularity.

The convert is similar.

□

Lemma A.2.4. *Let* $(\mathcal{L}, \mathsf{L})$ *be a behavioural type program and* $\Theta \cdot \mathcal{C}[\, f_\kappa \,] \to \Theta' \cdot \mathcal{C}[\, \mathsf{L}' \,]$*, where* $\Theta(f) = \lambda X.\mathtt{m}(\alpha', \overline{\mathbb{X}}', X)$*,* $\mathsf{L}' = \mathsf{L}_{\mathtt{m}}[\overline{\varphi}'/\overline{\varphi}][\alpha', \overline{\mathbb{X}}', \kappa/\alpha, \overline{\mathbb{X}}, X]$ *and* $\Theta' = \Theta \cup \Theta_{\mathtt{m}}[\overline{\varphi}'/\overline{\varphi}][\alpha', \overline{\mathbb{X}}', \kappa/\alpha, \overline{\mathbb{X}}, X]$*.*

The following two properties are equivalent:

(1) $I(\Theta \cdot \mathcal{C}[\, f_\kappa \,])$ *has a circularity,*

(2) $I(\Theta' \cdot \mathcal{C}[\, \mathsf{L}' \,])$ *has a circularity.*

Proof. To show the implication $1 \Rightarrow 2$, suppose that $I(\Theta \cdot \mathcal{C}[\, f_\kappa \,])$ has a circularity. Then by Lemma A.2.2(1), there exists $\mathtt{R} \in I(\Theta \cdot f_\kappa)$ such that $I(\Theta \cdot \mathcal{C}[\, \mathtt{R} \,])$ has a circolarity. By definition of I, there exist k such that $\mathtt{R} \in I^{(k+1)}(\Theta \cdot f_\kappa)$. Thus, by a reasoning similar to the one in Lemma A.2.3, there exists

$$\mathtt{R}' \in I^{(k)}(\Theta \cdot \mathsf{L}') \Subset I(\Theta \cdot \mathsf{L}')$$

such that $I(\mathtt{R}')$ has a circularity. by Lemma A.2.2(1), therefore, $I(\Theta \cdot \mathsf{L}')$ has a circularity.

The converse is similar.

□

The following theorem states the correctness and completeness of our algorithm. Similarly to Giachino et al. [2014], 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 $I^{(k)}(\Theta' \cdot \mathsf{L}')$. Proofs are omitted because they are similar to those of Giachino et al. [2014].

Theorem 4.6.1. *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.*

Proof. (*If* direction) By definition, $(\mathcal{L}, \mathsf{L})$ has a circularity if there is $\Theta \cdot \mathsf{L} \to^* \Theta' \cdot \mathsf{L}'$ such that $I^{\perp}(\Theta' \cdot \mathsf{L}')$ has a circularity. By induction on the length of $\mathsf{L} \to^* \mathsf{L}'$. When the length is 0 then $I^{\perp}(\Theta' \cdot \mathsf{L}')$ has a circularity implies $I(\Theta' \cdot \mathsf{L}')$ has a circularity (by $I^{\perp}(\Theta' \cdot \mathsf{L}') = I^{(0)}(\Theta' \cdot \mathsf{L}')$ and Lemma A.2.1(2)). Assume $\Theta \cdot \mathsf{L} \to^* \Theta' \cdot \mathsf{L}'$ be equal to $\Theta \cdot \mathsf{L} \to \Theta'' \cdot \mathsf{L}'' \to^* \Theta' \cdot \mathsf{L}'$. By inductive hypothesis, we assume that the theorem holds on the computation $\Theta'' \cdot \mathsf{L}'' \to^*$

$\Theta' \cdot \mathsf{L}'$. Then, by Lemma A.2.4, if $I(\Theta'' \cdot \mathsf{L}'')$ has a circularity then $I(\Theta \cdot \mathsf{L})$ has a circularity. Therefore the theorem.

(*Only-if* direction) We demonstrate that, if $I(\Theta \cdot \mathsf{L})$ has a circularity then there is $\Theta \cdot \mathsf{L} \rightarrow^* \Theta' \cdot \mathsf{L}'$ such that $I^{\perp}(\Theta' \cdot \mathsf{L}')$ has a circularity. Let m be the least natural number such that $I = I^{(m)}$. Let $\mathsf{L} = \mathcal{C}[\, f^{i_1}_{\kappa_1}\,] \cdots [f^{i_n}_{\kappa_n}]$ where $\Theta(f^{i_j}) = \mathtt{m}_{i_j}(\alpha_j, \overline{\mathtt{x}}_j, X_j)$, such that $\mathcal{C}[\ ] \cdots [\ ]$ does not contain function invocations. Then

$$\Theta \cdot \mathsf{L} \rightarrow^n \Theta \cdot \mathcal{C}[\, \mathsf{L}_{i_1}\,] \cdots [\mathsf{L}_{i_n}] = \Theta'' \cdot \mathsf{L}''$$

Additionally, by Lemma A.2.3, $I^{(m-1)}(\Theta'' \cdot \mathsf{L}'')$ has a circularity because $I^{(m)}(\Theta' \cdot \mathsf{L}')$ has a circularity. Now, we reapply the same argument to $\Theta'' \cdot \mathsf{L}''$ since $I^{(m-1)}(\Theta'' \cdot \mathsf{L}'')$ has a circularity. After m steps we get $\Theta' \cdot \mathsf{L}'$ such that $I^{(0)}(\Theta' \cdot \mathsf{L}') = I^{\perp}(\Theta' \cdot \mathsf{L}')$ has a circularity.

□

Appendix B

Proofs of Chapter 5 - Deadlock

B.1 Proofs of Theorem 5.5.1

To demonstrate the correctness of the type system and the analysis we split 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. We suppose that the analysed program has only deterministic effects (see Definition 4). The correctness of our system guarantees that, if the deadlock-freedom is accessed for a behavioural type program associated to a gASP program with deterministic effects, 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 deadlocks are present in the behavioural type of the original program, then none of its executions can lead to a deadlock. To this end, we prove that: (i) if there is no circularity in the type of a runtime configuration then this configuration exhibits no deadlock, and (ii) if a configuration reduces to a configuration with a circularity then the original configuration already had a circularity. These two points ensure that if no circularity is found in the behavioural type of a gASP program then there is no deadlock in the original program. We state again the Theorem 5.5.1 as follow.

Theorem B.1.1. *Let P be a* gASP *program with deterministic effects (see Definition 4) and cn be a configuration of its operational semantics, with behavioural type $\Theta \cdot L$.*

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;*

In order to prove the points *1* and *2* of Theorem 5.5.1 we need an extension of the type system to type runtime configurations (Section B.1.1). Additionally, to prove the point *2* we also need to define a *later-stage* relation between behavioural types (Section B.1.3).

B.1.1 Runtime Type System for Deadlock detection

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

$\mathbb{r}$	$::=$	$\Box \mid f \mid \alpha[\overline{x : f}]$	basic type
$\mathbb{f}$	$::=$	$\mathbb{r} \mid \lambda X.\mathtt{m}(f, \overline{g}, X, \Gamma, E) \mid f \rightsquigarrow g.x$	future type
F	$::=$	$f \mid {}^{s}f$	extended futures
κ	$::=$	$\star \mid \alpha \mid X$	synchronisers
L	$::=$	$\mathsf{0} \mid (\kappa, \alpha) \mid f_\kappa \mid \mathsf{L} + \mathsf{L} \mid \mathsf{L} \,\&\, \mathsf{L}$	behavioural type
K	$::=$	$\mathsf{L} \mid (\nu\, \overline{\varkappa})(\Theta \cdot \mathsf{L}) \mid \mathsf{K} \,\&\, \mathsf{K}$	behavioural type for configuration

F F has been introduced for distinguishing between two kinds of future names: i) f that has been used in the type system as a static time representation of a future, but that it is now used for 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 5.3 where we are focusing only on the deadlock analysis aspects, and we leaves out what is related with the effect analysis. The typing judgements are also simpler than the judgements of Section 5.2, 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 synchronised by the method $\mathtt{m}$ (this set of future names is a subset of the domain of $\Gamma_{\mathtt{m}}$) and the fields of the arguments modified by $\mathtt{m}$, represented by elements

configuration and processes

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

$$\frac{\Delta \vdash f : \lambda X.\mathtt{m}(\overline{g}, X, \Delta_{\mathtt{m}}, E_{\mathtt{m}})}{\Delta \vdash f(\bot) : \mathsf{0}}\ (\text{TR-Future-Undef}) \qquad \frac{\Delta \vdash f : \lambda X.\mathtt{m}(\overline{g}, X, \Delta_{\mathtt{m}}, E_{\mathtt{m}})^{\checkmark} \qquad \Delta \vdash w : f}{\Delta \vdash f(w) : \mathsf{0}}\ (\text{TR-Future-Eval})$$

$$\frac{\Delta(\alpha) = \alpha[\overline{y : f}] \qquad \Delta \vdash \overline{v} : \overline{f} \qquad \Delta' = \Delta + this : \alpha[\overline{y : f}] \qquad \Delta' \vdash p : \mathsf{K}_0 \qquad \forall i \in 1..n.\, \Delta' \vdash q_i : \mathsf{K}_i}{\Delta \vdash \alpha(\{\overline{y \mapsto v}\}, p, \{q_1, \cdots, q_n\}) : \overset{n}{\underset{i=0}{\&}} \mathsf{K}_i}\ (\text{TR-Actor})$$

$$\frac{\Delta \vdash cn_1 : \mathsf{K}_1 \qquad \Delta \vdash cn_2 : \mathsf{K}_2}{\Delta \vdash cn_1\ cn_2 : \mathsf{K}_1 \,\&\, \mathsf{K}_2}\ (\text{TR-Parallel})$$

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

Figure B.1 – Typing rules for runtime configurations.

like $g.x$ (g is the argument and x is the field of g in which $\mathtt{m}$ has stored a future).

2) the $rt\_unsync(\cdot)$ function on environments Δ is similar to $rt\_unsync(\cdot)$ in Section 5.2, except that it now grabs all ${}^{s}f$ and all futures f. More precisely we define $\mathtt{Fut}_R(\Delta)$, $\mathtt{AFut}_R(\Delta)$, and
$rtunsync\Delta$ to be the functions

$$\begin{aligned} \mathtt{Fut}_R(\Delta) &\stackrel{def}{=} \{F \mid F \in \mathrm{dom}(\Delta)\} \\ \mathtt{AFut}_R(\Delta) &\stackrel{def}{=} \{F \in \mathtt{Fut}_R(\Delta) \mid \Delta(F) = \Delta(F)^{\times}\} \\ rt\_unsync(\Delta) &\stackrel{def}{=} \&_{F \in \mathtt{AFut}_R(\Delta)} F_{\star}, \end{aligned}$$

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

values, variables and method names

- judgements used: $\Delta \vdash x : \mathbb{r}$ and $\Gamma \vdash \mathtt{m} : (\overline{f}, X, \Gamma') \to (E, A)$

$$\frac{v \quad \textit{integer-value} \text{ or } \mathtt{null}}{\Delta \vdash : v\square}\ (\text{TR-Val}) \qquad \frac{\Delta(x) = F}{\Delta \vdash : xF}\ (\text{TR-Var}) \qquad \frac{\Delta(F) = \mathbb{f}^{[\checkmark]}}{\Delta \vdash : F\mathbb{f}^{[\checkmark]}}\ (\text{TR-Fut})$$

$$\frac{\Delta(\textit{this}.x) = F}{\Delta \vdash : xF}\ (\text{TR-Field}) \qquad \frac{\Delta(\mathtt{m}) = (\overline{F}, X, \Gamma) \to (E) \qquad \sigma \text{ renaming}}{\Gamma \vdash m : (\sigma(F), \sigma(\overline{g}), \sigma(X), \Gamma \circ \sigma) \to (E \circ \sigma)}\ (\text{TR-Method-Sign})$$

synchronisations

- judgement used: $\Gamma, E \overset{\oplus}{\vdash}_S \ v : \mathsf{L} \ \triangleright \ \Gamma', E'$

$$\frac{\Delta, E \vdash v : F \qquad \Delta \vdash F : \mathbb{f}^{\checkmark}}{\Delta, E \overset{\oplus}{\vdash}_S \ v : \mathtt{0} \ \triangleright \ \Delta, E}\ (\text{TR-Synchronized})$$

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

$$\frac{\begin{array}{c}\Delta \vdash \textit{this} : \alpha[\cdots] \qquad \Delta \vdash x : F \\ \Delta \vdash F : G \rightsquigarrow \textit{this}.x \qquad \Delta' = \Delta[F^{\checkmark}]\end{array}}{\Delta, E \overset{\oplus}{\vdash}_S \ x : F_\alpha \,\&\, rt\_unsync(\Delta') \ \triangleright \ \Delta', E}\ (\text{TR-Sync-Field})$$

$$\frac{\begin{array}{c}\Delta \vdash \textit{this} : \alpha[\cdots] \qquad \Delta \vdash x : F \\ \Delta \vdash F : \mathbb{f} \qquad F \in S \qquad \Delta' = \Delta[F^{\checkmark}]\end{array}}{\Delta, E \overset{\oplus}{\vdash}_S \ x : F_\alpha \,\&\, rt\_unsync(\Delta') \ \triangleright \ \Delta', E \cup \{F\}}\ (\text{TR-Sync-Param})$$

Figure B.2 – Runtime typing rules for values, variables, method names and synchronisations

expressions with side effects

- judgement used: $\Delta, E \vdash_S z : f, \mathsf{L} \triangleright \Delta', E'$

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

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

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

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

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

Figure B.3 – Runtime typing rules expressions with side effects

B.1.2 Proof of Theorem 5.5.1.1

Since we have a type system for configurations we can now prove the first statement of the Theorem 5.5.1.

Lemma B.1.2. *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 3, 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 $(\nu\, \overline{\varkappa})\big(\Theta \cdot (f_\alpha + \mathsf{L}_s)\,\&(X, \alpha)\big)\ \&\ (\overset{n}{\underset{i=1}{\&}} \mathsf{K}_i)$.
Having that:

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

statements

- judgement used: $\Delta, E \vdash_S s : \mathsf{L} \triangleright \Delta', E'$

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

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

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

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

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

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

(TR-Return-Fut)
$$\frac{\Delta \vdash v : F \quad \Delta \vdash F : \mathbb{f} \quad \Delta(\mathtt{future}) = X \quad \Delta(\mathtt{destiny}) = f' \quad \Delta \vdash f' : \lambda X.\mathtt{m}(\overline{g}, X, \Gamma_\mathtt{m}, E_\mathtt{m})}{\Delta, E \vdash_S \mathtt{return}\ v : F_X \,\&\, \mathit{rt\_unsync}(\Delta \setminus F) \triangleright \Delta, E}$$

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

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

Figure B.4 – Runtime typing rules for statements

we can infer 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)

□

$$\text{(LS-RuntimeEmpty)}\quad \mathsf{K} \succeq_\Delta \mathsf{0} \qquad \text{(LS-Global)}\quad \frac{\mathsf{K}_1 \succeq_\Delta \mathsf{K}_1'}{\mathsf{K}_1 \,\&\, \mathsf{K} \succeq_\Delta \mathsf{K}_1' \,\&\, \mathsf{K}} \qquad \text{(LS-Behavior)}\quad \frac{\mathsf{K} = (\nu\,\overline{\varphi})(\Theta \cdot \mathsf{L}) \qquad \mathsf{K}' = (\nu\,\overline{\varphi'})(\Theta \cdot \mathsf{L}') \qquad \mathsf{L} \succeq_\Delta \mathsf{L}'}{\mathsf{K} \succeq_\Delta \mathsf{K}'}$$

$$\text{(LS-Empty)}\quad \mathsf{L} \succeq_\Delta \mathsf{0} \qquad \text{(LS-Plus)}\quad \mathsf{L}_1 + \mathsf{L}_2 \succeq_\Delta \mathsf{L}_i \qquad \text{(LS-Parallel)}\quad \frac{\mathsf{L}_1 \succeq \mathsf{L}_1'}{\mathsf{L}_1 \,\&\, \mathsf{L} \succeq_\Delta \mathsf{L}_1' \,\&\, \mathsf{L}}$$

$$\text{(LS-Invk)}\quad \frac{\begin{array}{c}\Delta(f) = \lambda X.\mathtt{m}(\alpha', \overline{\mathbb{z}'}, X) \\ \Delta(m) = \big((\alpha, \overline{\mathbb{x}}, X) \to \mathbb{r}, \mathsf{K}_\mathtt{m}\big) \qquad \Delta \vdash \mathtt{m} : (\alpha', \overline{\mathbb{x}'}, X) \to \mathbb{r}' \\ \overline{\alpha} = fn(\mathsf{K}) \setminus fn(\alpha, \overline{\mathbb{x}}, \mathbb{r}) \qquad \overline{\alpha'} \cup fn(\alpha', \overline{\mathbb{x}'}, \mathbb{r}') = \varnothing\end{array}}{(\nu\,\overline{\varphi})(\Theta \cdot (f_\kappa \,\&\, \mathsf{L}) + \mathsf{L}_s) \succeq_\Delta (\nu\,\overline{\varphi})(\Theta \cdot \mathsf{L}_s) \,\&\, \mathsf{K}_\mathtt{m}[\overline{\alpha'}/\overline{\alpha}][\alpha', \overline{\mathbb{x}'}, \mathbb{r}'/\alpha, \overline{\mathbb{x}}, \mathbb{r}]}$$

Figure B.5 – The later-stage relation.

B.1.3 Later stage relation

As we said before, we need to define a *later-stage* relation between behavioural types (denoted $\succeq_\Delta$), which is a syntactic relationship between behavioural 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 behaviour, and a sum type is larger than each element of the sum. The later-stage relation is the least congruence with respect to runtime behavioural type that contains the rules in Figure B.5.

B.1.4 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 $cn \to cn'$ 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 B.1.3 (Subject Reduction)**.** *Let* $\Delta \vdash_R cn : \mathsf{K}$ *and* $cn \to cn'$*. Then there exist* Δ'*,* K'*, and an injective renaming of actor and future names* $\imath$ *such that*

- $\Delta' \vdash_R cn' : \mathsf{K}'$ *and*
- $\imath(\mathsf{K}) \succeq_\Delta \mathsf{K}'$

Proof of Theorem 5.5.3 (Subject Reduction)

The proof is a case analysis on the reduction rules used in $cn \to cn'$.

Case: Serve

Serve

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

Proof. Let $\Delta, \mathsf{K}_p, \mathsf{K}_1 \cdots \mathsf{K}_n$ exist, by rule TR-Actor we obtain that $\Delta \vdash \alpha(a, \varnothing, \overline{q} \cup \{p\}) \ : \ \mathsf{K}_p \,\&(\overset{n}{\underset{i=1}{\&}} \mathsf{K}_i)$.

With the same Δ we can type the configuration $\alpha(a, p, \overline{q})$ by applying the rule TR-Actor and we gain that $\Delta \vdash \alpha(a, p, \overline{q}) \ : \ \mathsf{K}_p \,\&(\overset{n}{\underset{i=1}{\&}} \mathsf{K}_i)$.

It is trivial to demonstrate that the relation $\mathsf{K}_p \,\&(\overset{n}{\underset{i=1}{\&}} \mathsf{K}_i) \succeq \Delta \mathsf{K}_p \,\&(\overset{n}{\underset{i=1}{\&}} \mathsf{K}_i)$ holds. □

Case: Return

Return

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

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

Proof. We can distinguish two cases:

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

   By rules TR-Actor andTR-Process we obtain that

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

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

– there exists a set collecting effects E;

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

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

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

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

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

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

– there exists a set collecting effects E;

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

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

□

Case: Update

UPDATE

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

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

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: $\Delta \vdash \alpha(a, \{\ell \mid s\}, \overline{q})\ f(w) :\ (\nu\, \overline{\varkappa})\big(\Theta \cdot \mathsf{L}\big) \,\&\, (\overset{n}{\underset{i=1}{\&}} \mathsf{K}_i)$.
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 $\Delta' \vdash \alpha(a', \{\ell' \mid s\}, \overline{q})\ f(w)\ :\ (\nu\, \overline{\varkappa})\big(\Theta \cdot \mathsf{L}\big) \,\&\, (\overset{n}{\underset{i=1}{\&}} \mathsf{K}_i)$.
It is trivial to proof that $(\nu\, \overline{\varkappa})\big(\Theta \cdot \mathsf{L}\big) \,\&\, (\overset{n}{\underset{i=1}{\&}} \mathsf{K}_i) \succeq_\Delta (\nu\, \overline{\varkappa})\big(\Theta \cdot \mathsf{L}\big) \,\&\, (\overset{n}{\underset{i=1}{\&}} \mathsf{K}_i)$.

□

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 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 ($x \notin \mathit{fields}(\texttt{Act})$)

   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:
   $\Delta \vdash \alpha(a, \{\ell \mid x = e; s\}, \overline{q}) : \ (\nu\, \overline{\varkappa})\big(\Theta \cdot \mathsf{L}_e + \mathsf{L}_s\big) \,\&\, (\overset{n}{\underset{i=1}{\&}} \mathsf{K}_i)$.
   Let us chose $\Delta' = \Delta_4$ by rules TR-Parallel, TR-Actor and TR-Process we obtain that:
   $\Delta' \vdash \alpha(a, \{\ell \mid x = e; s\}, \overline{q}) : \ (\nu\, \overline{\varkappa})\big(\Theta \cdot \mathsf{L}_s\big) \,\&\, (\overset{n}{\underset{i=1}{\&}} \mathsf{K}_i)$.

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 $(\nu\,\overline{\varkappa})\big(\Theta \cdot \mathsf{L}_e{+}\mathsf{L}_s\big) \,\&\, (\overset{n}{\underset{i=1}{\&}}\mathsf{K}_i) \succeq_\Delta (\nu\,\overline{\varkappa})\big(\Theta \cdot \mathsf{L}_s\big) \,\&\, (\overset{n}{\underset{i=1}{\&}}\mathsf{K}_i)$.

2) x is a field (x $\in$ *fields*(`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 $E'' = E' \cup \{this.x\}$ (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

New

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

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 $\Delta'', E \vdash_S \texttt{new Act}(\overline{v}) : f,\ \mathsf{0} \triangleright \Delta''[f \mapsto \beta[\overline{a : g}]^{\checkmark}], E$ 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:
$\Delta \vdash \alpha(a, \{\ell | x = v.\mathtt{m}(\overline{v}); s\}, \overline{q}) : (\nu\, \overline{\varkappa})\big(\Theta \cdot (\mathsf{0} + \mathsf{L}_s)\,\&(X, \alpha)\big)\,\&\,(\overset{n}{\underset{i=1}{\&}}\mathsf{K}_i)$
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:
$\Delta' \vdash \alpha(a, \{\ell \mid x = \beta; s\}, \overline{q})\ \beta([\overline{y} \mapsto \overline{w}], \varnothing, \varnothing) : (\nu\, \overline{\varkappa})\big(\Theta \cdot (\mathsf{0}+\mathsf{L}_s)\,\&(X, \alpha)\big)\,\&\,(\overset{n}{\underset{i=1}{\&}}\mathsf{K}_i)\,\&\,\mathsf{0}.$
It is trivial to verify that:
$(\nu\, \overline{\varkappa})\big(\Theta \cdot (\mathsf{0}+\mathsf{L}_s)\,\&(X, \alpha)\big)\,\&\,(\overset{n}{\underset{i=1}{\&}}\mathsf{K}_i) \succeq_\Delta (\nu\, \overline{\varkappa})\big(\Theta \cdot (\mathsf{0} + \mathsf{L}_s)\,\&(X, \alpha)\big)\,\&\,(\overset{n}{\underset{i=1}{\&}}\mathsf{K}_i)\,\&\,\mathsf{0}.$
□

Case: Invk

Invk

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

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 $\Delta'', E \vdash_S v.\mathtt{m}(\overline{v}) :^s f\ ,\ {}^sf_\star \,\&\, rt\_unsync(\Delta'') \triangleright \Delta''', E$ such that $\Delta''' = \Delta''[f \mapsto \lambda X'.\mathtt{m}(g, \overline{g'}, X', \Delta_\mathtt{m}, E_\mathtt{m})]$ where ${}^sf$ 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
$\Delta \vdash \alpha(a, \{\ell | x = v.\mathtt{m}(\overline{v}); s\}, \overline{q}) : (\nu\, \overline{\varkappa}, {}^sf)\big(\Theta \cdot ({}^sf_\star \,\&\, rt\_unsync(\Delta'') + \mathsf{L}_s)\,\&(X, \alpha)\big) \,\&\, (\overset{n}{\underset{i=1}{\&}} \mathsf{K}_i)$.
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 $\imath({}^sf) = f$ 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:

- $\Delta' \vdash \alpha(a, \{\ell \mid x = f; s\}, \overline{q}) : (\nu\, \overline{\varphi}, f)\big(\Theta \cdot (\mathsf{0} + \mathsf{L}_s)\,\&(X, \alpha)\big) \,\&\, (\overset{n}{\underset{i=1}{\&}} \mathsf{K}_i)$
- $\Delta' \vdash \beta(a', p', \overline{q'} \cup \{p''\})\ f(\bot) : \mathsf{K}'_p \,\&\, (\overset{n}{\underset{i=1}{\&}} \mathsf{K}'_i) \,\&\, \mathsf{K}_{p''}$ 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 $(\nu\, \overline{\varkappa}, f)\big(\Theta \cdot ({f^s_\star} \,\&\, rt\_unsync(\Delta'') + \mathsf{L}_s)\,\&(X, \alpha)\big) \succeq_\Delta (\nu\, \overline{\varphi}, f)\big(\Theta \cdot (\mathsf{0} + \mathsf{L}_s)\,\&(X, \alpha)\big) \,\&\, \mathsf{K}_{p''}$.

□

Appendix C

Proofs of Chapter 5 - Effects

C.1 Proof of Section 5.5

The aim of this section is to prove the correctness of our effect analysis, which mostly means prove the following theorem.

Theorem C.1.1. *Let P be a* gASP *program, if $\Gamma \vdash P$ then P has deterministic effects.*

In order to prove this theorem we need to extend our type system to runtime configurations. This extension is presented in the next section.

C.1.1 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 5.3 where we are focusing only on the effect analysis leaving out all the aspects related with deadlock. This is the reason why the behavioural type syntax and also the typing judgments are simpler then the corresponding shown in Section 5.2.

$$
\begin{array}{rcll}
\mathbb{r} & ::= & \square \mid f \mid \alpha[\overline{x:f}] & \text{basic type} \\
\mathbb{f} & ::= & \mathbb{r} \mid \lambda X.\mathtt{m}(f,\overline{g},\Gamma,E) & \text{future type} \\
F & ::= & f \mid {}^{s}f & \text{extended futures}
\end{array}
$$

configuration and processes:
- judgements used: $\Delta \vdash cn \ \triangleright \ E$ and $\Delta, E, A \vdash p \ \triangleright \ E, A$

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

$$\frac{\begin{array}{c}\Delta(\alpha) = \alpha[\overline{y : f}] \qquad \Delta \vdash \overline{v} : \overline{f} \qquad \Delta' = \Delta + this : \alpha[\overline{y : f}] \\ \Delta', \varnothing, \varnothing \vdash p \ \triangleright \ E_0, A_0 \\ \forall i \in 1..n.\, \Delta', [this \mapsto \varnothing], \varnothing \vdash q_i \ \triangleright \ E_i, A_i \qquad (E_k \# E_j)^{k,j \in [1,n] \wedge k \neq j}\end{array}}{\Delta \vdash \alpha(\{\overline{y \mapsto v}\}, p, \{q_1, \cdots, q_n\})}\ (\text{TR-Actor}) \qquad \frac{\Delta \vdash cn_1 \qquad \Delta \vdash cn_2}{\Delta \vdash cn_1 \ cn_2}\ (\text{TR-Parallel})$$

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

Figure C.1 – Runtime typing rules for configurations

The main differences with the type system presented in Section 5.3 are:
- future types does not present delegation and future results do not contain the place holder X;
- we define *extended futures* F which 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).
- now judgments only associate types to atoms and expressions and anymore behavioural type to statements.

C.1.2 Proof of Theorem 5.5.4

The proof of the Theorem 5.5.4 can be split in two steps. The former involves in the proof that given a program P and a configuration cn, reached during the

values and method names

- judgements used: $\Delta \vdash x : \mathbb{r}$, $\Gamma \vdash f : \mathbb{f}$ and $\Delta \vdash m : (\overline{f}, \Delta') \to (E, A)$

$$\frac{v \quad \textit{integer-value} \text{ or } \texttt{null}}{\Delta, E, A \vdash v : \square \triangleright E} \text{(TR-Val-Int)} \qquad \frac{\Delta(x) = f}{\Delta, E, A \vdash x : f \triangleright E} \text{(TR-Var)} \qquad \frac{\Delta(\mathit{this}) = \alpha[x : f, \cdots] \quad E' = E[\alpha.x \mapsto^{\sqcup} \mathtt{r}]}{\Delta, E \vdash x : f \triangleright E'} \text{(TR-Field)}$$

$$\frac{\Delta(f) = \mathbb{f}}{\Delta \vdash f : \mathbb{f}} \text{(TR-Var)} \qquad \frac{\Delta(\mathtt{m}) = (\overline{f}, \Delta_{\mathtt{m}}) \to (E, A) \qquad \sigma \text{ renaming} \quad E' = \mathit{instanceof}(E, \sigma) \qquad A' = \mathit{instanceof}(A, \sigma)}{\Delta \vdash \mathtt{m} : (\sigma(\overline{f}), \Delta_{\mathtt{m}} \circ \sigma) \to (E', A')} \text{(TR-Method-Sign)}$$

synchronisations: $\Delta, E, A \overset{\oplus}{\vdash}_S v \triangleright \Delta', E', A'$

$$\frac{\Delta, E, A \vdash x : f \triangleright E' \quad \Delta \vdash f : \lambda X.\mathtt{m}(f, \overline{g}, \Delta_{\mathtt{m}}, E_{\mathtt{m}})}{\Delta, E \overset{\oplus}{\vdash}_S x \triangleright \Delta, E' \sqcup E_{\mathtt{m}}|_S} \text{(TR-Sync-Invk)} \qquad \frac{\Delta, E \vdash v : f \triangleright E' \qquad \Delta \vdash f : \mathbb{f} \quad \mathbb{f} \neq \lambda X.\mathtt{m}(f, \overline{g}, \Delta_{\mathtt{m}}, E_{\mathtt{m}})}{\Delta, E \overset{\oplus}{\vdash}_S v \triangleright \Delta, E'} \text{(TR-Synchronized)}$$

expressions with side effects

- judgement used: $\Delta, E, A \vdash_S z : f \triangleright \Delta', E', A'$

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

$$\frac{\Delta, E \overset{\oplus}{\vdash} v \triangleright \Delta', E' \quad \Delta', E' \overset{\oplus}{\vdash} v' \triangleright \Delta'', E''}{\Delta, E, A \vdash_S v \oplus v' : \square \triangleright \Delta'', E'', A} \text{(TR-Expression)} \qquad \frac{\Delta, E, A \vdash \overline{v} : \overline{G} \triangleright E' \qquad F, \beta \text{ fresh}}{\Delta, E, A \vdash_S \mathtt{new}\ \mathtt{Act}(\overline{v}) : F \triangleright \Delta[f \mapsto \beta[\overline{x : G}]], E', A} \text{(TR-New)}$$

$$\frac{\begin{array}{c}\Delta, E, A \vdash v : F \triangleright E \\ \Delta, E, A \vdash \overline{v} : \overline{F'} \triangleright E' \qquad \Delta \vdash \mathtt{m} : (F, \overline{F'}, \Delta_{\mathtt{m}}) \to (E_{\mathtt{m}}, A_{\mathtt{m}}) \\ {}^{s}g \text{ fresh} \qquad \Delta' = \Delta[{}^{s}g \mapsto \lambda X.\mathtt{m}(F, \overline{F'}, \Delta_{\mathtt{m}}, E_{\mathtt{m}})] \\ \left(\mathit{Effects}(\Delta')(\beta) \# y^{(E_{\mathtt{m}} \sqcup A)(\beta.y)}\right)^{\beta \in \mathrm{dom}(E_{\mathtt{m}} \uplus A) \wedge y \in \mathit{fields}(\mathtt{Act})}\end{array}}{\Delta, E, A \vdash_S v.\mathtt{m}(\overline{v}) : g \triangleright \Delta', E', A \sqcup A_{\mathtt{m}}|_S} \text{(TR-Invk)}$$

Figure C.2 – Runtime typing rules for values, variables, method names, synchronisations and expressions with side effects

statements

- judgement used: $\Gamma, E, A \vdash_S s \ \triangleright \ \Gamma', E', A$

(TR-Assign-Var-Exp)

$$\frac{x \notin \mathit{fields}(\texttt{Act}) \qquad \Delta, E, A \vdash z : F \ \triangleright \ \Delta', E', A'}{\Delta, E, A \vdash_S x = z \ \triangleright \ \Delta'[x \mapsto F], E', A'}$$

(TR-Assign-Field-Exp)

$$\frac{x \in \mathit{fields}(\texttt{Act}) \qquad \Delta \vdash \mathit{this} : \alpha[\cdots] \qquad \Delta, E, A \vdash z : F \ \triangleright \ \Delta', E', A' \qquad \mathit{Effects}(\Delta')(\alpha) \# x^{\mathtt{w}} \qquad A'(\alpha) \# x^{\mathtt{w}}}{\Delta, E, A \vdash_S x = z \ \triangleright \ \Delta'[\mathit{this}.x \mapsto F], E'[\alpha.x \mapsto^{\sqcup} \mathtt{w}], A}$$

(TR-Skip)

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

(TR-Seq)

$$\frac{\Delta, E, A \vdash s_1 \ \triangleright \ \Delta_1, E_1, A_1 \qquad \Delta_1, E_1, A_1 \vdash s_2 \ \triangleright \ \Delta_2, E_2, A_2}{\Delta, E, A \vdash_S s_1; s_2 \ \triangleright \ \Delta_2, E_2, A_2}$$

(TR-Return)

$$\frac{\Delta, E, A \vdash v : F \ \triangleright \ E' \qquad \Delta(\texttt{destiny}) = f' \qquad \Delta \vdash f' : \lambda X.\mathtt{m}(\overline{g}, X, \Gamma_{\mathtt{m}}, E_{\mathtt{m}})}{\Delta, E, A \vdash_S \texttt{return} \ v \ \triangleright \ \Delta, E', A}$$

(TR-If)

$$\frac{\Delta, E, A \vdash e : f \ \triangleright \ \Delta', E', A' \qquad \Delta', E', A' \vdash s_1 \ \triangleright \ \Delta_1, E_1, A_1 \qquad \Delta', E', A' \vdash s_2 \ \triangleright \ \Delta_2, E_2, A_2 \qquad \Delta_1 =_{\texttt{unsync}} \Delta_2}{\Delta, E, A \vdash_S \texttt{if} \ e \ \{ \, s_1 \, \} \ \texttt{else} \ \{ \, s_2 \, \} \ \triangleright \ \Delta_1 + \Delta_2, E_1 \sqcup E_2, A_1 \sqcup A_2}$$

Figure C.3 – Runtime typing rules for statements.

execution of P, our type system can type cn only if cn has has non deterministic effects, as we formally state in the following lemma.

Lemma C.1.2. *Let P be a* gASP*, cn be a runtime configuration, and let Δ be such that $\Delta \vdash cn$; then cn has a queue with deterministic effects.*

By Definition 4 we know that a configuration cn has *deterministic effects* if every active object of this configuration has a *queue with deterministic effects*, then to prove lemma 5.5.5 we need to prove that given an active object $\alpha(a, p, \{q_1, \cdots, q_n\})$ if Δ exists such that $\Delta \vdash \alpha(a, p, \{q_1, \cdots, q_n\})$ then $\alpha(a, p, \{q_1, \cdots, q_n\})$ has a queue with deterministic effects.

Proof. Let $\alpha(a, p, \{q_1, \cdots, q_n\})$ be an active object with non deterministic effects,

this implies, by Definition 4, that one of the following predicates holds:

1. $x^{\mathtt{w}} \in q_i \wedge x^{\mathtt{w}} \in q_j$ where $x \in dom(a)$ and $i, j \in [1, n]$ with $i \neq j$
2. $x^{\mathtt{w}} \in q_i \wedge x^{\mathtt{r}} \in q_j$ where $x \in dom(a)$ and $i, j \in [1, n]$ with $i \neq j$

Let us also suppose that there exists Δ such that $\Delta \vdash \alpha(a, p, \{q_1, \cdots, q_n\})$.

Case 1: $x^{\mathtt{w}} \in q_i \wedge x^{\mathtt{w}} \in q_j$.

For each pair of $i, j \in [1, n]$ with $i \neq j$, by rule TR-Actor and TR-Process we gain that there exist Δ'_i and Δ'_j that extend Δ such that $\Delta'_i, \varnothing, \varnothing \vdash_{S_i} q_i \triangleright E_i, A_i$ and $\Delta'_j, \varnothing, \varnothing \vdash_{S_j} q_j \triangleright E_j, A_j$, and also the rule TR-Actor checks if the effects of q_i and the effects of q_j are compatible ($E_i \# E_j$).
By (1) we have that there exist $x \in dom(a)$ (which is equal to say that $x \in \mathit{fields}(\mathtt{Act})$) such that $x^{\mathtt{w}} \in q_i$ and $x^{\mathtt{w}} \in q_j$ that by definition means that there exist a statement s_i in q_i and a statement s_j in q_j such that $x = z$ for some z. To type the statement $x = z$ in q_i and q_j we have to apply the rule TR-Assign-Field which implies that $E_i(\alpha.x) = \mathtt{w}$ and $E_j(\alpha.x) = \mathtt{w}$ that makes the condition $E_i \# E_j$ false.

Case 2: $x^{\mathtt{w}} \in q_i \wedge x^{\mathtt{r}} \in q_j$.

For each pair of $i, j \in [1, n]$ with $i \neq j$, by rule TR-Actor and TR-Process we gain that there exist Δ'_i and Δ'_j that extend Δ such that $\Delta'_i, \varnothing, \varnothing \vdash_{S_i} q_i \triangleright E_i, A_i$ and $\Delta'_j, \varnothing, \varnothing \vdash_{S_j} q_j \triangleright E_j, A_j$, and also the rule TR-Actor checks if the effects of q_i and the effects of q_j are compatible ($E_i \# E_j$).
By (2) we have that there exist $x \in dom(a)$ such that $x^{\mathtt{w}} \in q_i$ and $x^{\mathtt{r}} \in q_j$ that by definition means that there exist a statement s_i in q_i such that $x = z$ for some z and a statement s_j in q_j such that x appear in the left side of an assignment or in the guard of an if statement. To type the statement $x = z$ in q_i we have to apply the rule TR-Assign-Field which implies that $E_i(\alpha.x) = \mathtt{w}$, whereas to type s_j we are going to apply the rule TR-Field which implies that $E_j(\alpha.x) = \mathtt{r}.$ that makes the condition $E_i \# E_j$ false.

As we have seen it is not possible that our type system is able to type a configuration if it presents an active object with a queue with non deterministic effects.

□

The second step to prove Theorem 5.5.4 is to prove that if we type a config-

uration cn and $cn \to cn'$, than there exist an environment Δ' such that Δ' types cn', as formalised in the follow.

Lemma C.1.3. *Let $\Delta \vdash cn$ and $cn \to cn'$. Then there exist Δ' such that $\Delta' \vdash cn'$.*

Proof. The proof of this lemma is a case analysis on the reduction rule used in $cn \to cn'$.

Case: Invk

Invk

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

By hypothesis we know that there exists Δ such that $\Delta \vdash \alpha(a, \{\ell \mid x = v.\mathtt{m}(\overline{v}) \ ; \ s\}, \overline{q}) \ \beta(a', p, \overline{q'})$ which implies that both $\overline{q}$ and $\overline{q'}$ have deterministic effects. We want to show that there exist Δ' such that $\Delta' \vdash \alpha(a, \{\ell \mid x = f \ ; \ s\}, \overline{q}) \ \beta(a', p', \overline{q'} \cup \{p'\}) \ f(\bot)$ which implies that after one execution step the queues of the two active objects still have deterministic effects. Whereas the queue of the actor α does not change, it is trivial to guarantee that the queue still has deterministic effects. The following proof will demonstrate that the queue of active object β that becames $\overline{q'} \cup \{p'\}$ preserves the properties.

Proof. To assert $\Delta \vdash \alpha(a, \{\ell \mid x = v.\mathtt{m}(\overline{v}) \ ; \ s\}, \overline{q}) \ \beta(a', p, \overline{q'})$ we need to apply the rules TR-Parallel, TR-Actor and TR-Process, then we have that:

- there exist Δ_1 that extends Δ as in the application of the rules TR-Actor and TR-Process
- there exist a set of future names S such that $S \subseteq \text{dom}(\Delta_1)$ as in the application of TR-Process;
- there exist E_p and A_p such that $\Delta \vdash \{\ell \mid x = v.\mathtt{m}(\overline{v}) \ ; \ s\} \ \triangleright \ E_p, A_p$
- there exist $E_1, \cdots, E_n$ and $A_1, \cdots, A_n$ such that $\Delta \vdash q_i \ \triangleright \ E_i, A_i$ where $i \in [1, n]$
- there exist $E_{p'}$ and $A_{p'}$ such that $\Delta \vdash p \ \triangleright \ E_{p'}, A_{p'}$

– there exist $E'_1, \cdots, E'_m$ and $A'_1, \cdots, A'_m$ such that $\Delta \vdash \overline{q'_i} \triangleright E_{q'_i}, A_{q'_i}$ where $i \in [1, m]$.

Applying the rule TR-Invk we gain that $\Delta, \varnothing, \varnothing \vdash S : v.\mathtt{m}(\overline{v}) \triangleright {}^{s}f\Delta', E', A_{\mathtt{m}}|_S$ where $\Delta' = \Delta[{}^{s}f \mapsto \lambda X.\mathtt{m}(g, \overline{g'}, \Delta_{\mathtt{m}}, E_{\mathtt{m}})]$ and $E' = [\beta.v' \mapsto^{\sqcup} \mathtt{r}]^{v' \in \overline{v}}$.

We want to notice that we can type the process $\{\ell \mid x = v.\mathtt{m}(\overline{v}) \ ; \ s\}$ as if $x = v.\mathtt{m}(\overline{v})$ were the first instruction of the method. We can do this because each instruction can be preceded by two kind of statements:

– statements where there are operations of reading or writing on a field
– method invocations.

The operations of reading and writing on fields, done by the running process, has not impact in the definition of queue with non deterministic effects, then this effects can be not considered, this is the reason why the environment which collect effects at start is empty. Whereas, the method invocations done previously can influence the definition of queue with non deterministic effects. All the statement that there are before $x = v.\mathtt{m}(\overline{v})$ have already been executed. We know that for each method invocation we create a future and the environment Δ is updated adding an entry that maps this future with the corresponding method result (i.e $f \mapsto \lambda X.\mathtt{m}(g, \overline{g'}, \Delta_{\mathtt{m}}, E_{\mathtt{m}})$). The method result contains the effects of the method just invoked, then we can conclude then that all the information related to the effects of the method invocations done before the current instruction are stored in Δ'.

By hypothesis we know that $\Delta \vdash: \ell_{q'}(\mathtt{destiny})f_i$ such that $\Delta \vdash: f_i\lambda X.\mathtt{m}_i(\overline{g_i}, \Delta_{\mathtt{m}_i}, E_{q'_i})$; then the last premise of the rule TR-invk checks if the effects of $\mathtt{m}$ are compatible with all the effect of all the methods stored in Δ that are not yet terminated $(\bigwedge_{i \in [1,m]} E_{\mathtt{m}} \# E_{q'_i})$.

Let us chose $\Delta' = \Delta[f \mapsto \lambda X.\mathtt{m}(F, \overline{F'}, \Delta_{\mathtt{m}}, E_{\mathtt{m}})]$ where $f = \imath({}^{s}f)$, we want to type the target configuration. By applying the rules TR-Parallel, TR-Actor and TR-Process we have that Δ' can type $beta(a', p', \overline{q'} \cup \{p'\})$ only if the last promise of the rule TR-Actor holds. This promise checks if the effects of all the processes in the current active object are compatible, more precisely checks if $(E_{q'_k} \# E_{q'_j})^{k,j \in [1,m] \wedge k \neq j}$ and if $(\bigwedge_{i \in [1,m]} E_{\mathtt{m}} \# E_{q'_i})$. Whereas the first condition holds by hypothesis (q' was already correctly typed) we can state that the second condition

holds because it corresponds to the condition that we have checked typing the source configuration. Then since the source configuration was correctly typed we have that the condition holds.

□

Case: Return

RETURN

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

The absence of non deterministic affects in the target configuration is guaranteed by the hypothesis because the queue does not change during the reduction. In the follow we will show how to chose Δ' such that $\Delta' \vdash \alpha(a', \{\ell' \mid s\}, \overline{q})\ f(w)$.

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 $\Delta' \vdash \alpha(a, \varnothing, \overline{q})\ f(w)$.

□

Case: Update

UPDATE

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

The absence of non deterministic affects in the target configuration is guaranteed by the hypothesis because the queue does not change during the reduction. In the follow we will show how to chose Δ' such that $\Delta' \vdash \alpha(a', \{\ell' \mid s\}, \overline{q})\ f(w)$.

Proof. Let Δ exists, such that $\Delta \vdash \alpha(a, \{\ell \mid s\}, \overline{q})\ f(w)$, then we can chose $\Delta' = \Delta[x \mapsto \Delta(w)]$ to type the target configuration applying the rules TR-Actor, TR-Process and TR-Future-Eval and in particular we have that $\Delta' \vdash (a'+\ell')(x) : \Delta'(x)$.

□

Case: Assign

Assign

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

By hypothesis we know that there exists Δ such that $\Delta \vdash \alpha(a, \{\ell \mid x = e\ ;\ s\}, \overline{q})$ which implies that the queue of the active object α has deterministic effects. We want to show that there exist Δ' such that $\Delta' \vdash \alpha(a', \{\ell' \mid s\}, \overline{q})$, which implies that after an operation of assignment the queues of of α still have deterministic effects.

Proof. The absence of non deterministic affects in the target configuration is guaranteed by the hypothesis because the queue does not change during the reduction. In the follow we will show how to chose Δ' such that $\Delta' \vdash \alpha(a', \{\ell' \mid s\}, \overline{q})$.

We can distinguish two cases:

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

   By applying the rules TR-Actor, TR-Process, TR-Expression and TR-Assign-Var-Exp we have that there exist Δ_1 and S like in the application of TR-Process, such that $\Delta_1, \varnothing, \varnothing \vdash_S x = e \ \triangleright\ \Delta_3, E_1, \varnothing$, where:

   - $\Delta_3 = \Delta_2[x \mapsto f]$, where f is the type of the evaluation of the expression e and Δ_2 is the update of Δ_1, which are obtained by applying TR-Expression to type e ($\Delta_1, \varnothing, \varnothing \vdash_S e : f \ \triangleright\ \Delta_2, E_1, \varnothing$);
   - E_1 is the environment storing effects that we obtain by typing e.

   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 $\Delta' \vdash \alpha(a, \{\ell' \mid s\}, \overline{q})$.

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

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 $E_1[\alpha.x \mapsto^{\sqcup} \texttt{w}]$.

Let us chose $\Delta' = \Delta_3$ we can type the target configuration, in particular we have $\Delta' \vdash a'(x) : \Delta'(x)$. We can type the target configuration applying the rules TR-Actor and TR-Process and we gain that $\Delta' \vdash \alpha(a, \{\ell' \mid s\}, \overline{q})$.

□

Case: New

New

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

By hypothesis we know that there exists Δ such that $\Delta \vdash \alpha(a, \{\ell \mid x = \texttt{new Act}(\overline{v}) \texttt{ ; } s\}, \overline{q})$ which implies that the queue of process to be executed of the active object α has deterministic effects. We want to show that there exist Δ' such that $\Delta' \vdash \alpha(a, \{\ell \mid x = \beta \texttt{ ; } s\}, \overline{q}) \;\; \beta([\overline{y} \mapsto \overline{w}], \varnothing, \varnothing)$ which implies that after one execution step the queues of the two active objects still have deterministic effects.

Proof. The absence of non deterministic affects in the target configuration is guaranteed by the hypothesis because the queue of α does not change during the reduction and the queue of β is empty. In the follow we will show how to chose Δ' such that $\Delta' \vdash \alpha(a, \{\ell \mid x = \beta \texttt{ ; } s\}, \overline{q}) \;\; \beta([\overline{y} \mapsto \overline{w}], \varnothing, \varnothing)$.

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 $\Delta_1, \varnothing, \varnothing \vdash_S \texttt{new Act}(\overline{v}) : f \triangleright \Delta_2, E_1, \varnothing$, where $\Delta_2 = \Delta_1[f \mapsto \beta[\overline{a : g}]^{\checkmark}]$ and $E_1 = [\alpha.v \mapsto^{\sqcup} \texttt{r}]^{v \in \overline{v}}$. Finally by applying the rules TR-Assign-Var-Exp and TR-Seq we obtain that $\Delta_2, E_1, \varnothing \vdash_S x = \texttt{new Act}(\overline{v}) \texttt{ ; } s \triangleright \Delta_3, E_2, A$ 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 $\Delta' \vdash \overline{w} : \Delta'(f.y)$.

□

□

Appendix D

Extended Abstract in French

D.1 Motivation

L'évolution de l'infrastructure Internet, le développement de technologies qui ont énormément augmenté la bande passante de connexion et l'augmentation exponentielle du nombre d'appareils mobiles ont ouvert la voie à de nouvelles technologies et de nouveaux paradigmes. De nos jours, les calculs sont de moins en moins centralisés et de plus en plus distribués.

Deux paradigmes deviennent de plus en plus importants, ayant un fort impact sur l'industrie et sur la recherche, ils sont: le cloud computing [Mell and Grance, 2011] et l'Internet of Things paradigm (IoT) [Al-Fuqaha et al., 2015]. De nos jours, l'industrie se précipite pour lancer de nouveaux produits basés sur le cloud ou l'IoT, tandis que le laboratoire de recherche et l'université tentent de développer une technique qui pourrait rendre l'implémentation de ces systèmes distribués complexes plus facile et plus évolutive.

Une des questions ouvertes que la recherche tente de résoudre est de trouver un modèle de calcul qui convient le mieux aux exigences de ces systèmes distribués. Grâce à ses caractéristiques, de plus en plus souvent, le modèle d'objet actif a été choisi comme modèle de calcul pour le système distribué.

Le modèle d'objet Active fournit une alternative au modèle de concurrence classique qui repose sur la synchronisation de l'état mutable partagé à l'aide de verrous. L'approche proposée par le modèle objet actif, qui repose sur des interactions

non bloquantes via des appels de méthode asynchrones, correspond parfaitement aux requêtes dans la programmation de plateformes distribuées complexes.

Par exemple, de par leur conception, les systèmes IoT impliquent une grande quantité de dispositifs qui, en interagissant les uns avec les autres et en exécutant des tâches minimales, changent leur état interne, dans les limites du coût et de la puissance. De la même manière, le modèle d'objet actif avec sa conception légère est capable de bien évoluer sans consommer trop de ressources informatiques, en décomposant la logique métier en tâches minimales exécutées par chaque objet actif.

Un autre élément qui relie les objets actifs et IoT est que, lors du développement de systèmes distribués complexes, il est presque impossible de collecter un grand nombre de périphériques pour faire fonctionner le système, des simulations sont utilisées pour tester ces systèmes. En fait, le modèle d'objet actif est très approprié pour développer la simulation d'énormes systèmes distribués grâce au fait que chaque objet actif a la possibilité de créer de nouveaux objets actifs avec des stratégies de supervision programmables. La génération d'un nouvel objet actif rend ce modèle également très approprié pour simuler des gestionnaires de périphériques et des groupes hiérarchiques de périphériques.

De plus, les fonctionnalités fournies par le modèle d'objet Active en font un bon choix également pour les applications cloud. La propriété d'évolutivité garantie par le modèle d'objet Active est précisément l'une des raisons principales qui a poussé Microsoft à baser son framework sur elle. En fait, comme l'a déclaré Microsoft, Orléans a comme objectif principal la construction d'applications informatiques distribuées et à grande échelle, sans avoir besoin d'apprendre et d'appliquer des modèles complexes de concurrence ou d'autres modèles de mise à l'échelle.

Il y a aussi de nombreuses autres raisons qui rendent le modèle d'objet actif approprié pour implémenter un système distribué complexe, tel que le couplage libre et l'isolation stricte des objets actifs, du fait que chaque tâche exécutée par un objet actif ne peut que modifier le état de l'objet actif qui l'exécute.

Bien que la simplification du développement d'un système distribué complexe soit un problème notable, il est également très important de vérifier que ces systèmes ne présentent pas de bogues de concurrence connus. En effet, les problèmes tels que les interblocages, les conditions de course et les courses de données ne sont

pas rares dans les programmes distribués.

Le modèle d'objet actif permet la formalisation et la vérification grâce au fait que le schéma de communication est bien défini ainsi que les accès aux objets et aux variables. Les caractéristiques du modèle d'objet actif facilitent le développement d'outils pour l'analyse statique et la validation des programmes, qui peuvent être utilisés par les programmeurs pour éviter et détecter les bogues de concurrence.

La possibilité de développer une analyse efficace avec les autres fonctionnalités déjà mentionnées, a amené le modèle d'objet actif à être utilisé dans certains projets européens importants, tels que le projet Envisage, dans le domaine du cloud computing.

Pour toutes ces raisons, l'approche objet actif peut être considérée comme un très bon modèle pour développer des systèmes distribués complexes en raison du couplage lâche, de l'isolation stricte, de l'absence de mémoire partagée et de la responsabilité de formalisation et de vérification.

Par conséquent, de nombreux modèles et langages basés sur des objets actifs ont été développés au cours des années. Ces modèles et ces langages diffèrent généralement en ce qui concerne la façon dont les futurs (objets spéciaux retournés par des invocations asynchrones, utilisés pour gérer les synchronisations) sont traités et comment les modèles de synchronisation peuvent être construits.

Un autre aspect important qui différencie ces modèles est de savoir combien ils peuvent cacher au programmeur tous les aspects qui concernent la concurrence et la distribution. La transparence joue des règles importantes dans la définition d'un modèle d'objet actif, notamment parce que les objets actifs veulent être une abstraction bien intégrée pour la concurrence et la distribution qui laisse les programmeurs concentrés sur les exigences fonctionnelles. Cependant, la plupart des modèles d'objets Active fournissent une définition future explicite et des opérations spéciales pour gérer les synchronisations, ce qui permet au programmeur de vérifier si la méthode est terminée et en même temps d'extraire le résultat de la méthode. Les informations fournies par les programmeurs facilitent le développement d'outils de vérification, qui peuvent également démontrer un bon niveau de précision dans l'analyse des modèles de synchronisation. Cependant, avec cette approche, les programmeurs doivent savoir comment gérer les synchronisations, et peuvent être tentés d'ajouter trop de points de synchronisation pour simplifier le raisonnement

sur le programme.

D.2 Objectifs

Nous avons vu que le développement de bons outils de vérification pour l'analyse des modèles de synchronisation est important autant que la transparence du modèle, notamment parce que dans les systèmes distribués, les problèmes comme les interblocages, les courses de données et les conditions de course représentent des menaces insidieuses et récurrentes.

Avec cette thèse, nous voulons aider les programmeurs qui utilisent le modèle objet actif à implémenter des systèmes distribués, en fournissant une technique d'analyse de logiciel capable d'analyser les modèles de synchronisation afin de détecter la présence de blocages et d'analyser les effets. En outre, voulant aider davantage le programmeur, avec notre analyse nous voulons cibler un modèle d'objet actif avec un très haut niveau de transparence. Le modèle actif que nous voulons cibler est un modèle dans lequel les futurs ne sont pas explicitement identifiés, alors les programmeurs ne doivent pas faire de distinction entre les valeurs et le futur, et où les synchronisations sont implicites et effectuées uniquement sur la disponibilité du résultat d'une invocation de méthode .

De nombreuses techniques d'analyse statique et dynamique différentes ont été utilisées pour analyser les blocages et plus généralement les modèles de synchronisation, tels que: l'interprétation abstraite, la vérification de modèle, l'exécution symbolique, l'analyse de flux de données. Malheureusement, comme nous le verrons dans la Section 2.8, la plupart de ces techniques manquent dans l'analyse des interblocages pour les systèmes de récurrence mutuelle et de création dynamique de ressources. De plus, un engagement fort du programmeur est nécessaire pour annoter les programmes afin de traiter ces analyses.

Dans cette thèse, nous voulons également fournir une analyse qui peut être réalisée avec très peu d'interaction humaine, ou mieux de manière entièrement automatique, qui n'a pas d'impact sur la performance du système, et qui peut facilement évoluer.

Une autre caractéristique importante qu'une analyse de programme devrait avoir est la possibilité d'être facilement étendu, amélioré et même adapté pour

analyser différents modèles. Afin d'assurer une bonne adaptabilité et maintenabilité de l'analyse, nous souhaitons développer une approche modulaire qui permette également de combiner plusieurs techniques.

Enfin, en prenant en compte toutes les difficultés que nous pouvons rencontrer lors du développement d'une telle analyse, nous terminons la thèse par une réflexion sur les caractéristiques essentielles et les meilleures stratégies de synchronisation qu'un modèle d'objet actif devrait avoir, afin d'identifier les bon mélange entre transparence et aspects de vérification qu'un objet actif devrait avoir.

D.3 Contributions

La contribution globale de cette thèse est de fournir une technique d'analyse statique pour la vérification de l'absence de blocages et une analyse des effets dans les programmes d'objets actifs avec des futurs transparents. Le développement de cette technique d'analyse statique nous donne aussi la possibilité de mieux comprendre comment différentes approches de synchronisation peuvent impacter la vérification statique. La contribution principale de cette thèse peut être résumée en trois points principaux décrits ci-dessous. Plus de détails sur le contenu des chapitres sont disponibles dans la Section 1.4.

D.3.1 Technique d'analyse d'interblocage pour le modèle d'objet actif.

La première contribution propose une technique d'analyse statique basée sur des types comportementaux dans laquelle on a supprimé la possibilité d'avoir un objet actif avec état, afin de se concentrer sur les deux principales caractéristiques du modèle: l'absence de types futurs explicite et de synchronisations implicites. Comme nous le verrons plus en détail dans le Chapitre 4, la combinaison de ces deux caractéristiques du modèle peut amener à la nécessité de produire un ensemble de dépendances entre objets actifs pouvant être illimités en cas de méthodes récursives. Nous fournissons:

- un **système de type comportementaux** associant des types comportementaux aux méthodes de programme;
- une **analyse de type comportementaux** capable de traduire des types comportementaux en un graphe potentiellement illimité de dépendances "en attente de";
- une **adaptation de l'analyse des circularités** proposée dans [Kobayashi and Laneve, 2017, Giachino and Laneve, 2014], qui, en prenant comme entrée le système de type comportementaux et l'analyse de type comportementaux, est capable de détecter les blocages;
- la **preuve de l'exactitude** de l'analyse de l'impasse proposée.

D.3.2 Analyse d'effet pour les programmes d'objets actifs.

La deuxième contribution principale de cette thèse est une technique d'analyse d'effets basée sur des systèmes de type comportementaux capables de détecter la présence de conditions de course pouvant introduire un non-déterminisme dans l'exécution d'un programme d'objets-actifs. Dans ce cas également, **la preuve de l'exactitude** du système d'effets a été fournie.

D.3.3 Technique d'analyse d'interblocage gérant un objet actif avec état.

La troisième contribution principale de cette thèse est l'extension de la technique d'analyse statique proposée au Chapitre 4, qui prend en compte les informations fournies par l'analyse des effets et permet de détecter les blocages dans un programme d'objets actifs avec des objets actifs avec état.... Dans cette contribution, nous avons fourni: **un système de type comportementaux**, une **analyse de type comportementaux**, **l'adaptation de l'analyse des circularités**, et les **preuves de la justesse** de l'analyse de l'impasse proposée.

D.4 Conclusion

À l'heure où les logiciels ne sont plus centralisés et où les paradigmes de l'informatique en nuage et de l'Internet des objets dominent dans l'environnement de l'industrie et de la recherche, les modèles Actor et Active deviennent de plus en plus importants.

Malgré le fait que ces deux modèles facilitent le développement de systèmes distribués complexes, la mise en œuvre de ces systèmes reste une tâche non triviale. Cela est dû au fait que les bogues de concurrence connus tels que les interblocages, les conditions de concurrence et les courses de données ne sont pas rares dans les systèmes distribués.

Avec cette thèse, nous aidons les programmeurs qui utilisent le modèle d'objet actif à implémenter des systèmes distribués complexes, en fournissant une technique d'analyse de logiciel pour analyser les modèles de synchronisation. L'analyse proposée permet, sans nécessiter d'interaction humaine, de détecter l'absence d'interblocages et d'analyser les effets. Notre objectif était de fournir une analyse de programme modulaire et combinant plusieurs techniques différentes.

La première contribution présentée dans cette thèse propose une technique d'analyse statique basée sur des types comportementaux. L'analyse statique que nous avons développée est composée de: un système de type comportemental qui associe les types comportementaux aux méthodes du programme; une analyse de type comportemental qui traduit les types comportementaux en un graphique potentiellement illimité des dépendances «en attente»; et une analyse des circularités qui prend en entrée le programme de type comportemental et détecte les blocages en temps fini.

Le développement de cette analyse est axé sur la façon de gérer les futurs types implicites et les synchronisations wbn. En fait, la combinaison de ces deux fonctions peut nécessiter de définir un ensemble illimité de dépendances entre les objets actifs, si nous synchronisons une méthode récursive qui renvoie un futur.

Le premier problème est d'identifier quels paramètres sont des futures ou des valeurs. Il a été résolu en tapant chaque paramètre comme un futur potentiel non associé à une invocation de méthode. Ensuite, nous avons retardé l'identification de la nature du paramètre (futur ou valeur) et la liaison du nom futur à l'invocation de la méthode à l'analyse comportementale.

Le deuxième problème, lié à l'imbrication de futurs qui peuvent produire un ensemble illimité de paires de dépendances, a été résolu en déplaçant la génération des dépendances vers l'analyse de type comportemental. Nous fournissons le type comportemental de chaque méthode avec la paire spéciale (X, α), où X est un espace réservé pour le nom de l'objet actif qui va synchroniser la méthode, alors que α est le nom de l'objet actif exécutant la méthode en cours. L'espace réservé X sera instancié, lors de l'analyse de type comportemental, uniquement dans le cas de synchronisations et non pour des invocations de méthodes.

La deuxième contribution de cette thèse est le développement d'une analyse d'effets, basée sur des systèmes de types comportementaux, pour détecter la présence de conditions de course susceptibles d'introduire un non-déterminisme dans l'exécution d'un programme d'objets-actifs. Le système de type comportemental associe à chaque méthode une fonction qui trace les effets. Cette fonction mappe les futurs noms des paramètres consultés lors de l'exécution de la méthode sur un ensemble de noms de champs. Ces noms de champs sont étiquetés avec `r` ou `w`, indiquant respectivement un accès en lecture ou en écriture. Merci à cette information supplémentaire, nous sommes en mesure de taper uniquement les programmes avec des effets déterministes. L'impossibilité d'avoir des files d'attente avec des effets non déterministes garantit que: si une méthode stocke un futur dans le champ d'un argument, alors l'accès suivant au champ devrait avoir lieu après la fin de cette méthode.

La troisième contribution principale de cette thèse est une extension de la première contribution afin d'analyser le programme d'objets actifs avec des objets actifs avec état. Cette analyse prend en compte les informations fournies par l'analyse des effets et identifie l'absence d'impasses. L'analyse statique que nous avons développée, comme la précédente, est composée de: un système de type comportemental; une analyse de type comportemental; et une adaptation de l'analyse des circularités. Le principal problème abordé par cette analyse est le suivi des contrats à terme stockés dans des champs d'objets. En effet, les informations fournies par l'analyse des effets nous permettent seulement de savoir quand une méthode déjà synchronisée a stocké un futur dans un champ d'argument. Comment savoir l'identité de ce futur et l'invocation de la méthode qui lui est associée en dehors du contexte qui a invoqué cette méthode, était toujours un problème ouvert. Ce

problème a été résolu en présentant les délégations. Les délégations qui ont la forme $g \rightsquigarrow o.\, x$, indiquent que la méthode liée au futur g a stocké un futur dans le champ x de l'objet o. Les délégations nous permettent de déplacer l'identification de l'invocation de la méthode, pour un futur stocké dans un champ, à l'analyse du comportement, dans lequel nous avons des informations plus globales.

Dans l'ensemble, nous construisons une analyse éprouvée et complète des modèles de synchronisation pour les programmes implémentés à l'aide du modèle d'objet Active. Nous avons développé deux analyses: une technique d'analyse de blocage basée sur des types comportementaux dans laquelle nous avons développé un système de type comportemental, une analyse de type comportemental, et nous avons étendu l'analyse des circulaires présentées dans [Giachino et al., 2014, Kobayashi and Laneve, 2017]; et une analyse des effets également basée sur les types de comportement. Nous abordons dans cette thèse les problèmes liés à l'analyse d'un modèle Actor avec les types futurs implicites, la synchronisation *wait-by-necessity* et les objets actifs avec état. Les futurs types implicites et la synchronisation *wait-by-necessity* nécessitent la création d'ensembles non bornés de paires de dépendances dans le cas de synchronisations, ainsi que l'identification de futurs et de points de synchronisation. La gestion d'objets actifs avec état requiert de manipuler ou de détecter le non-déterminisme causé par l'ordre non-déterministe des accès sur les champs, et nécessite le suivi des futurs noms et invocations de méthodes synchronisés dans un contexte différent de celui dans lequel ils sont créés.

D.5 Perspectives

Les résultats présentés dans cette thèse sont prometteurs pour le modèle d'objet actif et sa vérification. Le travail que nous avons fait est un bon point de départ pour plusieurs sujets de recherche, que nous aborderons dans les prochaines sous-sections.

D.5.1 Amélioration de l'analyse

En ce qui concerne les travaux futurs, nous pensons immédiatement aux améliorations possibles de notre analyse. La première façon d'améliorer notre analyse est certainement liée à la suppression de certaines restrictions. La majeure partie de la restriction imposée à `gASP` et énoncée dans la Section 4.1 et la section 5.1 ont été imposées afin de simplifier le système de type comportemental. Comment supprimer ces restrictions est bien connu pour nous et il a été discuté dans la Section 4.9. La seule restriction présente dans nos deux travaux, et qui n'a pas été étudiée dans cette thèse, est l'impossibilité d'avoir des types de données récursifs. En fait, dans la Section 4.1 et la Section 5.1, il est dit que le champ d'un objet doit être de type `Int`, alors ils peut être seulement un entier ou un futur d'un entier. Comment gérer le type de données récursif est une tâche non triviale, en fait cela nécessite d'étendre notre analyse pour la rendre capable de gérer des enregistrements d'objets illimités.

Grâce à la modularité de notre analyse, nous pensons que la gestion des enregistrements d'objets non bornés peut être réalisée par une analyse séparée. Une telle analyse devrait être capable d'aplatir la structure des objets, afin de donner à notre analyse des types d'objets sous la forme d'un enregistrement, comme c'est le cas actuellement. La technique précise qui peut être utilisée pour aplatir les structures de données non bornées n'a pas été investie.

L'amélioration de notre analyse peut également être étudiée en dehors du champ des restrictions. Notre analyse, perdant un peu en général, peut gagner en précision si un algorithme d'ordonnancement précis est fixé. Prendre en compte une approche d'ordonnancement précise (c'est-à-dire l'ordonnancement FIFO), nous donne beaucoup plus d'informations liées à l'entrelacement réel des processus. Comme nous l'avons vu dans cette thèse, la quantité d'informations que l'analyse est capable d'extraire du programme est directement proportionnelle à sa précision. En fait, dans l'état actuel, l'analyse des interblocages et des effets tient compte du fait que deux invocations de méthodes, même si elles sont effectuées par la même méthode, peuvent être planifiées dans n'importe quel ordre.

Considérer la programmation FIFO dans notre analyse ne nécessite que des modifications mineures dans la règle de typage de la méthode. En fait, nous devons

diviser le type comportemental de chaque méthode en deux parties. La première partie doit être liée au comportement du corps de cette méthode, tandis que la seconde partie doit collecter l'invocation de méthode effectuée par la méthode. Cette solution nous permet de définir une composition parallèle plus précise des comportements des invocations de méthode, notamment dans le cas de plusieurs invocations effectuées par une méthode sur le même objet.

D.5.2 Nouveau paradigme pour la conception de futurs

Le développement d'une analyse pour les modèles de synchronisation souligne l'impact que la conception et la mise en œuvre des futurs a sur la facilité d'utilisation pour les programmeurs et sur la complexité et la précision de l'analyse.

D'un côté, les futurs explicites donnent plus de contrôle au programmeur; permettre des opérations complexes sur des contrats à terme (c'est-à-dire le multithreading coopératif ou le chaînage futur); et permettent également une identification plus précise des points de synchronisation. D'un autre côté, les contrats à terme implicites empêchent le programmeur de savoir si le flux de contrôle fonctionne sur des valeurs normales ou sur des contrats à terme; bloquer le programme seulement quand un futur est vraiment nécessaire; permettre une meilleure réutilisation du code (les méthodes sont écrites de la même manière indépendamment des variables qui contiennent un futur ou une valeur); et admet également le retour du futur pour les invocations de méthodes récursives.

Nous pouvons résumer en disant que, bien que les contrats à terme implicites imposent la synchronisation des flux de données, les contrats à terme explicites implémentent la synchronisation des flux de contrôle à travers une déclaration qui remplit le futur.

Alors que nous étions confrontés à tous les problèmes liés au développement d'une telle analyse, nous essayions également de comprendre s'il est possible de mélanger les deux approches afin d'obtenir les avantages des deux.

En fait, l'un des futurs travaux possibles sera la définition d'un nouveau paradigme pour la conception et la mise en œuvre de futurs.

Dans la suite, nous présenterons les caractéristiques que ce nouveau modèle devrait avoir et comment ces caractéristiques donneront l'avantage à la fois de

l'approche du flux de contrôle et de la synchronisation des flux de données.

La première caractéristique de ce modèle est l'utilisation de **types futurs explicites**. Les futurs à terme explicites de ce modèle ne doivent pas être considérés comme l'avenir du langage objet actif en tant qu'ABS, dans lequel les types futurs sont définis en tant que types paramétriques. Les contrats à terme doivent être explicitement dactylographiés, mais ce type indique seulement que nous nous référons à un futur possible (une variable typée comme future peut encore stocker une valeur), mais aucune distinction n'est faite entre un futur d'un entier et un futur ou un futur de un nombre entier. Taper le futur de cette façon, nous avons cela: le programmeur a plus de contrôle sur le point de synchronisation et est exposé aux points de synchronisation qui se produisent dans son programme; et il est plus facile de suivre les contrats à terme statiques. Ce sont les deux principaux avantages d'une approche avec des types futurs explicites. De plus, cette nouvelle définition des futurs types explicites permet également une meilleure réutilisation du code et permet des méthodes récursives qui retournent les futures.

La deuxième caractéristique de ce nouveau modèle devrait être une approche **synchronisation de flux de données explicite**, dans laquelle une opération de synchronisation simple est définie, comme dans l'approche de synchronisation de flux de contrôle, mais avec une orientation de flux de données. Cette opération de synchronisation résout un futur retournant une valeur, même dans le cas de futurs imbriqués (c'est-à-dire la fonction factorielle), exactement comme le fait la synchronisation *wait-by-necessity*. Avoir une opération de synchronisation explicite définie de cette manière donne plus de contrôle au programmeur et permet la définition d'une seule fonction récursive de synchronisation.

ACKNOWLEDGMENT: This work was partly funded by the French Government (National Research Agency, ANR) through the "Investments for the Future" Program reference ANR-11-LABX-0031-01.

Bibliography

The Associated Press. general electric acknowledges northeastern blackout bug. http://www.securityfocus.com/news/8032.

Martin Abadi, Cormac Flanagan, and Stephen N. Freund. Types for safe locking: Static race detection for java. *ACM Trans. Program. Lang. Syst.*, 28(2):207–255, March 2006. ISSN 0164-0925. doi: 10.1145/1119479.1119480. URL http://doi.acm.org/10.1145/1119479.1119480.

Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, USA, 1986. ISBN 0-262-01092-5.

Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich, editors. *Deductive Software Verification - The KeY Book - From Theory to Practice*, volume 10001 of *Lecture Notes in Computer Science*. Springer, 2016. ISBN 978-3-319-49811-9. doi: 10.1007/978-3-319-49812-6. URL http://dx.doi.org/10.1007/978-3-319-49812-6.

A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, and M. Ayyash. Internet of things: A survey on enabling technologies, protocols, and applications. *IEEE Communications Surveys Tutorials*, 17(4):2347–2376, 2015. ISSN 1553-877X. doi: 10.1109/COMST.2015.2444095.

Elvira Albert, Antonio E. Flores-Montoya, and Samir Genaim. *Analysis of May-Happen-in-Parallel in Concurrent Objects*, pages 35–51. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. ISBN 978-3-642-30793-5. doi: 10.1007/978-3-642-30793-5_3. URL https://doi.org/10.1007/978-3-642-30793-5_3.

Elvira Albert, Puri Arenas, Antonio Flores-Montoya, Samir Genaim, Miguel Gómez-Zamalloa, Enrique Martin-Martin, German Puebla, and Guillermo Román-Díez. *SACO: Static Analyzer for Concurrent Objects*, pages 562–567. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014. ISBN 978-3-642-54862-8. doi: 10.1007/978-3-642-54862-8_46. URL http://dx.doi.org/10.1007/978-3-642-54862-8_46.

Frances E. Allen. Control flow analysis. In *Proceedings of a Symposium on Compiler Optimization*, pages 1–19, New York, NY, USA, 1970. ACM. doi: 10.1145/800028.808479. URL http://doi.acm.org/10.1145/800028.808479.

Brian Amedro, Denis Caromel, Fabrice Huet, Vladimir Bodnartchouk, Christian Delbé, and Guillermo L. Taboada. HPC in Java: Experiences in Implementing the NAS Parallel Benchmarks. *APPLIED INFORMATICS AND COMMUNICATIONS*, August 2010. URL https://hal.inria.fr/inria-00504630.

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. ISSN 2352-2208. doi: http://dx.doi.org/10.1016/j.jlamp.2017.02.003. URL http://www.sciencedirect.com/science/article/pii/S2352220817300287.

Davide Ancona, Viviana Bono, Mario Bravetti, Joana Campos, Giuseppe Castagna, Pierre-Malo Deniélou, Simon J. Gay, Nils Gesbert, Elena Giachino, Raymond Hu, Einar Broch Johnsen, Francisco Martins, Viviana Mascardi, Fabrizio Montesi, Rumyana Neykova, Nicholas Ng, Luca Padovani, Vasco T. Vasconcelos, and Nobuko Yoshida. Behavioral types in programming languages. *Foundations and Trends® in Programming Languages*, 3(2-3):95–230, 2016. ISSN 2325-1107. doi: 10.1561/2500000031. URL http://dx.doi.org/10.1561/2500000031.

L. Baduel, F. Baude, and D. Caromel. Object-oriented spmd. In *Proceedings of the Fifth IEEE International Symposium on Cluster Computing and the Grid (CCGrid'05) - Volume 2 - Volume 02*, CCGRID '05, pages 824–831, Washington,

DC, USA, 2005. IEEE Computer Society. ISBN 0-7803-9074-1. URL http://dl.acm.org/citation.cfm?id=1169223.1169589.

Laurent Baduel, Françoise Baude, Denis Caromel, Arnaud Contes, Fabrice Huet, Matthieu Morel, and Romain Quilici. *Grid Computing: Software Environments and Tools*, chapter Programming, Composing, Deploying for the Grid, pages 205–229. Springer London, London, 2006. ISBN 978-1-84628-339-0. doi: 10.1007/1-84628-339-6_9. URL http://dx.doi.org/10.1007/1-84628-339-6_9.

Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, 2008. ISBN 026202649X. URL http://www.amazon.com/Principles-Model-Checking-Christel-Baier/dp/026202649X%3FSubscriptionId%3D13CT5CVB80YFWJEPWS02%26tag%3Dws%26linkCode%3Dxm2%26camp%3D2025%26creative%3D165953%26creativeASIN%3D026202649X.

Henry. G. Baker Jr. and Carl Hewitt. The incremental garbage collection of processes. In *Proc. Symp. on Artificial Intelligence and Programming Languages*, pages 55–59. New York, NY, USA, 1977.

Françoise Baude, Ludovic Henrio, and Cristian Ruz. Programming distributed and adaptable autonomous components—the gcm/proactive framework. *Software: Practice and Experience*, 45(9):1189–1227, 2015. ISSN 1097-024X. doi: 10.1002/spe.2270. URL http://dx.doi.org/10.1002/spe.2270.

Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt. *Verification of Object-oriented Software: The KeY Approach*. Springer-Verlag, Berlin, Heidelberg, 2007. ISBN 3-540-68977-X, 978-3-540-68977-5.

Saddek Bensalem and Klaus Havelund. Dynamic deadlock analysis of multi-threaded programs. In *Proceedings of the First Haifa International Conference on Hardware and Software Verification and Testing*, HVC'05, pages 208–223, Berlin, Heidelberg, 2006. Springer-Verlag. ISBN 3-540-32604-9, 978-3-540-32604-5. doi: 10.1007/11678779_15. URL http://dx.doi.org/10.1007/11678779_15.

Philip A. Bernstein and Sergey Bykov. Developing cloud services using the orleans virtual actor model. *IEEE Internet Computing*, 20(5):71–75, 2016.

Nikolaos Bezirgiannis and Frank Boer. *SOFSEM 2016: Theory and Practice of Computer Science: 42nd International Conference on Current Trends in Theory and Practice of Computer Science, Harrachov, Czech Republic, January 23-28, 2016, Proceedings*, chapter ABS: A High-Level Modeling Language for Cloud-Aware Programming, pages 433–444. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016. ISBN 978-3-662-49192-8. doi: 10.1007/978-3-662-49192-8_35. URL http://dx.doi.org/10.1007/978-3-662-49192-8_35.

K. Birman and T. Joseph. Exploiting virtual synchrony in distributed systems. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, SOSP '87, pages 123–138, New York, NY, USA, 1987. ACM. ISBN 0-89791-242-X. doi: 10.1145/41457.37515. URL http://doi.acm.org/10.1145/41457.37515.

Frank De Boer, Vlad Serbanescu, Reiner Hähnle, Ludovic Henrio, Justine Rochas, Crystal Chang Din, Einar Broch Johnsen, Marjan Sirjani, Ehsan Khamespanah, Kiko Fernandez-Reyes, and Albert Mingkun Yang. A survey of active object languages. *ACM Comput. Surv.*, 50(5):76:1–76:39, October 2017. ISSN 0360-0300. doi: 10.1145/3122848. URL http://doi.acm.org/10.1145/3122848.

Ahmed Bouajjani and Michael Emmi. Analysis of recursively parallel programs. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 203–214, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1083-3. doi: 10.1145/2103656.2103681. URL http://doi.acm.org/10.1145/2103656.2103681.

Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '02, pages 211–230, New York, NY, USA, 2002. ACM. ISBN 1-58113-471-1. doi: 10.1145/582419.582440. URL http://doi.acm.org/10.1145/582419.582440.

Stephan Brandauer, Elias Castegren, Dave Clarke, Kiko Fernandez-Reyes, Einar Broch Johnsen, Ka I. Pun, S. Lizeth Tapia Tarifa, Tobias Wrigstad, and Albert Mingkun Yang. *Formal Methods for Multicore Programming: 15th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2015, Bertinoro, Italy, June 15-19, 2015, Advanced Lectures*, chapter Parallel Objects for Multicores: A Glimpse at the Parallel Language Encore, pages 1–56. Springer International Publishing, Cham, 2015. ISBN 978-3-319-18941-3. doi: 10.1007/978-3-319-18941-3_1. URL `http://dx.doi.org/10.1007/978-3-319-18941-3_1`.

Sergey Bykov, Alan Geller, Gabriel Kliot, James R. Larus, Ravi Pandya, and Jorgen Thelin. Orleans: cloud computing for everyone. In Jeffrey S. Chase and Amr El Abbadi, editors, *ACM Symposium on Cloud Computing in conjunction with SOSP, SOCC, Cascais, Portugal*. ACM, 2011. ISBN 978-1-4503-0976-9.

Richard Carlsson and Hakan Millroth. On cyclic process dependencies and the verification of absence of deadlocks in reactive systems, 1997.

D. Caromel, C. Delbé, A. di Costanzo, and M. Leyton. ProActive: an integrated platform for programming and running applications on grids and P2P systems. *Computational Methods in Science and Technology*, 12(1):69–77, 2006.

Denis Caromel and Ludovic Henrio. *A Theory of Distributed Objects*. Springer Publishing Company, Incorporated, 1st edition, 2005. ISBN 3642058841, 9783642058844.

Denis Caromel, Ludovic Henrio, and Bernard Paul Serpette. Asynchronous and deterministic objects. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, pages 123–134, New York, NY, USA, 2004. ACM. ISBN 1-58113-729-X. doi: 10.1145/964001.964012. URL `http://doi.acm.org/10.1145/964001.964012`.

Elias Castegren and Tobias Wrigstad. Reference capabilities for concurrency control. In *Proceedings of 30th European Conference on Object-oriented Programming (ECOOP)*. Springer, 2016.

Dave Clarke, Tobias Wrigstad, Johan Östlund, and Einar Broch Johnsen. Minimal ownership for active objects. In *Proceedings of the 6th Asian Symposium on Programming Languages and Systems*, APLAS '08, pages 139–154, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-89329-5. doi: 10.1007/978-3-540-89330-1_11. URL http://dx.doi.org/10.1007/978-3-540-89330-1_11.

E. G. Coffman, M. Elphick, and A. Shoshani. System deadlocks. *ACM Comput. Surv.*, 3(2):67–78, June 1971. ISSN 0360-0300. doi: 10.1145/356586.356588. URL http://doi.acm.org/10.1145/356586.356588.

Melvin E. Conway. Design of a separable transition-diagram compiler. *Commun. ACM*, 6(7):396–408, July 1963. ISSN 0001-0782. doi: 10.1145/366663.366704. URL http://doi.acm.org/10.1145/366663.366704.

Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM. doi: 10.1145/512950.512973. URL http://doi.acm.org/10.1145/512950.512973.

Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '79, pages 269–282, New York, NY, USA, 1979. ACM. doi: 10.1145/567752.567778. URL http://doi.acm.org/10.1145/567752.567778.

Alfons Crespo, Juan Antonio Vila Carbó, and M Garcia-Valls. Resource management for mobile operating systems based on the active object model. *Computer Systems Science and Engineering*, 28(4):225–235, 2013.

Haskell B. Curry and Robert Feys. *Combinatory Logic*, volume 1. North Holland, 1958. Second edition, 1968.

Ole-Johan Dahl, Bjørn Myhrhaug, and Kristen Nygaard. *SIMULA 67 Common Base Language, (Norwegian Computing Center. Publication)*. 1968. ISBN B0007JZ9J6.

B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 2002.

Frank S. de Boer, Dave Clarke, and Einar Broch Johnsen. A complete guide to the future. In *Proceedings of the 16th European Conference on Programming*, ESOP'07, pages 316–330, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 978-3-540-71314-2. URL `http://dl.acm.org/citation.cfm?id=1762174.1762205`.

Frank S. de Boer, Mario Bravetti, Immo Grabe, Matias Lee, Martin Steffen, and Gianluigi Zavattaro. *A Petri Net Based Analysis of Deadlocks for Active Objects and Futures*, pages 110–127. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. ISBN 978-3-642-35861-6. doi: 10.1007/978-3-642-35861-6_7. URL `https://doi.org/10.1007/978-3-642-35861-6_7`.

Jessie Dedecker, Tom Van Cutsem, Stijn Mostinckx, Theo D'Hondt, and Wolfgang De Meuter. Ambient-oriented programming in ambienttalk. In *Proceedings of the 20th European Conference on Object-Oriented Programming*, ECOOP'06, pages 230–254, Berlin, Heidelberg, 2006. Springer-Verlag. ISBN 3-540-35726-2, 978-3-540-35726-1. doi: 10.1007/11785477_16. URL `http://dx.doi.org/10.1007/11785477_16`.

Crystal Chang Din, Richard Bubel, and Reiner Hähnle. *Automated Deduction - CADE-25: 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings*, chapter KeY-ABS: A Deductive Verification Tool for the Concurrent Modelling Language ABS, pages 517–526. Springer International Publishing, Cham, 2015a. ISBN 978-3-319-21401-6. doi: 10.1007/978-3-319-21401-6_35. URL `http://dx.doi.org/10.1007/978-3-319-21401-6_35`.

Crystal Chang Din, S. Lizeth Tapia Tarifa, Reiner Hähnle, and Einar Broch Johnsen. *Formal Methods and Software Engineering: 17th International Conference on Formal Engineering Methods, ICFEM 2015, Paris, France,*

November 3-5, 2015, Proceedings, chapter History-Based Specification and Verification of Scalable Concurrent and Distributed Systems, pages 217–233. Springer International Publishing, Cham, 2015b. ISBN 978-3-319-25423-4. doi: 10.1007/978-3-319-25423-4_14. URL http://dx.doi.org/10.1007/978-3-319-25423-4_14.

Mark Dowson. The ariane 5 software failure. *SIGSOFT Softw. Eng. Notes*, 22(2): 84–, March 1997. ISSN 0163-5948. doi: 10.1145/251880.251992. URL http://doi.acm.org/10.1145/251880.251992.

Dawson Engler and Ken Ashcraft. Racerx: Effective, static detection of race conditions and deadlocks. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 237–252, New York, NY, USA, 2003. ACM. ISBN 1-58113-757-5. doi: 10.1145/945445.945468. URL http://doi.acm.org/10.1145/945445.945468.

John Erickson, Madanlal Musuvathi, Sebastian Burckhardt, and Kirk Olynyk. Effective data-race detection for the kernel. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 151–162, Berkeley, CA, USA, 2010. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1924943.1924954.

Kiko Fernandez-Reyes, Dave Clarke, and Daniel S. McCain. Part: An asynchronous parallel abstraction for speculative pipeline computations. In Alberto Lluch-Lafuente and José Proença, editors, *Coordination Models and Languages - 18th IFIP WG 6.1 International Conference, COORDINATION 2016, Held as Part of the 11th International Federated Conference on Distributed Computing Techniques, DisCoTec 2016, Heraklion, Crete, Greece, June 6-9, 2016, Proceedings*, volume 9686 of *Lecture Notes in Computer Science*, pages 101–120. Springer, 2016. ISBN 978-3-319-39518-0. doi: 10.1007/978-3-319-39519-7_7. URL https://doi.org/10.1007/978-3-319-39519-7_7.

C. Flanagan and Mattias Felleisen. The semantics of future and an application. *Journal of Functional Programming*, 9(1):1–31, January 1999. ISSN 0956-

7968. doi: 10.1017/S0956796899003329. URL http://dx.doi.org/10.1017/S0956796899003329.

Cormac Flanagan and Matthias Felleisen. The semantics of future and its use in program optimization. pages 209–220, 1995. ISBN 0-89791-692-1.

Cormac Flanagan and Stephen N. Freund. Fasttrack: Efficient and precise dynamic race detection. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 121–133, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-392-1. doi: 10.1145/1542476.1542490. URL http://doi.acm.org/10.1145/1542476.1542490.

Cormac Flanagan and Shaz Qadeer. A type and effect system for atomicity. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, pages 338–349, New York, NY, USA, 2003. ACM. ISBN 1-58113-662-5. doi: 10.1145/781131.781169. URL http://doi.acm.org/10.1145/781131.781169.

Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 234–245, New York, NY, USA, 2002. ACM. ISBN 1-58113-463-0. doi: 10.1145/512529.512558. URL http://doi.acm.org/10.1145/512529.512558.

Elena Giachino and Cosimo Laneve. *A Beginner's Guide to the DeadLock Analysis Model*, pages 49–63. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. ISBN 978-3-642-41157-1. doi: 10.1007/978-3-642-41157-1_4. URL https://doi.org/10.1007/978-3-642-41157-1_4.

Elena Giachino and Cosimo Laneve. *Deadlock Detection in Linear Recursive Programs*, pages 26–64. Springer International Publishing, Cham, 2014. ISBN 978-3-319-07317-0. doi: 10.1007/978-3-319-07317-0_2. URL https://doi.org/10.1007/978-3-319-07317-0_2.

Elena Giachino, Naoki Kobayashi, and Cosimo Laneve. *Deadlock Analysis of Unbounded Process Networks*, pages 63–77. Springer Berlin Heidelberg, Berlin,

Heidelberg, 2014. ISBN 978-3-662-44584-6. doi: 10.1007/978-3-662-44584-6_6. URL https://doi.org/10.1007/978-3-662-44584-6_6.

Elena Giachino, Cosimo Laneve, and Michael Lienhardt. A framework for deadlock detection in core abs. *Software & Systems Modeling*, pages 1–36, 2015. ISSN 1619-1374. doi: 10.1007/s10270-014-0444-y. URL http://dx.doi.org/10.1007/s10270-014-0444-y.

Elena Giachino, Ludovic Henrio, Cosimo Laneve, and Vincenzo Mastandrea. Actors may synchronize, safely! In *Proceedings of the 18th International Symposium on Principles and Practice of Declarative Programming*, PPDP '16, pages 118–131, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4148-6. doi: 10.1145/2967973.2968599. URL http://doi.acm.org/10.1145/2967973.2968599.

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.

Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea. *Java Concurrency in Practice*. Addison-Wesley, 2006.

A. Göransson. *Efficient Android Threading: Asynchronous Processing Techniques for Android Applications*. Programming / Android. Oreilly & Associates Incorporated, 2014. ISBN 9781449364137. URL https://books.google.fr/books?id=T141ngEACAAJ.

Philipp Haller. On the integration of the actor model in mainstream technologies: The scala perspective. In *Proceedings of the 2Nd Edition on Programming Systems, Languages and Applications Based on Actors, Agents, and Decentralized Control Abstractions*, AGERE! 2012, pages 1–6, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1630-9. doi: 10.1145/2414639.2414641. URL http://doi.acm.org/10.1145/2414639.2414641.

Philipp Haller and Martin Odersky. Scala actors: Unifying thread-based and event-based programming. *Theor. Comput. Sci.*, 410(2-3):202–220, February

2009. ISSN 0304-3975. doi: 10.1016/j.tcs.2008.09.019. URL http://dx.doi.org/10.1016/j.tcs.2008.09.019.

Robert H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538, October 1985. ISSN 0164-0925. doi: 10.1145/4472.4478. URL http://doi.acm.org/10.1145/4472.4478.

R. T. Hammel and D. K. Gifford. Fx-87 performance measurements: Dataflow implementation. Technical report, Cambridge, MA, USA, 1988.

J. W. Havender. Avoiding deadlock in multitasking systems. *IBM Syst. J.*, 7 (2):74–84, June 1968. ISSN 0018-8670. doi: 10.1147/sj.72.0074. URL http://dx.doi.org/10.1147/sj.72.0074.

Ludovic Henrio and Florian Kammüller. Multiactive objects: Formalisation of the semantics with multiasp, 2015. URL http://www-sop.inria.fr/members/Ludovic.Henrio/misc.html.

Ludovic Henrio, Muhammad Uzair Khan, Nadia Ranaldo, and Eugenio Zimeo. First class futures: Specification and implementation of update strategies. In *Proceedings of the 2010 Conference on Parallel Processing*, Euro-Par 2010, pages 295–303, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-21877-4. URL http://dl.acm.org/citation.cfm?id=2031978.2032019.

Ludovic Henrio, Oleksandra Kulankhina, and Eric Madelaine. Integrated environment for verifying and running distributed components. In *Proc. International Conference on Fundamental Approaches to Software Engineering (FASE 2016)*, LNCS. Springer, 2016.

Ludovic Henrio, Cosimo Laneve, and Vincenzo Mastandrea. *Analysis of Synchronisations in Stateful Active Objects*, pages 195–210. Springer International Publishing, Cham, 2017. ISBN 978-3-319-66845-1. doi: 10.1007/978-3-319-66845-1_13. URL https://doi.org/10.1007/978-3-319-66845-1_13.

Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd International Joint*

Conference on Artificial Intelligence, IJCAI'73, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc. URL http://dl.acm.org/citation.cfm?id=1624775.1624804.

Rich Hickey. Agents and asynchronous actions. URL https://clojure.org/reference/agents.

Atsushi Igarashi and Naoki Kobayashi. A generic type system for the pi-calculus. *SIGPLAN Not.*, 36(3):128–141, January 2001. ISSN 0362-1340. doi: 10.1145/373243.360215. URL http://doi.acm.org/10.1145/373243.360215.

Lightbend Inc. Akka—simpler concurrency., 2017. URL http://akka.io.

Einar Broch Johnsen and Olaf Owe. An asynchronous communication model for distributed concurrent objects. 6(1):35–58, March 2007.

Einar Broch Johnsen, Olaf Owe, and Marte Arnestad. Combining active and reactive behavior in concurrent objects. In Dag Langmyhr, editor, *Proc. of the Norwegian Informatics Conference (NIK'03)*, pages 193–204. Tapir Academic Publisher, November 2003.

Einar Broch Johnsen, Olaf Owe, and Ingrid Chieh Yu. Creol: A type-safe object-oriented model for distributed concurrent systems. *Theor. Comput. Sci.*, 365(1): 23–66, November 2006. ISSN 0304-3975. doi: 10.1016/j.tcs.2006.07.031. URL http://dx.doi.org/10.1016/j.tcs.2006.07.031.

Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. ABS: A core language for abstract behavioral specification. In Bernhard Aichernig, Frank S. de Boer, and Marcello M. Bonsangue, editors, *Proc. 9th Intl. Symp. on Formal Methods for Components and Objects (FMCO)*, volume 6957, pages 142–164. 2011a.

Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. Abs: A core language for abstract behavioral specification. In *Proceedings of the 9th International Conference on Formal Methods for Components*

and Objects, FMCO'10, pages 142–164, Berlin, Heidelberg, 2011b. Springer-Verlag. ISBN 978-3-642-25270-9. doi: 10.1007/978-3-642-25271-6_8. URL http://dx.doi.org/10.1007/978-3-642-25271-6_8.

Baris Kasikci, Cristian Zamfir, and George Candea. Data races vs. data race bugs: Telling the difference with portend. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 185–198, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-0759-8. doi: 10.1145/2150976.2150997. URL http://doi.acm.org/10.1145/2150976.2150997.

Eric Kerfoot, Steve McKeever, and Faraz Torshizi. Deadlock freedom through object ownership. In *International Workshop on Aliasing, Confinement and Ownership in Object-Oriented Programming*, IWACO '09, pages 3:1–3:8, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-546-8. doi: 10.1145/1562154.1562157. URL http://doi.acm.org/10.1145/1562154.1562157.

Gary A. Kildall. A unified approach to global program optimization. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '73, pages 194–206, New York, NY, USA, 1973. ACM. doi: 10.1145/512927.512945. URL http://doi.acm.org/10.1145/512927.512945.

James C. King. Symbolic execution and program testing. *Commun. ACM*, 19 (7):385–394, July 1976. ISSN 0001-0782. doi: 10.1145/360248.360252. URL http://doi.acm.org/10.1145/360248.360252.

Naoki Kobayashi. A partially deadlock-free typed process calculus. *ACM Trans. Program. Lang. Syst.*, 20(2):436–482, March 1998. ISSN 0164-0925. doi: 10.1145/276393.278524. URL http://doi.acm.org/10.1145/276393.278524.

Naoki Kobayashi. *A New Type System for Deadlock-Free Processes*, pages 233–247. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006. ISBN 978-3-540-37377-3. doi: 10.1007/11817949_16. URL https://doi.org/10.1007/11817949_16.

Naoki Kobayashi. TyPiCal. http://www-kb.is.s.u-tokyo.ac.jp/~koba/typical/, 2007.

Naoki Kobayashi and Cosimo Laneve. Deadlock analysis of unbounded process networks. *Information and Computation*, 252:48 – 70, 2017. ISSN 0890-5401. doi: http://dx.doi.org/10.1016/j.ic.2016.03.004. URL http://www.sciencedirect.com/science/article/pii/S0890540116000419.

Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. Efficient state merging in symbolic execution. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 193–204, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1205-9. doi: 10.1145/2254064.2254088. URL http://doi.acm.org/10.1145/2254064.2254088.

R. Greg Lavender and Douglas C. Schmidt. Pattern languages of program design 2. chapter Active Object: An Object Behavioral Pattern for Concurrent Programming, pages 483–499. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996. ISBN 0-201-895277. URL http://dl.acm.org/citation.cfm?id=231958.232967.

Mohsen Lesani and Antonio Lain. Semantics-preserving sharing actors. In *Proceedings of the 2013 Workshop on Programming Based on Actors, Agents, and Decentralized Control*, AGERE! 2013, pages 69–80, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2602-5. doi: 10.1145/2541329.2541332. URL http://doi.acm.org/10.1145/2541329.2541332.

N. G. Leveson and C. S. Turner. An investigation of the therac-25 accidents. *Computer*, 26(7):18–41, July 1993. ISSN 0018-9162. doi: 10.1109/MC.1993.274940. URL http://dx.doi.org/10.1109/MC.1993.274940.

J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, pages 47–57, New York, NY, USA, 1988. ACM. ISBN 0-89791-252-7. doi: 10.1145/73560.73564. URL http://doi.acm.org/10.1145/73560.73564.

Kin-Keung Ma, Khoo Yit Phang, Jeffrey S. Foster, and Michael Hicks. Directed symbolic execution. In *Proceedings of the 18th International Conference*

on Static Analysis, SAS'11, pages 95–111, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-23701-0. URL http://dl.acm.org/citation.cfm?id=2041552.2041563.

Peter M. Mell and Timothy Grance. The nist definition of cloud computing. Technical report, Gaithersburg, MD, United States, 2011.

Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to analysis for java. *ACM Trans. Softw. Eng. Methodol.*, 14(1):1–41, January 2005. ISSN 1049-331X. doi: 10.1145/1044834.1044835. URL http://doi.acm.org/10.1145/1044834.1044835.

Mark S. Miller, E. Dean Tribble, and Jonathan Shapiro. Concurrency among strangers: Programming in e as plan coordination. In *Proceedings of the 1st International Conference on Trustworthy Global Computing*, TGC'05, pages 195–229, Berlin, Heidelberg, 2005. Springer-Verlag. ISBN 3-540-30007-4, 978-3-540-30007-6. URL http://dl.acm.org/citation.cfm?id=1986262.1986274.

Benjamin Morandi, Sebastian S. Bauer, and Bertrand Meyer. SCOOP – A contract-based concurrent object-oriented programming model. In Peter Müller, editor, *Advanced Lectures on Software Engineering*, volume 6029 of *Lecture Notes in Computer Science*, pages 41–90. Springer, 2010.

Anders Møller and Michael I. Schwartzbach. Static program analysis. https://cs.au.dk/~amoeller/spa/spa.pdf.

Satish Narayanasamy, Zhenghao Wang, Jordan Tigani, Andrew Edwards, and Brad Calder. Automatically classifying benign and harmful data races using replay analysis. *SIGPLAN Not.*, 42(6):22–31, June 2007. ISSN 0362-1340. doi: 10.1145/1273442.1250738. URL http://doi.acm.org/10.1145/1273442.1250738.

Robert H. B. Netzer and Barton P. Miller. Improving the accuracy of data race detection. *SIGPLAN Not.*, 26(7):133–144, April 1991. ISSN 0362-1340. doi: 10.1145/109626.109640. URL http://doi.acm.org/10.1145/109626.109640.

J. Niehren, J. Schwinghammer, and G. Smolka. A concurrent lambda calculus with futures. *Theor. Comput. Sci.*, 364(3):338–356, November 2006. ISSN 0304-3975. doi: 10.1016/j.tcs.2006.08.016. URL http://dx.doi.org/10.1016/j.tcs.2006.08.016.

Joachim Niehren, David Sabel, Manfred Schmidt-Schauß, and Jan Schwinghammer. Observational semantics for a concurrent lambda calculus with reference cells and futures. In *23rd Conference on Mathematical Foundations of Programming Semantics*, ENTCS, New Orleans, April 2007. Accepted.

Ka. I Pun. *behavioural static analysis for deadlock detection.* PhD thesis, Faculty of Mathematics and Natural Sciences, University of Oslo, Norway, 2013.

Franz Puntigam and Christof Peter. Types for active objects with static deadlock prevention. *Fundam. Inf.*, 48(4):315–341, December 2001. ISSN 0169-2968. URL http://dl.acm.org/citation.cfm?id=1220086.1220088.

H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953. ISSN 00029947. URL http://www.jstor.org/stable/1990888.

Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multi-threaded programs. *SIGOPS Oper. Syst. Rev.*, 31(5):27–37, October 1997. ISSN 0163-5980. doi: 10.1145/269005.266641. URL http://doi.acm.org/10.1145/269005.266641.

Jan Schäfer and Arnd Poetzsch-Heffter. Jcobox: Generalizing active objects to concurrent components. In *Proceedings of the 24th European Conference on Object-oriented Programming*, ECOOP'10, pages 275–299, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-14106-4, 978-3-642-14106-5. URL http://dl.acm.org/citation.cfm?id=1883978.1883996.

V. Serbanescu, K. Azadbakht, F. de Boer, C. Nagarajagowda, and B. Nobakht. A design pattern for optimizations in data intensive applications using abs and java 8. *Concurrency and Computation: Practice and Experience*, 28(2):374–385, 2016. ISSN 1532-0634. doi: 10.1002/cpe.3480. URL http://dx.doi.org/10.1002/cpe.3480. cpe.3480.

Marjan Sirjani. *Rebeca: Theory, Applications, and Tools*, pages 102–126. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007. ISBN 978-3-540-74792-5. doi: 10.1007/978-3-540-74792-5_5. URL https://doi.org/10.1007/978-3-540-74792-5_5.

Matt Staats and Corina Păsăreanu. Parallel symbolic execution for structural test generation. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA '10, pages 183–194, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-823-0. doi: 10.1145/1831708.1831732. URL http://doi.acm.org/10.1145/1831708.1831732.

Jørgen Staunstrup, Henrik Reif Andersen, Henrik Hulgaard, Jørn Lind-Nielsen, Kim G. Larsen, Gerd Behrmann, Kwre Kristoffersen, Arne Skou, Henrik Leerberg, and Niels Bo Theilgaard. Practical verification of embedded software. *Computer*, 33(5):68–75, May 2000. ISSN 0018-9162. doi: 10.1109/2.841786. URL http://dx.doi.org/10.1109/2.841786.

Kenjiro Taura, Satoshi Matsuoka, and Akinori Yonezawa. Abcl/f: A future-based polymorphic typed concurrent object-oriented language - its design and implementation. In *Proceedings of the DIMACS workshop on Specification of Parallel Algorithms*, pages 275–292. American Mathematical Society, 1994.

Vasco Thudichum Vasconcelos, Francisco Martins, and Tiago Cogumbreiro. Type inference for deadlock detection in a multithreaded polymorphic typed assembly language. In *Proceedings Second International Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software, PLACES 2009, York, UK, 22nd March 2009.*, pages 95–109, 2009. doi: 10.4204/EPTCS.17.8. URL https://doi.org/10.4204/EPTCS.17.8.

Willem Visser, Klaus Havelund, Guillaume Brat, Seungjoon Park, and Flavio Lerda. Model checking programs. *Automated Software Engg.*, 10(2):203–232, April 2003. ISSN 0928-8910. doi: 10.1023/A:1022920129859. URL http://dx.doi.org/10.1023/A:1022920129859.

Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. Relay: Static race detection on millions of lines of code. In *Proceedings of the the 6th Joint Meeting of the*

European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, ESEC-FSE '07, pages 205–214, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-811-4. doi: 10.1145/1287624.1287654. URL http://doi.acm.org/10.1145/1287624.1287654.

Scott West, Sebastian Nanz, and Bertrand Meyer. *A Modular Scheme for Deadlock Prevention in an Object-Oriented Programming Model*, pages 597–612. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010. ISBN 978-3-642-16901-4. doi: 10.1007/978-3-642-16901-4_39. URL https://doi.org/10.1007/978-3-642-16901-4_39.

Ann Wollrath, Roger Riggs, and Jim Waldo. A distributed object model for the javatm system. In *Proceedings of the 2Nd Conference on USENIX Conference on Object-Oriented Technologies (COOTS) - Volume 2*, COOTS'96, pages 17–17, Berkeley, CA, USA, 1996. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1268049.1268066.

Derek Wyatt. *Akka Concurrency*. Artima, 2013.

Akinori Yonezawa, Jean-Pierre Briot, and Etsuya Shibayama. Object-oriented concurrent programming abcl/1. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, OOPLSA '86, pages 258–268, New York, NY, USA, 1986. ACM. ISBN 0-89791-204-7. doi: 10.1145/28697.28722. URL http://doi.acm.org/10.1145/28697.28722.

List of Acronyms

CFG Control Flow Graphs 35

FIFO First In First Out 11

GCM Grid Component Model 23

HPC High Performance Computing 22

JVM Java Virtual Machine 25

RMI Remote Method Invocation 23

SIMD Single Instruction Multiple Data 24