Université de nice sophia antipolis

École Doctorale des Sciences et Technologies de l'Information et de la Communication

Towards a Safe and Secure Synchronous Language

Vers un langage synchrone sûr et sécurisé

Pejman Attar

Thèse de doctorat
Spécialité informatique

Soutenue le 12 Décembre 2013
devant le jury composé de :

President:	Jean-Paul Rigault
Reviewers:	Roberto Amadio
	Michele Bugliesi
Examiners:	Juliusz Chroboczek
	Pascal Raymond
Supervisors:	Frédéric Boussinot
	Ilaria Castellani

Contents

Abstract **7**

1 Introduction **11**
1.1 The Synchronous Model . . . 12
1.1.1 Memory in Synchronous Models . . . 13
1.1.2 Causality Cycles . . . 14
1.1.3 Non-terminating Instants . . . 14
1.1.4 Real Parallelism . . . 14
1.1.5 The Reactive Approach . . . 15
1.2 Security . . . 15
1.2.1 Access Control . . . 15
1.2.2 Secure Information Flow . . . 16
1.3 State of the Art . . . 16
1.3.1 Concurrency . . . 17
1.3.2 Parallelism and Multi-core architectures . . . 20
1.3.3 Safety . . . 21
1.3.4 Security . . . 21
1.4 Document Structure . . . 22
1.4.1 DSL . . . 23
1.4.2 CRL . . . 25
1.4.3 SSL . . . 25
1.4.4 DSLM . . . 26
1.4.5 Conclusion . . . 27

I DSL **29**

2 The Dynamic Synchronous Language DSL **31**
2.1 Language Description . . . 32

2.1.1 Scripts . . . 33
2.1.2 Sites . . . 34
2.1.3 Events . . . 35
2.1.4 Basic Properties . . . 35
2.1.5 Example . . . 35
2.2 Semantics . . . 37
2.2.1 Expressions . . . 37
2.2.2 Scripts . . . 37
2.2.3 Least Fix-Point . . . 42
2.2.4 Programs and Sites . . . 43
2.3 Examples and Semantical Variants . . . 44
2.3.1 Examples . . . 44
2.3.2 Variants of the Semantics . . . 47

3 DSL Implementation 49
3.1 FunLoft Variant . . . 49
3.1.1 Dynamic Adding of Instructions . . . 50
3.1.2 Reactive Engine . . . 51
3.1.3 Functions and Tasks . . . 52
3.1.4 Instantaneous Loops . . . 54
3.1.5 Static Checks . . . 54
3.1.6 Execution of Instructions . . . 56
3.1.7 Translation in FunLoft . . . 57
3.2 Bigloo/Scheme Variant . . . 58
3.2.1 Sites . . . 58
3.2.2 Functions and Tasks . . . 59
3.2.3 Translation in Bigloo . . . 60
3.3 Benchmarks . . . 60
3.3.1 FunLoft . . . 60
3.3.2 SugarCubes . . . 61
3.3.3 ReactiveML . . . 62
3.3.4 Scheme/Bigloo . . . 62
3.3.5 Interpretation . . . 63

II CRL 65

4 The Core Reactive Language CRL 67
4.1 Syntax . . . 67
4.1.1 Expressions . . . 68

4.1.2 Scripts . . . 68
4.2 Semantics . . . 69
4.3 Reactivity . . . 75

III SSL **85**

5 Secure Synchronous Language SSL **87**
5.1 Fine-grained and Coarse-grained Bisimilarity . . . 88
5.2 Security property . . . 91
5.4 Type System . . . 93
5.5.1 Example . . . 104

IV DSLM **109**

6 DSL with Memory: The language DSLM **111**
6.1 Informal Language Description . . . 112
6.1.1 Scripts and Expressions . . . 113
6.1.2 Agents, Sites and Systems . . . 115
6.1.3 Example . . . 116
6.2 Domains . . . 119
6.3 Semantics of Scripts . . . 123
6.3.1 Expressions . . . 123
6.3.2 Suspension Predicate . . . 125
6.3.3 Transition Relation . . . 125
6.3.4 Semantic Properties . . . 129
6.4 Semantics of Sites and Systems . . . 130
6.4.1 Sites . . . 131
6.4.2 End of Instants . . . 132
6.4.3 Reconditioning Function for Next Instant . . . 133

7 Typing System for DSLM **135**

8 DSLM Implementation **141**
8.1 Implementation Semantics . . . 143
8.1.1 Domains . . . 143
8.1.2 Suspension Predicate . . . 143
8.1.3 Sites and Systems . . . 144
8.2 Load Balancing . . . 147

V Conclusion **151**

9 Conclusion and Future Work **153**

Abstract

French

Cette thèse propose une nouvelle approche du parallélisme et de la concurrence, posant les bases d'un langage de programmation à la fois sûr et "secure" (garantissant la sécurité des données), fondé sur une sémantique formelle claire et simple, tout en étant adapté aux architectures multi-cores.

Nous avons adopté le paradigme synchrone, dans sa variante réactive, qui fournit une alternative simple à la programmation concurrente standard en limitant l'impact des erreurs dépendant du temps ("data-races"). Dans un premier temps, nous avons considéré un langage réactif d'orchestration, DSL, dans lequel on fait abstraction de la mémoire (Partie 1).

Dans le but de pouvoir traiter la mémoire et la sécurité, nous avons ensuite étudié (Partie 2) un noyau réactif, CRL, qui utilise un opérateur de parallélisme déterministe. Nous avons prouvé la réactivité bornée des programmes de CRL.

Nous avons ensuite équipé CRL de mécanismes pour contrôler le flux d'information (Partie 3). Pour cela, nous avons d'abord étendu CRL avec des niveaux de sécurité pour les variables et les évènement, puis nous avons défini dans le langage étendu, SSL, un système de types permettant d'éviter les fuites d'information.

Parallèlement (Partie 4), nous avons ajouté la mémoire à CRL, en proposant le modèle DSLM. En utilisant une notion d'agent, nous avons structuré la mémoire de telle sorte qu'il ne puisse y avoir de "data-races". Nous avons également étudié l'implémentation de DSLM sur les architectures multi-cores, fondée sur la notion de site et de migration d'un agent entre les sites.

L'unification de SSL et de DSLM est une piste pour un travail futur.

English

This thesis proposes a new approach to parallelism and concurrency, laying the basis for the design of a programming language with a clear and simple formal semantics, enjoying both safety and security properties, while lending itself to an implementation on multi-core architectures.

We adopted the synchronous programming paradigm, in its reactive variant, which provides a simple alternative to standard concurrent programming by limiting the impact of time-dependent errors ("data-races"). As a first step (Part 1), we considered a reactive orchestration language, DSL, which abstracts away from the memory.

To set the basis for a formal treatment of memory and security, we then focused on a reactive kernel, CRL, equipped with a deterministic parallel operator (Part 2). We proved bounded reactivity of CRL programs.

Next, we enriched CRL with mechanisms for information flow control (Part 3). To this end, we first extended CRL with security levels for variables and events. We then defined a type system on the extended language, SSL, which ensures the absence of information leaks.

Finally, we added memory to CRL, as well as the notions of agent and site, thus obtaining the model DSLM (Part 4). We structured the memory in such a way that data-races cannot occur, neither within nor among agents. We also investigated the implementation of DSLM on multi-core architectures, using the possibility of agent migration between sites.

The unification of SSL and DSLM is left for future work.

Remerciements:

Trois années de dur labeur passées en compagnie de personnes qui m'ont aidé et soutenu dans la réalisation de ma thèse. Aujourd'hui que celle-ci se termine, je tiens à remercier toutes ces personnes qui ont été présentes tout au long de ces trois ans.

Tout d'abord, je remercie mes deux encadrants Frédéric Boussinot et Ilaria Castellani avec qui j'ai eu une relation très amicale et qui ont dû me supporter bien que cela n'eut pas dû être facile tous les jours. Ils m'ont beaucoup aidé dans la rédaction de cette thèse et ont été très à l'écoute, m'ont apporté de précieux conseils; sans eux ma thèse aurait beaucoup plus ressemblé à un poème persan qu'à un texte anglais.

Je remercie également Jean-Paul Rigault qui a accepté de présider le jury, Roberto Amadio et Michele Bugliesi pour avoir eu la gentillesse de relire ma thèse en tant que rapporteurs, et le reste des membres du jury Juliusz Chroboczek et Pascal Raymond pour avoir accepté de participer au jury.

Je remercie Bernard Serpette et José Santos pour avoir accepté de partager mes réflexions aussi bien sensées qu'insensées, apporté leurs différents avis et contribué à me guider dans mes résultats. Merci également à Nathalie Bellesso pour m'avoir aidé à m'y retrouver dans le labyrinthe administratif, particulièrement hermétique pour un étranger. Je remercie aussi tous les autres membres de l'équipe INDES, pour m'avoir offert un environnement de travail amical et stimulant.

Et encore une fois un grand merci à Juliusz Chroboczek et Gabriel Kerneis pour m'avoir ouvert le chemin de la recherche et permis d'aboutir à ce résultat.

Enfin, je tiens à remercier ma famille, particulièrement mon oncle, ma tante et mes deux cousines sans qui je ne serais pas en France. Merci également à tous mes amis pour avoir été présents, tous ceux qui ont été à mes côtés et particulièrement la Colectividad.

Et enfin, un grand merci à Annie pour son soutien, et pour m'avoir aidé à rédiger cette note de remerciements :).

Chapter 1

Introduction

Concurrency and parallelism are among the main problems of systems and programming languages. We define concurrency as the composition of independently executing processes and parallelism as the simultaneous execution of (possibly related) computations on different hardware components [57].[1]

Originally, issues concerning parallelism and concurrency were concentrated at the operating system level and left to experts of this domain.

Nowadays, multi-core machines are everywhere: in servers, PCs and even in mobile phones. These machines are widely used by the public. Concurrency problems are no more expert problems but they now also concern software programmers.

There exist several models of concurrent programming, like the Actor model [6], Petri nets [47], process calculi [63], Transactional memory [39] and the shared memory concurrency model [33] which is the most widely used in programming languages and in operating systems. We are going to describe more deeply the last approach, which is the one we adopted in our work.

The first and most used variant of the shared memory model is called *preemptive multi-threading*. In this context, concurrent programs are system threads scheduled and preempted by the system in an arbitrary way. The major problem of this variant is the freedom schedulers have in choosing the threads to be executed; this leads to so-called *time-dependent errors*, which are generally considered as extremely difficult to tame and to debug [38] .

Another variant of the shared memory model is *cooperative multi-threading*. In this variant, the system loses the possibility to arbitrarily preempt threads.

[1] However, the terms *concurrent* and *parallel* will often be used interchangeably when their use is clear from the context.

In order to be given to a new thread, the control must be explicitly released by the currently executing thread. Thanks to this, time-dependent errors do not occur anymore. However, the cooperative approach suffers from an obvious drawback as a single thread can freeze the whole system if it never releases the control, thus preventing the system from giving it to the other threads.

The intrinsic difficulty of problems raised by concurrency calls for *formal* techniques, and more specifically for formal semantics. Formal semantics for concurrent programs are usually *operational semantics*. Operational semantics can be separated in two categories: small-step semantics (structural operational), and big-step semantics (natural semantics). Small-step semantics is close to program execution and describes each step of evaluation by an abstract interpretor. On the other hand, big-step semantics describes how the overall execution result is obtained, possibly using abstract means such as least fix-points of functionals. Small-step semantics are closer to implementation than big-step semantics, but more difficult to reason with.

1.1 The Synchronous Model

Synchronous programming is an approach to concurrent programming which is at the basis of our work. Synchronous programming simplifies concurrency, compared to standard approaches based on the exclusive use of the classic model of threads (pthreads or Java threads). The simplification basically results from a cleaner and simpler semantics than the classical concurrent model, which reduces the number of possible interleavings in parallel computations. However, standard synchronous languages introduce specific issues: non-termination of instants, dynamic creation, causality cycles, and they have difficulties to deal with memory. Moreover, they are generally not able to fully benefit from real parallelism, as that provided by multi-core machines.

We are going to present the synchronous approach in more detail, and show the advantages and disadvantages of this model by means of examples written in the first synchronous language Esterel [18].

In the synchronous approach, the interaction of the system with its environment is discrete. A global clock is used to sample an environment of present and absent objects. At each clock tick, the system reacts to the modifications of its environment and produces a new environment. The time of the global clock is not a physical time but a logical time. The interval between two clock ticks is called *instant*.

Let us consider a program made of two parallel statements, one awaiting for an event `ev1`, then producing event `ev2`, and the other producing event `ev1`. Such a program is written in Esterel as:

```
P1 = await immediate ev1; emit ev2 || emit ev1
```

Due to the synchronous parallelism used in Esterel, the program `P1` immediately emits both `ev1` and `ev2`, and this is its only possible outcome. Actually, the standpoint of Esterel is that the I/O behavior of programs within each instant should be deterministic. Hence, although its small-step semantics may be non-deterministic, the parallel operator (||) is required to be confluent within instants. The small-step semantics of `P1` allows the various possible interleavings of the two parallel components, and lets the control progress until both emissions of `ev1` and `ev2` are performed. By contrast, the big-step semantics guesses that `ev1` and `ev2` are present, and then verifies that this is a coherent outcome.

1.1.1 Memory in Synchronous Models

As mentioned above, the synchronous model requests a confluent parallelism, which is difficult to combine with the presence of memory [37], [18]. Indeed, uncontrolled concurrent accesses to the memory may produce a non-deterministic behavior. Consider the following program:

```
P2 = x:=1 || x:=2
```

The outcome of `P2` can be either $x = 1$ (if the second branch is executed first) or $x = 2$ otherwise. Moreover, non-determinism can result from non-atomic accesses to the memory. Thus deterministic concurrent programming demands for means to get atomic memory accesses.

Esterel chooses a rather drastic solution to this problem: a variable cannot be read by one branch of a parallel statement and written by the other [18]. Thus, the previous program `P2` is rejected by the compiler. However, Esterel does not control concurrent accesses made at a lower level by procedures and functions. Consider for example the following program where two functions are called in parallel:

```
P3 = f1 () || f2 ()
```

The Esterel compiler is unable to verify that no concurrent accesses occur through the calls of $f1$ and $f2$ when executing `P3`. Therefore, if $f1$ and $f2$ are sharing memory there could be a non-deterministic execution of the program `P3`.

Actually, one may think that the Esterel solution to avoid concurrent accesses to the memory is over-restrictive, specially in the context of multi-core programming in which memory accesses are the basic communication and synchronization means.

1.1.2 Causality Cycles

In the synchronous model, events are a means for communication. At each instant, an event is either absent, or present if it is produced by one of the concurrent threads. However, in Esterel, *causality cycles* can appear when no coherent solution can be found for the absence/presence status of an event. For example, consider the program:

```
P4 = present ev else emit ev end
```

There is a causality cycle in `P4`, as the status of `ev` cannot be determined: if `ev` is absent, then it is emitted, which is contradictory; on the other hand, if `ev` is present, then it is not emitted, which is also a contradiction. Thus, `P4` has no coherent solution in determining the status of `ev`.

1.1.3 Non-terminating Instants

Moreover, the synchronous model has to face another problem, called *instant non-termination*, which arises when one of the concurrent threads prevents the system to reach the end of instant. This problem is closely related to the freezing problem of the cooperative model. To solve this problem we can use code analysis techniques, but the main difficulty remains, which is to deal with function calls and be able to predict their termination. The reactive approach which we describe later gives a partial solution to this problem.

1.1.4 Real Parallelism

In order to take benefit of real parallelism in a shared memory model, as in multi-core architectures, a synchronous language should not only be able to avoid causality cycles and non-terminating instants, but also to control

memory accesses at the lowest level, in order to avoid problems like data-races. Presently, there exists no synchronous language covering all these aspects.

1.1.5 The Reactive Approach

In this thesis, we address the previously discussed problems, and propose to focus on a particular brand of synchronous programming, called the *reactive* approach, which was first embodied in the synchronous language *SL* [27, 10], an offspring of Esterel, and later incorporated into various programming environments, such as C, Java, Caml and Scheme. The model of *SL* departs from that of ESTEREL in that it assumes the reaction to the absence of an event to be postponed until the end of the instant. This assumption helps disambiguate programs and simplifying the implementation of the language. It is also essential to ensure the monotonicity of programs (as transformers on sets of events) and their reactivity to the environment. As a consequence of these properties, causality cycles do not appear any more.

Our other main concern is real parallelism. In order to take advantage of multi-core architectures, we will use the notion of *Synchronized schedulers* which was first presented in FunLoft [26] by F. Boussinot. This notion allows us to use multi-cores by executing programs on separate schedulers in real parallelism and synchronizing them at the end of each instant.

1.2 Security

Nowadays we are all concerned about security, not only when communicating over the web but also for protecting our local data. The application area of security is really wide. In this thesis, we are interested more particularly in the issue of protecting *data confidentiality*. This problem has two facets, which are *access control* and *secure information flow*. We now discuss each of them in some more detail.

1.2.1 Access Control

Access control [40, 32] is a means to allow only authorized users to access sensitive data. Firewalls [56] and anti-viruses [45] can be seen as software which implements forms of access control. This approach is largely used in computer science to manage the permission of users to access objects.

However, access control by itself is not sufficient to ensure data confidentiality: once an authorized program receives confidential data, there exists

no way to make sure that this information is handled correctly and the program will not leak this information. Hence, access control needs to be combined with an analysis of the flow of information through the program. This is the object of secure information flow.

1.2.2 Secure Information Flow

In this thesis, we will be concerned with ensuring confidentiality in synchronous reactive programs. Information can flow between programs, where each program can manipulate these data (read, write, diffuse, etc.). An information flow is considered *secure* if it respects a certain *security policy*, which specifies which information should be accessible where, when and to whom. This notion is often formalized by means of *non-interference*.

Non-interference was originally introduced by Goguen and Meseguer in [35]. Then, Volpano et al. proposed in [69] the first security type system ensuring non-interference for a sequential programming language. Concurrency poses new challenges for information flow security by introducing non-deterministic and non-terminating behaviors for programs. The first extension for concurrency was proposed by Smith and Volpano who presented a notion of *possibilistic* non-interference [67]. Soon after, Sabelfeld and Sands presented in [61] the first explicit formulation of non-interference in terms of bisimulation, using a notion of *probabilistic* non-interference, as opposed to possibilistic. Subsequently, various studies on non-interference for concurrency have been carried out, both in probabilistic and possibilistic settings, such as [60, 66, 8, 22].

Here, we will focus on a notion of non-interference for a reactive language. Our work is strongly inspired by that of Almeida Matos et al. [8], where a property of non-interference was studied for the synchronous reactive kernel of the language ULM [21]. Our work is also related to that of Russo and Sabelfeld in [59] where a notion of non-interference in a multi-threaded cooperative model is investigated.

1.3 State of the Art

In this section we briefly review existing work around our main domains of interest: parallelism, multi-core architectures and techniques to use them efficiently, safety and security.

1.3.1 Concurrency

Standard Threads

Threads are the basic standard means to deal with parallelism at system level and in programming languages. A thread is the smallest sequence of instructions that can be managed independently by the operating system. There exist two ways to schedule threads: *preemptive multi-tasking* and *cooperative multi-tasking*.

Preemptive multi-tasking is used in Posix [53] and in Java [55], and in almost all operating systems. In preemptive multi-tasking the operating system is free to decide when context switches (shifting the control from one thread to another) should happen. Several techniques are associated with preemptive multi-tasking, such as locks and priorities. These techniques raise a number of well-known problems such as deadlocks, livelocks, and priority inversions.

Cooperative multi-tasking, on the other hand, relies on the threads themselves to decide when to perform the context switches. The cooperative approach suffers from a major problem which is called *freezing*: one thread can freeze the whole system if it never releases the control. Due to this problem, preemptive multi-tasking is usually preferred to cooperative multi-tasking.

Synchronous Model

1) Classical Synchronous Languages

Esterel is the first synchronous programming language, proposed in the 80's by Berry and co-workers [18, 19]. Esterel is both a programming language and a compiler which translates Esterel programs into finite-state machines (version 3 of the language) or into sets of equations (versions 5 and 7). In Esterel there is no means for dynamic creation, hence only static programs are allowed.

Dataflow is based on the idea that changing the value of a variable should automatically force recalculation of all variables which depend on it. There exist several synchronous languages which use this model. We briefly review some of them.

Signal [17] is a dataflow synchronous language whose formal model provides the capability to describe systems with several clocks as relational specifications. Relations are useful as partial specifications and as specifications of non-deterministic devices or external processes.

Lustre [36] is a dataflow synchronous language designed for programming reactive systems as well as for describing hardware. The dataflow aspect of Lustre makes it very close to usual description tools in these domains (block-diagrams, networks of operators, dynamical sample-systems, etc.), and its synchronous interpretation makes it well-suited for handling time in programs. Moreover, this synchronous interpretation allows it to be compiled into an efficient sequential program.
Lucid Synchrone [29] is a more recent, ML-based dataflow language. It is an extension of Lustre with some ML characteristics like type inference and higher order functions. In contrast to classical synchronous languages, in Lucid Synchrone parallel components can be dynamically created and threads executions are scheduled statically.

2) Reactive Approach

The reactive approach [27] is a variant of the synchronous programming approach described above. This model removes causality cycles and allows dynamic creation of parallel components by forbidding immediate reaction to the absence of signals. However, due to dynamic creation, the expressive power of the model increases and verifying programs and reasoning about their execution becomes more difficult.

This approach gave birth to several synchronous reactive languages like ReactiveC, SugarCubes, Loft, FairThreads, ReactiveML and FunLoft, which are described next.

ReactiveC was presented in [23]. It adds synchronous constructions to the C language. ReactiveC is the first reactive programming language.

SugarCubes [28] is a library made for reactive programming in Java. This library incorporates the reactive approach in Java. The latest version of SugarCubes [68] tries to take advantage of multi-cores and GPUs by delaying the elementary operations to the end of instant. OpenCL is used to deploy all the elementary operations over the CPU/GPU grid.

FairThreads [25] proposes a cooperative thread library based on the reactive model, for the C language. This library allows cooperative and preemptive scheduling to be mixed. In FairThreads, a thread can be executed either by a scheduler in a cooperative (synchronous) way, or be detached and executed asynchronously (*unlinked thread*). Schedulers are executed asynchronously in parallel by a native thread. Several schedulers can be linked

together and become a set of synchronized schedulers: they are executed in full parallelism but they share the same instants and events (*Synchronized schedulers*).

Loft [24] is an extension of C based on FairThreads, which facilitates the FairThreads programming by simplifying the syntax and semantics.

FunLoft is a functional synchronous reactive language with type inference, implemented using FairThreads. The main objective is to get a safe language in which the resources, CPU and memory, are controlled. Moreover, in FunLoft, memory leaks cannot occur and programs always react in finite time. The possibility of such a control results from the work of Dabrowski [30].

ReactiveML [41] is a higher order programming language based on the reactive model, and embedded in an ML language (actually OCaml). ReactiveML is dedicated to the implementation of interactive systems as found in graphical user interfaces, video games or simulation problems.

3) Other Approaches

ULM [21] is a functional programming model which focuses on the control of resources in a global computing context. ULM is an instance of the *GALS* (Globally Asynchronous Locally Synchronous) model, where each site is a reactive machine running independently and the communications between them are executed asynchronously. In ULM, programs attempting to access the memory of a distant site remain blocked until they migrate on the site. This can be seen as a means to insure safety (we propose here a different technique for the same purpose, also based on memory protection in case of migration).

Synchronous Process Calculi Process calculi are one of the most popular models for describing the interaction of concurrent process that communicate by exchanging messages. Most of these calculi allow processes to proceed asynchronously. Early calculi that introduce a notion of synchrony are SCCS [31] and CBS [58]. More recently, Amadio proposed the Synchronous π-calculus [9], which is based on the SL language. This is the process calculus that is the closest to our work.

ORC [44] is a novel orchestration language for distributed and concurrent programming which provides uniform access to computational services, including distributed communication and data manipulation, through sites. Using four simple concurrency primitives, the programmer orchestrates the invocation of sites to achieve a goal, while managing timeouts, priorities, and failures. ORC is based on process calculi and is close to our proposal (DSL, Part 1).

1.3.2 Parallelism and Multi-core architectures

Another concern of our work is multi-core architectures. There are several techniques, languages and systems to deal with this issue.

The first approach is to use all the available cores in the machine, including the graphic card. To be able to take advantage of graphic card cores, several languages have been created. Here, we shall discuss three of them.

CUDA [54] is a programming language which allows users to exploit the power of the graphics processing units (GPU) by using GPU-accelerated libraries. In this way, users are able to replace or augment CPU-only programs by recoursing to the power of graphic cards. This allows all the computational units to be exploited. The disadvantage of CUDA is that it is only supported by the graphic cards made by NVIDIA.

OpenCL (Open Computing Language) [46] is a framework for writing programs that execute across heterogeneous platforms consisting of central processing units (CPUs), graphics processing units (GPUs) and other processors. OpenCL includes a language (based on C99) for writing kernels (functions that execute on OpenCL devices), plus application programming interfaces (APIs) that are used to define and then control the platforms. OpenCL provides parallel computing using task-based and data-based parallelism.

SugarCubes latest version tries to take benefit of CPUs and GPUs by using OpenCL [68].

Ptolemy [34] is a platform on which multi-core architectures can be modeled. The Ptolemy project involves modeling, simulation, and design of concurrent, real-time, embedded systems. The focus is on assembling concurrent components. The key underlying principle is the use of well-defined models of computation that govern the interaction between components. A

major problem area being addressed is the use of heterogenous mixtures of models of computation.

1.3.3 Safety

Preventing time-dependent errors and finding ways to debug them are among the main problems of concurrency. Proposing solutions to these problems is thus the basis for providing a safe language. There already exists several solutions. We recall some of them in the next paragraphs.

Transactional Memory [39, 5, 4] is an attempt to simplify concurrent programming languages by allowing the atomic execution of a group of load and store instructions. The motivation of transactional memory is to transparently support the definition of regions of code that are considered as transactional and try to group the memory accesses of each region and execute them atomically.

PACT (PArtially Cooperative Threads). In his PhD thesis [30] Dabrowski propose a solution to monitor the resources (memory and CPU) in a formalisation close to Sπ-calculus. He proposed a formal method to ensure reactivity and safety in FairThreads, which has strongly inspired the design of the new language called FunLoft [26].

1.3.4 Security

Security is one of our main concerns. By security, we mean here securing the confidentiality of manipulated data. The way to ensure end-to-end protection of data confidentiality is secure information flow. More precisely, in this thesis, we focus on the notion of non-interference for reactive concurrent languages.

JFlow [49], is the first extension of a real programming language (Java) with secure information flow. Programs written in JFlow can be statically checked by the JFlow compiler, which prevents information leaks through storage channels.

JFlow supports the decentralized label model [50], which allows multiple principals to protect their privacy even in case of mutual distrust. It also supports safe, statically-checked declassification, allowing a principal to relax its own privacy policies.

JIF [51] is a security-typed programming language that extends Java with support for information flow control and access control, enforced at both compile time and run time. Jif is written in Java and is built using the Polyglot extensible Java compiler framework. Jif extends Java by adding labels that express restrictions on how information may be used.

Flow Caml [65] is a prototype implementation of an information flow analyzer for the Caml language. It consists in an extension of OCaml with a type system ensuring secure information flow. Its purpose is basically to allow the programmer to write "real" programs and to automatically check that they obey some confidentiality or integrity policy. In Flow Caml, standard ML types are annotated with security levels chosen in a user-definable lattice. Each annotation gives an approximation of the information that the described expression may convey. Because it has full type inference, the system verifies, without requiring source code annotations, that every information flow caused by the analyzed program is legal with respect to the security policy specified by the programmer.

Flow Caml is also one of the first real-size implementations of a programming language equipped with a security type system which features simultaneously subtyping, polymorphism and full type inference.

SPARK [2] is a formally-defined language based on the Ada language [70], which provides means for information flow policies (integrity and confidentiality).

1.4 Document Structure

In this thesis, we attempt to define a synchronous reactive model which enjoys both safety and security properties while being able to benefit fully from the multi-cores architectures. By safety, we mean the absence of time-dependent errors during execution. By security, we mean the absence of information leaks.

In order to reach this goal, we shall examine four different languages: 1) DSL (Dynamic Synchronous Language); 2) CRL (Core Reactive Language); 3) DSLM (Dynamic Synchronous Language with Memory); 4) SSL (Secure Synchronous Language). Each of these languages is studied in detail in a part of this thesis, which concludes with a discussion of future work. We give now the synopsis of each part.

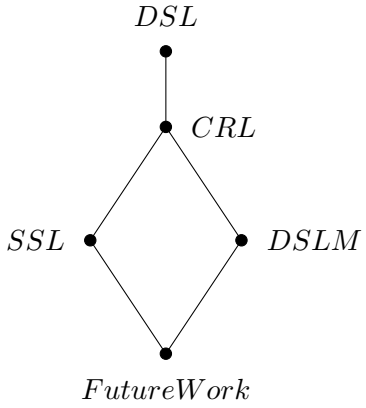


Figure 1.1: Document Structure

1.4.1 DSL

At the start of our work is the design and implementation of the DSL language. DSL is a member of the family of synchronous languages: basically, parallel components are sharing the same notion of instant, and during instants, they communicate and synchronize using broadcast events. As in most synchronous formalisms, the parallel operator of DSL is non-deterministic. DSL is based on the reactive variant of the synchronous approach: the reaction to the absence of an event is postponed to the next instant (recall that immediate reaction to absence is a source of causality cycles). Actually, DSL extends the standard reactive-synchronous model by introducing the notion of site and the possibility for parallel components (called scripts) to migrate between sites. The various sites forming a program are executed asynchronously (they do not share instants). From this point of view, DSL is thus an instance of the GALS model, mixing synchronous and asynchronous parallelism. Concerning memory, DSL makes a drastic simplification: the memory is viewed in an abstract way, and, from the programming level, it is only accessed through functions. More precisely, a change in the memory can only result from a function call, and a function call is mandatory to read the memory. In this respect, DSL can be seen as an orchestration language: the only things that matter are the moments (the instants) and the places (the sites) where functions are called. There are thus two almost completely separated levels in DSL: the orchestration level, in which the execution of scripts and functions is orchestrated, using events and migration orders; and the "host" level, in which the memory is

accessed and transformed by functions.

To summarize the main characteristics of DSL, we can say that it is a distributed (sites), reactive (instants), orchestration (host level) programming language. Moreover, it exhibits two forms of parallelism: the synchronous parallelism expressed by the standard synchronous operator (for scripts), and the asynchronous parallelism implicit in the presence of sites.

The work on DSL is described in Part 1. The programming model and the language are informally described in Chapter 2. Then, a formal semantics is given for the reactive part of the language, which describes the execution of scripts. This semantics is a "big-step" one: a rewriting of a script represents its execution during a whole instant. The possibility to give such an abstract formulation of the semantics basically results from two facts: the reactive character of the language, and the abstraction from the memory. The semantics of site execution and of migration between sites is given in Section 2.2.4. Finally, the implementation of DSL is considered in Chapter 3. Actually, we have implemented DSL in two functional languages (FunLoft and Scheme); in both cases, the orchestration and the host levels are implemented in the same language. Chapter 3 ends with a comparison of the various implementations of DSL (the previously mentioned ones, plus two others in ReactiveML and SugarCubes).

Shortcomings of DSL

The language DSL raises several issues. The first concerns memory abstraction. Indeed, this abstraction makes programming difficult in many contexts where one has to focus on memory related aspects. In these contexts, the orchestration character of DSL is more a negative than a positive aspect. In other words, DSL cannot definitely be considered as a general-purpose programming language.

The second issue is that in DSL there is no way to ensure that the memory is accessed in a correct way (for example, without data-races). This of course results from the memory abstraction. But it is also related to the distributed nature of DSL: how to prevent the memory to be accessed in an incorrect way by scripts executed asynchronously on distinct sites?

The third issue concerns security. The importance of security concerns is continuously increasing, as more and more data need to be protected from undesired accesses. DSL, as almost all programming languages, does not provide any means for data protection.

The fourth issue concerns the possibility to run DSL programs in an efficient way on multi-core machines. The usage of cores (and more generally

of CPUs) cannot be controlled by syntactic means in DSL. Moreover, to avoid incorrect accesses to the memory, the implementation has to choose a conservative approach, which more or less means that the memory cannot actually be shared. This certainly constitutes a major drawback for an optimized usage of multi-core machines.

The subsequent parts of the document are devoted to the analysis of these issues and to proposals to solve them.

1.4.2 CRL

The issues of security, on one hand, and of safety and optimized use of multi-cores on the other hand, are rather orthogonal. With the objective to reconcile them, we choose to focus in the first place on a kernel language, which will be later extended in several ways. This kernel, described in Part 2, is called CRL (Core Reactive Language) and can be viewed as DSL without migration (thus, with a unique site), and without host level. In CRL, data are manipulated directly, using a fixed set of operators (thus, there is no way to define functions). The use of operators simplifies the detection of non-terminating instants, compared to the more general use of functions. Moreover, we choose to use a deterministic asymmetric parallel operator (actually, a left-right operator), instead of the standard symmetric synchronous parallel operator of DSL. This change of parallel operator is basically motivated by the aim to incorporate a deterministic scheduling in the language itself, thus ensuring deterministic behavior without having to introduce an explicit scheduler (such an approach was first proposed in [22] and resumed in [8], but our variant has some novelties with respect to both). To allow fine tracking of security violations, one can no more use a big-step semantics; the formal semantics of CRL is thus expressed in a small-step framework. Using this semantics, we prove that the reactivity of scripts is still guaranteed, and we give a static bound for the number of steps it requires. The language CRL is described in Part 2.

1.4.3 SSL

The language SSL is an extension of CRL with security levels for data. It is a kind of minimal language for studying the problem of secure information flow in synchronous reactive programs. In the perspective of studying secure information flow, we also address the question of defining a semantic equivalence on programs. We define two bisimulation equivalences on SSL programs, corresponding to two different notions of observation (fine-

grained and coarse-grained). We prove that the first equivalence is strictly included in the second. We then define two non-interference properties based on these bisimulations which we call fine-grained reactive non-interference and coarse-grained reactive non-interference. We finally introduce a security type system, which we prove to ensure both non-interference properties. Thanks to the design choices of the language, this type system allows for a precise treatment of termination leaks, improving on previous work. The language SSL is describeddsl in Part 3.

1.4.4 DSLM

The language DSLM, considered in Part 4, extends CRL by introducing memory on one hand, and sites and migration on the other hand. DSLM provides two means for ensuring correct accesses to the memory: first, one defines the notion of agent, which has its own memory, only accessible by itself; second, events (with possible associated values) are the only means for agents to synchronize and communicate, and this can happen only when they are located on the same site. We show that data-races are not possible in this framework.

To increase expressivity, functions are also considered in DSLM. We make two basic assumptions on functions: first, they should always terminate; second, they should only access the memory of the calling agent. These assumptions have to be checked by the implementation of DSLM.

Concerning multi-core architectures, we have used the notion of synchronised scheduler to implement the sites. Each site actually contains a set of schedulers which run the agents present on the site. These schedulers share the same instants and the same events (they are synchronised). The isolation of the memory of each agent and the sharing of events among the schedulers allows the implementation to run each scheduler on a specific core and to transparently migrate an agent from a scheduler to another provided it belongs to the same site. In this way, we can let the implementation load-balance the agents onto the schedulers. The implementation is also free to adapt the number of schedulers used by the various sites, which constitutes another way to optimise the use of the cores.

The definition of DSLM and its formal semantics (in a small-step style) are described in Chapter 6. The implementation of DSLM, based on the notion of scheduler, is described at a semantical level in Chapter 8. We have implemented DSLM using FunLoft and C. In this implementation, functions are coded in FunLoft, and are thus proved to always terminate by the FunLoft compiler.

1.4.5 Conclusion

In Part 5, we conclude the document and propose some tracks for future work. One of them would be to design a unified formalism mixing together DSLM and SSL. The perspective would be to produce as outcome a general-purpose concurrent language, safe and secure, and able to fully benefit from multi-core architectures.

Part I

DSL

Chapter 2

The Dynamic Synchronous Language DSL

DSL (Dynamic Synchronous Language) is a core distributed synchronous reactive scripting language based on the GALS [48] model. In DSL, systems are composed of several sites executed asynchronously (possibly on different processing resources), and each site is running scripts in a synchronous parallel way (same notion of instant). Basically, DSL introduces sequence and non-deterministic parallel operators, event-based primitives, and a way to execute a script on a remote site. Scripts may call functions that are considered in an abstract way: their effect on the memory is not considered, but only their "orchestration" i.e. the organization of their calls in time (the instant at which they occur) and in place (the site where they are called).

The computing model we consider is the following: there are N sites, each of them being composed of two levels, the orchestration level and the host level. The sites are completely autonomous and are run asynchronously (possibly on different processing resources).

At the orchestration level, each site runs a script which is fundamentally parallel. At that level, inputs are:

- new scripts dropped by the other sites, by the external world, or by the host level of the site; these new scripts are put in parallel with the one currently executed by the site; events are input as simple scripts generating them.
- boolean values coming from the host level and used by `if` instructions;
- integer values coming from the host level and used by `repeat` statements.

The outputs of the orchestration level are new scripts sent to other sites.

Scripts run by the same site synchronize by means of local events which are broadcast in the whole site. Two properties are assumed in DSL: reactivity of sites, and absence of interferences between sites (i.e. sites do not share instants, nor memory, nor events). To assure these two properties we have two solutions : either we implement DSL in FunLoft and let the language take care of the verification, which would prevent us to fully benefit from multi-core architectures; or we can add memory and functions to the language level: then we will have to ensure these properties within the language but we will be able to benefit from multi-core architectures. In this thesis we rather choose the first solution.

Sites bear an analogy with a musical orchestra: the orchestration level corresponds to the orchestra conductor who leads the host level corresponding to the music players. The conductor follows a music partition (script) and communicates with the musicians by sending them orders and signals (modelled by events) and by listening to the music they play (also modelled by events; imagine events corresponding to sound waves produced by the instruments or by the voices). The conductor must be able to do several things in parallel: she must direct and listen to all instruments at the same time. Of course, as needed by any orchestra, a common clock is defined by the conductor; in our model, this clock which should be shared by the whole orchestra is given by instants. One can see the presence of several orchestras (sites) playing asynchronously as what happens sometimes in music festivals, when several stages are used independently, possibly simultaneously (of course in this case, the absence of interferences between distinct stages is mandatory: nothing should be shared by distinct sites).

DSL is designed with a simple formal semantics, describing without ambiguity how the system evolves. Our approach is an alternative to the use of locks for memory protection in a classic threading context.

The language DSL has been presented in the article [13], and it is also fully described in a technical report [14].

The language is first described informally in Section 2.1, and its formal semantics is given in Section 2.2. Some examples and several semantics variants are considered in 2.3.

2.1 Language Description

A program is composed of several independent sites, each of them executing a script made of parallel components. To add a new script into a site, one

puts the script in parallel with the already existing parallel components.

In the current version, DSL does not provide any means to define functions. However, scripts may call functions defined in a "host" language (different variants of DSL correspond to different host languages). These functions have parameters of basic types only (integer, boolean, string).

Modules are special functions whose execution is not immediate; actually, execution of a module does not start immediately, but at the next instant; moreover, the execution of a module can last several instants or even never terminate. Modules are called using a specific keyword (`launch`). Scripts "orchestrate" the execution of functions and modules on the various sites that compose a program.

We first present scripts in 2.1.1, then introduce sites in 2.1.2 and events in 2.1.3. The basic properties of DSL are presented in 2.1.4. Finally, an example which will be used to benchmark implementations is described in 2.1.5.

2.1.1 Scripts

Scripts are made of basic instructions, where syntax is the following:

$s \in$ **Script** ::= `nothing`
| `cooperate`
| $f\,(v_1, \ldots, v_n)$
| `launch` $m(ev, exp_1, ..., exp_n)$
| $s; s$
| $s \parallel s$
| `if` exp `then` s `else` s `end`
| `loop` s `end`
| `repeat` exp `do` s `end`
| `generate` ev
| `await` ev
| `do` s `watching` ev
| `drop` s `in` $site$

The informal semantics of scripts can be described as follows:

- `nothing` does nothing
- `cooperate` terminates the execution for the current instant. Execution resumes at the next instant.
- $f\,(v_1, \ldots, v_n)$ calls the function f with the parameters $v_1, ..., v_n$. Execution starts immediately and is instantaneous. To call a non-existing function is considered as an empty statement.

- `launch` $m\,(ev, v_1, \ldots, v_n)$ launches the module m with the parameters $v_1, ..., v_n$ and a fresh event ev which is generated when the module execution is over. Execution takes several (at least, one) instants to terminate, or may even never terminate. To call a non-existing module is considered as an empty statement.
- $s_1; s_2$ runs the two scripts s_1 and s_2 in sequence.
- $s_1 \parallel s_2$ runs the two scripts s_1 and s_2 in parallel. The parallel script terminates as soon as both s_1 and s_2 are terminated. The parallel in *DSL*, as the one in Esterel, is non-deterministic but confluent.
- `if` *exp* `then` s_1 `else` s_2 `end` runs the script s_i corresponding to the result of the evaluation of the boolean expression *exp*.
- `loop` s `end` cyclically runs the script s. Execution of s is restarted as soon as it terminates, except if it terminates instantly (i.e. in the same instant where it is started); in this last case, the loop waits for the next instant to restart s. There is thus no possibility to get an *instantaneous loop* which would cycle forever during the same instant.
- `repeat` *exp* `do` s `end` runs n times the script s, where n is the result of the evaluation of the integer expression *exp*. Note that we are using two different constructs for loops and iteration, in replacement of the standard *while* operator. This allows for a clear separation between non-terminating and iterative while loops.
- `generate` ev generates the event ev.
- `await` ev blocks execution while the event ev is not generated. Execution resumes as soon as ev is generated.
- `do` s `watching` ev executes the script s while the event ev is not generated. The execution of s is aborted when ev is generated. The `watching` instruction terminates normally when s terminates.
- `drop` s `in` *site* adds the script s in the remote site *site*. Execution continues immediately without waiting for the completion of s.

2.1.2 Sites

Sites are asynchronous (i.e., each site is possibly run by a distinct native thread). On the contrary, scripts are executed synchronously on a site: they share the same instants and thus proceed at the same pace.

The `drop` instruction is the means by which a script can influence remote sites. Note that, if nothing remains to be done after a `drop` instruction, one can see it as a *migration* to the remote site.

The creation of sites is not specified in the language; we suppose that for each script of the form `drop` *s* `in` *site*, *site* always exists and is accessible.

2.1.3 Events

Events are boolean values that are present or absent during instants. Events are not transmitted among sites. Once an event is generated during an instant, it remains present up to the end of the instant. Events are automatically reset to absent at the beginning of each instant. Events used by the three instructions `generate`, `await`, and `watching` are created if they are not already existing on the site of execution.

2.1.4 Basic Properties

DSL requires that the two following fundamental properties are valid:

- No site can be prevented from passing to the next instant (*reactivity* property). This means that functions and modules run by a site should not use all of the computing power of the site.
- No data-race can occur between scripts, functions and modules (*interference freeness* property).

In the FunLoft variant, the fundamental properties are checked by the compiler which verifies that:

- Functions always terminate instantaneously.
- modules always cooperate.
- Memory can only be shared by functions or modules to the same site.

In the other variants, the validity of the two fundamental properties is left to the programmer.

2.1.5 Example

We consider a system composed of three sites `site1`, `site2`, and `site3`, and a script supposed to be run by `site1`; the script is made of two sub-parts

executed on `site2` and `site3`. Each sub-part calls the `consume` function (which heavily uses the CPU, according to the value of its parameter) and then drops back a script on `site1` to signal its termination. The two events generated upon termination are awaited in parallel. The code is:

```
repeat 1000 do
 (
  drop
    print("0");
    consume(1E7);
    drop generate done0 in site1
  in site2
 ||
  drop
    print("1");
    consume(1E7);
    drop generate done1 in site1
  in site3
 ||
  await done0
 ||
  await done1
 );
 cooperate
end
```

Note that there are similar parts in the code (for example, the two calls to `consume`). Actually, the DSL language does not give any means to share or parameterize scripts. In this respect, scripts are not very friendly and should thus be produced from some higher-level language; the definition of such a language is not in the scope of this thesis.

The two calls of `consume` can be executed in real parallelism (for example, on a dual-core machine). It is assumed that no interferences appear between them (for example, resulting from the sharing of a global counter). This assumption is statically verified in the FunLoft variant of DSL, while it is the responsibility of the programmer in the other variants. We shall return on this example later, when implementation is considered.

Remark: the body of a `repeat` statement is not demanded to be non-instantaneous, unlike the body of a `loop` statement. Indeed, a `repeat` script always terminates (provided its body terminates), and thus cannot prevent the other scripts to get the control. In the previous code, the justification for the `cooperate` is to prevent an instantaneous termination of the `repeat`

if both `done0` and `done1` are received in the same instant; this is actually possible because of the asynchronous execution of sites.

2.2 Semantics

We give DSL a semantics expressed with rewriting rules. The semantics is "big-step": one rewriting of a term represents the global execution of the term during one instant (by contrast "small-step" semantics would describe the various execution steps occurring during the instant).

Evaluation of expressions is considered in 2.2.1. The (big-step) rewriting of scripts is first described in 2.2.2; then, fix-points are considered in 2.2.3; site execution is described in 2.2.4; three examples are considered in 2.3; finally, three variants of the semantics are analyzed in 2.3.2.

2.2.1 Expressions

Expressions are either basic values (of type integer, boolean, or string), or calls of functions of the form $f\,(v_1, \ldots, v_n)$ where the v_i are basic values. We adopt the following notation: we write $f\,(v_1, \ldots, v_n)\downarrow$ if there is no function named f which is defined, or if the call is not well typed; in this case we say that we have a *wrong call*; we write $f\,(v_1, \ldots, v_n)\uparrow$ otherwise.

The evaluation of a basic value returns itself. There are two cases for the evaluation of $f\,(v_1, \ldots, v_n)$:

- if $f\,(v_1, \ldots, v_n)\uparrow$, the evaluation of the call returns the value of $\mathbf{f}$ applied to the list of values v_i, where $\mathbf{f}$ is the function associated to f;
- if $f\,(v_1, \ldots, v_n)\downarrow$, then the value returned is the default value of the expected (basic) type (0 for integers, `false` for booleans, and the empty string `""` for strings).

The evaluation of the expression exp returning a value v is noted $exp \rightsquigarrow v$.

As with functions, we write $m\,(v_1, \ldots, v_n)\downarrow$ if the module m does not exist or if the call is not well typed, and we write $m\,(v_1, \ldots, v_n)\uparrow$ otherwise.

2.2.2 Scripts

The general format of the script semantics is:

$$P \vdash s \xrightarrow{b} s', G, D$$

- P is the set of present events; events not belonging to P are absent;

- s is the script which is rewritten;
- s' is the *residual* script ("what remains to do at the next instant");
- G is the set of events generated by the rewriting of s;
- D is the multi-set of *dropped scripts* of the form $site{\downarrow}u$, where $site$ is a site name and u is a script; the union of multi-sets is noted $\uplus$;
- b is a boolean which is true (tt) if s' is terminated and false (ff) otherwise; the boolean conjunction is noted $\wedge$.

The semantics of scripts is given by the following rules:

Nothing

$$P \vdash \texttt{nothing} \xrightarrow{tt} \texttt{nothing}, \emptyset, \emptyset \tag{2.1}$$

Cooperate

$$P \vdash \texttt{cooperate} \xrightarrow{ff} \texttt{nothing}, \emptyset, \emptyset \tag{2.2}$$

The execution of a cooperate terminates instantaneously for the current instant.

Drop

$$P \vdash \texttt{drop}\ s\ \texttt{in}\ site \xrightarrow{tt} \texttt{nothing}, \emptyset, \{site{\downarrow}s\} \tag{2.3}$$

Sequence

$$\frac{P \vdash s_1 \xrightarrow{ff} s_1', G, D}{P \vdash s_1; s_2 \xrightarrow{ff} s_1'; s_2, G, D} \tag{2.4}$$

$$\frac{P \vdash s_1 \xrightarrow{tt} s_1', G_1, D_1 \quad P \vdash s_2 \xrightarrow{b} s_2', G_2, D_2}{P \vdash s_1; s_2 \xrightarrow{b} s_2', G_1 \cup G_2, D_1 \uplus D_2} \tag{2.5}$$

The semantics of a sequence considers the case where the first branch is not terminated (ff), and the case where it is (tt). In the first case, the execution of the sequence is over for the current instant. In the second case, the second script is executed.

Parallel

$$\frac{P \vdash s_1 \xrightarrow{b_1} s_1', G_1, D_1 \quad P \vdash s_2 \xrightarrow{b_2} s_2', G_2, D_2}{P \vdash s_1 \parallel s_2 \xrightarrow{b_1 \wedge b_2} s_1' \parallel s_2', G_1 \cup G_2, D_1 \uplus D_2} \tag{2.6}$$

Parallel branches are executed at the same time.

Loop

$$\frac{P \vdash s \parallel \texttt{cooperate} \xrightarrow{ff} s', G, D}{P \vdash \texttt{loop}\ s\ \texttt{end} \xrightarrow{ff} s'; \texttt{loop}\ s\ \texttt{end}, G, D} \tag{2.7}$$

A loop statement executes its body cyclically: a `cooperate` instruction is systematically added in parallel to its body to avoid instantaneous loops.

Generate

$$P \vdash \texttt{generate}\ ev \xrightarrow{tt} \texttt{nothing}, \{ev\}, \emptyset \tag{2.8}$$

The generate instruction produces an event in the environment.

Await

$$\frac{ev \in P}{P \vdash \texttt{await}\ ev \xrightarrow{tt} \texttt{nothing}, \emptyset, \emptyset} \tag{2.9}$$

An await instruction terminates if the awaited event is present in the environment. If the event is not present, the execution is over for the current instant:

$$\frac{ev \notin P}{P \vdash \texttt{await}\ ev \xrightarrow{ff} \texttt{await}\ ev, \emptyset, \emptyset} \tag{2.10}$$

Watching

$$\frac{P \vdash s \xrightarrow{tt} s', G, D}{P \vdash \texttt{do}\ s\ \texttt{watching}\ ev \xrightarrow{tt} \texttt{nothing}, G, D} \tag{2.11}$$

A watching statement executes its body. If the body is terminated (i.e. it is nothing), then the watching statement rewrites in a `nothing` instruction. If the body is not terminated and the watching event is present, it rewrites to `nothing`:

$$\frac{ev \in P \quad P \vdash s \xrightarrow{ff} s', G, D}{P \vdash \texttt{do}\ s\ \texttt{watching}\ ev \xrightarrow{ff} \texttt{nothing}, G, D} \tag{2.12}$$

As long as the body is not terminated and the watching event is not present, the watching statement rewrites its body:

$$\frac{ev \notin P \quad P \vdash s \xrightarrow{ff} s', G, D}{P \vdash \texttt{do}\ s\ \texttt{watching}\ ev \xrightarrow{ff} \texttt{do}\ s'\ \texttt{watching}\ ev, G, D} \tag{2.13}$$

Evaluation of expressions (function calls are expressions) appears in the following rules which are thus less formal than the previous ones; indeed, evaluation of expressions is not totally captured by the semantics of DSL.

Function

$$\frac{f\,(v_1, \ldots, v_n) \rightsquigarrow v}{P \vdash f\,(v_1, \ldots, v_n) \xrightarrow{tt} \texttt{nothing}, \emptyset, \emptyset} \tag{2.14}$$

Execution of a function call is equivalent to its evaluation; the returned value is of no use, and the call is actually only evaluated for its side-effects (a wrong call does nothing and has no side-effect).

Modules

$$\frac{m\,(ev, v_1, \ldots, v_n)\downarrow}{P \vdash \texttt{launch}\ m\,(ev, v_1, \ldots, v_n) \xrightarrow{tt} \texttt{nothing}, \emptyset, \emptyset} \tag{2.15}$$

$$\frac{m\,(ev, v_1, \ldots, v_n)\uparrow}{P \vdash \texttt{launch}\ m\,(ev, v_1, \ldots, v_n) \xrightarrow{ff} \texttt{await}\ ev, \emptyset, \emptyset} \tag{2.16}$$

Three points should be noted:

- Rule 2.15 states that a wrong call of a module is equivalent to a `nothing` statement.
- In rule 2.16, ev is a new event[1] which signals the termination of the launched module; it is automatically generated by the system when the call of m turns to be completely terminated.
- In case of real preemption, i.e. when rule 2.12 applies, the waiting for termination is abandoned and the module is not actually started.

Repeat

$$\frac{exp \rightsquigarrow n \quad P \vdash \overbrace{s; \ldots; s}^{n\ times} \xrightarrow{b} s', G, D}{P \vdash \texttt{repeat}\ exp\ \texttt{do}\ s\ \texttt{end} \xrightarrow{b} s', G, D} \tag{2.17}$$

Two points should be noted:

- Evaluation of exp is performed when the rule is applied, that is at execution time (not at compile time).
- In case exp is a wrong function call, n is equal to 0, and the sequence is equal to `nothing`[2]. The `repeat` statement is thus in this case equivalent to `nothing`.

[1] a mechanism to produce new fresh events is assumed.
[2] a sequence of $n \leq 0$ elements is by definition equal to `nothing`.

If

$$\frac{exp \rightsquigarrow tt \quad P \vdash s_1 \xrightarrow{b} s_1', G, D}{P \vdash \texttt{if}\ exp\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{end} \xrightarrow{b} s_1', G, D} \tag{2.18}$$

$$\frac{exp \rightsquigarrow ff \quad P \vdash s_2 \xrightarrow{b} s_2', G, D}{P \vdash \texttt{if}\ exp\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{end} \xrightarrow{b} s_2', G, D} \tag{2.19}$$

Note that if *exp* is a wrong function call, its evaluation returns *ff*, and thus s_2 is chosen.

2.2.3 Least Fix-Point

It is easy to see (by inspecting the rules) that the execution of scripts is deterministic:

if $P \vdash s \xrightarrow{b_1} s_1, G_1, D_1$ and $P \vdash s \xrightarrow{b_2} s_2, G_2, D_2$, then $s_1 = s_2$, $G_1 = G_2$, $D_1 = D_2$, and $b_1 = b_2$.

Let s be a script; the determinism property allows one to define the function f_s which, given a set P of present events, returns the set G of events generated by s:

$$f_s(P) = G \text{ where } P \vdash s \xrightarrow{b} s', G, D$$

It may also be easily shown that the function f_s has two main characteristics: it is total and it is increasing. It is total because, for each script and each set of present events, there exists a (unique) rewriting:

$$\forall s, P, \exists s', G, D, b \quad P \vdash s \xrightarrow{b} s', G, D$$

The function f_s is increasing (for the set inclusion order):

$$\text{if } P_1 \subseteq P_2 \text{ then } f_s(P_1) \subseteq f_s(P_2)$$

The function f_s thus has a least fix-point μf_s (Kleene theorem) verifying:

$$f_s(\mu f_s) = \mu f_s$$

that is:

$$\mu f_s \vdash s \xrightarrow{b} s', \mu f_s, D$$

and:

$$f_s(Q) = Q \text{ implies } \mu f_s \subseteq Q$$

We know that the least fix-point μf_s is the limit of the sequence of approximations $X_0, X_1, \ldots$ defined by:

$$X_0 = \emptyset \text{ and } X_{n+1} = f_s(X_n)$$

which is noted:

$$\mu f_s = \bigcup f_s^n(\emptyset)$$

Finally, when the value of the least fix-point is not required, we write:

$$s \Rightarrow s', D$$

instead of:

$$\mu f_s \vdash s \xrightarrow{b} s', \mu f_s, D$$

2.2.4 Programs and Sites

A site is a couple $(site, s)$ made of a site name $site$ and a script s; it is noted $site : s$.

A *program* is a (finite) multi-set of sites and of dropped scripts waiting to be incorporated into sites. A program is thus a multi-set S where each element is either a site $site_i : s_i$ or a dropped script $site_i \downarrow s_i$. One supposes that there is at least one site and that all sites have distinct names:

$$\forall site_i : s_i, site_j : s_j \in S, i \neq j \Rightarrow site_i \neq site_j$$

Note that the same dropped element can appear several times in a program (it is a multi-set), as for example in:

$$\{site : \texttt{nothing}, site \downarrow f(), site \downarrow f()\}$$

The execution of a program S_0 is a sequence of rewritings of the form:

$$S_0 \mapsto S_1 \mapsto S_2 \mapsto ...$$

where the arrow $\mapsto$ is defined by rules 2.20, 2.21, and 2.22 given below.

Site execution

$$\frac{s \Rightarrow s', D}{\{..., site : s, ...\} \mapsto \{..., site : s', ...\} \uplus D} \tag{2.20}$$

The dropped scripts resulting from a site execution are added in the program by rule 2.20; they are waiting to be absorbed by rule 2.21 below. In the definition of $\Rightarrow$, note that the least fix-point is not explicitly built: the semantics is not effective in this respect as it does not indicates *how* to compute it.

Absorption of dropped scripts

$$\{..., site : s, site{\downarrow}u, ...\} \mapsto \{..., site : s \parallel u, ...\} \tag{2.21}$$

Rule 2.21 represents the absorption of a dropped script u by the appropriate site *site*: the dropped script is simply put in parallel with the script s already present in *site*.

Inputs The dynamic adding of a script s in the site *site* of a program S is modeled by:

$$S \mapsto S \uplus \{site{\downarrow}s\} \tag{2.22}$$

Program inputs are dropped events: the input of the event e in the site *site* is simply modeled by the rewriting:

$$S \mapsto S \uplus \{site{\downarrow}\texttt{generate}\ e\}$$

2.3 Examples and Semantical Variants

2.3.1 Examples

We give several examples: the first shows the computation of the semantics by successive approximations; the second illustrates the links between the fix-point semantics and the notion of causality; the third example concerns the `drop` primitive; the fourth illustrates the relation between the `watching` and `launch` instructions; finally, the last example shows the global execution of a program.

Approximations

Let us consider the following script s:

$$\texttt{generate}\, ev_1; \texttt{await}\, ev_2 \parallel \texttt{await}\, ev_1; \texttt{generate}\, ev_2$$

Actually, one can prove that:

$$\{ev_1, ev_2\} \vdash s \xrightarrow{tt} \texttt{nothing} \parallel \texttt{nothing}, \{ev_1, ev_2\}, \emptyset$$

Let us show that this corresponds to the least fix-point μf_s of f_s (using the previous notations). Let $X_0 = \emptyset$. One has:

$$X_0 \vdash s \xrightarrow{ff} \texttt{await}\, ev_2 \parallel \texttt{await}\, ev_1; \texttt{generate}\, ev_2, \{ev_1\}, \emptyset$$

Let $X_1 = \{ev_1\}$. We have:

$$X_1 \vdash s \xrightarrow{ff} \texttt{await}\, ev_2 \parallel \texttt{nothing}, \{ev_1, ev_2\}, \emptyset$$

Let $X_2 = \{ev_1, ev_2\}$. Since:

$$X_2 \vdash s \xrightarrow{tt} \texttt{nothing} \parallel \texttt{nothing}, X_2, \emptyset$$

we get the result:

$$\mu f_s = \bigcup f_s^n(\emptyset) = X_2 = \{ev_1, ev_2\}$$

.

Minimality

Minimality of fix-points is mandatory to reject "violations of causality". Indeed, consider the following script $s = \texttt{await}\, ev; \texttt{generate}\, ev$. Two fix-points, $\{ev\}$ and $\emptyset$, exist:

1. $\{ev\} \vdash s \xrightarrow{tt} \texttt{nothing}, \{ev\}, \emptyset$
2. $\emptyset \vdash s \xrightarrow{ff} s, \emptyset, \emptyset$.

The least fix-point is thus $\emptyset$. Note that in the first rewriting, the generation of ev "results" from the test of presence of ev, and thus does not correspond to any "causal" execution. In a sense, the minimality of fix-points is a way to rule out non-causal executions.

Asynchrony

Let us consider the script:

$$\texttt{drop generate}\ ev \parallel \texttt{await}\ ev; print(\text{“}ok\text{”})\ \texttt{in}\ site_1$$

The message will always be printed, because the dropped script is incorporated in $site_1$ as a whole. This would not be the case with:

$$\begin{array}{l}\texttt{drop generate}\ ev\ \texttt{in}\ site_1;\\ \texttt{drop await}\ ev; print(\text{“}ok\text{”})\ \texttt{in}\ site_1\end{array}$$

Indeed, $site_1$ may incorporate the first script and may react *before* the incorporation of the second script; in this case, the message is not printed because the generation of ev is lost.

Module Abortion

Let us consider the immediate preemption of a module launched by the body of a `watching` statement:

$$\begin{array}{l}\texttt{generate}\ ev;\\ \texttt{do launch}\ m(ev')\ \texttt{watching}\ ev\end{array}$$

If module `m` does not exist or is incorrectly called, then the global instruction terminates immediately (rules 2.15 and 2.11). Otherwise (`m` exists and is correctly called), the module is not launched by the executive system (last remark, rule 2.16), and the instruction will terminate at the next instant (rule 2.12).

Program Input

Let us consider the following program made of a unique site:

$$S = \{site : \texttt{await}\ ev; print(msg)\}$$

The only rewriting that can be made is:

$$S \mapsto S$$

Suppose a new input is given to the program, which becomes S':

$$S \mapsto S' = S \uplus \{site{\downarrow}\texttt{generate}\ ev\}$$

There are two possible rewritings for S':

$$S' \mapsto S'$$

and (rule 2.21):

$$S' \mapsto S'' = \{site : \texttt{await}\ ev; print(msg) \parallel \texttt{generate}\ ev\}$$

Now, one can prove that the only possible rewriting of S'' is:

$$S'' \mapsto \{site : \texttt{nothing}\}$$

During this rewriting, the function *print* is called and a message is printed as a side-effect of the call.

2.3.2 Variants of the Semantics

In this section, we discuss three aspects of the semantics: instantaneous loops are first considered; then, a variant of the drop instruction is analyzed; finally, the watching instruction is discussed.

Instantaneous Loops The fact that the function f_s is total basically results from the rule 2.7 that "fixes" instantaneous loops. Note that without the fix, some loops could have no rewriting at all; this would be for instance the case with the rule:

$$\frac{P \vdash s; \texttt{loop}\ s\ \texttt{end} \xrightarrow{b} s', G, D}{P \vdash \texttt{loop}\ s\ \texttt{end} \xrightarrow{b} s', G, D} \qquad (2.7')$$

in which the execution of a loop basically means to unfold it. The reactivity property of DSL would thus be lost by using this rule instead of rule 2.7.

Packed Drop Let us consider a possible variant of the semantics in which the dropped scripts are grouped by destination. The idea is that, instead of dropping one after the other several scripts intended for the same site, one drops the parallel composition of these scripts, in one unique drop action. This reduces the asynchrony of site execution and thus makes the reasoning about programs easier. To model this variant, we first define the *pack* function which takes a multi-set of dropped scripts and returns the set which is the compact version of it:

- $pack\ (D) = D$ if $\forall d_1 = (site_1 \downarrow s_1) \in D, \forall d_2 = (site_2 \downarrow s_2) \in D,\ d_1 \neq d_2$ implies $site_1 \neq site_2$
- $pack\ (D \uplus \{(site \downarrow s_1), (site \downarrow s_2)\}) =$
  $pack\ (D \uplus \{(site, s_1 \parallel s_2)\})$

The site execution rule 2.20 becomes:

$$\frac{s \Rightarrow s', D}{\{..., site : s, ...\} \mapsto \{..., site : s', ...\} \uplus pack\ (D)} \qquad (2.20')$$

Note that the two `drop` scripts of 2.3.1 become equivalent in this variant of the semantics. To implement the variant, dropped scripts have to be stored, up to the end of the current instant, before being compacted and actually sent to remote sites.

Preemption Operator The basic assumption of the model resides in the couple of rules 2.12 and 2.13 which state that the body of a `watching` instruction is executed in both cases of presence and of absence of ev. The alternative proposed by Esterel, called "strong preemption", corresponds to the following two rules:

$$\frac{ev \in P}{P \vdash \texttt{do}\ s\ \texttt{watching}\ ev \xrightarrow{tt} \texttt{nothing}, \emptyset, \emptyset} \qquad (2.12')$$

$$\frac{ev \notin P \quad P \vdash s \xrightarrow{b} s', G, D}{P \vdash \texttt{do}\ s\ \texttt{watching}\ ev \xrightarrow{b} \texttt{do}\ s'\ \texttt{watching}\ ev, G, D} \qquad (2.13')$$

With these rules, the body is immediately executed *in absence* of ev (rule 2.13′), and it is not executed at all when ev is present (rule 2.12′). One thus has an immediate reaction to the absence of ev, which introduces "causality cycles" (e.g. `do generate` ev `watching` ev). Causality cycles are a major obstacle to the introduction of dynamic thread creation in Esterel. It is thus clear that strong preemption cannot be, in a way or another, introduced coherently in DSL.

We could have replaced rule 2.11 by the following:

$$\frac{P \vdash s \xrightarrow{tt} s', G, D}{P \vdash \texttt{do}\ s\ \texttt{watching}\ ev \xrightarrow{ff} \texttt{nothing}, G, D} \qquad (2.11')$$

This alternative rule gives a more uniform treatment of the preemption operator that actually would *never* terminate instantaneously. However, we prefer to keep rule 2.11 because it entails the following intuitive invariant: `do` s `watching` ev is strictly equivalent to s if ev is never present. This invariant would be violated with the alternative rule 2.11′.

Chapter 3

DSL Implementation

DSL has been implemented in four different languages: SugarCubes, ReactiveML, FunLoft and Bigloo/Scheme. In this chapter we present the two implementations that we have realized in FunLoft and Bigloo/Scheme. The two other implementations have been done by Louis Mandel and Jean-Ferdinand Susini and are presented in [14]. Finally, we compare the four implementations on the same benchmark of Section 2.1.5.

3.1 FunLoft Variant

In the FunLoft variant of DSL, a script is first translated into an instruction of the type `instruction_t` defined in FunLoft, before being compiled by the FunLoft compiler. The definition of FunLoft insures the reactivity and memory protection properties of the compiled code (actually, the static checks for bounded resource consumption are switched-off in the FunLoft compiler, but the remaining checks are sufficient to insure reactivity and memory protection).

The translation has the following characteristics:

- The notion of an instant is re-built: an instant of a script is made of several micro-steps of the target FunLoft program, each micro-step corresponding actually to one instant of the translated FunLoft program.
- Events are represented by strings. A hashtable (of the type `aa_t`) associating strings with events is available on each site.
- The generation of a DSL event is sustained during the following micro-steps, up to the end of instant.

- A valued event is used to deal with the dynamic adding of new scripts in a site (it has type `(instruction_t) event_t`).

3.1.1 Dynamic Adding of Instructions

A special event is associated with each site, used to add the scripts dropped in the site. The module `dynamic` awaits this event and collects its associated values using the `get_all_values` instruction of FunLoft; the collected instructions are processed by calling the function `incorporate`, defined below. The code of `dynamic` is:

```
let module dynamic (eng) =
  let inst_list = ref Nil_list in
    loop
      let add = Engine.add (eng) in
        begin
          await add;
          get_all_values add in inst_list;
          incorporate (eng,!inst_list);
          generate Engine.wakeup (eng);
          continue_instant (eng);
        end
```

The `continue_instant` function just sets the flag `move` of the engine:

```
let continue_instant (eng) =
   Engine.move (eng) := true
```

The function `incorporate` is recursively defined and the FunLoft compiler checks that it always terminates:

```
let incorporate (eng,inst_list) =
  match inst_list with
    Nil_list do ()
  | Cons_list (head,tail) do
      begin
        thread evaluate (eng,head,event);
        incorporate (eng,tail);
      end
  end
```

To drop an instruction in a site, basically means to generate the special event of the site engine with the instruction as value:

```
let module send_to (site,inst) =
  link Site.sched (site) do
    let engine = !Site.rengine (site) in
      generate Engine.add (engine) with inst
```

3.1.2 Reactive Engine

The reactive engine of a site basically sustains the generated events and decides when instants are terminated. Each time an event is generated, it is stored in the list `sustain` of the engine, in order to be re-generated at each micro-step, up to the end of instant. Moreover, the `move` flag of the engine is set (by calling `continue_instant`) to resume execution of scripts awaiting the event, if there are such scripts:

```
let dsl_generate (eng,evt) =
  let e = event_lookup (eng,evt) in
    begin
      generate e;
      let s = Engine.sustain (eng) in
        s := Cons_list (e,!s);
      continue_instant (eng);
      generate Engine.wakeup (eng);
    end
```

The algorithm of the engine is the following: micro-steps are executed cyclically while the `move` flag is set; when the `move` flag has not been set by the last micro-step, the `pre_eoi` flag is set to let `watching` statements proceed, in case of preemption. Indeed, in this case, a `watching` has to let its body react, in order to choose safely between rules 2.11 and 2.12. Cyclic execution is then resumed as previously, while there are new moves. The end of the current DSL instant is decided when the setting of `pre_eoi` does not produce any new move; in this case the `eoi` flag is set, to indicate the end of the current instant. This algorithm corresponds to the following code:

```
let module react (eng) =
  let move = Engine.move (eng) in
    loop
      begin
```

```
        move := false;
        sustain_all (eng);
        cooperate;
        if not !move then
          begin
            generate Engine.pre_eoi (eng);
            cooperate;
            if not !move then
              begin
                close_instant (eng);
                return;
              end
            end
          end
        end
      end
```

The `sustain_all` function maintains the generated events during the next micro-steps, up to the end of instant. The `close_instant` function generates `eoi`, stops the sustainment of generated events, and increments the instant counter:

```
let close_instant (eng,trace) =
  let instant = Engine.instant (eng) in
  begin
    generate Engine.eoi (eng);
    Engine.sustain (eng) := Nil_list;
    instant++;
 end
```

Note that the flag `pre_eoi` becomes useless if we replace the rule 2.11 by the alternative rule 2.11′. The alternative rule would thus simplify the implementation in FunLoft (however, this would not be the case for the other variants of DSL).

3.1.3 Functions and Tasks

Functions are called using the `Call` instruction. Functions and parameters are represented as character strings. The function `call_dispatch` analyses the first string passed to `Call` and calls the corresponding function. Functions that are used in a program must always be called through the call dispatcher. Here is an example of call dispatcher:

```
let call_dispatch (eng,fun,params) =
  if fun = "print_string" then
    let p1 = get_param (params,0) in
      print_string (p1)
  else if fun = "print_int" then
    let p1 = string2int(get_param(params,0)) in
      print_int (p1)
  else if fun = "quit" then
    quit (0)
  else
    warning ("unknown call")
```

Tasks are launched in sites with the `Launch` instruction. Like functions, tasks and parameters are represented as character strings. The module `task_dispatch` analyses the first string and launches the appropriate thread. An event given as parameter is generated at the end of the task execution. Here is an example of a task dispatcher:

```
let module task_dispatch (eng,task,params,evt) =
  begin
    if task = "print_getchar" then
      run print_getchar ()
    else if task = "getchar" then
      run getchar ()
    ...
    generate evt;
  end
```

Note that the compiler checks that non-cooperative FunLoft functions are always called while unlinked. Here is an example of use of the non-cooperative FunLoft function `fl_getchar` (which basically corresponds to the `getchar` function of C):

```
let getchar_result = ref ' '

let module getchar () =
  let loc = local ref ' ' in
    begin
      unlink loc := fl_getchar ();
      link main_scheduler do
```

```
        getchar_result := !loc;
    end
```

The compiler complains if the `unlink` statement is omitted. Note the use of a local reference to store the read character; an intermediate local reference is mandatory because it is not possible to access the global reference `getchar_result` while unlinked (otherwise, data-races could occur while accessing it).

Note that the executing engine is given as parameter to `call_dispatch` and `task_dispatch`; this allows the function `dsl_generate` to be called by functions and tasks. Communication through events can occur by this means from the host level to the orchestration level.

3.1.4 Instantaneous Loops

The FunLoft compiler statically checks for the absence of instantaneous loops; more precisely, it rejects a program in which a loop body has the possibility to terminate instantly. In the present context, this means that there is no possibility for an instruction to cycle during the same micro-step. According to the semantics of DSL, loops that would cycle during the same (DSL) instant are dynamically detected and "corrected"; this is for example the case with:

```
Loop (Nothing)
```

The implementation proceeds as follows: the DSL instant is stored when the body of a loop starts to execute and the system checks that the instant is different from the stored instant when the body terminates. When the two instants are the same (the body thus terminates instantly), the system forces the body to wait for the next DSL instant (it is as if a `cooperate` statement were dynamically inserted at the end of the body when it terminates instantly).

3.1.5 Static Checks

The static checks performed by the compiler are the ones of FunLoft, excepted those insuring that the consumption of resources is bounded (actually, the execution engine defined does not run in bounded memory, basically because new scripts can always be dropped dynamically in sites).

Reactivity

Basically, the reactivity property comes from the fact that there is no way for a script to prevent the execution of the other scripts by exhausting the CPU. More precisely:

- It is not possible to define recursive scripts in DSL. A script can only launch an already defined task.
- Recursive tasks (recursive FunLoft modules) are allowed because they are needed by the implementation in FunLoft; however, execution of a launched task does not begin immediately, but at the next DSL instant; thus, there is no possibility for a recursively defined task to cycle forever during the same micro-step. Hence, no task could cycle forever during the same DSL instant.
- Functions are proved to always terminate. An example of correct function is:

```
let length(l) =
  match l with Nil_list -> 0
        | Cons_list (h,t) -> 1 + length(t)
  end
```

On the contrary, the following function is rejected:

```
let f(l) =
  match l with Nil_list -> 0
        | default -> f (l)
```

Memory Protection

Each site (scheduler) has its own memory, which is protected from accesses by instructions run by the other sites. As a consequence, no data-race can occur from the parallel execution of two scripts run on two distinct sites (thus, run by two distinct native threads).

Moreover, tasks may have a local private memory and the system verifies that this memory cannot be accessed by the other tasks. Basically, a task is not allowed to store one of its private references into a public reference accessible by the other tasks.

To illustrate the memory protection technique, let us consider the following task dispatcher:

```
let r = ref 0

let module task_dispatch (eng,task,params,evt) =
  if task = "tst1" then
    thread tst1 (r)
  else if task = "tst2" then
    thread tst2 (r)
  ...
```

An error is detected if both tasks `tst1` and `tst2` access the reference `r` while on different sites; indeed, in this case, there could be a data-race while accessing `r`. A way to fix the bug is to force the two tasks to be in the same site:

```
let module task_dispatch (eng,task,params,evt) =
  if task = "tst1" then
    link site1_sched do thread tst1 (r)
  else if task = "tst2" then
    link site1_sched do thread tst2 (r)
  else
    ...
```

3.1.6 Execution of Instructions

Let us return to the script of Section 2.1.5. An equivalent FunLoft program is:

```
#include "dsl3.fl"

let turns = 1000
let consume_value =
  Cons_list ("10000000",Nil_list)

let remote (from,target,msg,done) =
  Drop (target,
    Seq (Print (msg),
    Seq (Call ("consume",consume_value),
         Drop (from,Generate (done)))))

let parallel =
  Repeat (IntConst (turns),
```

```
    Seq (
      Par (remote (site1,site2,"0","done0"),
      Par (remote (site1,site3,"1","done1"),
      Par (Await ("done0"),Await ("done1")))),
      Cooperate) )

let module dsl_main () =
  drop_in_site1 (parallel)
```

The include directive of the file `dsl3.fl` defines the types and the constructors used for instructions, and three sites `site1`, `site2`, and `site3`. The function `remote` is called twice by the instruction `parallel`. Finally, the module `dsl_main` is defined; it is actually the program entry point. The body of `dsl_main` simply drops the instruction `parallel` in `site1`.

3.1.7 Translation in FunLoft

A translator of DSL in FunLoft is implemented; it translates the script of Section 2.1.5 into the following instruction:

```
Repeat
 (IntConst (1000),
 Seq (Par (Par (Par
   (Drop (site2, Seq (Seq (Print ("0"),
                           Call ("consume",
                                 Cons_list ("10000000", Nil_list))),
                      Drop (site1,Generate ("done0")))),
    Drop (site3,
                  Seq
                      (Seq (Print ("1"),
                            Call ("consume",
                                  Cons_list ("10000000", Nil_list))),
                       Drop (site1,Generate ("done1"))))),
    Await ("done0")),
    Await ("done1")),
 Cooperate))
```

To execute it, one just replaces the definition of `parallel` by this instruction, in the FunLoft program of Section 3.1.6.

3.2 Bigloo/Scheme Variant

The main points of the implementation of DSL in Scheme/Bigloo are:

- Sites are coded in Scheme/Bigloo and executed by a native thread.
- Script execution basically follows the semantic rules described in previous section.
- As there is no notion of instant in Scheme/Bigloo, we introduce the notion of reactive machine to implement it.
- In Scheme/Bigloo there is no difference between functions and tasks.

3.2.1 Sites

In our implementation a reactive machine (site) is made of four lists:

- The first one contains the executing scripts.
- The second one contains the waiting scripts.
- The third one contains scripts whose execution is finished for the current instant.
- The last list is the event environment.

We also need a boolean to indicate if the fixed point is reached (`eoi`). A site is thus coded as:

```
(define site
   (list
      (list) (list) (list) (list)
       eoi))
```

The behaviour of the reactive machine is extremely simple: it executes the function `one_step` which defines one instant of DSL, up to the least fixed point. This function returns the new state of the reactive machine. The code of `one_step` is:

```
(define (one_step site)
 (when (event_generated site)
  (set! site (wakeup_waiting_script site)))
```

```
(if (eoi site)
 site
 (let ((res (sem_action
      (first_script site)
      (get_env site)
      (get_eoi site))))

  (if (execution_finished res)
   (one_step
     (list (next_script site)
     (get_waiting_script site)
     (cons (get_script_res res)
           (get_finished_script site))
     (get_env_res res)
     (get_eoi res)))
   (one_step
     (list (next_script site)
     (cons (get_script_res res)
           (get_waiting_script site))
     (get_finished_script site)
     (get_env_res res)
     (get_eoi_res res))))))))
```

First, we check if there are generated events (`event_generated`). If it is the case, we wake-up all the waiting scripts (`wakeup_waiting_script`). Then, we should verify if the fixed point (*eoi*) is already reached. In that case, there is no need to continue; otherwise, we try to execute a new script (`sem_action`). If `execution_finished` returns true, the current script is terminated for the current instant. Otherwise, the script is waiting for an event. In both cases, we pass to the next script after storing the script in the appropriate list.

3.2.2 Functions and Tasks

In DSL, we require that functions are instantaneous, but tasks can take several instants. In the current implementation, the only way for the user to implement a task is to define a Scheme/Bigloo function executed as a native thread.

3.2.3 Translation in Bigloo

There are two possibilities in Scheme/Bigloo to execute a script: either it is translated in Scheme/Bigloo, then compiled, and executed as a standard Scheme/Bigloo program; or it is given as input to a top-level interpretor that analyses the script, translates it in an abstract tree, and runs it using a native Scheme/Bigloo thread. From the script of Section 2.1.5, the DSL to Bigloo translator produces a code which is exactly the same as the one produced by the FunLoft variant in 3.1.7. To execute the script, we drop it into the appropriate reactive machine.

3.3 Benchmarks

In this section, we execute the script defined in Section 2.1.5 with the four variants of DSL and compare the results.

3.3.1 FunLoft

In FunLoft, the function `consume` called by the script to consume the computing resource can be defined by:

```
let consume_intern (n) =
  let x = local ref 0 in
     repeat n do x++
```

Alternatively, one could define `consume` as an extern C function, introduced in FunLoft by:

```
extern "C" consume : int -> unit
```

The definition of `consume` in C is then:

```
#include "val.h"
value consume (value vn)
{
    long n = val2int (vn);
    long k, x = 0;
    for ( k = 0; k < n; k++ ) x++;
    return val_unit;
}
```

The extern function is much more efficient than the function defined in FunLoft (see below). However, with the extern function, the compiler loses the possibility to detect an error in the C implementation (this is recalled to the user by a message issued at compile time).

The execution machine we use is a dual-core machine[1]. The execution time is obtained with the Unix command `time`.

Results with the FunLoft variant are shown in Figure 3.1; on the left, the (intern) FunLoft definition of `consume` is used, while on the right it is defined in C (extern).

FL	intern	extern
real	2m58.616s	0m32.922s
user	5m47.278s	1m4.652s
sys	0m2.516s	0m0.208s

Figure 3.1: FunLoft variant

3.3.2 SugarCubes

Figure 3.2 shows the results with the SugarCubes variant (the `consume` method is directly implemented in Java). The counter used by `consume` is both implemented as an integer of type `int` and as a long integer of type `long`. The implementation with `int` is:

```
class FUN_consume implements Fun
{
   public void exec(final String arg){
      int len = Integer.valueOf(arg);
      int x = 0;
      for(int i = 0 ; i < len ;i++) x++;
   }
}
```

Figure 3.3 shows the execution time when the JIT of Java is switched off (option `-Xint`).

[1] machine characteristics: Dell Latitude, Linux 2.6.35 processor Intel Core i5, 2.4 GHz, 4GB of memory

SC + JIT	long	int
real	0m30.658	0m1.899s
user	1m7.288s	0m3.692s
sys	0m0.160s	0m0.028

Figure 3.2: SugarCubes variant with JIT

SC - JIT	long	int
real	6m32.884s	3m34.355s
user	19m5.076s	10m25.387s
sys	0m13.617s	0m5.044s

Figure 3.3: SugarCubes variant without JIT

3.3.3 ReactiveML

Figure 3.4 shows the results with the ReactiveML variant (the `consume` method is directly implemented in ReactiveML).

The code of `consume` is:

```
let consume n =
  let x = ref 0 in
  for i = 1 to n do
    x := !x + 1
  done
```

RML	
real	1m27.292s
user	0m39.946s
sys	0m0.584s

Figure 3.4: ReactiveML variant

3.3.4 Scheme/Bigloo

The results for the Scheme/Bigloo variant are shown in Figure 3.5. We have considered the two cases, where the counter is implemented as an integer and as a long.

Scheme	long	int
real	47m34.820s	4m40.714s
user	76m53.744s	8m0.614s
sys	4m9.536s	0m4.572s

Figure 3.5: Scheme/Bigloo variant

3.3.5 Interpretation

With the FunLoft, SugarCubes and Scheme/Bigloo variants, we see that the user time is (more or less) twice the real time, which shows that the two cores are running simultaneously at full speed. The two instances of `consume` are indeed executed in real parallelism by the two cores. The ReactiveML variant does not use the two cores in an optimal way; it seems to be slowed down by the presence of JoCaml which introduces a communication overhead.

The efficiency of the SugarCubes variant heavily depends on the JIT of Java. We see also that it depends on the use of integers instead of longs. Note that in the FunLoft variant, integers are coded as `long long` integers of C.

The dependence on the choice between integers and longs is also clear in the Scheme/Bigloo variant. Note that in this variant, no optimization is performed in the reactive engine; optimization was not the focus and the efficiency of this variant could be clearly improved in this respect.

Part II

CRL

Chapter 4

The Core Reactive Language CRL

In this chapter we focus on a core reactive language (CRL) without memory. We choose a deterministic parallel operator with a small-step semantics. In CRL which is a variation of the left-right merge which is already present in SugarCubes or, in a different form, in [7].

As in DSL, all CRL programs are reactive, thanks to a clear separation between the loop construct `loop` s `end` and the iteration construct `repeat` exp `do` s `end`, and to our semantics for loops, which requires them to yield the control at each iteration of their body. This chapter and the next one will be more formal than the previous ones. We will study the small-step semantics of CRL in depth and give the formal proof of reactivity, given that the operator $\upharpoonright$ has not been studied previously. In the proofs, we shall only consider the most interesting cases, referring to [15] for complete proofs.

The rest of this chapter is organized as follows. Sections 4.1 and 4.2 present the syntax and the semantics of the language CRL. Section 4.3 is devoted to proving reactivity of CRL scripts.

4.1 Syntax

In this section we introduce the syntax of CRL. Let Val be a set of values, ranged over by v, v', Var a set of variables, ranged over by x, y, z, and $Events$ a set of events, ranged over by ev, ev'. A fixed valuation function $V : Var \to Val$ for open terms is used to evaluate open expressions and execute open terms. There are two syntatic categories: expressions and scripts. Expression execution finishes immediately, but script execution can

take more than one instant or never terminate (a *looping* script).

4.1.1 Expressions

An expression $exp \in Exp$ is either a basic value, or a variable, or the value returned by an operator. Letting $\overrightarrow{exp}$ denote a tuple of expressions $exp_1, \ldots, exp_n$, the syntax of expressions is:

$$exp \in Exp ::= v \mid x \mid op(\overrightarrow{exp})$$

We note that the evaluation of an operator $op(\overrightarrow{exp})$ is instantaneous, and therefore so is the evaluation of an expression exp, denoted by $exp \rightsquigarrow v$, which is formally defined by the three rules:

$$\frac{}{v \rightsquigarrow v} \qquad \frac{V(x) = v}{x \rightsquigarrow v} \qquad \frac{\forall i \in \{1, \ldots, n\} \,.\, exp_i \rightsquigarrow v_i \qquad op(v_1, \ldots, v_n) = v}{op(\overrightarrow{exp}) \rightsquigarrow v}$$

In the rest of this thesis we will not make any difference between operators and functions.

4.1.2 Scripts

We now present the syntax of *CRL* scripts. Alongside with typical sequential operators, *CRL* like *DSL* includes four operators that are commonly found in reactive languages, `cooperate`, `generate` ev, `await` ev and `do` s `watching` ev, as well as a binary *asymmetric parallel operator*, denoted by $\upharpoonright$, which performs a deterministic scheduling on its components. This operator is very close to that used in [8] and, earlier on, in the implementation of *SugarCubes* [28]. However, while in [8] and [28] each parallel component was executing as long as possible, our operator $\upharpoonright$ implements a form of *prioritized scheduling*, where the first component yields the control only when terminating or suspending (*late cooperation*), while the second yields it as soon as it generates an event that unblocks the first component (*early cooperation*).

The syntax of scripts is defined by:

$$
\begin{array}{rl}
s \in \textbf{Scripts} ::= & \texttt{nothing} \\
& \mid s; s \\
& \mid s \nmid s \\
& \mid \texttt{cooperate} \\
& \mid \texttt{generate}\ ev \\
& \mid \texttt{await}\ ev \\
& \mid \texttt{do}\ s\ \texttt{watching}\ ev \\
& \mid \texttt{loop}\ s\ \texttt{end} \\
& \mid \texttt{repeat}\ exp\ \texttt{do}\ s\ \texttt{end} \\
& \mid \texttt{if}\ exp\ \texttt{then}\ s\ \texttt{else}\ s\ \texttt{end}
\end{array}
$$

4.2 Semantics

This section presents the operational semantics of CRL. Programs proceed through a succession of instants, transforming sets of events. There are two transition relations, both defined on *configurations* of the form $\langle s, E\rangle$, where s is a script and $E \subseteq Events$ is an *event environment*, i.e. a set of events that are currently *present*.

Let us first give the general idea of these two transition relations:

1. The *small-step transition relation* describes the step-by-step execution of a configuration within an instant. The general format of a transition is:

   $$\langle s, E\rangle \rightarrow \langle s', E'\rangle$$

   where:

   - s is the script to execute;
   - E is the starting event environment;
   - s' is the rewritten script (*residual script*);
   - E' is the resulting event environment: E' coincides with E if the transition does not generate any new event. Otherwise $E' = E \cup \{ev\}$, where ev is the new generated event.

2. The *tick transition relation* describes the passage from one instant to the next, and applies only to suspended configurations. A transition of this kind has always the form:

$$\langle s, E\rangle \hookrightarrow \langle [s]_E, \emptyset\rangle$$

where the resulting event environment is empty and $[s]_E$ is a "reconditioning" of script s for the next instant, possibly allowing it to resume execution at the next instant even without the help of new events from the environment (e.g. *cooperate*).

Before formally defining $\langle s, E\rangle \rightarrow \langle s', E'\rangle$ and $\langle s, E\rangle \hookrightarrow \langle [s]_E, \emptyset\rangle$, we introduce the *suspension predicate* $\langle s, E\rangle\ddagger$, which holds when s is suspended in the event environment E, namely when s waits for some event not contained in E, or when s deliberately yields the control for the current instant by means of a `cooperate` instruction. The rules defining the predicate $\ddagger$ are given below:

$$\langle \mathit{cooperate}, E\rangle\ddagger \quad (\mathit{coop}) \qquad \frac{\mathit{ev} \notin E}{\langle \texttt{await}\ \mathit{ev}, E\rangle\ddagger}\ (\mathit{wait}_s)$$

$$\frac{\langle s, E\rangle\ddagger}{\langle \texttt{do}\ s\ \texttt{watching}\ \mathit{ev}, E\rangle\ddagger}\ (\mathit{watch}_s) \qquad \frac{\langle s_1, E\rangle\ddagger}{\langle s_1; s_2, E\rangle\ddagger}\ (\mathit{seq}_s)$$

$$\frac{\langle s_1, E\rangle\ddagger \qquad \langle s_2, E\rangle\ddagger}{\langle s_1 \nmid s_2, E\rangle\ddagger}\ (\mathit{par}_s)$$

The reconditioning function $[s]_E$ prepares s for the next instant: it erases all guarding `cooperate` instructions, as well as all guarding $\texttt{do}\ s'\ \texttt{watching}\ \mathit{ev}$ instructions whose time-out event ev is in E (i.e. has been generated). The rules of the reconditioning function are given below:

$$[\texttt{cooperate}]_E = \texttt{nothing} \qquad [\texttt{await}\ \mathit{ev}]_E = \texttt{await}\ \mathit{ev}$$

$$[\texttt{do}\ s\ \texttt{watching}\ \mathit{ev}]_E = \begin{cases} \texttt{nothing} & \mathit{if} \quad \mathit{ev} \in E \\ \texttt{do}\ [s]_E\ \texttt{watching}\ \mathit{ev} & \mathit{otherwise} \end{cases}$$

$$[s_1; s_2]_E = [s_1]_E\,; s_2 \quad [s_1 \nmid s_2]_E = [s_1]_E \nmid [s_2]_E$$

The tick transition relation is then defined by the unique rule:

$$\frac{\langle s, E\rangle \ddagger}{\langle s, E\rangle \hookrightarrow \langle [s]_E, \emptyset\rangle} \ (tick)$$

The small-step transition relation is defined by the following rules. We note that to deal with the possibility for a script to get suspended, we chose the smallest possible grain of parallelism; for example, a full step is devoted to the evaluation of the boolean expression of a test, while the execution of the chosen branch is left for a future step. In this way, the execution of the test can progress *even if the chosen branch is suspended*. In other words, we chose to make each step as elementary as possible.

We assume scripts are well-typed with respect to a standard type system that ensures that in `if` *exp* `then` s_1 `else` s_2 `end` and `repeat` *exp* `do` s `end` the expression *exp* evaluates respectively to a boolean and to an integer bigger or equal to one.

We give next the semantics of *CRL* scripts. We comment on the most interesting transition rules.

Sequence

$$\frac{\langle s_1, E\rangle \to \langle s_1', E'\rangle}{\langle s_1; s_2, E\rangle \to \langle s_1'; s_2, E'\rangle} \tag{4.1}$$

$$\langle \texttt{nothing}\,; s, E\rangle \to \langle s, E\rangle \tag{4.2}$$

Parallel

$$\frac{\langle s_1, E\rangle \to \langle s_1', E'\rangle}{\langle s_1 \nmid s_2, E\rangle \to \langle s_1' \nmid s_2, E'\rangle} \tag{4.3}$$

The execution of a parallel script always starts with its left branch.

$$\langle \texttt{nothing} \nmid s, E\rangle \to \langle s, E\rangle \tag{4.4}$$

Once the left branch is over, the script reduces to its right branch.

$$\frac{\langle s_1, E\rangle \ddagger \quad \langle s_2, E\rangle \to \langle s_2', E'\rangle}{\langle s_1 \nmid s_2, E\rangle \to \langle s_1 \nmid s_2', E'\rangle} \tag{4.5}$$

If the left branch is suspended, then the right branch executes until unblocking the left branch.

$$\frac{\langle s, E\rangle \ddagger}{\langle s \nmid \texttt{nothing}, E\rangle \to \langle s, E\rangle} \tag{4.6}$$

Thus *early cooperation* is required in the right branch. To avoid non-determinism, a terminated right branch can only be eliminated if the left branch is suspended.

Generate

$$\langle \texttt{generate}\ ev, E\rangle \to \langle \texttt{nothing}, E \cup \{ev\}\rangle \tag{4.7}$$

Await

$$\frac{ev \in E}{\langle \texttt{await}\ ev, E\rangle \to \langle \texttt{nothing}, E\rangle}\ (wait) \tag{4.8}$$

Watching

$$\frac{\langle s, E\rangle \to \langle s', E'\rangle}{\langle \texttt{do}\ s\ \texttt{watching}\ ev, E\rangle \to \langle \texttt{do}\ s'\ \texttt{watching}\ ev, E'\rangle} \tag{4.9}$$

A $\texttt{do}\ s\ \texttt{watching}\ ev$ executes its body until termination or suspension, reducing to `nothing` when its body terminates, and getting processed by the reconditioning function when its body suspends.

$$\langle \texttt{do nothing watching}\ ev, E\rangle \to \langle \texttt{nothing}, E\rangle \tag{4.10}$$

We note that the semantics of watching statement is not the same as in *DSL*, where the watching body is killed as soon as the event is generated.

Loop

$$\langle \texttt{loop}\ s\ \texttt{end}, E\rangle \to \langle (s \nmid \texttt{cooperate}); \texttt{loop}\ s\ \texttt{end}, E\rangle \tag{4.11}$$

A `loop` s `end` executes its body cyclically: a `cooperate` instruction is systematically added in parallel to its body to avoid *instantaneous loops*, i.e. divergence within an instant[1].

Repeat

$$\frac{exp \rightsquigarrow n \qquad n \geq 1}{\langle \texttt{repeat}\ exp\ \texttt{do}\ s\ \texttt{end}, E\rangle \to \langle \underbrace{s; \ldots; s}_{n\ times}, E\rangle} \tag{4.12}$$

$$\frac{exp \rightsquigarrow n \qquad n < 1}{\langle \texttt{repeat}\ exp\ \texttt{do}\ s\ \texttt{end}, E\rangle \to \langle \texttt{nothing}, E\rangle} \tag{4.13}$$

If

$$\frac{exp \rightsquigarrow tt}{\langle \texttt{if}\ exp\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{end}, E\rangle \to \langle s_1, E\rangle} \tag{4.14}$$

$$\frac{exp \rightsquigarrow ff}{\langle \texttt{if}\ exp\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{end}, E\rangle \to \langle s_2, E\rangle} \tag{4.15}$$

The small-step transition relation satisfies two simple properties: determinism and incremental production of events throughout an instant.

Corollary 4.2.1 (Determinism)

Let $s \in$ **Scripts** *and* $E \subseteq$ *Events. Then:*

$$s \neq \texttt{nothing} \quad \Rightarrow \quad \textit{either}\ \langle s, E\rangle \ddagger \ \ \textit{or}\ \ \exists!\, s', E'.\ \langle s, E\rangle \to \langle s', E'\rangle$$

[1] In general, we shall call "instantaneous" any property that holds within an instant.

Proof By inspecting the suspension and transition rules, it is immediate to see that at most one transition rule applies to each configuration $\langle s, E\rangle$. □

Lemma 4.2.1 (Event persistence)

Let $s \in$ **Scripts** *and* $E \subseteq Events$. *Then:* $\quad \langle s, E\rangle \rightarrow \langle s', E'\rangle \;\Rightarrow\; E \subseteq E'$

Proof Straightforward, since the only transition rule that changes the event environment E is the rule for `generate` ev, which adds the event ev to E. □

We define now the notion of *instantaneous convergence*, which is at the basis of the reactivity property of CRL programs. Let us first introduce some notation.

The *timed multi-step transition relation* $\langle s, E\rangle \Rightarrow_n \langle s', E'\rangle$ is defined by:

$\langle s, E\rangle \Rightarrow_0 \langle s, E\rangle$

$\langle s, E\rangle \rightarrow \langle s', E'\rangle \;\wedge\; \langle s', E'\rangle \Rightarrow_n \langle s'', E''\rangle \quad \Rightarrow \quad \langle s, E\rangle \Rightarrow_{n+1} \langle s'', E''\rangle$

Then the *multi-step transition relation* $\langle s, E\rangle \Rightarrow \langle s', E'\rangle$ is given by:

$$\langle s, E\rangle \Rightarrow \langle s', E'\rangle \;\Leftrightarrow\; \exists\, n \,.\, \langle s, E\rangle \Rightarrow_n \langle s', E'\rangle$$

The *immediate convergence* predicate is defined by:

$$\langle s, E\rangle \overset{\ddagger}{\Downarrow} \;\Leftrightarrow\; \langle s, E\rangle \ddagger \;\vee\; s = \texttt{nothing}$$

We define now the relation and the predicate of *instantaneous convergence*:

Definition 4.2.1 (Instantaneous convergence)

$$\begin{array}{lll} \langle s, E\rangle \Downarrow \langle s', E'\rangle & \text{if} & \langle s, E\rangle \Rightarrow \langle s', E'\rangle \;\wedge\; \langle s', E'\rangle \overset{\ddagger}{\Downarrow} \\ \langle s, E\rangle \Downarrow & \text{if} & \exists s', E' \,.\, \langle s, E\rangle \Downarrow \langle s', E'\rangle \end{array}$$

The relation and predicate of *instantaneous termination* are defined similarly:

Definition 4.2.2 (Instantaneous termination)

$$\begin{array}{lll} \langle s, E\rangle \;\Downarrow\!\!\!\!\!\!\Downarrow\; E' & \text{if} & \langle s, E\rangle \Downarrow \langle \texttt{nothing}, E'\rangle \\ \langle s, E\rangle \;\Downarrow\!\!\!\!\!\!\Downarrow & \text{if} & \exists\, E' \,.\, \langle s, E\rangle \;\Downarrow\!\!\!\!\!\!\Downarrow\; E' \end{array}$$

The relation $\langle s, E\rangle \Downarrow \langle s', E'\rangle$ defines the overall effect of the program s within an instant, starting with the set of events E.

As an immediate corollary of Proposition 4.2.1, the relation $\Downarrow$ is a function.

The *timed* versions of $\langle s, E\rangle \Downarrow \langle s', E'\rangle$, $\langle s, E\rangle \Downarrow$, $\langle s, E\rangle \mathrel{\overset{\mathrm{T}}{\Downarrow}} \langle s', E'\rangle$ and $\langle s, E\rangle \mathrel{\overset{\mathrm{T}}{\Downarrow}}$ are defined in the expected way.

In the next section we prove an important property of CRL, namely that every configuration $\langle s, E\rangle$ instantaneously converges. This is true in particular for initial configurations, where $E = \emptyset$. As for DSL, this property is called *reactivity*. Since CRL has a new parallel operator ($\nmid$) which has not been studied previously, we give a formal proof of reactivity.

4.3 Reactivity

In this section we prove the reactivity of CRL programs. In fact, we shall prove a stronger property than reactivity, namely that every configuration $\langle s, E\rangle$ instantaneously converges in a number of steps which is bounded by the *instantaneous size* of s, denoted by $\mathit{size}(s)$. The intuition for $\mathit{size}(s)$ is that the portion of s that sequentially follows a `cooperate` instruction should not be taken into account, as it will not be executed in the current instant. Moreover, if s is a loop, $\mathit{size}(s)$ should cover a single iteration of its body.

To formally define the function $\mathit{size}(s)$, we first introduce an auxiliary function $\mathit{dsize}(s)$ (where "d" stands for "decorated") that assigns to each program an element of $(\mathbf{Nat} \times \mathbf{Bool})$. Then $\mathit{size}(s)$ will be the first projection of $\mathit{dsize}(s)$. Intuitively, if $\mathit{dsize}(s) = (n, b)$, then n is an upper bound for the number of steps that s can execute within an instant; and b is tt or ff depending on whether or not a `cooperate` instruction is reached within the instant. For conciseness, we let $n^\wedge$ stand for (n, tt), n stand for (n, ff), and n° range over $\{n^\wedge, n\}$.

The difference between $n^\wedge$ and n will essentially show when computing the size of a sequential composition: if the decorated size of the first component has the form $n^\wedge$, then a `cooperate` has been met and the counting will stop; if it has the form n, then n will be added to the decorated size of the second component.

Definition 4.3.1 (Instantaneous size)

The function size : **Scripts** $\to$ **Nat** *is defined by:*

$$size(s) = n \quad if \quad (dsize(s) = n \ \vee \ dsize(s) = n^{\wedge}).$$

where the function dsize : **Scripts** $\to$ (**Nat** $\times$ **Bool**) *is given inductively by:*

$$dsize(\texttt{nothing}) = 0 \qquad\qquad dsize(\texttt{cooperate}) = 0^{\wedge}$$

$$dsize(\texttt{generate}\ ev) = dsize(\texttt{await}\ ev) = 1$$

$$dsize(s_1; s_2) = \begin{cases} {n_1}^{\wedge} & if \ dsize(s_1) = {n_1}^{\wedge} \\ (1 + n_1 + n_2)^{\circ} & if \ dsize(s_1) = n_1 \ \wedge \ dsize(s_2) = {n_2}^{\circ} \end{cases}$$

$$dsize(s_1 \nmid s_2) = \begin{cases} (1 + n_1 + n_2)^{\wedge} & if \ dsize(s_1) = {n_1}^{\wedge} \ \wedge \ dsize(s_2) = n_2 \\ (1 + n_1 + n_2)^{\wedge} & if \ dsize(s_1) = n_1 \ \wedge \ dsize(s_2) = {n_2}^{\wedge} \\ (1 + n_1 + n_2)^{\circ} & if \ dsize(s_1) = {n_1}^{\circ} \ \wedge \ dsize(s_2) = {n_2}^{\circ} \end{cases}$$

$$dsize(\texttt{repeat}\ exp\ \texttt{do}\ s\ \texttt{end}) = (m + (m \times n))^{\circ} \qquad if \ dsize(s) = n^{\circ} \ \wedge \ exp \rightsquigarrow m$$

$$dsize(\texttt{loop}\ s\ \texttt{end}) = (2 + n)^{\wedge} \qquad if \quad dsize(s) = n^{\circ}$$

$$dsize(\texttt{do}\ s\ \texttt{watching}\ ev) = (1 + n)^{\circ} \quad if \ dsize(s) = n^{\circ}$$

$$dsize(\texttt{if}\ exp\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{end}) = \begin{cases} 1)(1 + max\{n_1, n_2\})^{\wedge}, \\ \quad if \ dsize(s_i) = {n_i}^{\wedge}, i = 1, 2 \\ 2)(1 + max\{n_1, n_2\}), \\ \quad if \ \ for \ \ i \neq j \ dsize(s_i) = n_i \wedge \\ \qquad\qquad dsize(s_j) = {n_j}^{\circ} \end{cases}$$

The following lemma establishes that $size(s)$ decreases at each step of a small-step execution:

Lemma 4.3.1 (Size reduction within an instant)

$$\forall s\, \forall E \quad (\, \langle s, E\rangle \rightarrow \langle s', E'\rangle \;\Rightarrow\; size(s') < size(s)\,)$$

Proof The use of *dsize* in the definition of $size(s)$ makes the proof not entirely straightforward. We prove by induction on the structure of s the following stronger statement, where $dsize(s)\lceil_2$ indicates the second projection of $dsize(s)$:

$$\langle s, E\rangle \rightarrow \langle s', E'\rangle \;\Rightarrow\; \begin{cases} i) & size(s') < size(s) \\ ii) & dsize(s)\lceil_2 = tt \;\Rightarrow\; dsize(s')\lceil_2 = tt \end{cases}$$

Here we are only going to consider the most interesting cases:

- *Basic cases:*
  - $s = \texttt{generate}\, ev$. In this case, the transition is inferred by Rule (4.7) and we have:

    $$\langle \texttt{generate}\, ev, E\rangle \rightarrow \langle \texttt{nothing}, E \cup \{ev\}\rangle$$

    Since $size(\texttt{nothing}) = 0 < 1 = size(\texttt{generate}\, ev)$, we may conclude.

- *Inductive cases:*
  - $s = \texttt{loop}\, s_0\, \texttt{end}$. In this case the transition is inferred by Rule (4.11):

    $$\langle \texttt{loop}\, s\, \texttt{end}_0, E\rangle \rightarrow \langle s_0 \fatsemi \texttt{cooperate}; \texttt{loop}\, s\, \texttt{end}_0, E\rangle$$

    Since $dsize(s_0 \fatsemi \texttt{cooperate}) = (1 + size(s_0))^\wedge$, we also have:

    $$dsize(s_0 \fatsemi \texttt{cooperate}; \texttt{loop}\, s\, \texttt{end}_0) = (1 + size(s_0))^\wedge$$

    and thus $size(s') = 1 + size(s_0) < 2 + size(s_0) = size(s)$.

    Moreover, it is immediate to see that Clause *ii*) holds too, since we have both $dsize(s)\lceil_2 = tt$ and $dsize(s')\lceil_2 = tt$.

– $s = \texttt{repeat}\ exp\ \texttt{do}\ s_0\ \texttt{end}$. In this case the expression exp evaluates to some natural number $m \geq 1$ and the transition is inferred by Rule (4.12):

$$\frac{exp \rightsquigarrow m}{\langle \texttt{repeat}\ exp\ \texttt{do}\ s\ \texttt{end}_0, E\rangle \rightarrow \langle \overbrace{s_0; \ldots ; s_0}^{m\ times}, E\rangle}$$

Then, if $dsize(s_0) = n_0^{\wedge}$, we have $size(s') = n_0$. If instead $dsize(s_0) = n_0$, then $size(s') = (m-1)+(m\times n_0) < m+(m\times n_0) = size(s)$.

Moreover, since by definition $dsize(\texttt{repeat}\ exp\ \texttt{do}\ s_0\ \texttt{end})\restriction_2 = tt$ if and only if $dsize(s_0)\restriction_2 = tt$, Clause ii) holds too.

– $s = s_1 \nmid s_2$. Then the transition is inferred by one of the four Rules (4.3-4.6). We examine the four cases in turn.

1. Rule (4.3). In this case the component s_1 moves:

$$\frac{\langle s_1, E\rangle \rightarrow \langle s_1', E'\rangle}{\langle s_1 \nmid s_2, E\rangle \rightarrow \langle s_1' \nmid s_2, E'\rangle}$$

Here $s' = s_1' \nmid s_2$, and by induction $size(s_1') < size(s_1)$, hence $size(s') = 1 + size(s_1') + size(s_2) < 1 + size(s_1) + size(s_2) = size(s)$. As for Clause ii), note that $dsize(s_1 \nmid s_2)\restriction_2 = tt$ implies $dsize(s_i)\restriction_2 = tt$ for at least one $i \in \{1, 2\}$. If $dsize(s_2)\restriction_2 = tt$, then we obtain $dsize(s_1' \nmid s_2)\restriction_2 = tt$ by definition. If $dsize(s_1)\restriction_2 = tt$, then $dsize(s_1')\restriction_2 = tt$ follows by induction, whence we deduce again $dsize(s_1' \nmid s_2)\restriction_2 = tt$.

2. Rule (4.4). In this case $s_1 = \texttt{nothing}$ and we have:

$$\langle s_1 \nmid s_2, E\rangle \rightarrow \langle s_2, E\rangle$$

Here $s' = s_2$ and $size(s') = size(s_2) < 1 + size(s_2) = size(s)$. Moreover, it is immediate to see that Clause ii) holds too, since $dsize(s)\restriction_2 = tt \Leftrightarrow dsize(s')\restriction_2 = tt$.

3. Rule (4.5). Here $\langle s_1, E\rangle\ddagger$ and the component s_2 moves:

$$\frac{\langle s_1, E\rangle \ddagger \quad \langle s_2, E\rangle \rightarrow \langle s_2', E'\rangle}{\langle s_1 \nmid s_2, E\rangle \rightarrow \langle s_1 \nmid s_2', E'\rangle}$$

In this case we apply induction on s_2 and the reasoning is completely symmetric to that for Rule (4.3) above.

4. Rule (4.6). In this case $s_2 = \texttt{nothing}$ and we have:

$$\frac{\langle s_1, E\rangle\ddagger}{\langle s_1 \upharpoonright \texttt{nothing}, E\rangle \rightarrow \langle s_1, E\rangle}$$

In this case, the reasoning is symmetric to that for Rule (4.4) above.

□

As a consequence of Lemma (4.3.1), when a script executes n steps, its size will decrease by at least n, as expressed by the following:

Corollary 4.3.1 *Let* $s \in$ **Scripts**. *For any* $E \subseteq$ *Events:*

$$\langle s, E\rangle \Rightarrow_n \langle s', E'\rangle \quad \Rightarrow \quad n \leq size(s) - size(s')$$

Proof Immediate, by induction on the length n of the computation. □

We are now ready to prove our main result, namely that every program s instantaneously converges in a number of steps that is bounded by $size(s)$.

Theorem 4.3.2 (Script reactivity) $\forall s, \forall E \quad (\exists n \leq size(s) \ \ \langle s, E\rangle \Downarrow_n)$

Proof By simultaneous induction on the structure and on the size of s. The basic cases are the same for both inductions. In the inductive cases, the induction will always be on the structure except for the case $s = s_1 \upharpoonright s_2$, where it will be on the size.

We consider only the main cases:

- *Basic cases* (s *has no proper subterms and* $size(s) \in \{0, 1\}$)

  1. $s = \texttt{cooperate}$. In this case $\langle s, E\rangle\ddagger$, and thus we get again the result for $n = 0$, since by definition $\langle s, E\rangle\ddagger \Rightarrow (\langle s, E\rangle \Downarrow_0 \langle s, E\rangle)$. Here we have $n = 0 \leq size(\texttt{cooperate}) = 0$.

2. $s = \texttt{generate}\ ev$. In this case, by Rule (4.7) there is a transition:

$$\langle \texttt{generate}\ ev, E\rangle \rightarrow \langle \texttt{nothing}, E \cup \{ev\}\rangle$$

Since $\langle \texttt{nothing}, E \cup \{ev\}\rangle \Downarrow_0 \langle \texttt{nothing}, E \cup \{ev\}\rangle$, by definition of $\Downarrow_n$ it follows that $\langle \texttt{generate}\ ev, E\rangle \Downarrow_1 \langle \texttt{nothing}, E \cup \{ev\}\rangle$.

Here we have $n = 1 \leq \mathit{size}(\texttt{generate}\ ev) = 1$.

- *Inductive cases* (s *has proper subterms and* $\mathit{size}(s) \geq 1$)

1. $s = s_1 \nmid s_2$. By induction $\exists\, n_1 \leq \mathit{size}(s_1)\,.\ \langle s_1, E\rangle \Downarrow_{n_1} \langle s_1', E_1\rangle$. We distinguish two cases:
   - If $s_1' = \texttt{nothing}$, then by applying Rule (4.4) we obtain:

$$\langle s_1' \nmid s_2, E_1\rangle \rightarrow \langle s_2, E_1\rangle$$

By induction $\exists\, n_2 \leq \mathit{size}(s_2)\,.\ \langle s_2, E_1\rangle \Downarrow_{n_2} \langle s_2', E_2\rangle$. Hence there is a transition sequence:

$$\langle s_1 \nmid s_2, E\rangle \Rightarrow_{n_1} \langle s_1' \nmid s_2, E_1\rangle \rightarrow \langle s_2, E_1\rangle \Downarrow_{n_2} \langle s_2', E_2\rangle$$

We conclude that $\langle s_1 \nmid s_2, E\rangle \Downarrow_n \langle s_2', E_2\rangle$, with $n = n_1 + 1 + n_2$.

Since $n_i \leq \mathit{size}(s_i)$ for $i = 1, 2$, we have that $n = n_1 + 1 + n_2 \leq \mathit{size}(s_1) + 1 + \mathit{size}(s_2) = \mathit{size}(s_1 \nmid s_2)$.

Note that this case is entirely similar to sequential composition $s = s_1\,;s_2$ when $s_1' = \texttt{nothing}$.
   - If $\langle s_1', E_1\rangle\ddagger$, there are two subcases:
     (a) $\langle s_2, E_1\rangle\ddagger$, in which case we have $\langle s_1' \nmid s_2, E_1\rangle\ddagger$ by Rule (par_s) of the suspension predicate, and thus $\langle s_1 \nmid s_2, E\rangle \Downarrow_{n_1} \langle s_1' \nmid s_2, E_1\rangle$. Here $n = n_1 \leq \mathit{size}(s_1) \leq \mathit{size}(s_1 \nmid s_2)$.
     (b) $\neg\langle s_2, E_1\rangle\ddagger$. There are two possibilities:
     - If $s_2 = \texttt{nothing}$, then by Rule (4.6) we have:

$$\langle s_1' \nmid s_2, E_1\rangle \rightarrow \langle s_1', E_1\rangle$$

Hence there is a transition sequence:

$$\langle s_1 \nmid s_2, E\rangle \Rightarrow_{n_1} \langle s_1' \nmid s_2, E_1\rangle \rightarrow \langle s_1', E_1\rangle\ddagger$$

and thus we have $\langle s_1 \nmid s_2, E\rangle \Downarrow_n \langle s_1', E_1\rangle$, with $n = n_1 + 1$.

Since $n_1 \leq \mathit{size}(s_1)$, we have $n = n_1 + 1 \leq \mathit{size}(s_1) + 1 + \mathit{size}(s_2) = \mathit{size}(s_1 \nmid s_2)$.

- If $s_2 \neq \texttt{nothing}$, then we may apply induction to get $\exists\, n_2 \leq size(s_2)\,.\, \langle s_2, E_1\rangle \Downarrow_{n_2} \langle s_2', E_2\rangle$. Since $\neg\langle s_2, E_1\rangle\ddagger$, we know that $s_2' \neq s_2$ and therefore $size(s_2') < size(s_2)$. This implies that $size(s_1' \restriction s_2') < size(s_1 \restriction s_2)$, thus we may now apply induction on the size of $s_1' \restriction s_2'$ to obtain:

$$\exists\, n' \leq size(s_1' \restriction s_2')\,.\, \langle s_1' \restriction s_2', E_2\rangle \Downarrow_{n'} \langle s', E'\rangle.$$

In conclusion we have:

$$\langle s_1 \restriction s_2, E\rangle \Rightarrow_{n_1} \langle s_1' \restriction s_2, E_1\rangle \Rightarrow_{n_2} \langle s_1' \restriction s_2', E_2\rangle \Downarrow_{n'} \langle s', E'\rangle$$

and thus $\langle s_1 \restriction s_2, E\rangle \Downarrow_n \langle s', E'\rangle$, with $n = n_1 + n_2 + n'$. By Corollary 4.3.1 $(n_1+n_2) \leq size(s_1 \restriction s_2) - size(s_1' \restriction s_2')$. By induction $n' \leq size(s_1' \restriction s_2')$. Then $n = (n_1+n_2)+n' \leq (size(s_1 \restriction s_2) - size(s_1' \restriction s_2')) + size(s_1' \restriction s_2') = size(s_1 \restriction s_2)$.

2. $s = \texttt{repeat}\ exp\ \texttt{do}\ s_1\ \texttt{end}$. If $exp \rightsquigarrow m$ for some $m \geq 1$, Rule (4.12) gives:

$$\langle \texttt{repeat}\ exp\ \texttt{do}\ s_1\ \texttt{end}, E\rangle \rightarrow \langle \overbrace{s_1; \ldots; s_1}^{m\ times}, E\rangle$$

By induction $\exists\, n_1 \leq size(s_1)\,.\, \langle s_1, E\rangle \Downarrow_{n_1} \langle s_1', E_1\rangle$. We distinguish two cases.

(a) If $m = 1$, we can immediately conclude that $\langle s, E\rangle \Downarrow_{1+n_1} \langle s_1', E_1\rangle$.

(b) If $m > 1$, let $s_2 = \overbrace{s_1; \ldots; s_1}^{m-1\ times}$. From $\langle s_1, E\rangle \Downarrow_{n_1} \langle s_1', E_1\rangle$, by Rule (4.1) we deduce $\langle s_1; s_2, E\rangle \Rightarrow_{n_1} \langle s_1'; s_2, E_1\rangle$. There are two possibilities:

- $\langle s_1', E_1\rangle\ddagger$. Then also $\langle s_1'; s_2, E_1\rangle\ddagger$ by Rule (4.2) of the suspension predicate. Thus $\langle s_1; s_2, E\rangle \Downarrow_{n_1} \langle s_1'; s_2, E_1\rangle\ddagger$. Whence we deduce that $\langle s, E\rangle \Downarrow_{1+n_1} \langle s_1'; s_2, E_1\rangle$.
- $s_1' = \texttt{nothing}$. In this case there exist $n_2, \ldots, n_m$ and $E_2, \ldots, E_m$ such that:

$$\begin{array}{lll} \langle s_1; s_2, E\rangle & \Rightarrow_{n_1} & \langle \texttt{nothing}; s_2, E_1\rangle \rightarrow \langle s_2, E_1\rangle \\ & \Rightarrow_{n_2} & \langle \texttt{nothing}; \overbrace{s_1; \ldots; s_1}^{m-2\ times}, E_1\rangle \rightarrow \langle \overbrace{s_1; \ldots; s_1}^{m-2\ times}, E_2\rangle \\ & \vdots & \\ & \Rightarrow_{n_m} & \langle \texttt{nothing}, E_m\rangle \end{array}$$

By induction $n_k \leq \mathit{size}(s_1)$ for each $k = 1, \ldots, m$, thus we may conclude that $\langle s_1; s_2, E\rangle \Downarrow_{n'}$ with $n' = (m-1) + n_1 + \cdots + n_m \leq (m-1) + (m \times \mathit{size}(s_1))$. Therefore we may conclude that $\langle s, E\rangle \Downarrow_n$ with $n = m + n_1 + \cdots + n_m \leq m + (m \times \mathit{size}(s_1))$.

3. $s = \texttt{loop}\ s_1\ \texttt{end}$. By Rule (4.11) we have

   $(*)$ $\langle \texttt{loop}\ s_1\ \texttt{end}, E\rangle \to \langle s_1 \fatsemi \texttt{cooperate}; \texttt{loop}\ s_1\ \texttt{end}, E\rangle$

   By induction $\exists\, n_1 \leq \mathit{size}(s_1)\,.\ \langle s_1, E\rangle \Downarrow_{n_1} \langle s_1', E_1\rangle$. This means that $\langle s_1, E\rangle \Rightarrow_{n_1} \langle s_1', E_1\rangle$. Then, by repeated applications of the Rules (4.3) and (4.1), we get:

   $(**)$ $\langle s_1 \fatsemi \texttt{cooperate}; \texttt{loop}\ s_1\ \texttt{end}, E\rangle \Rightarrow_{n_1} \langle s_1' \fatsemi \texttt{cooperate}; \texttt{loop}\ s_1\ \texttt{end}, E_1\rangle$

   There are now two possibilities:

   (a) If $\langle s_1', E_1\rangle\ddagger$, then by the Rules (par_s) and (seq_s) of the suspension predicate also $\langle s_1' \fatsemi \texttt{cooperate}; \texttt{loop}\ s_1\ \texttt{end}, E_1\rangle\ddagger$, and thus we may conclude that $\langle \texttt{loop}\ s_1\ \texttt{end}, E\rangle \Downarrow_{1+n_1} \langle s_1' \fatsemi \texttt{cooperate}; \texttt{loop}\ s_1\ \texttt{end}, E_1\rangle$.
   Since $n_1 \leq \mathit{size}(s_1)$, we get $n = 1 + n_1 \leq 1 + \mathit{size}(s_1) = \mathit{size}(s)$.

   (b) If $s_1' = \texttt{nothing}$, then by Rule (4.4) we have:

   $$\langle s_1' \fatsemi \texttt{cooperate}, E_1\rangle \to \langle \texttt{cooperate}, E_1\rangle$$

   whence by Rule (4.1):

   $(***)$ $\langle s_1' \fatsemi \texttt{cooperate}; \texttt{loop}\ s_1\ \texttt{end}, E_1\rangle \to \langle \texttt{cooperate}; \texttt{loop}\ s_1\ \texttt{end}, E_1\rangle$

   Since $\langle \texttt{cooperate}; \texttt{loop}\ s_1\ \texttt{end}, E_1\rangle\ddagger$ by the Rules (coop_s) and (seq_s) of the suspension predicate, we may now recompose $(*)$, $(**)$ and $(***)$ to get:

   $$\langle \texttt{loop}\ s_1\ \texttt{end}, E\rangle \Downarrow_{1+n_1+1} \langle \texttt{cooperate}; \texttt{loop}\ s_1\ \texttt{end}, E_1\rangle$$

   In this case we have $n = 2 + n_1 \leq 2 + \mathit{size}(s_1) = \mathit{size}(s)$.

□

The idea of "slowing down" loops by forcing them to yield the control at each iteration, which is crucial for our reactivity result, was already used in [28] for a similar purpose. A similar instrumentation of loops was proposed in [59]. However, while in our work and in [28] a `cooperate` instruction is added *in parallel* with each iteration of the body of the loop, in [59] it is added *after* each iteration of the body. We believe that in a language that allows a parallel program to be followed in sequence by another program (which is not the case in [59]), our solution is more efficient in that it avoids introducing an additional suspension in case the body of the loop already contains one.

In [11], Amadio and Dabrowski studied the notion of reactivity for a standard parallel operator. In addition to this property, they give an upper bound to the usage of computation resources and memory, based on a sophisticated control of functions. Given the absence of memory and functions, CRL allows for a much simpler bound, formalized by the notion of size.

Part III

SSL

Chapter 5

Secure Synchronous Language SSL

We are now going to introduce *secure information flow* into CRL, obtaining a Secure Synchronous Language called SSL.

Secure information flow is often formalized via the notion of *non-interference* (NI), expressing the absence of dependency between secret inputs and public outputs (or more generally, between inputs of some confidentiality level and outputs of lower or incomparable level). The non-interference property is usually grounded on a notion of semantic equivalence (e.g. bisimulation).

SSL is a kind of minimal language for studying the problem of secure information flow in synchronous reactive programs. We define two non-interference properties for SSL, a fine-grained one and a coarse-grained one, based on corresponding bisimulation equivalences. We also introduce a security type system, which we prove to ensure both non-interference properties. Thanks to the design choices of the language, this type system allows for a precise treatment of termination leaks, improving on previous work.

The work presented in this chapter is based on the paper [16], which in turn builds on the work of Almeida Matos et al. in [8], which laid the basis for the study of non-interference in a synchronous reactive language. As regards the notion of bisimulation, our work is also related to that of Amadio [10].

The main contributions of SSL are : 1) the definition of two bisimulation equivalences for synchronous reactive programs, of different granularity; 2) the proposal of two properties of reactive non-interference, based on the above bisimulations, and 3) the presentation of a type system ensuring both non-interference properties.

We note that the domains, syntax and semantics of SSL are the same as the ones of CRL. The only difference between SSL and CRL is the introduction of security levels for events and variables..

The rest of this chapter is organised as follows. Section 5.1 introduces the two bisimulation equivalences and gives some properties of them. In Section 5.2 we define our two NI properties. Finally, Section 5.4 presents our security type system and the proof of its soundness.

5.1 Fine-grained and Coarse-grained Bisimilarity

We introduce two bisimulation equivalences (aka *bisimilarities*) on scripts, which differ for the granularity of the underlying notion of observation. The first bisimulation formalizes a *fine-grained observation* of scripts: the observer is viewed as a script, which is able to interact with the observed script at any point of its execution. The second reflects a *coarse-grained observation* of scripts: here the observer is viewed as part of the environment, which interacts with the observed script only at the start and the end of instants.

Let us start with an informal description of the two bisimilarities:

1. *Fine-grained bisimilarity* $\approx^{fg}$. In the bisimulation game, each small step must be simulated by a (possibly empty) sequence of small steps, and each instant change must be simulated either by an instant change, in case the continuation is observable, or by an unobservable behaviour otherwise.

2. *Coarse-grained bisimilarity* $\approx^{cg}$. Here, each converging sequence of steps must be simulated by a converging sequence of steps, at each instant. For instant changes, the requirement is the same as for fine-grained bisimulation.

As may be expected, the latter equivalence is more abstract than the former, as it only compares the I/O behaviours of scripts (as functions on sets of events) at each instant, while the former also compares their intermediate results. Let us move now to the formal definitions of the equivalences $\approx^{fg}$ and $\approx^{cg}$. For technical convenience, we first extend the reconditioning function to non-suspended programs as follows:

Notation. $\llcorner s \lrcorner_E \stackrel{\text{def}}{=} \begin{cases} [s]_E & \text{if } \langle s, E\rangle\ddagger \\ s & \text{otherwise} \end{cases}$

Definition 5.1.1 (Fine-grained bisimulation)
A symmetric relation $\mathcal{R}$ on scripts is a fg-bisimulation *if $s_1 \,\mathcal{R}\, s_2$ implies, for any $E \subseteq Events$:*

$$1)\ \langle s_1, E\rangle \to \langle s_1', E'\rangle \ \Rightarrow\ \exists\, s_2' .\ (\ \langle s_2, E\rangle \Rightarrow \langle s_2', E'\rangle \ \wedge\ s_1' \,\mathcal{R}\, s_2'\ )$$

$$2)\ \langle s_1, E\rangle \ddagger \ \Rightarrow\ \exists\, s_2' .\ (\ \langle s_2, E\rangle \Downarrow \langle s_2', E\rangle \ \wedge\ \llcorner s_1 \lrcorner_E \ \mathcal{R}\ \llcorner s_2' \lrcorner_E\ )$$

Then s_1, s_2 are fg-bisimilar*, $s_1 \approx^{fg} s_2$, if $s_1 \,\mathcal{R}\, s_2$ for some fg-bisimulation $\mathcal{R}$.*

The equivalence $\approx^{fg}$ is *time-insensitive*, and thus insensitive to internal moves. It is also *termination-insensitive*, as it cannot distinguish proper termination from suspension (recall that no divergence is possible within an instant and thus the execution of a diverging script always spans over an infinity of instants). On the other hand, $\approx^{fg}$ is sensitive to the order of generation of events and to repeated emissions of the same event ("stuttering"). Typical examples are:

$$(\texttt{nothing}\,;\texttt{generate}\ ev) \ \approx^{fg}\ \texttt{generate}\ ev \ \not\approx^{fg}\ (\texttt{generate}\ ev\,;\texttt{generate}\ ev)$$

$$(\texttt{generate}\ ev_1 \nmid \texttt{generate}\ ev_2) \quad \not\approx^{fg} \quad (\texttt{generate}\ ev_2 \nmid \texttt{generate}\ ev_1)$$

$$\texttt{nothing} \quad \approx^{fg} \quad \texttt{cooperate} \quad \approx^{fg} \quad \texttt{loop cooperate end}$$

Definition 5.1.2 (Coarse-grained bisimulation)
A symmetric relation $\mathcal{R}$ on scripts is a cg-bisimulation *if $s_1 \,\mathcal{R}\, s_2$ implies, for any $E \subseteq Events$:*

$$\langle s_1, E\rangle \Downarrow \langle s_1', E'\rangle \ \Rightarrow\ \exists\, s_2' .\ (\ \langle s_2, E\rangle \Downarrow \langle s_2', E'\rangle \ \wedge\ \llcorner s_1' \lrcorner_{E'} \ \mathcal{R}\ \llcorner s_2' \lrcorner_{E'}\ )$$

Then s_1, s_2 are cg-bisimilar*, $s_1 \approx^{cg} s_2$, if $s_1 \,\mathcal{R}\, s_2$ for some cg-bisimulation $\mathcal{R}$.*

Like $\approx^{fg}$, the equivalence $\approx^{cg}$ is both time-insensitive and termination-insensitive. Furthermore, it is also *stuttering-insensitive* and *generation-order-insensitive* (that is, it ignores the generation order of events within an instant). Typically:

$$\texttt{generate}\ ev \ \approx^{cg}\ (\texttt{generate}\ ev\,;\texttt{generate}\ ev)$$

$$(\texttt{generate}\ ev_1 \nmid \texttt{generate}\ ev_2) \quad \approx^{cg} \quad (\texttt{generate}\ ev_2 \nmid \texttt{generate}\ ev_1)$$

More generally, the equivalence $\approx^{cg}$ views $\nmid$ as a commutative operator:

Theorem 5.1.1 (Commutativity of $\upharpoonright$ up to $\approx^{cg}$)
$\forall s_1, s_2 \quad (s_1 \upharpoonright s_2 \approx^{cg} s_2 \upharpoonright s_1)$

Finally, we prove that $\approx^{fg}$ is strictly included in $\approx^{cg}$ (the strictness of the inclusion being witnessed by the last two examples above):

Theorem 5.1.2 (Relation between the bisimilarities)

$$\approx^{fg} \subset \approx^{cg}$$

Proof To prove $\approx^{fg} \subseteq \approx^{cg}$ It is enough to show that $\approx^{fg}$ is a cg-bisimulation. Let $s_1 \approx^{fg} s_2$. Suppose that $\langle s_1, E\rangle \Downarrow \langle s_1', E'\rangle$. This means that there exists $n \geq 0$ such that:

$$\langle s_1, E_1\rangle = \langle s_1^0, E^0\rangle \rightarrow \langle s_1^1, E^1\rangle \rightarrow \cdots \rightarrow \langle s_1^n, E^n\rangle = \langle s_1', E'\rangle \overset{\ddagger}{\Upsilon}$$

Since $s_1 \approx^{fg} s_2$, by Clauses 1 and 2 of Definition 5.1.1 we have correspondingly:

$$\langle s_2, E\rangle = \langle s_2^0, E^0\rangle \Rightarrow \langle s_2^1, E^1\rangle \Rightarrow \cdots \Rightarrow \langle s_2^n, E^n\rangle = \langle s_2', E'\rangle \Downarrow \qquad (*)$$

where $s_1^i \approx^{fg} s_2^i$ for every $i < n$ and $\llcorner s_1' \lrcorner_{E'} \approx^{fg} \llcorner s_2' \lrcorner_{E'}$. Then we may conclude since $(*)$ can be rewritten as $\langle s_2, E\rangle \Downarrow \langle s_2', E'\rangle$. □

Our coarse-grained bisimilarity is very close to the semantic equivalence proposed by Amadio in [10] for a slightly different reactive language, equipped with a classical non-deterministic parallel operator. By contrast, the non-interference notion of [8] was based on a fine-grained bisimilarity (although bisimilarity was not explicitly introduced in [8], it was *de facto* used to define non-interference). Note that reactivity was not a concern in either [8] or [10] (nevertheless, it had been thoroughly studied in previous work by Amadio et al. [9]). We believe that coarse-grained bisimilarity makes full sense when coupled with reactivity, since it ignores instantaneously diverging computations and thus, if applied to non-reactive programs, it would equate any two instantaneously diverging programs.

Finally, it should be noted that, since our left-parallel operator $\upharpoonright$ is deterministic, we could as well use trace-based equivalences rather than bisimulation-based ones. However, bisimulation provides a convenient means for defining non-interference in a concurrent setting. Moreover, as our study for *CRL* is meant to provide the basis for the treatment of a fully-fledged distributed reactive language, including a notion of site and asynchronous

parallelism between sites, we chose to adopt bisimulation-based equivalences from the start.

This concludes our discussion about semantic equivalences. We turn now to the definition of non-interference, which is grounded on that of bisimulation.

5.2 Security property

In this section we define two non-interference properties for scripts. As usual when dealing with secure information flow, we assume a finite lattice $(\mathcal{S}, \leq)$ of *security levels*, ranged over by τ, σ, ϑ. We denote by $\sqcup$ and $\sqcap$ the join and meet operations on the lattice, and by $\bot$ and $\top$ its minimal and maximal elements.

In SSL, the objects that are assigned a security level are events and variables. An *observer* is identified with a downward-closed set of security levels (for short, a dc-set), i.e. a set $\mathcal{L} \subseteq \mathcal{S}$ satisfying the property: $(\tau \in \mathcal{L} \wedge \tau' \leq \tau) \Rightarrow \tau' \in \mathcal{L}$.

A type environment Γ is a mapping from variables and events to their types, which are just security levels τ, σ. Given a dc-set $\mathcal{L}$, a type environment Γ and an event environment E, the subset of E to which Γ assigns security levels in $\mathcal{L}$ is called the *$\mathcal{L}$-part* of E under Γ. Similarly, if $V : Var \to Val$ is a valuation, the subset of V whose domain is given levels in $\mathcal{L}$ by Γ is the *$\mathcal{L}$-part* of V under Γ.

Two event environments E_1, E_2 or two valuations V_1, V_2 are $=^{\Gamma}_{\mathcal{L}}$-equal, or indistinguishable by a $\mathcal{L}$-observer, if their $\mathcal{L}$-parts under Γ coincide:

Definition 5.2.1 ($\Gamma\mathcal{L}$-equality of event environments and valuations)

Let $\mathcal{L} \subseteq \mathcal{S}$ be a dc-set, Γ a type environment and V a valuation. Define:

$$E_1 =^{\Gamma}_{\mathcal{L}} E_2 \quad \textit{if} \quad \forall\, ev \in Events \;\; (\, \Gamma(ev) \in \mathcal{L} \;\Rightarrow\; (\, ev \in E_1 \Leftrightarrow ev \in E_2 \,) \,)$$
$$V_1 =^{\Gamma}_{\mathcal{L}} V_2 \quad \textit{if} \quad \forall\, x \in Var \;\; (\, \Gamma(x) \in \mathcal{L} \;\Rightarrow\; V_1(x) = V_2(x) \,)$$

Let $\rightarrow_V, \Rightarrow_V, \Downarrow_V$ denote our various semantic arrows under the valuation V. Then we may define the indistinguishability of two scripts by a fine-grained or coarse-grained $\mathcal{L}$-observer, for a given Γ, by means of the following two notions of $\Gamma\mathcal{L}$-bisimilarity:

Definition 5.2.2 (Fine-grained $\Gamma\mathcal{L}$-bisimilarity)
A symmetric relation $\mathcal{R}$ on scripts is a fg-$\Gamma\mathcal{L}$-V_1V_2-bisimulation *if $s_1 \mathcal{R} s_2$ implies, for any E_1, E_2 such that $E_1 =^{\Gamma}_{\mathcal{L}} E_2$:*

1) $\langle s_1, E_1\rangle \rightarrow_{V_1} \langle s_1', E_1'\rangle$
$\Rightarrow \exists\, s_2', E_2' \,.\, (\, \langle s_2, E_2\rangle \Rightarrow_{V_2} \langle s_2', E_2'\rangle \wedge E_1' =^{\Gamma}_{\mathcal{L}} E_2' \wedge s_1' \,\mathcal{R}\, s_2' \,)$

2) $\langle s_1, E_1\rangle\ddagger \;\Rightarrow \exists\, s_2', E_2' \,.\, (\, \langle s_2, E_2\rangle \Downarrow_{V_2} \langle s_2', E_2'\rangle \wedge E_1 =^{\Gamma}_{\mathcal{L}} E_2' \wedge \llcorner s_1 \lrcorner_{E_1} \,\mathcal{R}\, \llcorner s_2' \lrcorner_{E_2'} \,)$

3) and 4) : *Symmetric to* 1) *and* 2) *for $\langle s_2, E_2\rangle$ under valuation V_2.*

Then scripts s_1, s_2 are fg-$\Gamma\mathcal{L}$-bisimilar*, $s_1 \approx^{fg}_{\Gamma\mathcal{L}} s_2$, if for any V_1, V_2 such that $V_1 =^{\Gamma}_{\mathcal{L}} V_2$, $s_1 \mathcal{R} s_2$ for some* fg-$\Gamma\mathcal{L}$-V_1V_2-bisimulation $\mathcal{R}$.

Definition 5.2.3 (Coarse-grained $\Gamma\mathcal{L}$-bisimilarity)
A symmetric relation $\mathcal{R}$ on scripts is a cg-$\Gamma\mathcal{L}$-V_1V_2-bisimulation *if $s_1 \mathcal{R} s_2$ implies, for any E_1, E_2 such that $E_1 =^{\Gamma}_{\mathcal{L}} E_2$:*

$$\langle s_1, E_1\rangle \Downarrow_{V_1} \langle s_1', E_1'\rangle \;\Rightarrow\; \exists\, s_2', E_2' \,.\quad (\, \langle s_2, E_2\rangle \Downarrow_{V_2} \langle s_2', E_2'\rangle \wedge E_1' =^{\Gamma}_{\mathcal{L}} E_2' \wedge \llcorner s_1' \lrcorner_{E_1'} \,\mathcal{R}\, \llcorner s_2' \lrcorner_{E_2'} \,)$$

Two scripts s_1, s_2 are cg-$\Gamma\mathcal{L}$-bisimilar*, $s_1 \approx^{cg}_{\Gamma\mathcal{L}} s_2$, if for any V_1, V_2 such that $V_1 =^{\Gamma}_{\mathcal{L}} V_2$, $s_1 \mathcal{R} s_2$ for some* cg-$\Gamma\mathcal{L}$-V_1V_2-bisimulation $\mathcal{R}$.

Our *reactive non-interference* (RNI) properties are now defined as follows:

Definition 5.2.4 (Fine-grained and Coarse-grained RNI)
A script s is fg-secure in Γ if $s \approx^{fg}_{\Gamma\mathcal{L}} s$ for every dc-set $\mathcal{L}$.
A script s is cg-secure in Γ if $s \approx^{cg}_{\Gamma\mathcal{L}} s$ for every dc-set $\mathcal{L}$.

In examples, we use superscripts to indicate the level of variables and events.

Example 5.3 *The following script is cg-secure but not fg-secure:*

$$s = \texttt{if}\ x^{\top} = 0 \quad \texttt{then}\ \texttt{generate}\ ev_1^{\perp} \nmid \texttt{generate}\ ev_2^{\perp}$$
$$\texttt{else}\ \texttt{generate}\ ev_2^{\perp} \nmid \texttt{generate}\ ev_1^{\perp}$$

If we replace the second branch of s by $\texttt{generate}\ ev_1^{\perp}\,;\texttt{generate}\ ev_2^{\perp}$, then we obtain a script s' that is both *fg*-secure and *cg*-secure.

In general, from all the equivalences/inequivalences of Definitions 5.1.1 and 5.1.2, we may obtain secure/insecure scripts for the corresponding RNI property by plugging the two equivalent/inequivalent scripts in the branches of a high conditional.

Theorem 5.3.1 (Relation between the RNI properties)

Let $s \in Programs$. If s is fg-secure then s is cg-secure.

We conclude this section by comparing our work on SSL with that of Almeida Matos and al. in [8].

The language examined in [8] is similar to SSL but strictly more expressive, including imperative constructs, local declarations and a memory. Here, however, we adopt a slightly different semantics for $s \upharpoonright s'$, prescribing an *early cooperation* on the right (as described in Section 4.2). This simple change forces the scheduler to serve the same thread at the start of each instant, thus avoiding the so-called *scheduling leaks* of [8], and allowing for a more relaxed typing rule for $\upharpoonright$, which is just the standard rule for symmetric parallel composition.

Moreover, reactivity was not a concern in [8]: as soon as they contained *while loops*, scripts were not guaranteed to terminate or suspend within an instant. Hence, it only made sense to consider a fine-grained notion of non-interference. By contrast, in SSL all scripts are reactive, thanks to a clear separation between the loop construct `loop` s `end` and the iteration construct `repeat` exp `do` s `end`, and to our semantics for loops, which requires them to yield the control at each iteration of their body. This makes it possible to define a notion of coarse-grained *reactive non-interference* (RNI), which accounts only for the I/O behaviour of scripts within each instant. The coarse-grained RNI property has an advantage over the fine-grained one: it exploits in a more direct way the structure of reactive computations, and it recovers the flavor of big-step semantics within each instant, offering a more abstract NI notion for reactive scripts.

5.4 Type System

We present now our security type system for SSL, which is based on that of [8], which in turn was inspired by those proposed by Boudol and Castellani in [22] and by smith in [67] for a standard while concurrent language. The originality of these type systems is that they associate pairs (τ, σ) of security levels with scripts, where τ is a lower bound on the level of "writes" and σ is an upper bound on the level of "reads". This allows the level of reads to be recorded, and then to be used to constrain the level of writes in the remainder of the script. In this way, it is possible to obtain a more precise treatment of *termination leaks*[1] than in standard type systems.

[1]Leaks due to different termination behaviors, typically in the branches of a conditional. In classical parallel while languages, termination leaks may also arise in while loops. This

Recall that a type environment Γ is a mapping from variables and events to security levels τ, σ. Moreover, Γ associates a type of the form $\overrightarrow{\tau} \to \tau$ to functions, where $\overrightarrow{\tau}$ is a tuple of types $\tau_1, \ldots, \tau_n$. Type judgments for expressions and scripts have the form $\Gamma \vdash exp : \tau$ and $\Gamma \vdash s : (\tau, \sigma)$ respectively.

The intuition for $\Gamma \vdash exp : \tau$ is that τ is an *upper bound* on the levels of variables occurring in *exp*. According to this intuition, subtyping for expressions is *covariant*. The intuition for $\Gamma \vdash s : (\tau, \sigma)$ is that τ is a *lower bound* on the levels of events generated in s (the "writes"of s), and σ is an *upper bound* on the levels of events awaited or watched in s and of variables tested in s (the "reads" or *guards* of s, formally defined in Definition 5.4.2). Accordingly, subtyping for scripts is *contravariant* in its first component, and *covariant* in the second.

The typing rules for expressions and scripts are presented respectively in Figure 5.1 and Figure 5.2. The three rules that increase the guard type are (Watching), (Repeat) and (Cond1), and those that check it against the write type of the continuation are (Seq), (Repeat) and (Loop). Note that there are two more rules for the conditional, factoring out the cases where either both branches terminate in one instant or both branches are infinite: indeed, in these cases no termination leaks can arise and thus it is not necessary to increase the guard level. In Rule (Cond2), *FIN* denotes the set of scripts that *terminate* in one instant, namely those built without using the constructs `await` *ev*, `loop` and `cooperate`. In Rule (Cond3), *INF* denotes the set of infinite or *non-terminating* scripts, defined inductively as follows:

- `loop` s `end` $\in INF$;
- $s \in INF \Rightarrow$ `repeat` exp `do` s `end` $\in INF$;
- $s_1 \in INF \Rightarrow s_1; s_2 \in INF$;
- $s_1 \in FIN \wedge s_2 \in INF \Rightarrow s_1; s_2 \in INF$;
- $s_1 \in INF \vee s_2 \in INF \Rightarrow s_1 \nmid s_2 \in INF$
- $s_1 \in INF \wedge s_2 \in INF \Rightarrow$ `if` exp `then` s_1 `else` s_2 `end` $\in INF$

Note that $FIN \cup INF \subset Programs$. Examples of scripts that are neither in FIN nor in INF are: `await` *ev*, `if` *exp* `then nothing else` (`loop` s `end`) `end`, and `do` (`loop` s `end`) `watching` *ev*.

is not possible in SSL, given the simple form of the `loop` construct. On the other hand, is *SSL* termination leaks can also occur in repeat statements, and in await statement.

$$(\textsc{Val})\ \Gamma \vdash v : \bot \qquad\qquad (\textsc{Var})\ \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}$$

$$(\textsc{SubExp})\ \frac{\Gamma \vdash exp : \sigma, \sigma \leq \sigma'}{\Gamma \vdash exp : \sigma'}$$

$$(\textsc{Fun})\ \frac{\Gamma \vdash \overrightarrow{exp} : \overrightarrow{\tau}, \Gamma(f) = \overrightarrow{\tau} \to \tau, \forall i\,.\tau_i \leq \tau}{\Gamma \vdash f(\overrightarrow{exp}) : \tau}$$

Figure 5.1: Typing rules for expressions

Definition 5.4.1 (Safe conditionals)
A conditional `if` exp `then` s_1 `else` s_2 `end` *is* safe *if* $s_1, s_2 \in FIN$ *or* $s_1, s_2 \in INF$.

The reason for calling such conditionals "safe" is that they cannot introduce termination leaks, since their two branches have the same termination behaviour.

We now prove that typability implies security via the classical steps:

Lemma 5.4.1 (Subject Reduction)
Let $\Gamma \vdash s : (\tau, \sigma)$. *Then:*

i) $\langle s, E\rangle \to \langle s', E'\rangle$ *implies* $\Gamma \vdash s' : (\tau, \sigma)$;

ii) $\langle s, E\rangle\ddagger$ *implies* $\Gamma \vdash [s]_E : (\tau, \sigma)$.

Proof By induction on the proof of $\Gamma \vdash s : (\tau, \sigma)$. We will prove either Clause *i)* or Clause *ii)* depending on whether $\langle s, E\rangle$ is able to move or is suspended. We are going only to present the important cases.

- Rule (Await). In this case $s =$ `await` ev. If $ev \notin E$, then it is immediate to see that Clause *ii)* holds, since $[\texttt{await}\ ev]_E = \texttt{await}\ ev$. Assume now $ev \in E$. Then Rule (4.8) gives the transition:

$$\frac{ev \in E}{\langle \texttt{await}\ ev, E\rangle \to \langle \texttt{nothing}, E\rangle}$$

$$\text{(Nothing)}\quad \Gamma \vdash \texttt{nothing} : (\top, \bot)$$

$$\text{(Cooperate)}\quad \Gamma \vdash \texttt{cooperate} : (\top, \bot)$$

$$\text{(Seq)}\quad \frac{\Gamma \vdash s_1 : (\tau_1, \sigma_1), \Gamma \vdash s_2 : (\tau_2, \sigma_2), \sigma_1 \leq \tau_2}{\Gamma \vdash s_1 \,;\, s_2 : (\tau_1 \sqcap \tau_2, \sigma_1 \sqcup \sigma_2)}$$

$$\text{(Par)}\quad \frac{\Gamma \vdash s_1 : (\tau_1, \sigma_1), \Gamma \vdash s_2 : (\tau_2, \sigma_2)}{\Gamma \vdash s_1 \nmid s_2 : (\tau_1 \sqcap \tau_2, \sigma_1 \sqcup \sigma_2)}$$

$$\text{(Generate)}\quad \frac{\Gamma(ev) = \tau}{\Gamma \vdash \texttt{generate}\, ev : (\tau, \bot)}$$

$$\text{(Await)}\quad \frac{\Gamma(ev) = \sigma}{\Gamma \vdash \texttt{await}\, ev : (\top, \sigma)}$$

$$\text{(Watching)}\quad \frac{\Gamma(ev) = \vartheta, \Gamma \vdash s : (\tau, \sigma), \vartheta \leq \tau}{\Gamma \vdash \texttt{do}\, s\, \texttt{watching}\, ev : (\tau, \vartheta \sqcup \sigma)}$$

$$\text{(Loop)}\quad \frac{\Gamma \vdash s : (\tau, \sigma), \sigma \leq \tau}{\Gamma \vdash \texttt{loop}\, s\, \texttt{end} : (\tau, \sigma)}$$

$$\text{(Repeat)}\quad \frac{\Gamma \vdash exp : \vartheta\, \Gamma \vdash s : (\tau, \sigma), \vartheta \sqcup \sigma \leq \tau}{\Gamma \vdash \texttt{repeat}\, exp\, \texttt{do}\, s\, \texttt{end} : (\tau, \vartheta \sqcup \sigma)}$$

$$\text{(Cond1)}\quad \frac{\Gamma \vdash exp : \vartheta, \Gamma \vdash s_i : (\tau, \sigma), i = 1, 2, \vartheta \leq \tau}{\Gamma \vdash \texttt{if}\, exp\, \texttt{then}\, s_1\, \texttt{else}\, s_2\, \texttt{end} : (\tau, \vartheta \sqcup \sigma)}$$

$$\text{(Cond2)}\quad \frac{\Gamma \vdash exp : \vartheta, (\Gamma \vdash s_i : (\tau, \sigma) \wedge s_i \in FIN, i = 1, 2\,), \vartheta \leq \tau}{\Gamma \vdash \texttt{if}\, exp\, \texttt{then}\, s_1\, \texttt{else}\, s_2\, \texttt{end} : (\tau, \sigma)}$$

$$\text{(Cond3)}\quad \frac{\Gamma \vdash exp : \vartheta, (\Gamma \vdash s_i : (\tau, \sigma) \wedge s_i \in INF, i = 1, 2\,), \vartheta \leq \tau}{\Gamma \vdash \texttt{if}\, exp\, \texttt{then}\, s_1\, \texttt{else}\, s_2\, \texttt{end} : (\tau, \sigma)}$$

$$\text{(SubProg)}\quad \frac{\Gamma \vdash s : (\tau, \sigma), \tau' \leq \tau, \sigma \leq \sigma'}{\Gamma \vdash s : (\tau', \sigma')}$$

Figure 5.2: Typing rules for scripts

Again, we conclude that Clause i) holds using the typing Rules (NOTHING) and (SUBPROG).

- Rule (SEQ). In this case $s = s_1; s_2$, and $\Gamma \vdash s : (\tau, \sigma)$ is deduced from $\Gamma \vdash s_1 : (\tau_1, \sigma_1)$ and $\Gamma \vdash s_2 : (\tau_2, \sigma_2)$, where $\tau = \tau_1 \sqcap \tau_2$, $\sigma = \sigma_1 \sqcup \sigma_2$, and $\sigma_1 \leq \tau_2$. Assume first that $\langle s, E\rangle$ can move, so we need to prove Clause i). There are two cases:

  - $s_1 \neq \texttt{nothing}$. Then the transition is obtained by Rule (4.1):

    $$\frac{\langle s_1, E\rangle \to \langle s_1', E'\rangle}{\langle s_1; s_2, E\rangle \to \langle s_1'; s_2, E'\rangle}$$

    Here $s' = s_1'; s_2$. By induction $\Gamma \vdash s_1' : (\tau_1, \sigma_1)$. Then by the typing Rule (SEQ) we obtain $\Gamma \vdash s_1'; s_2 : (\tau_1 \sqcap \tau_2, \sigma_1 \sqcup \sigma_2)$ and we may conclude.
  - $s_1 = \texttt{nothing}$. Then the transition is inferred by Rule (4.2):

    $$\langle \texttt{nothing}; s_2, E\rangle \to \langle s_2, E\rangle$$

    Here $s' = s_2$. Since $\tau_1 \sqcap \tau_2 \leq \tau_2$ and $\sigma_1 \sqcup \sigma_2 \geq \sigma_2$, by the typing Rule (SUBPROG) we get:

    $$\Gamma \vdash s_2 : (\tau_1 \sqcap \tau_2, \sigma_1 \sqcup \sigma_2)$$

    and we may conclude.

  Suppose now $\langle s, E\rangle\ddagger$. In this case we need to prove Clause ii). The statement $\langle s_1; s_2, E\rangle\ddagger$ is deduced from $\langle s_1, E\rangle\ddagger$. By induction $\Gamma \vdash [s_1]_E : (\tau_1, \sigma_1)$. Since $[s_1; s_2]_E = [s_1]_E; s_2$, we may then use the typing Rule (SEQ) to obtain $\Gamma \vdash [s_1]_E; s_2 : (\tau_1 \sqcap \tau_2, \sigma_1 \sqcup \sigma_2)$ and we may conclude.

- Rules (COND1), (COND2) and (COND3). Then $s = \texttt{if}\ exp\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{end}$ and $\neg\langle s, E\rangle\ddagger$. We distinguish two cases:

  1. $s_i \notin FIN$ for both $i \in \{1, 2\}$ and $s_i \notin INF$ for both $i \in \{1, 2\}$. Then $\Gamma \vdash s : (\tau, \sigma)$ is deduced by the typing Rule (COND1) from the hypotheses $\Gamma \vdash exp : \vartheta$ and $\Gamma \vdash s_i : (\tau, \sigma')$ for $i \in \{1, 2\}$,

where $\sigma = \vartheta \sqcup \sigma'$. Consider the case where *exp* evaluates to *true*. Then the transition is inferred by Rule (4.14):

$$\frac{exp \rightsquigarrow tt}{\langle \texttt{if}\ exp\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{end}, E\rangle \rightarrow \langle s_1, E\rangle}$$

Here $s' = s_1$. Since $\Gamma \vdash s_1 : (\tau, \sigma')$ and $\sigma' \leq \sigma$, by the typing Rule (SubProg) we have $\Gamma \vdash s_1 : (\tau, \sigma)$ and we may conclude.

2. $s_i \in FIN$ for both $i \in \{1, 2\}$ or $s_i \in INF$ for both $i \in \{1, 2\}$. Then, $\Gamma \vdash s : (\tau, \sigma)$ is deduced by one of the two typing Rules (Cond2) or (Cond3), from the hypotheses $\Gamma \vdash exp : \vartheta$ and $\Gamma \vdash s_i : (\tau, \sigma)$ for $i \in \{1, 2\}$. Again, assuming *exp* evaluates to *true*, we get a transition $\langle s, E\rangle \rightarrow \langle s_1, E\rangle$. Since $\Gamma \vdash s_1 : (\tau, \sigma)$, we may immediately conclude.

□

Definition 5.4.2 (Guards and Generated Events)

1) For any s, Guards(s) is the union of the set of events ev such that s contains an `await` *ev or a* `do` s' `watching` *ev instruction (for some* s'*), together with the set of variables x that occur in s as argument of a function or in the control expression exp of an instruction* `repeat` *exp* `do` s' `end` *or of an* unsafe *conditional* `if` *exp* `then` s_1 `else` s_2 `end` *in s.*

2) For any s, Gen(s) is the set of events ev such that `generate` *ev occurs in s.*

Lemma 5.4.2 (Guard Safety for expressions)

If $\Gamma \vdash exp : \vartheta$*, then* $\Gamma(g) \leq \vartheta$ *for every* $g \in Guards(exp)$*.*

Proof Immediate, by inspecting the typing rules for expressions. □

The following Lemma establishes that if $\Gamma \vdash s : (\tau, \sigma)$, then τ is a *lower bound* on the levels of events in $Gen(s)$ and σ is an *upper bound* on the levels of events and variables in $Guards(s)$.

Theorem 5.4.1 (Guard Safety and Confinement)

i) If $\Gamma \vdash s : (\tau, \sigma)$ *then* $\Gamma(g) \leq \sigma$ *for every* $g \in \mathit{Guards}(s)$*;*

ii) If $\Gamma \vdash s : (\tau, \sigma)$ *then* $\tau \leq \Gamma(ev)$ *for every* $ev \in \mathit{Gen}(s)$*.*

Proof By induction of the inference of $\Gamma \vdash s : (\tau, \sigma)$. We proceed by case analysis on the last typing rule used in the inference. We prove Clauses *i*) and *ii*) at the same time for each case. We are focussing on the most interesting cases.

- *Basic cases:*

  1. Rule (AWAIT). Here $s = \texttt{await}\ ev$, $\mathit{Guards}(s) = \{ev\}$ and $\mathit{Gen}(s) = \emptyset$. Then Clause *ii*) holds trivially, and Clause *i*) holds because the statement $\Gamma \vdash s : (\tau, \sigma)$ is deduced from the hypothesis $\Gamma(ev) = \sigma$.

- *Inductive cases:*

  1. Rule (SEQ). Here $s = s_1; s_2$ and $\Gamma \vdash s : (\tau, \sigma)$ is deduced from the hypotheses $\Gamma \vdash s_1 : (\tau_1, \sigma_1)$, $\Gamma \vdash s_2 : (\tau_2, \sigma_2)$, $\tau = \tau_1 \sqcap \tau_2$, $\sigma = \sigma_1 \sqcup \sigma_2$ and $\sigma_1 \leq \tau_2$. We consider the two Clauses in turn.
     - Clause *i*). We have $\mathit{Guards}(s) = \mathit{Guards}(s_1) \cup \mathit{Guards}(s_2)$. By induction, $\Gamma(g_i) \leq \sigma_i$ for every $g_i \in \mathit{Guards}(s_i)$. Since $\sigma_i \leq \sigma_1 \sqcup \sigma_2 = \sigma$, we can conclude.
     - Clause *ii*). We have $\mathit{Gen}(s) = \mathit{Gen}(s_1) \cup \mathit{Gen}(s_2)$. By induction $\tau_i \leq \Gamma(ev_i)$ for every $ev_i \in \mathit{Gen}(s_i)$. Since $\tau = \tau_1 \sqcap \tau_2 \leq \tau_i$, we can conclude.
  2. Rules (COND1), (COND2) and (COND3).

     Here $s = \texttt{if}\ \mathit{exp}\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{end}$ and in all cases we have $\mathit{Gen}(s) = \mathit{Gen}(s_1) \cup \mathit{Gen}(s_2)$. In the case of Rule (COND1), we have $\mathit{Guards}(s) = \mathit{Guards}(s_1) \cup \mathit{Guards}(s_2) \cup \mathit{Guards}(\mathit{exp})$ and the statement $\Gamma \vdash s : (\tau, \sigma)$ is deduced from the hypotheses $\Gamma \vdash \mathit{exp} : \vartheta$, $\Gamma \vdash s_1 : (\tau, \sigma')$ and $\Gamma \vdash s_2 : (\tau, \sigma')$, for some σ' such that $\vartheta \sqcup \sigma' = \sigma$. In the other two cases, $\Gamma \vdash s : (\tau, \sigma)$ is deduced from the hypotheses $\Gamma \vdash \mathit{exp} : \vartheta$, $\Gamma \vdash s_1 : (\tau, \sigma)$ and $\Gamma \vdash s_2 : (\tau, \sigma)$, and $\mathit{Guards}(s) = \mathit{Guards}(s_1) \cup \mathit{Guards}(s_2)$. Thus in all cases there exists $\sigma' \leq \sigma$ such that $\Gamma \vdash s_i : (\tau, \sigma')$ for $i \in \{1, 2\}$. This is all we need to treat the three cases uniformly. We prove Clauses *i*) and *ii*) in turn.

- Clause *i*). By induction, for each i we have $\Gamma(g_i) \leq \sigma'$ for every $g_i \in \mathit{Guards}(s_i)$. Since $\sigma' \leq \sigma$, this is enough to conclude in the case of Rules (Cond2) and (Cond3). In the case of Rule (Cond1), we additionally need to prove $\Gamma(g) \leq \sigma$ for each $g \in \mathit{Guards}(\mathit{exp})$. Recall that in this case $\sigma = \sigma' \sqcup \vartheta$. Then we may conclude since $\Gamma(g) \leq \vartheta$ by Lemma 5.4.2.
- Clause *ii*). We have $\mathit{Gen}(s) = \mathit{Gen}(s_1) \cup \mathit{Gen}(s_2)$. By induction $\tau \leq \Gamma(ev_i)$ for every $ev_i \in \mathit{Gen}(s_i)$. Then we can immediately conclude.

3. Rule (Loop). Straightforward inductive case.
4. Rule (SubProg). In this case $\Gamma \vdash s : (\tau, \sigma)$ is deduced from the hypothesis $\Gamma \vdash s' : (\tau', \sigma')$ for some τ', σ' such that $\tau \leq \tau'$ and $\sigma' \leq \sigma$. By induction $\Gamma(g) \leq \sigma'$ for every $g \in \mathit{Guards}(s)$ and $\tau' \leq \Gamma(ev)$ for every $ev \in \mathit{Gen}(s)$. A fortiori $\Gamma(g) \leq \sigma$ and $\tau \leq \Gamma(ev)$.

□

Definition 5.4.3 ($\Gamma\mathcal{L}$-Highness) *Let Γ be a type environment and $\mathcal{L}$ be a downward-closed set of security levels.*

1. *The set of $\mathcal{H}^{\Gamma,\mathcal{L}}_{syn}$ of* syntactically $\Gamma\mathcal{L}$-high programs *is defined by:*

$$\mathcal{H}^{\Gamma,\mathcal{L}}_{syn} = \{s \in \mathit{Programs} \mid \forall\, ev \in \mathit{Gen}(s) \quad \Gamma(ev) \notin \mathcal{L}\}$$

2. *The set of $\mathcal{H}^{\Gamma,\mathcal{L}}_{sem}$ of* semantically $\Gamma\mathcal{L}$-high programs *is defined coinductively by:*

$$s \in \mathcal{H}^{\Gamma,\mathcal{L}}_{sem} \;\Rightarrow\; \forall E \subseteq \mathit{Events} \quad \begin{cases} \langle s, E\rangle \rightarrow \langle s', E'\rangle \;\Rightarrow\; (\, E =^{\Gamma}_{\mathcal{L}} E' \\ \qquad\qquad\qquad\qquad \wedge\; s' \in \mathcal{H}^{\Gamma,\mathcal{L}}_{sem}\,) \\ \langle s, E\rangle \ddagger \;\Rightarrow\; [s]_E \in \mathcal{H}^{\Gamma,\mathcal{L}}_{sem} \end{cases}$$

Since a syntactically $\Gamma\mathcal{L}$-high program does not contain generated events of level in $\mathcal{L}$, its behaviour is unobservable for an $\mathcal{L}$-observer. Indeed, it is easy to show that every syntactically $\Gamma\mathcal{L}$-high program is also semantically $\Gamma\mathcal{L}$-high:

Lemma 5.4.3 (Syntactic $\Gamma\mathcal{L}$-highness implies semantic $\Gamma\mathcal{L}$-highness) *For any type environment Γ and dc-set of security levels $\mathcal{L}$: $\mathcal{H}_{syn}^{\Gamma,\mathcal{L}} \subseteq \mathcal{H}_{sem}^{\Gamma,\mathcal{L}}$.*

Clearly, this inclusion is strict. For instance, the program:

$$s = \texttt{if } tt \texttt{ then nothing else generate } ev^{\perp} \texttt{ end}$$

is semantically $\Gamma\perp$-high but not syntactically $\Gamma\perp$-high.

When Γ and $\mathcal{L}$ are clear from the context, we will omit them and talk simply of "syntactically high" and "semantically high" programs.

We introduce now the notions of $\Gamma\mathcal{L}$-guardedness (borrowed from [8]) and of $\Gamma\mathcal{L}$-unaffectingness, which will be used to prove our soundness result.

Both these notions are syntactic. Let us briefly explain the intuition behind them. A program s is $\Gamma\mathcal{L}$-guarded if it can be typed in Γ with a type (τ, σ) such that σ belongs to $\mathcal{L}$; since σ is an upper bound for the level of guards in s (by the Guard Safety Lemma), this means that s contains only guards of level in $\mathcal{L}$ (although the level of expressions controlling safe conditionals may not be in $\mathcal{L}$); hence its behaviour will be deterministic for an $\mathcal{L}$-observer. On the other hand, a program s is $\Gamma\mathcal{L}$-unaffecting if it can be typed in Γ with a type (τ, σ) such that τ does *not* belong to $\mathcal{L}$; since τ is a lower bound for the level of generated in s (by the Confinement Lemma), this means that s does not contain generated events of level in $\mathcal{L}$; hence the behaviour of s will be unobservable for an $\mathcal{L}$-observer.

Definition 5.4.4 ($\Gamma\mathcal{L}$-Guardedness) *Let Γ be a type environment and $\mathcal{L}$ be a downward-closed set of security levels. A program s is* $\Gamma\mathcal{L}$-guarded *if there exist τ, σ such that $\Gamma \vdash s : (\tau, \sigma)$ and $\sigma \in \mathcal{L}$.*

Definition 5.4.5 ($\Gamma\mathcal{L}$-Unaffectingness) *Let Γ be a type environment and $\mathcal{L}$ be a downward-closed set of security levels. A program s is* $\mathcal{L}$-unaffecting *if there exist τ, σ such that $\Gamma \vdash s : (\tau, \sigma)$ and $\tau \notin \mathcal{L}$. Conversely, s is* $\mathcal{L}$-affecting *if it is typable and such that $\Gamma \vdash s : (\tau, \sigma)$ implies $\tau \in \mathcal{L}$.*

Note that if s is $\Gamma\mathcal{L}$-guarded or $\Gamma\mathcal{L}$-(un)affecting, then by definition s is typable. Moreover, as a simple consequence of Theorem 5.4.1 (Subject Reduction), both $\Gamma\mathcal{L}$-guardedness and $\Gamma\mathcal{L}$-unaffectingness are preserved by execution.

Lemma 5.4.4 ($\Gamma\mathcal{L}$-Unaffectingness implies Syntactic $\Gamma\mathcal{L}$-highness) *Let Γ be a type environment and $\mathcal{L}$ be a downward-closed set of security levels. If s is $\Gamma\mathcal{L}$-unaffecting, then $s \in \mathcal{H}_{syn}^{\Gamma,\mathcal{L}}$.*

Note that the reverse implication does not hold in general: indeed, by definition $\Gamma\mathcal{L}$-Unaffectingness implies typability, while a syntactically $\Gamma\mathcal{L}$-high program is not necessarily typable. Consider for example the following program, assuming a three-level security lattice $\{\bot, \ell, \top\}$:

$$\texttt{if } x^{\top} \texttt{ then generate } ev^{\ell} \texttt{ else nothing end}$$

This program is $\Gamma\mathcal{L}$-high for $\mathcal{L} = \{\bot\}$. However, it is not typable (since it has a level drop from its control expression of level $\top$ to its first branch of level ℓ), hence it cannot be $\Gamma\mathcal{L}$-unaffecting.

In fact, the reverse implication only holds in one particular case, namely when $\mathcal{L} = \mathcal{S} - \{\top\}$. In this case, by definition of syntactic $\Gamma\mathcal{L}$-highness we have $\Gamma(ev) = \top$ for every ev in $Gens$ and therefore the program s is typable. Indeed, it can be easily shown that every program whose generated events are all of level $\top$ is typable, and dually, that every program whose guards and control expressions of conditionals (including the safe ones) are all of level $\bot$ is typable.

We are now ready to prove the two preliminary results that underpin the soundness theorem.

Theorem 5.4.2 (Behaviour of $\mathcal{L}$-guarded programs) *Let s be typable and $\mathcal{L}$-guarded in Γ. Then for any V_1, V_2 such that $V_1 =^{\Gamma}_{\mathcal{L}} V_2$ and for any E_1, E_2 such that $E_1 =^{\Gamma}_{\mathcal{L}} E_2$:*

1. $(\langle s, E_1\rangle \rightarrow_{V_1} \langle s', E_1'\rangle) \;\Rightarrow\; \exists s'', E_2' \,.\, (\langle s, E_2\rangle \Rightarrow_{V_2} \langle s'', E_2'\rangle \wedge E_1' =^{\Gamma}_{\mathcal{L}} E_2')$
2. $\langle s, E_1\rangle \ddagger \;\Rightarrow\; \langle s, E_2\rangle \overset{\ddagger}{\curlyvee}$

Theorem 5.4.3 (Behaviour of non $\mathcal{L}$-guarded programs) *Let s be typable and non $\mathcal{L}$-guarded in Γ. Then either s is $\mathcal{L}$-unaffecting, or for any V_1, V_2 such that $V_1 =^{\Gamma}_{\mathcal{L}} V_2$ and for any E_1, E_2 such that $E_1 =^{\Gamma}_{\mathcal{L}} E_2$::*

1. $(\langle s, E_1\rangle \rightarrow_{V_1} \langle s', E_1'\rangle) \;\Rightarrow\; \exists s'', E_2' \,.\, (\langle s, E_2\rangle \Rightarrow_{V_2} \langle s'', E_2'\rangle \wedge E_1' =^{\Gamma}_{\mathcal{L}} E_2')$
2. $\langle s, E_1\rangle \ddagger \;\Rightarrow\; \exists\, s', E_2' \,.\, (\langle s, E_2\rangle \Downarrow_{V_2} \langle s', E_2'\rangle \;\wedge\; E_1 =^{\Gamma}_{\mathcal{L}} E_2')$

Lemma 5.4.5 (Semantically $\Gamma\mathcal{L}$-high programs are all $\Gamma\mathcal{L}$-bisimilar) *For any type environment Γ and dc-set of security levels $\mathcal{L}$:*

$$\textit{If } s_1, s_2 \in \mathcal{H}^{\Gamma,\mathcal{L}}_{sem}, \textit{ then } s_1 \approx^{fg}_{\Gamma\mathcal{L}} s_2.$$

Proof We show that for any pair of valuations V_1, V_2 such that $V_1 =^{\Gamma}_{\mathcal{L}} V_2$, the relation $\mathcal{R}_{\Gamma,\mathcal{L}} = (\mathcal{H}^{\Gamma,\mathcal{L}}_{sem} \times \mathcal{H}^{\Gamma,\mathcal{L}}_{sem})$ is an fg-$\Gamma\mathcal{L}$-V_1V_2-bisimulation.

Suppose $(s_1, s_2) \in \mathcal{R}_{\Gamma,\mathcal{L}}$. We prove Clauses 1) and 2) of Definition 5.2.2 (this is enough since Clauses 3) and 4) are symmetric):

1. Let $\langle s_1, E_1\rangle \rightarrow_{V_1} \langle s_1', E_1'\rangle$. In this case, since s_1 is semantically high, we know that $E_1 =^{\Gamma}_{\mathcal{L}} E_1'$ and $s_1' \in \mathcal{H}^{\Gamma,\mathcal{L}}_{sem}$. Then s_2 may reply by staying idle, namely by the empty sequence of moves $\langle s_2, E_2\rangle \Rightarrow_{V_2} \langle s_2, E_2\rangle$, given that $E_1' =^{\Gamma}_{\mathcal{L}} E_1 =^{\Gamma}_{\mathcal{L}} E_2$ and both s_1' and s_2 are in $\mathcal{H}^{\Gamma,\mathcal{L}}_{sem}$.

2. Let $\langle s_1, E_1\rangle\ddagger$. Then, since s_1 is semantically high, we have $[s_1]_{E_1} \in \mathcal{H}^{\Gamma,\mathcal{L}}_{sem}$. Now, by Theorem 4.3.2 (Reactivity) we know that there exist s_2', E_2' such that $\langle s_2, E_2\rangle \Downarrow \langle s_2', E_2'\rangle$. Since s_2 is semantically high, we know that $E_2 =^{\Gamma}_{\mathcal{L}} E_2'$ and $s_2' \in \mathcal{H}^{\Gamma,\mathcal{L}}_{sem}$ (whether s_2' is `nothing` or a program suspended in E_2'). Then we may conclude, since $E_1 =^{\Gamma}_{\mathcal{L}} E_2 =^{\Gamma}_{\mathcal{L}} E_2'$ and $\llcorner s_1 \lrcorner_{E_1}$ and $\llcorner s_2' \lrcorner_{Elo_2}$ are in $\mathcal{H}^{\Gamma,\mathcal{L}}_{sem}$.

□

We may now prove the main result of this section, namely the soundness of the type system for fine-grained reactive noninterference.

Theorem 5.4.4 (Typability ⇒ Fine-grained RNI) *Let $s \in$ Programs. If s is typable in Γ then s is fg-secure in Γ.*

Proof We want to show that $s \approx^{fg}_{\Gamma\mathcal{L}} s$ for any downward-closed set $\mathcal{L}$. For any such $\mathcal{L}$, consider the relation $\mathcal{R}_{\Gamma,\mathcal{L}} = \mathcal{R}^1_{\Gamma,\mathcal{L}} \cup \mathcal{R}^2_{\Gamma,\mathcal{L}}$ on programs, where:

$$\begin{aligned} \mathcal{R}^1_{\Gamma,\mathcal{L}} &= \{(s_1, s_2) \mid s_i \text{ is } \Gamma\mathcal{L}\text{−unaffecting},\ i = 1, 2\} \\ \mathcal{R}^2_{\Gamma,\mathcal{L}} &= \{(s, s) \mid s \text{ is } \Gamma\mathcal{L}\text{−affecting}\} \end{aligned}$$

Note that for every s which is typable in Γ we have either $(s, s) \in \mathcal{R}^1_{\Gamma,\mathcal{L}}$ or $(s, s) \in \mathcal{R}^2_{\Gamma,\mathcal{L}}$. We show now that for any pair of valuations V_1, V_2 such that $V_1 =^{\Gamma}_{\mathcal{L}} V_2$, the relation $\mathcal{R}_{\Gamma,\mathcal{L}}$ is an fg-$\Gamma\mathcal{L}$-V_1V_2-bisimulation.

1. Suppose $(s_1, s_2) \in \mathcal{R}^1_{\Gamma,\mathcal{L}}$. Then by Lemma 5.4.4 we have $s_1, s_2 \in \mathcal{H}^{\Gamma,\mathcal{L}}_{syn}$, whence by Lemma 5.4.3 we obtain $s_1, s_2 \in \mathcal{H}^{\Gamma,\mathcal{L}}_{sem}$. At this point we proceed as in the proof of Lemma 5.4.5 to show that the pair (s_1, s_2) satisfies the two clauses of fg-$\Gamma\mathcal{L}$-V_1V_2-bisimulation, leading to a pair of residuals (s_1', s_2') which belongs again to $\mathcal{R}^1_{\Gamma,\mathcal{L}}$ (because $\Gamma\mathcal{L}$-unaffectingness is preserved by execution).

2. Suppose now $(s, s) \in \mathcal{R}^2_{\Gamma,\mathcal{L}}$. Then we show that the pair (s, s) satisfies the two clauses of fg-$\Gamma\mathcal{L}$-V_1V_2-bisimulation, using Theorem (5.4.2) or Theorem (5.4.3) depending on whether s is $\mathcal{L}$-guarded or not. Since $\mathcal{L}$-guardedness is preserved by execution, an $\mathcal{L}$-guarded program always remains $\mathcal{L}$-guarded and its behaviour is exactly the same for an $\mathcal{L}$-observer in both valuations V_1 and V_2. In this case, we get a pair of residuals (s', s') which falls back into the relation $\mathcal{R}^2_{\Gamma,\mathcal{L}}$. On the other hand, if s is not $\mathcal{L}$-guarded and is $\mathcal{L}$-affecting (recall that we already treated in 1. the case where s is $\mathcal{L}$-unaffecting), then it initially behaves as an $\mathcal{L}$-guarded program, giving rise to a pair of residuals (s', s') which either falls back into the relation $\mathcal{R}^2_{\Gamma,\mathcal{L}}$, or into the relation $\mathcal{R}^1_{\Gamma,\mathcal{L}}$ if s' is $\mathcal{L}$-unaffecting. Intuitively, this second case occurs when s first generates some events of level in $\mathcal{L}$, and then traverses some guard not in $\mathcal{L}$, from which point onwards it becomes $\mathcal{L}$-unaffecting.

□

Note that scripts s, s' of Example 5.3 are not typable (although *cg*-secure). We conclude with some examples illustrating the use of the rules for the conditional.

Example 5.5 *The following scripts s_i and s are all typable:*

$s_1 = \texttt{if}\ (x^\top = 0)\ \texttt{then await}\ ev_1^\top\ \texttt{else cooperate end}$ *type* $(\top, \top)$

$s_2 = \texttt{if}\ (x^\top = 0)\ \texttt{then nothing else cooperate end}$ *type* $(\top, \top)$

$s_3 = \texttt{if}\ (x^\top = 0)\ \texttt{then nothing else (loop nothing end) end}$ *type* $(\top, \top)$

$s_4 = \texttt{if}\ (x^\top = 0)\ \texttt{then (loop nothing end) else (loop cooperate end) end}$ *type* $(\top, \bot)$

$s = \texttt{generate}\ ev_2^\bot$ *type* $(\bot, \bot)$

Then $s_4; s$ is typable but not $s_1; s$, $s_2; s$ nor $s_3; s$.

We have extended the core reactive language *CRL* by adding security at the language level, obtaining *SSL*. In *SSL*, we proposed two reactive non-interference properties with a security type system ensuring them. This type system is more permissive than that of [8], thanks to the relaxed typing rule for parallel composition and to refined typing rules for the conditional. Both improvements are made possible by design choices of *SSL*.

5.5.1 Example

In this section, we give an example to illustrate the power of *SSL* and how we can control information flow over an everyday program. To this end,

we add function calls at the language level with the same requirement as in DSL (functions should terminate instantaneously).

Twitter News Application

Twitter [3] is an online social networking and micro-blogging service that enables its users to send and read text-based messages of up to 140 characters, known as "tweets". Users may subscribe to other users tweets (*follow* them) and get their tweets in real time.

Here, we are going to present an example of a twitter application containing three main frames:

1. The news frame which shows all followed users' tweets; information can be displayed in this area by using the function *print_news*.

2. The advertising frame which receives advertisements. With the help of the function *print_ads*, the ads will appear in this frame.

3. The last frame displays the local weather by taking into account the user location. Once the information is received, it can be showed into this frame with the *print_weather* function. The new location for the weather is set by means of the function *set_location*.

The example is shown in Figure 5.3. In this example, each time a new tweet arrives the event "new tweets" is generated. Then, we print the new tweet in the associated frame. In the weather frame, if the weather has changed and we have new information about the weather, an event called "update_weather" is generated; the frame will be updated as soon as this event is received. If the location is changed, which is signaled by "new_location", we change our current location. Finally, the ad frame updates its contents each time we receive "new_ads". The code is given in Figure 5.4.

The security problem raised by this application is the following: we do not want that anybody knows how frequently we receive news. Indeed, it is getting a real issue to protect personal data from advertising companies like Google. It is not a major security issue, but if an external observer can see how frequently the user receives new tweets, it can suggest different types of advertisements. Therefore, the security level of the event "new tweets" is set to high. For the location changes, we only want to allow weather functions to see these changes, thus the security level of "new_location" is not as high as "new tweets" but it is not low either (it will be an intermediate level l).

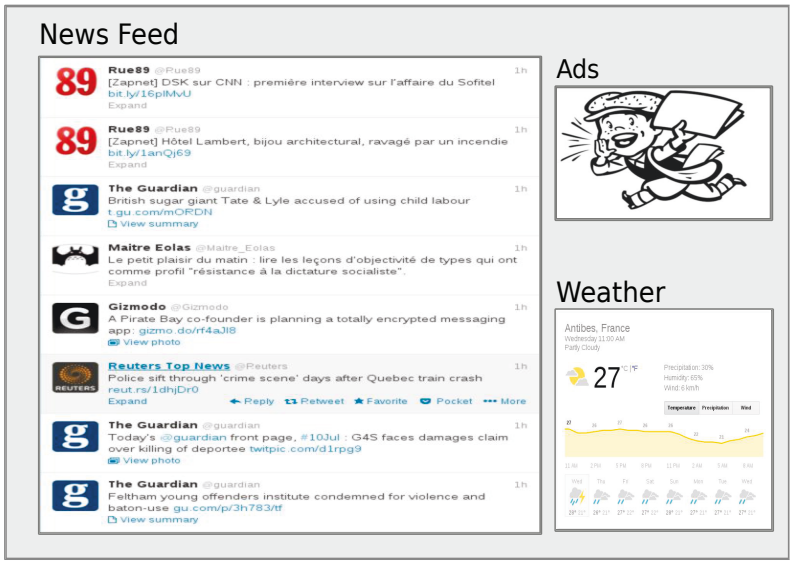


Figure 5.3: Twitter Application

We do not care about how often we receive new advertisements or weather updates. So, the security level of "new_ads" and "update_weather" is set to public (low).

Besides events, also functions need to be given a security level. The functions *print_ads* and *print_weather* have no argument and what they publish is something public, hence the security level of these functions is from public to public (we note $\bot \to \bot$). The function *set_location* has no argument but it uses the location which has the security level l, between public and private ($\bot < l < \top$). Therefore the security level of *set_location* is $\bot \to l$. The last function used in this example is *print_news* which has no argument and prints the new tweets, which is something totally private. Thus, the security level of this function is from public to private (we note $\bot \to \top$). This example is well typed in our type system which entails that no information leaks can appear.

Let us now consider the following piece of code and suppose that we add it in parallel with the application code:

$$\texttt{await}\, new\ tweets^{\top};\quad \texttt{generate}\, new\_ads^{\bot};$$

Now, an external observer can infer how frequently the news event is

```
(loop
  (await "new tweets"^⊤;
   print_news ()^{⊥→⊤})
 end)

┆

(loop
  ((await "new_location"^l;
    set_position(get_location())^{⊥→l})
   ┆
   (await "update_weather"^⊥;
    print_weather ()^{⊥→⊥}))
 end)

┆

(loop
  (await "new_ads"^⊥;
   print_ads ()^{⊥→⊥})
 end)
```

Figure 5.4: Code of Twitter Application

generated, thus there is an information leak. This script is rejected by our type system by the Rule (*Seq*), since the security level of `await` *new tweets* is $(\top, \top)$ and the security level of `generate` *new_ads* is $(\bot, \bot)$, and $\top \not\leq \bot$. Even if generating the event *new_ads* is secure by itself and waiting for new tweets is also secure, the combination of these two is not secure.

We have given an everyday example in *SSL*, where the absence of information leaks is guaranteed by our type system.

Part IV

DSLM

Chapter 6

DSL with Memory: The language DSLM

The language *DSLM* (Dynamic Synchronous Language with Memory) presented in this chapter is obtained by adding memory and distribution to *CRL*. The main purpose of *DSLM* is to create a general-purpose language with the ability of taking advantage of multi-core architectures.

Adding memory to *CRL* introduces the possibility of time-dependent errors (e.g. data-races). To prevent this kind of errors we introduce a new level of parallelism called *Agents*. Agents encapsulate the memory and prevent time-dependent errors like data-races. This property is assured by a type system which is presented in the next chapter.

Furthermore memory encapsulation by agents and the definition of sites as synchronized schedulers will allow us to benefit of multi-cores which is fully detailed in the Chapter 8. This part resumes the technical report [12].

DSLM contains four levels of parallelism: scripts, agents, sites and systems. Let us explain each of them:

- Scripts : scripts are the basic parallel components. They are composed by an asymmetric deterministic parallel operator. The syntax of scripts and their semantics is given in Section 6.1.

- Agents: an agent encapsulates a memory and a script. This script can be a parallel script made of several components (the components belonging to the agent) sharing the agent memory. The only parallel components that can access the memory of an agent are the ones belonging to it. An agent can be created during the execution by the

construct `create_agent`. When an agent is created its execution is delayed up to the next instant.

- Sites: a site is a location where execution of agents takes place. Each site runs one or more agents and manages a set of events shared by the site agents. There is no dynamic creation of sites. The model requires the uniqueness of site names. All the agents belonging to the same site are executed in synchronous parallelism, which means they are sharing the same instants and events.

- System: a *system* is composed of a set of sites which are run in real parallelism (asynchronously). Agents can *migrate* in a system from a site to another; the syntax is `migrate to` *site* for moving the executing agent to site *site*. This construct is the main means of communication and synchronization.

Execution of an agent consists in the execution of the agent's script in the context of the agent's memory. Execution of a site consists in the synchronous execution of all the agents belonging to the site, up to a state where they are all suspended or terminated. Then, the end of the current instant is decided and all the events which have not been generated during the instant are considered as absent. During site execution, the agents of the site can communicate and synchronize using the events they dynamically create. The model of *DSLM* is presented in Figure 6.1.

We note that in *DSLM* (as in *DSL*), no means is provided to define functions. However, scripts may call functions defined in a "host" language (the language in which we define our model is called the host language). Functions are required to terminate instantaneously (i.e. in the same instant they are started). On the contrary, the execution of modules can last several instants or even never terminate, but each module should reach a cooperation point during each instant. To assure these properties we are using FunLoft, which provides a means to check them.

The informal language description is given in 6.1. Then, the domains of language are exposed in 6.2. The formal semantics of scripts is given in 6.3 and the semantics properties are given in 6.3.4. At the end, in 6.4, the semantics of sites and systems is described.

6.1 Informal Language Description

This section contains the informal description of the language. First, we describe scripts and expressions. Then, a description of the model execution

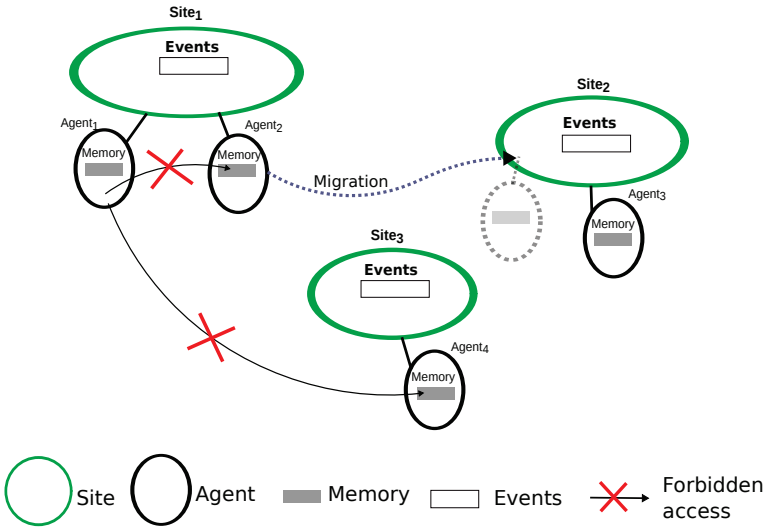


Figure 6.1: DSLM model

semantics is given.

6.1.1 Scripts and Expressions

Scripts are considered in a synchronous context and execution of a script at one instant has two possible outcomes:

- the script is terminated (nothing remains to be executed);
- the script is suspended: either it is waiting for an event to be generated, or it is waiting for the end of instant. In the first case, execution will either resume during the current instant, if the awaited event is generated (thus becoming present), or it will be blocked until the end of the current instant (and then the awaited event is considered as absent).

We describe now the informal semantics of $DSLM$:

- $s_1; s_2$ runs scripts s_1 and s_2 in sequence; execution of s_2 immediately starts when s_1 is terminated;
- $x := exp$ puts the value of exp in the memory location of x, and terminates;

- $s_1 \nmid s_2$ runs the scripts s_1 and s_2 in parallel. The execution is exactly the same as in *CRL*. First, s_1 is executed and yields the control only when its execution is terminated or suspended (*late cooperation*). Once s_1 is suspended or terminated, s_2 is executed and gets suspended as soon as it generates an event which unblocks s_1 (*early cooperation*).

- `let` $x = exp$ `in` s `end` defines a new variable x whose scope is s. A new location is associated to x, and the value of exp is stored in this location. Then, the script behaves like s;

- `cooperate` suspends the execution for the current instant, waiting for the next instant. At the next instant the cooperate instruction is replaced by `nothing`;

- `generate` ev `with` exp generates event ev with the value of the expression exp and terminates;

- `await` ev has no effect and terminates if event ev is present. Otherwise, it suspends execution waiting for the event ev;

- `get_all` ev `in` l stores all the values associated to the generations of event ev during the current instant into the location l; execution is suspended up to the end of instant;

- `do` s `watching` ev executes the script s while event ev is not present. Execution of s is aborted as soon as ev is generated; in case of abortion, execution of the watching statement is suspended up to the end of instant;

- `repeat` exp `do` s `end` runs the script s, n times in sequence, where n is the result of the evaluation of exp;

- `loop` s `end` cyclically runs the script s: execution of s is restarted as soon as it terminates. However, if s terminates instantaneously (i.e. in the same instant it is started), the loop waits for the next instant to restart s. There is thus no possibility to get an *instantaneous loop* which would cycle forever during the current instant, freezing the whole system;

- `launch` $m(ev, exp_1, ..., exp_n)$ launches the module m with the parameter $exp_1, ..., exp_n$. Execution cannot terminate instantaneously. Execution may never terminate;

- `if` *exp* `then` s_1 `else` s_2 `end` runs the script corresponding to the result of evaluation of the expression *exp* (s_1 corresponds to true, and s_2 to false);
- `migrate to` *site* makes a request for moving the executing agent from the current site to the site *site*. Execution of the instruction is suspended for the current instant and resumes after the migration is effective on *site*. Due to parallelism, there can be several migration requests during the same instant (we call this *schizophrenia*). In this case, only the first request is considered and the other ones are ignored;
- `createAgent` s `in` *site* creates a new agent which encapsulates s with an empty memory. The agent is added to the list of agents requesting to be incorporated to *site* and its execution starts when it is effectively incorporated.

Expressions are the following:

- a basic value v;
- a vector of expressions $e\vec{x}p$;
- a variable x whose value is the location associated to x;
- $!\,x$ whose value is the content of the location associated to x;
- `ref` *exp* which returns a new location where the value of *exp* is stored;
- $f(exp)$ which calls the function f with the value of *exp* (it can be a vector) as parameter. Execution of the call starts immediately and is required to be instantaneous, i.e. to terminate instantly.

6.1.2 Agents, Sites and Systems

Execution of an agent consists in the execution of the agent's script in the context of the agent's memory. Execution of an agent terminates when the agent's script is terminated.

Execution of a site consists in the synchronous execution of all the agents belonging to the site, up to a state where they are all suspended or terminated. During each instant, the agents of the site can communicate and synchronize using events.

The end of the current instant is decided at the site level when all the agents belonging to the site are suspended or terminated. Then, the following actions are performed:

- the requests of migration to other sites are processed;
- the agents requesting to be incorporated in the site are actually added to the site;
- the events which have not been generated during the instant are considered as absent and the suspended scripts are reconstructed for the next instant (i.e. `cooperate` is changed in `nothing`);
- finally, the site event set is reset to the empty set.

When these actions are performed, the execution of the site for the next instant can start.

A system is a statically defined set of sites, each of them with a distinct name. Execution of sites is completely asynchronous: sites are chosen and executed in a totally arbitrary way (even in real parallelism). Nothing is shared between different sites and the only means of communication is agent migration.

The semantics of the language (given in the next section) is divided into several levels describing systems, sites, agents, and scripts. The semantics at a given level uses lower levels; for example, the semantics of an agent is based on script semantics.

Here are the main characteristics of the semantics:

1. The semantics of agents and scripts is completely deterministic, even at the level of memory manipulations. In $DSLM$, we are using the same parallel operator as in CRL which is a totally deterministic parallel operator.
2. The semantics of sites is confluent, at the event level. This results from two facts: first, the synchronous execution of agents, and second, the memory encapsulation in agents. With these two elements, it becomes possible to get a confluent semantics for sites: during one instant of a site, not two different event sets can be produced.
3. The semantics of systems is completely *non-deterministic*, which makes possible to model distribution as well as multi-core aspects.

6.1.3 Example

This section describes the coding of a 2D simulation of colliding particles. The simulation is divided in two containers in which particles are moving

and are bouncing against the borders. Collisions are elastic ones. There is a "Migration point" in each container: when a particle falls into a migration point, it migrates into the other container (in the same state).

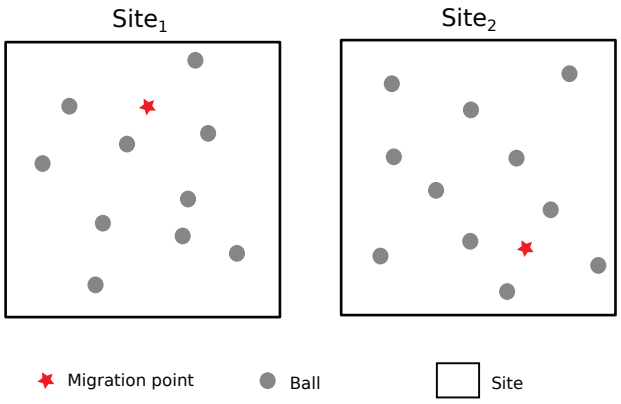


Figure 6.2: Example: Particle collision

The simulation is shown on Figure 6.2. It is made of two sites $site_1$ and $site_2$, one for each container. Initially, each site contains N particles. Each particle is implemented as one agent. As they belong to the same site, all the particles present in a container share the same instants, and communicate their position by generating the shared event `position`. At each instant, a particle with coordinates (x, y) generates event `position` with the couple (x, y) as value. Then, the particle gets all the values of `position` and calculates if there is a potential collision with another particle (using the function `collision`, not described here). Finally, the particle computes its next position (function `next_position`) according to its state.

The script of a particle is a loop whose body is a parallel instruction with three branches. The first branch implements the signaling of position and the passing to the next position previously described. The second branch draws the particle on screen (function `draw`). Finally, the third branch decides if a migration must occur (function `should_migrate`). The code for the first site is described in Figure 6.3.

The code for the second site is similar except that `site_1` and `site_2` are exchanged.

Note that the functions used by the system are defined in the host language (Section 6).

```
createAgent =
  repeat N do
    createAgent =
      let l = ref () in
      let (x, y) = ref random_position() in
      let d = ref random_direction() in
        loop
          generate "position" with (!x, !y);
          get_all "position" in l;
          d := collision(!l, !x, !y, !d);
          (x, y) := next_position(!x, !y, !d)
        ⦔
          draw(!x, !y)
        ⦔
          if (should_migrate_site2(!x, !y)) then
            migrate to site_2
          else
            if (should_migrate_site1(!x, !y)) then
              migrate to site_1
            else
              ()
        end
    in site_1
  end
in site_1
```

Figure 6.3: DSLM Example: Bouncing Balls

The example illustrates several aspects of the model and of the language:

1. Parallel components of agents are *naturally expressed* using the synchronous parallel operator. Each particle is described as an autonomous object which moves accordingly to its defining script and interacts with the other particles through broadcast events.

2. Agents can be executed in real parallelism: two agents belonging to distinct schedulers can be run in real parallelism, by different computing resources. However, if two agents belong to the same site, the efficiency of the real parallelism is moderated by the necessity for the schedulers to synchronize at each end of instant. On the other hand, two agents belonging to different sites can be executed in real parallelism without restriction, which can lead to efficient executions.

3. Each particle has its own memory containing its state (its coordinates), which is shared by all the components of its execution script. The language requires the absence of data-races in memory accesses, which is verified by the type system of Section 7. Thus, there is no risk of a data-races between any two agents, whether on the same site or not.

4. There is no possibility of an instantaneous loop which would prevent execution from passing to the next instant, due to the semantics of the loop operator (rule 6.16).

6.2 Domains

The following disjoint countable sets are defined: **LocName** (locations), **VarName** (variable names), **FunName** (function names), **ModuleName** (module names), **SiteName** (site names) and **EventName** (event names). Each set has an associated function which returns an unused element of the set (for example, each call of the function *new_loc* returns a new unused location in **LocName**).

We use the following notation to define domains:

- $A \times B$ denotes the cartesian product of the domains A and B;
- $A \oplus B$ denotes the disjoint union of the domains A and B;
- $\mathbb{N}^Z$ denotes the multi-set containing the elements of Z;
- $\uplus$ is the union of multi-sets;

- **None** is the domain that contains the unique distinguished element $None$. In the sequel, we do not distinguish between **None** and $None$;
- $\overrightarrow{A}$ denotes the domain of heterogeneous vectors of domain A.
- $A \to B$ is the domain of (partial) functions from A to B.

The set **Basic** is the set of basic values which, for simplicity, is defined as follows:

$$b \in \mathbf{Basic} = \mathbf{Bool} \oplus \mathbf{Integer} \oplus \mathbf{Double} \oplus \mathbf{String}$$

The set **Value** which contains basic values, locations, and vectors of values is defined as:

$$v \in \mathbf{Value} = \mathbf{Basic} \oplus \mathbf{LocName} \oplus \overrightarrow{\mathbf{Value}}$$

Memory

A memory M belonging to **Mem** is a partial function that associates a value with its location or variable.

$$M \in \mathbf{Mem} : \mathbf{VarName} \oplus \mathbf{LocName} \to \mathbf{Value}$$

One notes $M[l \leftarrow v]$ the memory M' defined by: $M'(l) = v$ and for $x \neq l$, $M'(x) = M(x)$. If $M(x)$ is a location, $M[x \leftarrow v]$ is an abbreviation for $M[M(x) \leftarrow v]$.

Events

Elements of **EventEnv** are multi-sets of pairs composed of an event name and an associated basic value:

$$E \in \mathbf{EventEnv} : \mathbb{N}^{(\mathbf{EventName} \times \mathbf{Basic})}$$

To simplify notation, one notes $ev \in E$ if there exists v such that $(ev, v) \in E$.

The function *get_values* is used to collect all the basic values associated with an event ev in a set of events E:

$$get\_values : \mathbf{EventName} \times \mathbf{EventEnv} \to \overrightarrow{\mathbf{Basic}}$$

Expressions

The set **Expr** denotes the expressions and is defined by the following grammar:

$$\begin{array}{lllll} exp \in \mathbf{Expr} ::= & v & \mid x & \mid\, !\, x \\ & \mid e\vec{x}p & \mid \texttt{ref}\ exp & \mid f(e\vec{x}p) \end{array}$$

Scripts

The set **Script** denotes the scripts. This set extends all the constructions from CRL, explained in Section 4.1 and adds memory manipulation constructs to create (*let*) and manipulate the memory (assignment). It also adds the possibility to generate events with values and to collect all the generated values of a particular event (*get_all*).

The **Script** set is defined by:

$$\begin{array}{rl} s \in \mathbf{Script} ::= & \texttt{nothing} \\ \mid & s; s \\ \mid & x := exp \\ \mid & s \nmid s \\ \mid & \texttt{let}\ x = exp\ \texttt{in}\ s\ \texttt{end} \\ \mid & \texttt{cooperate} \\ \mid & \texttt{generate}\ ev\ \texttt{with}\ e \\ \mid & \texttt{await}\ ev \\ \mid & \texttt{get\_all}\ ev\ \texttt{in}\ l \\ \mid & \texttt{do}\ s\ \texttt{watching}\ ev \\ \mid & \texttt{repeat}\ exp\ \texttt{do}\ s\ \texttt{end} \\ \mid & \texttt{loop}\ s\ \texttt{end} \\ \mid & \texttt{launch}\ m(ev, exp_1, ..., exp_n) \\ \mid & \texttt{if}\ exp\ \texttt{then}\ s\ \texttt{else}\ s\ \texttt{end} \\ \mid & \texttt{migrate to}\ site \\ \mid & \texttt{createAgent}\ s\ \texttt{in}\ site \end{array}$$

Agents

The set **Agent** denotes the agents which are triples of the form (s, M, η), where $s \in \mathbf{Script}$, $M \in \mathbf{Mem}$, and η is a migration request:

$$Ag \in \mathbf{Agent} = \mathbf{Script} \times \mathbf{Mem} \times \mathbf{Migr}$$

$$\eta \in \mathbf{Migr} = \mathbf{None} \oplus \mathbf{SiteName}$$

Drop orders code agent migration requests. A drop order can be either the **None** value to indicate the absence of migration request, or a demand for migration of the current agent, or a demand for migration of a newly created agent. The set **D** of drop orders is defined as:

$$d \in \mathbf{D} = \mathbf{None} \oplus \mathbf{SiteName} \oplus (\mathbf{Agent} \times \mathbf{SiteName})$$

A drop order $site \in \mathbf{SiteName}$ is a demand for the migration of the current agent to *site*. A drop order $(Ag, site) \in \mathbf{Agent} \times \mathbf{SiteName}$ is a demand for the migration to *site* of the newly created agent Ag.

We define the combination of two migration requests:

$$\blacktriangleright\colon \mathbf{SiteName} \times \mathbf{Migr} \to \mathbf{SiteName}$$

$$site \blacktriangleright \eta = \begin{cases} site & if\ \eta = None \\ site_\eta & if\ \eta = site_\eta \end{cases}$$

$site \blacktriangleright \eta$ is equal to *site* if it is the first migration request of agent during the current instant. Otherwise, the migration requests are ignored.

Sites

The set **Site** denotes the sites which are quadruples of the form:

$$S \in \mathbf{Site} = \mathbf{SiteName} \times \mathbb{N}^{\mathbf{Agent}} \times \mathbb{N}^{\mathbf{Agent}} \times \mathbf{EventEnv}$$

The site $(site, \mathcal{A}, \mathcal{I}, E)$ is interpreted as follows:

- *site* is the name of the site;
- $\mathcal{A}$ is the multi-set of the agents running in the site;
- $\mathcal{I}$ is the multi-set of agents which will be incorporated in the site at the next instant.
- E is the multi-set of the events generated in the site.

Let $S = (site, \mathcal{A}, \mathcal{I}, E)$ be a site; one notes $sn(S) \subseteq \mathbf{SiteName}$ the set of site names occurring in $\mathcal{A}$.

Systems

The set $\Sigma \in \mathbf{Sys}$ denotes the systems which are sets of sites:

$$\Sigma = \{S_1, ..., S_n\}$$

One says that a system $\Sigma = \{S_1, ..., S_n\}$ where $S_i = (site_i, \mathcal{A}_i, \mathcal{I}_i, E_i)$ is *well-formed* if the following two requirements are fulfilled:

1. No two sites have the same name:
   $\forall i, j \in \{1, \ldots, n\} : \quad i \neq j \;\Rightarrow\; site_i \neq site_j$;
2. The target of a migration is always defined:
   $\forall i \in \{1, \ldots, n\} : \quad site \in sn(S_i) \;\Rightarrow\; \exists j \,.\, site = site_j$.

Reconditioning Function

A reconditioning function Ω (Section 6.4.3) is used to prepare an agent for the next instant. This function is similar to what we previously presented in Section 4.2 in CRL over scripts. Its domain is:

$$\Omega : \mathbf{Agent} \times \mathbf{EventEnv} \rightarrow \mathbf{Agent}$$

6.3 Semantics of Scripts

This section presents the semantics of scripts. First, in Section 6.3.1 the semantics of expressions is defined. Then, the suspension predicate for scripts is presented in Section 6.3.2. Finally, the transition relation which defines the small-step semantics of scripts is defined in Section 6.3.3.

6.3.1 Expressions

The evaluation of an expression is noted:

$$exp, M \rightsquigarrow v, M' \tag{6.1}$$

where:

- exp is the initial expression;
- M is the memory in which the expression exp is evaluated;
- v is the result of the evaluation of exp;
- M' is the new memory after the evaluation of exp;

Evaluation of expressions is defined by the following rules:

- A value evaluates to itself:

$$v, M \rightsquigarrow v, M \tag{6.2}$$

- To access a variable, the variable must denote a location; the evaluation returns the value stored in it:

$$!\, x, M \rightsquigarrow M(x), M \tag{6.3}$$

- The elements of a vector of expressions are evaluated in increasing order:

$$\frac{exp_i, M_i \rightsquigarrow v_i, M_{i+1}}{(exp_1, ..., exp_n), M_1 \rightsquigarrow (v_1, ..., v_n), M_{n+1}} \tag{6.4}$$

- Evaluation of `ref` exp returns a new location in which the value of exp is stored:

$$\frac{exp, M \rightsquigarrow v, M' \quad l = new\_loc()}{\texttt{ref}\ exp, M \rightsquigarrow l, M'[l \leftarrow v]} \tag{6.5}$$

- Evaluation of a function call should be instantaneous. The only changes in the memory are the ones resulting from the evaluation of the arguments:

$$\frac{\overrightarrow{exp}, M \rightsquigarrow \overrightarrow{v}, M' \quad f(\overrightarrow{v}) = v'}{f(\overrightarrow{exp}), M \rightsquigarrow v', M'} \tag{6.6}$$

6.3.2 Suspension Predicate

Reactive programs suspend execution either waiting for events which are not already produced, or waiting for the end of current instant. We are going to extend the *suspension predicate* of scripts presented in CRL in Section 4.2 (noted ‡). Here, we only present the clause for the newly added construct *get_all* which is suspended in all environments:

$$\langle \texttt{get\_all}\ ev\ \texttt{in}\ x, E\rangle ‡ \tag{6.7}$$

The suspension predicate of scripts is naturally extended to agents:

$$\frac{\langle s, E\rangle ‡}{\langle (s, M, \eta), E\rangle ‡} \tag{6.8}$$

A site $S = (site, \mathcal{A}, \mathcal{I}, E)$ is suspended if all its agents are suspended or terminated (an agent is terminated if its script is):

$$\frac{\forall Ag \in \mathcal{A} \quad \langle Ag, E\rangle ‡ \quad \vee \quad Ag = (\texttt{nothing}, M, \eta)}{(site, \mathcal{A}, \mathcal{I}, E)\, ‡}$$

The predicate ♮ indicates the absence of migration request in a site:

$$\frac{\forall (s, M, \eta) \in \mathcal{A} \quad \eta = None}{(site, \mathcal{A}, \mathcal{I}, E)♮}$$

The transition of a site to the next instant will be described by the Rule 6.31. The end of current instant of a site is only possible when the two previous predicates are valid.

6.3.3 Transition Relation

The small-step semantics of scripts is presented as a set of rewriting rules. The general format of a script transition is:

$$\langle s,\ E,\ M\rangle \xrightarrow{d} \langle s',\ E',\ M'\rangle$$

where:

- s is the script which is rewritten;
- E is a multi-set of pairs (ev, v) where ev is a generated event and v is its associated value.
- s' is the *residual* script (what remains to be done at the next step);
- E' is the multi-set of events generated during the rewriting of script s, coupled with their values;
- M is the memory in which s is rewritten;
- M' is the new memory obtained after the rewriting of s;
- d is a drop order indicating if a migration request has been issued from the rewriting of s, and if it is the case, the nature of the request.

The semantics of instructions is given by the following rules and we describe the newly added rules compare to CRL:

Sequence

$$\frac{\langle s_1,\, E,\, M\rangle \xrightarrow{d} \langle s_1',\, E',\, M'\rangle}{\langle s_1; s_2,\, E,\, M\rangle \xrightarrow{d} \langle s_1'; s_2,\, E',\, M'\rangle} \tag{6.9}$$

$$\langle \texttt{nothing}; s_2,\, E,\, M\rangle \xrightarrow{\textbf{None}} \langle s_2,\, E,\, M\rangle \tag{6.10}$$

Parallel

$$\frac{\langle s_1,\, E,\, M\rangle \xrightarrow{d} \langle s_1',\, E',\, M'\rangle}{\langle s_1 \nmid s_2,\, E,\, M\rangle \xrightarrow{d} \langle s_1' \nmid s_2,\, E',\, M'\rangle} \tag{6.11}$$

$$\frac{\langle s_1, E\rangle \ddagger \quad \langle s_2,\, E,\, M\rangle \xrightarrow{d} \langle s_2',\, E',\, M'\rangle}{\langle s_1 \nmid s_2,\, E,\, M\rangle \xrightarrow{d} \langle s_1 \nmid s_2',\, E',\, M'\rangle} \tag{6.12}$$

$$\langle \texttt{nothing} \nmid s,\, E,\, M\rangle \xrightarrow{\textbf{None}} \langle s,\, E,\, M\rangle \tag{6.13}$$

Let

$$\frac{e, M \rightsquigarrow v, M'}{\langle \texttt{let}\ x = exp\ \texttt{in}\ s\ \texttt{end},\ E,\ M\rangle \xrightarrow{\textbf{None}} \langle s,\ E,\ M'[x \leftarrow v]\rangle} \tag{6.14}$$

A let instruction declares a variable x in the scope of a script s; the initial value of x is obtained by evaluating an expression exp

Assignment

$$\frac{exp, M \rightsquigarrow v, M'}{\langle x := exp,\ E,\ M\rangle \xrightarrow{\textbf{None}} \langle \texttt{nothing},\ E,\ M'[x \leftarrow v]\rangle} \tag{6.15}$$

An assignment puts a new value in the memory location associated to a variable. The type system of Chapter 7 insures that the value of the variable is always a location.

Loop

$$\langle \texttt{loop}\ s\ \texttt{end},\ E,\ M\rangle \xrightarrow{d} \langle (s \nmid \texttt{cooperate}); \texttt{loop}\ s\ \texttt{end},\ E',\ M'\rangle \tag{6.16}$$

Generate

$$\frac{exp, M \rightsquigarrow v, M' \quad E' = E \uplus \{(ev, v)\}}{\langle \texttt{generate}\ ev\ \texttt{with}\ exp,\ E,\ M\rangle \xrightarrow{\textbf{None}} \langle \texttt{nothing},\ E',\ M'\rangle} \tag{6.17}$$

A generate instruction produces an event in the environment and associates a value obtained from the evaluation of an expression to this production. The pair made of the event and its value is added in the event environment which is a multi-set. Thus, several productions of the same event with the same value are possible during the same instant.

Await

$$\frac{ev \in E}{\langle \texttt{await}\ ev,\ E,\ M\rangle \xrightarrow{\textbf{None}} \langle \texttt{nothing},\ E,\ M\rangle} \tag{6.18}$$

There is no rule corresponding to an event which is not present. In this case the instruction is suspended: $\langle \texttt{await}\ ev, E\rangle\ddagger$.

Watching

$$\frac{\langle s,\, E,\, M\rangle \xrightarrow{d} \langle s',\, E',\, M'\rangle}{\langle \texttt{do}\ s\ \texttt{watching}\ ev,\, E,\, M\rangle \xrightarrow{d} \langle \texttt{do}\ s'\ \texttt{watching}\ ev,\, E',\, M'\rangle} \tag{6.19}$$

$$\langle \texttt{do nothing watching}\ ev,\, E,\, M\rangle \xrightarrow{\textbf{None}} \langle \texttt{nothing},\, E,\, M\rangle \tag{6.20}$$

Module Call

$$\frac{\overrightarrow{exp}, M \rightsquigarrow \overrightarrow{v}, M' \quad m(ev, \overrightarrow{v}) \Uparrow}{\langle \texttt{launch}\ m\,(ev, exp_1, \ldots, exp_n),\, E,\, M\rangle \xrightarrow{\textbf{None}} \langle \texttt{await}\ ev,\, E,\, M'\rangle} \tag{6.21}$$

Repeat

$$\frac{exp, M \rightsquigarrow n, M'}{\langle \texttt{repeat}\ exp\ \texttt{do}\ s\ \texttt{end},\, E,\, M\rangle \xrightarrow{\textbf{None}} \langle \overbrace{s; \ldots; s}^{n\ times},\, E,\, M'\rangle} \tag{6.22}$$

If

$$\frac{exp, M \rightsquigarrow tt, M'}{\langle \texttt{if}\ exp\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{end},\, E,\, M\rangle \xrightarrow{\textbf{None}} \langle s_1,\, E,\, M'\rangle} \tag{6.23}$$

$$\frac{exp, M \rightsquigarrow ff, M'}{\langle \texttt{if}\ exp\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{end},\, E,\, M\rangle \xrightarrow{\textbf{None}} \langle s_2,\, E,\, M'\rangle} \tag{6.24}$$

The fact that the rewriting of the chosen branch is delayed to a future execution step is an essential feature of the small-step semantics of scripts, as previously discussed in Section 6.1.2.

Agent Creation

$$\langle \texttt{createAgent}\ s\ \texttt{in}\ site,\ E,\ M\rangle \xrightarrow{(s,\emptyset,\mathbf{None})\downarrow site} \langle \texttt{nothing},\ E,\ M\rangle \quad (6.25)$$

An agent creation produces a drop order $Ag \downarrow site$ to demand the migration in *site* of a new agent Ag containing a script s and a new empty memory $\emptyset$. The absorption of the newly created agent by the system is described in Rule (6.28).

Agent Migration

$$\langle \texttt{migrate to}\ site,\ E,\ M\rangle \xrightarrow{site} \langle \texttt{cooperate},\ E,\ M\rangle \quad (6.26)$$

A migration instruction produces a drop order *site* to demand the migration in *site* of the executing agent, and suspends up to the end of the current instant. The processing of the migration request is described in Rule (6.30) .

6.3.4 Semantic Properties

In this section we consider determinism and reactivity of scripts.

Theorem 6.3.1 (Determinism) *For any script s, event environment E and memory M, there is only one possible transition.*

Proof By inspecting the suspension and transition rules, it is immediate to see that at most one rule applies to each configuration $\langle s, E, M\rangle$. □

Definition 6.3.1 *The* multi-step transition relation $\langle s, E, M\rangle \overset{D}{\Rightarrow} \langle s', E', M'\rangle$ *is defined as follows:*

$$\langle s, E, M\rangle \overset{\mathbf{None}}{\Rightarrow} \langle s, E, M\rangle$$

$$\langle s,\ E,\ M\rangle \xrightarrow{\delta} \langle s',\ E',\ M'\rangle \quad \wedge \quad \langle s', E', M'\rangle \overset{D}{\Rightarrow} \langle s'', E'', M''\rangle \quad \Rightarrow \quad \langle s, E, M\rangle \overset{d\cdot D}{\Rightarrow} \langle s'', E'', M''\rangle$$

Definition 6.3.2 (Instantaneous convergence) *The instantaneous convergence of a script, noted* $\Downarrow_D$*, is defined by:*

$$\langle s, E, M\rangle \Downarrow_D \langle s', E', M'\rangle \quad \text{if} \quad \begin{array}{l}\langle s, E, M\rangle \stackrel{D}{\Rightarrow} \langle s', E', M'\rangle \quad \wedge \\ (s' = \texttt{nothing} \ \vee \ \langle s', E'\rangle \ddagger)\end{array}$$

$$\langle s, E, M\rangle \Downarrow_D \quad \text{if} \quad \exists s', E', M' \,.\, \langle s, E, M\rangle \Downarrow_D \langle s', E', M\rangle$$

The relation $\langle s, E, M\rangle \stackrel{D}{\Rightarrow} \langle s', E', M'\rangle$ defines the overall effect of the program s within an instant, starting with the set of events E and the memory M. At the end of instant the script s becomes s' and the new set of events and memory are E' and M'.

We note $\langle s, E, M\rangle \Downarrow_D$ in the same case where we are not interested in the residual script.

We can now state the central theorem of this section.

Theorem 6.3.2 *(***Script reactivity***)* *Let* $s \in$ **Script**. *Then :*

$$\forall E \subseteq \textbf{EventEnv}. \quad (\exists n \leq size(s), \exists D.\langle s, E, M\rangle_n \Downarrow_D)$$

Proof The proof of reactivity for *DSLM* is similar to the one of *CRL* given in Section 4.3. We need to extend the definition of size to the new constructs which are added in DSLM. The newly added construct cases are trivial, therefore omitted. □

6.4 Semantics of Sites and Systems

The small-step semantics of sites and systems is given in this section. The small-step execution of a site during one instant is described by the first three rules (6.27)-(6.29). The next two rules deal with end of instants. Migration requests are processed in Rule (6.30). The passing to the next instant is described by Rule (6.31). Finally, the transformation of the suspended terms for the next instant is described in Section 6.4.3.

The format for system rewriting is:

$$\Sigma \rightarrow \Sigma'$$

where Σ and Σ' are systems.

6.4.1 Sites

There are three rules for defining system rewriting. These rules describe the choice of a site S, the choice of an agent Ag in S, and the execution of Ag in the event environment of S.

- The first rule considers the case where no drop order is issued from the agent execution:

$$\frac{S = (site, \mathcal{A} \uplus (s, M, \eta), \mathcal{I}, E) \quad \langle s,\, E,\, M\rangle \xrightarrow{\textbf{None}} \langle s',\, E',\, M'\rangle}{\Sigma[S] \quad \rightarrow \quad \Sigma[(site, \mathcal{A} \uplus (s', M', \eta), \mathcal{I}, E')]} \tag{6.27}$$

  After execution, the agent is reintegrated in the site and the site event environment is replaced by the produced event set.

- The second rule corresponds to the production of the drop order of a new agent Ag in the target site. Agent Ag is put in the set of agents requesting to be incorporated into $site_2$:

$$\frac{\begin{array}{c}\langle s,\, E_1,\, M\rangle \xrightarrow{Ag \downarrow site_2} \langle s',\, E_1',\, M'\rangle \\ S_1 = (site_1, Ag_1 \uplus (s, M, \eta), \mathcal{I}_1, E_1) \quad S_2 = (site_2, Ag_2, \mathcal{I}_2, E_2) \\ S_1' = (site_1, Ag_1 \uplus (s', M', \eta), \mathcal{I}_1, E_1') \quad S_2' = (site_2, Ag_2, \mathcal{I}_2 \uplus Ag, E_2)\end{array}}{\Sigma[S_1][S_2] \rightarrow \Sigma[S_1'][S_2']} \tag{6.28}$$

- The third rule corresponds to the production of a migration request $site_0$ for the current agent. There are two cases: either a migration request is already present in the agent, and then $site_0$ is simply ignored (a way to prevent schizophrenia situation, where a script wants to migrate to several distinct sites at the same time); or, there is no previous migration request in the agent, and then $site_0$ becomes the agent migration request:

$$\frac{S = (site, \mathcal{A} \cup (s, M, \eta), \mathcal{I}, E) \quad \langle s,\, E,\, M\rangle \xrightarrow{site_0} \langle s',\, E',\, M'\rangle}{\Sigma[S] \quad \rightarrow \quad \Sigma[(site, \mathcal{A} \cup (s', M', site_0 \blacktriangleright \eta), \mathcal{I}, E')]} \tag{6.29}$$

6.4.2 End of Instants

The end of the current instant of a site is decided when the site is suspended, that is when all its agents are suspended. In this case, the site decides the end of the current instant, and can start the next instant.

Two rules are needed to process suspended sites. The first one considers migration requests and the second prepares the site for the next instant. In both cases, suspended agents are transformed to take into account absent events and the passing to the next instant. These two transformations are defined using the function Ω described in Section 6.4.3.

- The first rule considers the case where an agent Ag_1 of site $site_1$ requests to migrate to site $site_2$. First, suspended instructions of Ag_1 are processed in order to take into account the end of current instant (function Ω); then, the transformed agent is added to the set of agents requesting to be incorporated in $site_2$:

$$\frac{\begin{array}{c} S_1 \ddagger \\ S_1 = (site_1, \mathcal{A}_1 \uplus (s, M, site_2), \mathcal{I}_1, E_1) \quad S_2 = (site_2, \mathcal{A}_2, \mathcal{I}_2, E_2) \\ S'_1 = (site_1, \mathcal{A}_1, \mathcal{I}_1, E_1) \quad S'_2 = (site_2, \mathcal{A}_2, \mathcal{I}_2 \uplus \Omega\,((s, M, \mathbf{None}), E_1), E_2) \end{array}}{\Sigma[S_1][S_2] \quad \rightarrow \quad \Sigma[S'_1][S'_2]} \tag{6.30}$$

- The second rule considers the case where there is no migration request. In this case, suspended instructions are processed in order to take into account the end of current instant, and the agents requesting to be incorporated in the site are added to the agent set. Moreover, the site event environment is reset ($E = \emptyset$):

$$\frac{S \ddagger, S\natural \quad S = (site, \mathcal{A}, \mathcal{I}, E)}{\Sigma[S] \quad \hookrightarrow \quad \Sigma[(site, \Omega\,(\mathcal{A}, E) \cup \mathcal{I}, \emptyset, \emptyset)]} \tag{6.31}$$

In the rule, $\Omega\,(\mathcal{A}, E)$ means:

$$\Omega\,(\mathcal{A}, E) = \{\Omega\,(Ag, E) \quad | \quad Ag \in \mathcal{A}\}$$

6.4.3 Reconditioning Function for Next Instant

The reconditioning function Ω is used at each end of instant in order to reconstruct suspended agents, with regard to an event environment E, and to prepare them for execution at the next instant. To recondition an agent means to transform its script and this reconditioning can possibly modify the agent's memory.

There are four basic cases and two inductive cases for script reconditioning:

- `cooperate` is reconditioned in `nothing`:

$$\Omega\left((\texttt{cooperate}, M, \eta), E\right) = (\texttt{nothing}, M, \eta)$$

- `do` s `watching` ev is reconditioned in `nothing` if ev is present in E; otherwise, $(ev \notin E)$, the instruction is reconditioned in `do` s' `watching` ev where s' is the reconditioned of s in E:

$$\frac{ev \in E}{\Omega\left((\texttt{do}\ s\ \texttt{watching}\ ev, M, \eta), E\right) = (\texttt{nothing}, M, \eta)}$$

$$\frac{ev \notin E \quad \Omega\left((s, M, \eta), E\right) = (s', M', \eta)}{\Omega\left((\texttt{do}\ s\ \texttt{watching}\ ev, M, \eta), E\right) = (\texttt{do}\ s'\ \texttt{watching}\ ev, M', \eta)}$$

- `await` ev is reconditioned in itself:

$$\Omega\left((\texttt{await}\ ev, M, \eta), E\right) = (\texttt{await}\ ev, M, \eta)$$

- `get_all` ev `in` l is reconditioned in `nothing`; moreover, the values associated with ev in E are collected in a list which is assigned to l. Note that this is the only reconditioning step that possibly modifies the agent memory.

$$\Omega\left((\texttt{get\_all}\ ev\ \texttt{in}\ x, M, \eta), E\right) = (\texttt{nothing}, M[x \leftarrow \mathit{get\_values}(ev, E)], \eta)$$

- $s_1; s_2$ and $s_1 \nmid s_2$ are the inductive cases:

$$\frac{\Omega\,((s_1, M, \eta), E) = (s_1', M', \eta)}{\Omega\,((s_1; s_2, M, \eta), E) = (s_1'; s_2, M', \eta)}$$

$$\frac{\Omega\,((s_1, M, \eta), E) = (s_1', M_1, \eta) \quad \Omega\,((s_2, M_1, \eta), E) = (s_2', M_2, \eta)}{\Omega\,((s_1 \nmid s_2, M, \eta), E) = (s_1' \nmid s_2', M_2, \eta)}$$

The semantics of sites and systems is now completed and we are going to describe in the next chapter the type system to prevent the occurrence of data-races.

Chapter 7

Typing System for DSLM

In this chapter, we propose a type system whose purpose is twofold: first, to insure that values are correctly used; this is traditional type checking, to verify for instance that in `if` exp `then` s_1 `else` s_2 `end`, exp is a boolean expression; second, that no data-race occurs. For example, consider the following fragment:

```
let x = ref exp1 in
  createAgent !x in remote;
  x:=exp2;
end
```

There is a data-race as x is read by an agent belonging to site `remote`, while it is written in the current site. To prevent this kind of errors, the type system checks that a reference not belonging to an agent's memory cannot be accessed by the agent.

A *type* is either a basic type (*int, bool,* etc), or a reference on a *type*:

$$Basic ::= bool \mid unit \mid int \mid string$$

$$\tau ::= Basic \mid ref\ \tau \mid \overrightarrow{\tau}$$

A typing environment Γ is a possibly empty set[1] of elements of the form $x : \tau$, where x is a variable and τ is a type:

[1] In the sequel, the brackets of the standard set notation are omitted.

$$\Gamma ::= x_1 : \tau_1, \cdots, x_n : \tau_n$$

The general form of a typing judgment is:

$$\Gamma \vdash s : \tau \tag{7.1}$$

where :

- Γ is the typing environment;
- s is the script to be typed;
- τ is the type of s.

Typing Rules

A value has a unique type τ:

$$\Gamma \vdash v : \tau \tag{7.2}$$

To type a vector of expressions we should type each of the expressions belonging to the vector:

$$\frac{\overrightarrow{exp} = (exp_1, ..., exp_n) \quad \forall i \in \{1, ..., n\} \quad \Gamma \vdash exp_i : \tau_i}{\Gamma \vdash \overrightarrow{exp} : (\tau_1, ..., \tau_n)} \tag{7.3}$$

To type an assignment we should check the expression is well typed and the variable x is already defined in the typing environment:

$$\frac{\Gamma \vdash exp : \tau \quad \Gamma \vdash x : \texttt{ref}\ \tau}{\Gamma \vdash x := exp : \texttt{unit}} \tag{7.4}$$

To type a let statement defining a variable x as exp in s, we should first type the expression exp by τ_1; then, the script s should be typed in the new environment in which x has type τ_1:

$$\frac{\Gamma \vdash exp : \tau_1 \quad \Gamma \cup x : \tau_1 \vdash s : \tau_2}{\Gamma \vdash \texttt{let}\ x = exp\ \texttt{in}\ s\ \texttt{end} : \tau_2} \tag{7.5}$$

We suppose that the type of any function is known by the type system. The type of a function consists of the type of the function's arguments and the type of the result. To type a function call, the arguments are type checked and the call receives the type of the function result:

$$\frac{\Gamma \vdash \overrightarrow{exp} : \vec{\tau} \quad f : \vec{\tau} \rightarrow \tau' \quad \vec{\tau} \in \overrightarrow{\mathbf{Basic}} \wedge \tau' \in \mathbf{Basic}}{\Gamma \vdash f(\overrightarrow{exp}) : \tau'} \tag{7.6}$$

The type of an accessed variable must be a reference type present in the typing environment:

$$\frac{\Gamma \vdash x : \texttt{ref}\ \tau}{\Gamma \vdash ! x : \tau} \tag{7.7}$$

A reference is typed by typing the expression it points to:

$$\frac{\Gamma \vdash exp : \tau}{\Gamma \vdash \texttt{ref}\ exp : \texttt{ref}\ \tau} \tag{7.8}$$

To type a sequence, both branches must be typed:

$$\frac{\Gamma \vdash s_1 : \tau_1 \quad \Gamma \vdash s_2 : \tau_2}{\Gamma \vdash s_1\ ;\ s_2 : \texttt{unit}} \tag{7.9}$$

To type a conditional, the condition should be typed to a boolean, then the two branches s_1 and s_2 should be typed:

$$\frac{\Gamma \vdash exp : bool \quad \Gamma \vdash s_1 : \tau_1 \quad \Gamma \vdash s_2 : \tau_1}{\Gamma \vdash \texttt{if}\ exp\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{end} : unit} \tag{7.10}$$

To type a `repeat` exp `do` s `end` statement, the expression exp should be typed as an integer, before typing s:

$$\frac{\Gamma \vdash exp : \texttt{int} \quad \Gamma \vdash s : \tau}{\Gamma \vdash \texttt{repeat}\ exp\ \texttt{do}\ s : \texttt{unit}} \tag{7.11}$$

To type a loop, one types its body:

$$\frac{\Gamma \vdash s : \tau}{\Gamma \vdash \texttt{loop}\ s\ \texttt{end} : \texttt{unit}} \tag{7.12}$$

To type a parallel statement, one types both branches in the same environment:

$$\frac{\Gamma \vdash s_1 : \tau_1 \quad \Gamma \vdash s_2 : \tau_2}{\Gamma \vdash s_1 \nmid s_2 : \texttt{unit}} \tag{7.13}$$

We suppose that the type of any module is known by the type system. The type of a module contains the type of the module's arguments and the type of the result is always `unit`. To type a module call, the arguments are type checked and the call is typed as `unit`:

$$\frac{\Gamma \vdash \overrightarrow{exp} : \overrightarrow{\tau} \quad m : \vec{\tau} \rightarrow \texttt{unit} \quad \overrightarrow{\tau} \in \overrightarrow{\mathbf{Basic}}}{\Gamma \vdash \texttt{launch}\ m\,(ev, exp_1, \ldots, exp_n) : \texttt{unit}} \tag{7.14}$$

The cooperate statement is simply typed in `unit`:

$$\Gamma \vdash \texttt{cooperate} : \texttt{unit} \tag{7.15}$$

To type a generate statement, the associated value should be of a basic type, and the type of the statement is `unit`:

$$\frac{\Gamma \vdash exp : Basic}{\Gamma \vdash \texttt{generate}\ ev\ \texttt{with}\ exp : \texttt{unit}} \tag{7.16}$$

An await statement is simply typed in `unit`:

$$\Gamma \vdash \texttt{await}\ ev : \texttt{unit} \tag{7.17}$$

To type a watching statement means to type its body:

$$\frac{\Gamma \vdash s : \tau}{\Gamma \vdash \texttt{do}\ s\ \texttt{watching}\ ev : \texttt{unit}} \tag{7.18}$$

A *get_all* statement is simply typed in `unit`:

$$\frac{\Gamma \vdash x : \tau \quad \tau \in \textbf{Basic}}{\Gamma \vdash \texttt{get\_all}\ ev\ \texttt{in}\ x : \texttt{unit}} \tag{7.19}$$

To type an agent creation one should type its body in an empty typing environment:

$$\frac{\emptyset \vdash s : \tau}{\Gamma \vdash \texttt{createAgent}\ s\ \texttt{in}\ site : \texttt{unit}} \tag{7.20}$$

This is the central rule to prevent the possibility of data-races in the language.

The migration of the current agent to a site is typed in `unit`:

$$\Gamma \vdash \texttt{migrate to}\ site : \texttt{unit} \tag{7.21}$$

We now consider the issue of data-races (as presented in [52]). Informally, a data-race is the possibility of accessing (read/write) to same memory cell, at the same time, by two different sources.

In our case, to avoid data-races we should prevent an agent to access another agent memory and also to prevent transmit its memory location to other agents. Let us discuss how our semantics and type system ensure this.

At the semantics level, there is a clear separation between agents memories. First, each time a new agent is created, Rule (6.25), the newly created agent is started with an empty memory. Thus, any agent has a separated memory at its creation. The only way to extend its memory, is to use the

Rule 6.5, where the function *new_loc* creates a new fresh location from the set **LocName** and adds it to the agent memory. Therefore, each time a new location is created it cannot be shared by another agent. And at the end, by the help of our type system we assures that an agent can only access a variable ($!x$ and $x := e$) that it has previously created.

Now, we are going to discuss about the possibility of transmission of memory between agents, functions and modules. Since the type system assure that only basic values can be generated (**Basic**), an agent cannot generate any memory location and cannot transmit it to another agent. On the other hand, our type system also requires that the construct which collects the event values (*get_all*) will only get basic values since we can only generate basic values, thus agents cannot receive a memory location either. Finally, since functions and modules are not defined at the language level, we are not allowing agents to transmit memory locations as arguments to functions and modules (arguments should have a basic type). Thus, there is no possible transmission of a memory location from an agent to another.

As an agent can only access the locations that it creates, and cannot transmit a location to another agent, we may conclude that if an agent is typed, it is data-race free.

To give a formal proof of the absence of data-races, we should first introduce agent names, then mark each location with the name of the agent that created it. Next, we should add new rules to generate an error when an agent accesses a location which does not belong to it. At the end, we should show that if a script is well typed, there is no possible execution which leads to the error state. In his thesis [30], Dabrowski gives a formal proof of the absence of data-races, for a framework close to the one considered here.

Chapter 8

DSLM Implementation

In this chapter, we consider the implementation of DSLM and put the focus on multi-processor/multi-core architectures. The main goal is to allow DSLM to maximize the usage of computing resources (processors or cores).

We define an implementation model based on the notion of scheduler: a scheduler executes the agents which are linked to it in a way which is totally transparent to the users.

A site is implemented as a set of schedulers which share the same instants. These schedulers are said to be synchronized (they form a *Synchronized Scheduler*. This notion basically comes from [25]). Since they share the same instants, the agents of the site can use the same events to communicate and synchronize even if they are run by distinct schedulers.

The number of schedulers which implement a site can dynamically change during the execution, according to the load of agents that are present on the site and to the availability of computing resources. Due to the memory isolation of agents, agents can be *transparently redistributed* among the schedulers of the same site, to adapt load balance over it.

Schedulers within a site are supposed to run in real parallelism and to synchronize at the end of each instant (via a synchronization barrier). Each scheduler is executed by a distinct thread (for example, in a Linux-SMP architecture), or by a distinct processor (for example, in a cluster).

If n_1 is the number of sites defined in the system (which is statically fixed) and n_2 is the number of available computing resources (fixed by the machine), we choose to set the number of schedulers to the maximum of n_1 and n_2.

The implementation model is graphically represented in Figure 8.1.

The mapping of the schedulers to the computing resources is not stat-

ht

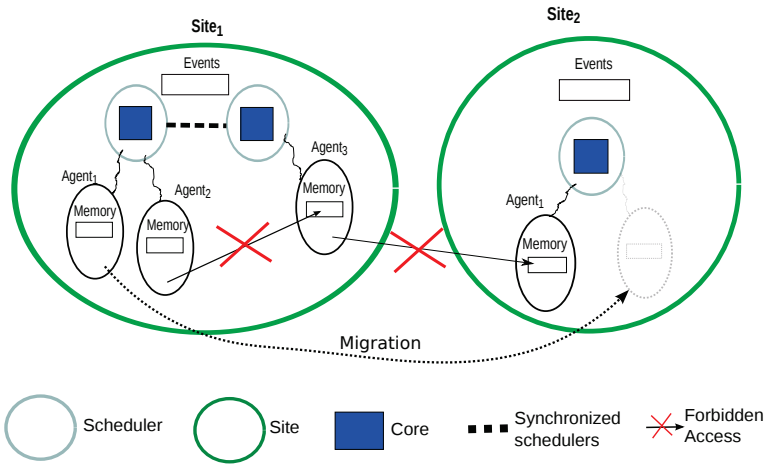


Figure 8.1: DSLM Implementation Model

ically fixed. This characteristic allows the system to use resources in an efficient way. For example, in a multi-core context, the system is free to optimize the mapping of the schedulers to the cores (and consequently the mapping of the agents) in a way that maximizes the use of the cores.

Initially, one scheduler is associated with each site. The remaining schedulers, if any, are the *unused schedulers*. At run time, two actions are possible for a site: first, the activation of an unused scheduler (which thus becomes used); second, the releasing of a scheduler (which becomes unused). The first action is called *site expansion* and the second *site contraction*. The conditions and the moments for performing expansions and contractions are not specified and are left to the implementation, in order to maximize the possibilities of optimization.

The way sites are implemented by sets of synchronized schedulers is formalized through a new implementation semantics described in the next section. This implementation semantics contains rules for site expansion, site contraction, and transparent agent migration within the same site. The rules for sites and systems are similar to those of the semantics of Section 6.4, and therefore omitted.

The rest of the Chapter is structured as follows: first, the implementation semantics is given in 8.1; then the load balancing algorithm is explained in

8.2.

8.1 Implementation Semantics

In this section, we give the implementation domains and explain the main changes between the implementation semantics and the `DSLM` semantics of 6.3. Then, we define the suspension predicate for the schedulers and for the sites. At the end, we give the implementation semantics of sites and systems.

8.1.1 Domains

The new definition of **Site** is:

$$site \in \mathbf{Site} = \mathbf{SiteName} \times \mathbf{SyncSched} \times \mathbb{N}^{\mathbf{Agent}} \times \mathbf{EventEnv}$$

A scheduler is a multi-set of agents and a synchronized scheduler is a multi-set of schedulers:

$$sched \in \mathbf{Sched} = \mathbb{N}^{\mathbf{Agent}}$$

$$\texttt{scheds} \in \mathbf{SyncSched} = \mathbb{N}^{\mathbf{Sched}}$$

8.1.2 Suspension Predicate

The suspension predicate defined in Section 6.3.2 is extended to schedulers and redefined for sites.

A scheduler is suspended if all the agents belonging to it are suspended or terminated:

$$\frac{\forall i. \quad \langle Ag_i, E\rangle \ddagger \quad \vee \quad Ag_i = (\texttt{nothing}, M_i, \eta_i)}{\langle \{Ag_1, \ldots, Ag_m\}, E\rangle \ddagger} \tag{8.1}$$

A set of synchronized schedulers is suspended if all the schedulers belonging to it are suspended:

$$\frac{\forall i. \quad \langle sched_i, E\rangle ‡}{\langle \{sched_1, ..., sched_n\}, E\rangle ‡} \tag{8.2}$$

A site is suspended if the synchronized schedulers in it are suspended:

$$\frac{\langle \texttt{scheds}, E\rangle ‡}{(site, \texttt{scheds}, I, E)\, ‡} \tag{8.3}$$

8.1.3 Sites and Systems

Using the new domains and the previously defined suspension predicate, we present site execution at implementation level. Then, the rules to calculate the end of instants are given. At the end, we give solutions to benefit from the multi-core architecture by expansion/contraction of sites, and transparent migration of agents.

Sites

The three rules 6.27-6.29 are adapted to describe the execution of the script of an agent chosen in a synchronized scheduler of a site.

The first rule corresponds to the absence of migration order (similar to rule 6.27):

$$\frac{sched = sched_0 \uplus (s, M, \eta) \quad \langle s,\ E,\ M\rangle \xrightarrow{\textbf{None}} \langle s',\ E',\ M'\rangle}{\Sigma[(site, \texttt{scheds}[sched], I, E)] \rightarrow \Sigma[(site, \texttt{scheds}[sched_0 \uplus (s', M', \eta)], I, E')]} \tag{8.4}$$

The creation of a new agent is described by the rule (similar to rule 6.28):

$$\frac{\begin{array}{cc} sched = sched_0 \uplus (s, M, \eta) & \langle s,\ E,\ M\rangle \xrightarrow{Ag \downarrow site_2} \langle s',\ E',\ M'\rangle \\ sched' = sched_0 \cup (s', M', \eta) & S' = (site_2, \texttt{scheds}_2, I_2 \cup Ag, E_2) \end{array}}{\begin{array}{c} \Sigma[(site_1, \texttt{scheds}_1[sched], I_1, E_1)][(site_2, \texttt{scheds}_2, I_2, E_2)] \\ \rightarrow \Sigma[(site_1, \texttt{scheds}'_1[sched'], I_1, E'_1)][S'] \end{array}} \tag{8.5}$$

Finally, the migration to another site is described by the rule (similar to rule 6.29):

$$\frac{\begin{array}{c} sched = sched_0 \uplus (s, M, \eta) \quad \langle s,\ E,\ M\rangle \xrightarrow{site'} \langle s',\ E',\ M'\rangle \\ sched' = sched_0 \uplus (s', M', site' \blacktriangleright \eta) \end{array}}{\Sigma[(site, \texttt{scheds}[sched], I, E)] \to \Sigma[(site, \texttt{scheds}[sched'], I, E')]} \tag{8.6}$$

End of Instant

When there exist migration orders, they are executed when the site is suspended (as in Rule 6.30):

$$\frac{\begin{array}{c} sched = sched_0 \uplus (s, M, site_2) \quad S \ddagger \quad S_1 = (site_!, \texttt{scheds}[sched_0], I_1, E_1) \\ S_2 = (site_2, \texttt{scheds}_2, I_2 \cup \Omega(s, M), E_2) \end{array}}{\Sigma[(site_1, \texttt{scheds}_1[sched], I_1, E_1)][(site_2, \texttt{scheds}_2, I_2, E_2)] \to \Sigma[S_1][S_2]} \tag{8.7}$$

The end of the current instant is reached when the site is suspended and there is no migration order to be processed (as in Rule 6.31):

$$\frac{S = (site, Sched, I, E) \quad S \ddagger \quad S\natural \quad S' = (site, \Omega'(\texttt{scheds}) \uplus I, E)}{\Sigma[S] \to \Sigma[S']} \tag{8.8}$$

In this rule, Ω' extends the reconstruction function Ω of Section 6.4.3 and is defined by:

$$\Omega'(sched_1 \uplus ... \uplus sched_n) = \Omega'(sched_1) \uplus ... \uplus \Omega'(sched_n)$$

and:

$$\Omega'(Ag_1 \uplus ... \uplus Ag_m) = \Omega(Ag_1) \uplus ... \uplus \Omega(Ag_m)$$

Expansion and Contraction

The two site expansion and site contraction actions depend on the number of unused schedulers, which is an integer global to the system. This integer is named `unused_schedulers`. The `free_scheduler`() function returns an arbitrary scheduler chosen among the unused schedulers, turns its state to used, and decrements the counter `unused_schedulers`. Conversely, the `kill_sched` function takes a used scheduler in parameter, turns its state to unused, and increments the counter `unused_schedulers`.

The two rules for site expansion and site contraction use freely the counter `unused_schedulers` and the previous functions.

Expansion of a site adds a new scheduler to the synchronized scheduler to a site:

$$\frac{S = (site, \texttt{scheds}, \mathcal{I}, E) \quad \texttt{unused\_schedulers} > 0 \quad sched = \texttt{free\_scheduler}()}{\Sigma[S] \rightarrow \Sigma[(site, \texttt{scheds} \uplus sched, \mathcal{I}, E)]} \tag{8.9}$$

The removal of a scheduler *sched* in a site cannot occur unless the number of agents of the scheduler, noted $\#sched$, is equal to zero. In this case, contraction means that the scheduler is removed from the site:

$$\frac{\texttt{scheds} = \texttt{scheds}_0 \uplus sched_i \quad \#sched_i = 0 \quad \texttt{kill\_scheduler}(sched_i)}{\Sigma[(site, \texttt{scheds}, \mathcal{I}, E)] \rightarrow \Sigma[(site, \texttt{scheds}_0, \mathcal{I}, E)]} \tag{8.10}$$

Transparent Migration

During execution of a site, the implementation can choose to arbitrarily transfer agents between the schedulers, in particular for optimization purposes. These transfers are called transparent migrations as they do not introduce any change in the execution of agents and are not observable at user level.

Transparent migration simply means to transfer an agent from a scheduler of a site to another scheduler of the same site:

$$\begin{aligned}\Sigma[(site, \texttt{scheds}[sched_1 \uplus Ag][sched_2], \mathcal{I}, E)] \rightarrow \\ \Sigma[(site, \texttt{scheds}[sched_1][sched_2 \uplus Ag], \mathcal{I}, E)]\end{aligned} \tag{8.11}$$

8.2 Load Balancing

Load balancing is a method for distributing workload over multiple CPUs. These CPUs could be over a network, on a computer cluster, or even on a multi-core machine.

In our work, load balancing means to balance the charge between different schedulers of the same site. There are several approaches to load balancing like *work sharing* or *work stealing* [20]. We choose to use a simpler algorithm, which is presented in [64]. This algorithm deploys the threads equally between the computing resources. We adapt this algorithm by replacing threads with agents, to spread them between schedulers.

To prevent oscillating migrations of a few agents between two schedulers, we introduce a threshold used to trigger the load balancing process. The load balancing is not executed at each instant, but only when the number of created agents exceeds the threshold.

Let us explain how the example of Section 6.1.3 works under the implementation semantics.

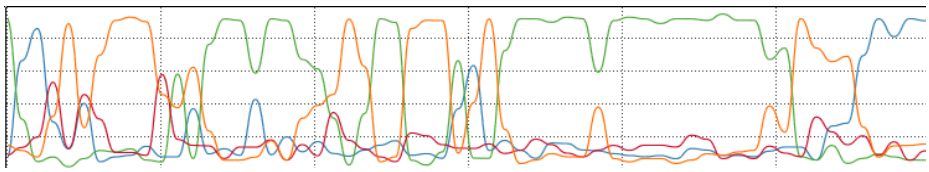
Let us presume that the execution machine is a quad-core machine. Therefore, the number of schedulers is four, which is the maximum between the number of sites and the number of cores.

At the beginning of the execution we have two sites. Each site contains one scheduler which executes all the agents belonging to the site. Since the charge over each site is high, $site_1$ executes the load balancing algorithm, and as free schedulers are available, the site expands itself and adds a new scheduler to its scheduler set. Now, the site contains two schedulers (which are synchronized). Then, the transparent migration between these two schedulers can happen. As a result, the charge of the first scheduler is divided by two, and half of the agents are now linked to the newly added scheduler (same process for $site_2$).

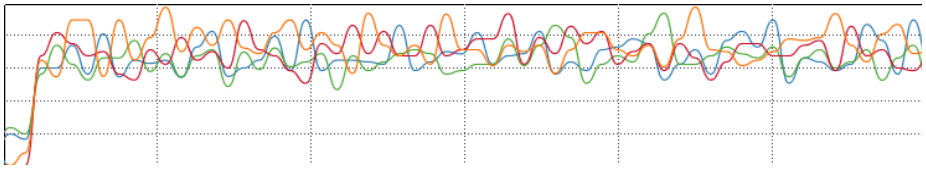
Let us suppose that a considerable amount of agents of $site_2$ migrates to the other site. In this way, the charge over $site_2$ is reduced. Then, first $site_2$ migrates transparently all the agents of one scheduler (`sched`) to the other and frees the scheduler. Afterwards, $site_2$ is contracted by removing `sched`. Now $site_1$ which has a huge load on each scheduler can expand by adding `sched` to its set of schedulers. Each scheduler of $site_1$ gives a third of its load to `sched`. In the resulting state of execution, $site_1$ contains thus three schedulers, and $site_2$ only one. This process continues up to the end of execution.

Now, we are going to compare two executions of this example, one in FunLoft and the other one in DSLM with load balancing. We consider 3000

bouncing balls in each site. The graphics in Figure 8.2 shows an execution of this example over a quad-core machine.



FunLoft Execution



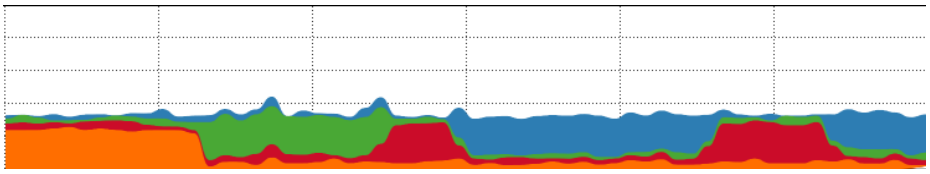
DSLM Execution

Figure 8.2: Two Different Executions of Bouncing Balls

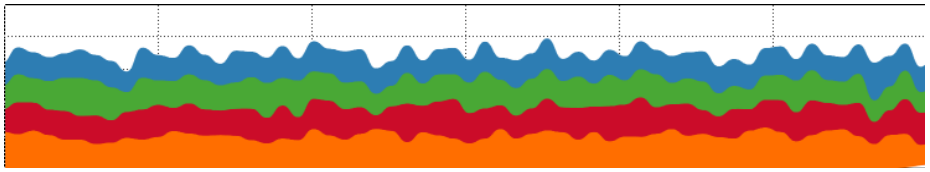
The first image shows the execution of the program in FunLoft where we have two sites and each of them is executed over a native thread which is attached to a core. The load balancing is left to the SMP system. In this execution the core usage is not regular and we are only partially taking benefit of the multi-core architecture. By observing the Figure 8.3, we can see that during the execution, the system only activates one core at each time and that the other three cores are not really used.

On the other hand, the second image shows the DSLM execution where we are using all available cores. The oscillation is the result of load balancing between schedulers. We can see the usage of all cores in the Figure 8.3 where DSLM takes benefit of all the cores at the same time (two times more than FunLoft).

The implementation semantics is in an experimental system available at [1]. The implementation is based on FairThreads and uses the FunLoft verification system.



FunLoft Execution



DSLM Execution

Figure 8.3: Two Different Executions of Bouncing Balls

Part V

Conclusion

Chapter 9

Conclusion and Future Work

We have presented several approaches of parallel programming, based on the synchronous - reactive model in the goal to design a general-purpose language which can take advantage of multi-core architectures and be secure. Synchronous programming is simpler than the traditional asynchronous approaches, based on the exclusive use of preemptive threads. However, four major issues are raised by synchronous programming: how to be sure that the program is indeed reactive? how to execute it efficiently on a multi-core architecture? how to be sure that there is no harmful interference between parallel computations (e.g. data-races)? how to have a secure reactive synchronous language? Our proposal gives answers to these questions.

First, we presented DSL, a core reactive language which is adapted to an orchestration model. In DSL, the memory management is left to the host language. Moreover, there is no means to define functions and modules at the language level; they are left to the host language. In DSL, each site is run by a separated thread linked to a core. For this reason, *DSL* can only partially take benefit of multi-core architectures.

Despite the fact that *DSL* is well-suited for orchestration, there are several issues which can be pointed out. First, the memory abstraction makes programming difficult and sometimes unusable. Particularly, *DSL* cannot be considered as a general-purpose programming language. The second one is at the function call level. There is no proper way to ensure the memory protection. The next issue concerns security. In *DSL*, as in almost all programming languages, there is no means for data protection. The last issue is about taking advantage of the multi-core machines. The *DSL* approach is to map each site to one core which is rather a conservative approach and remains a considerable weakness to optimize the usage of

multi-core machines.

We investigated these issues to propose solutions for them. Since security and optimization of multi-cores are quite orthogonal, we chose to consider as a first step a kernel language CRL which was then extended to deal with each issue. In CRL there is no notion of site nor of host language and data are manipulated directly using a fixed set of basic operators. Moreover, CRL makes use of a new parallel operator which is similar, but not identical to those of [28] and [8]. A big-step semantics cannot allow fine tracking of security violations; due to this we choose to express CRL in a small-step framework. Using this semantics, we prove that the reactivity of scripts is still guaranteed in the presence of the new parallel operator, and we give a static bound for the number of steps it requires.

Then, security levels are added to CRL to obtain SSL (Secure Synchronous Language). SSL is a minimal language proposed to study the problem of information flow security in the synchronous reactive model. To this goal, we first defined two bisimulation properties: a fine-grained bisimulation and a coarse-grained one. Based on these bisimulation, we proposed two non-interference properties. Then, we defined a security type system which we proved to ensure both non-interference properties.

At the end, we added memory and distribution to CRL to obtain a Dynamic Script Language with Memory ($DSLM$). The main goal of $DSLM$ is to provide a general-purpose language which can benefit from multi-core architectures with a sound semantics.

In $DSLM$, by adding memory at the language level, we have to face a new problem called memory safety. To resolve this problem, we propose to encapsulate memory and create a new level of parallelism called agents. We propose a type system which verifies the memory isolation of each agent.

We also proposed an implementation semantics for $DSLM$. The main goal of this semantics is to give a way to benefit from multi-core architectures. To this end, we add a new level of parallelism on top of agents, called schedulers. A scheduler is a native thread which executes agents. In this way, a site becomes a set of schedulers which are sharing the same notion of instant (referred to in the literature as synchronized schedulers). We give the possibility of creating and removing schedulers in each site (contraction and expansion). A site expands (or contracts) itself by taking into account the charge over the site and the available cores. Once the site is expanded we need to take benefit of the newly added scheduler. To this end, we introduce agent migration between schedulers of each site, called transparent migration (due to the transparency at user level). This is possible as a result of the clear memory separation between agents.

Note that the actual implementation of *DSLM* is based on FunLoft, which ensures the termination of functions calls and the reactivity of modules. In this perspective, *DSLM* is a safe reactive parallel programming language, adapted to multi-core/multi-processor architectures, which is, to our knowledge, something new.

We envision the following tracks for future work:

- In previously presented models, functions and modules are defined in the host language, thus there is no insurance that the required properties (instantaneous termination of functions, and non-instantaneous execution of modules) are satisfied. We plan to add function and module definitions directly at the language level. Thus, we could envision to statically check their required properties with a type system.

- We plan to merge *SSL* and *DSLM*. This will help us to reach our main goal to have a general-purpose language equipped with memory and able to deal with security and to take benefit of multi-core architectures. Security would then mean controlling information flows at all levels: memory, events, agent migrations, functions and modules.

- Declassification [62] has been systematically pointed out as a central issue to be dealt with by any approach to secure information flow. We aim at extending our current framework to handle declassification along the lines of [42, 43].

Bibliography

[1] DSLM : Dynamic Script Language with Memory. https://gforge.inria.fr/projects/partout/.

[2] Spark Language. http://www.spark-2014.org/.

[3] Twitter. http://www.twitter.com/.

[4] Martín Abadi, Andrew Birrell, Tim Harris, Johnson Hsieh, and Michael Isard. Dynamic Separation for Transactional Memory. *Tr-2008-43, Microsoft Research*, 2008.

[5] Martín Abadi, Andrew Birrell, Tim Harris, and Michael Isard. Semantics of Transactional Memory and Automatic Mutual Exclusion. *ACM SIGPLAN Notices*, 43(1):63–74, 2008.

[6] Gul Agha. Actors: a Model of Concurrent Computation in Distributed Systems, Series in Artificial Intelligence. *MIT Press*, 11(12):12, 1986.

[7] Ana Almeida Matos. Non-disclosure for Distributed Mobile Code. In *Proceedings FST-TCS'05*, volume 3821 of *Lecture Notes in Computer Science*, pages 177–188, 2005.

[8] Ana Almeida Matos, Gérard Boudol, and Ilaria Castellani. Typing Noninterference for Reactive Programs. *Journal of Logic and Algebraic Programming*, 72(2):124–156, 2007.

[9] Roberto M. Amadio. A Synchronous pi-Calculus. *Journal of Information and Computation*, 205(9):1470–1490, 2007.

[10] Roberto M. Amadio. The SL Synchronous Language, Revisited. *The Journal of Logic and Algebraic Programming*, 70(2):121–150, 2007.

[11] Roberto M. Amadio and Frédéric Dabrowski. Feasible Reactivity for Synchronous Cooperative Threads. In *Workshop on Expressiveness in Concurrency*, pages 33–43, San Francisco, 2006. ENTCS 154(3).

[12] Pejman Attar. DSLM : Dynamic Synchronous Language with Memory. Technical report, 2012. http://hal.archives-ouvertes.fr/hal-00779192.

[13] Pejman Attar and Frédéric Boussinot. Orchestration Synchrone et Au-delà. *Journal européen des systèmes automatisés*, 45(1-3):77–92, 2011.

[14] Pejman Attar, Frédéric Boussinot, Louis Mandel, and Jean-Ferdy Susini. Proposal for a Dynamic Synchronous Language. http://hal.archives-ouvertes.fr/hal-00590420, 2011.

[15] Pejman Attar and Ilaria Castellani. Fine-grained and coarse-grained reactive noninterference. Technical report, July 2013. http://hal.archives-ouvertes.fr/hal-00854136.

[16] Pejman Attar and Ilaria Castellani. Fine-grained and Coarse-grained Reactive Noninterference. 2013. In Proceedings of Trustworthy Global Computing (TGC'13).

[17] Albert Benveniste, Paul Le Guernic, and Christian Jacquemot. Synchronous Programming with Events and Relations: the SIGNAL Language and Its Semantics. *Science of computer programming*, 16(2):103–149, 1991.

[18] Gérard Berry and Laurent Cosserat. The ESTEREL Synchronous Programming Language and Its Mathematical Semantics. In Stephen Brookes, Andrew Roscoe, and Glynn Winskel, editors, *Seminar on Concurrency*, volume 197 of *Lecture Notes in Computer Science*, pages 389–448. Springer Berlin / Heidelberg, 1985.

[19] Gérard Berry and Georges Gonthier. The ESTEREL Synchronous Programming Language: Design, Semantics, Implementation. *Sci. of Comput. Programming*, 19, 1992.

[20] Robert D. Blumofe and Charles E. Leiserson. Scheduling Multithreaded Computations by Work stealing. *J. ACM*, 46(5):720–748, 1999.

[21] Gérard Boudol. ULM: A Core Programming Model for Global Computing. In David Schmidt, editor, *Programming Languages and Systems*, volume 2986 of *Lecture Notes in Computer Science*, pages 234–248. Springer Berlin Heidelberg, 2004.

[22] Gérard Boudol and Ilaria Castellani. Noninterference for Concurrent Programs and Thread Systems. *Theoretical Computer Science*, 281(1):109–130, 2002.

[23] Frédéric Boussinot. Reactive C: An Extension of C to Program Reactive Systems. *Software: Practice and Experience*, 21(4):401–428, 1991.

[24] Frédéric Boussinot. Concurrent Programming with FairThreads: The Loft Language. 2003. http://www-sop.inria.fr/meije/rp/LOFT/doc/book/book.html.

[25] Frédéric Boussinot. FairThreads: Mixing Cooperative and Preemptive Threads in C. *Concurrency and Computation: Practice and Experience*, 18(5):445–469, 2006.

[26] Frédéric Boussinot. *Safe Reactive Programming: The FunLoft Proposal*. Lambert Academic Publishing, 2010.

[27] Frédéric Boussinot and Robert De Simone. The SL Synchronous Language. *Software Engineering, IEEE Transactions on*, 22(4):256–266, 1996.

[28] Frédéric Boussinot and Jean-Ferdy Susini. The SugarCubes Tool Box: A Reactive Java Framework. *Software: Practice and Experience*, 28(14):1531–1550, 1998.

[29] Paul Caspi and Marc Pouzet. A functional extension to lustre. In *8th International Symposium on Languages for Intensional Programming*, 1995.

[30] Frédéric Dabrowski. Programmation Réactive Synchrone: langages et contrôle des ressources. 2007. PhD thesis.

[31] Robert De Simone. Higher-level Synchronising Devices in Meije-SCCS. *Theoretical Computer Science*, 37:245–267, 1985.

[32] Jack B Dennis and Earl C Van Horn. Programming Semantics for Multiprogrammed Computations. *Communications of the ACM*, 9(3):143–155, 1966.

[33] Michel Dubois and Christoph Scheurich. Memory Access Dependencies in Shared-memory Multiprocessors. *Software Engineering, IEEE Transactions on*, 16(6):660 –673, 1990.

[34] J. Eker, J.W. Janneck, E.A. Lee, Jie Liu, Xiaojun Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Yuhong Xiong. Taming Heterogeneity - the Ptolemy Approach. *Proceedings of the IEEE*, 91(1):127 – 144, 2003.

[35] Joseph A Goguen and José Meseguer. Security Policies and Security models. In *IEEE Symposium on Security and privacy*, volume 12, 1982.

[36] Nicholas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous Data Flow Programming Language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

[37] Nicolas Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer Academic Pub., 1993.

[38] Per Brinch Hansen. Concurrent Programming Concepts. *ACM Comput. Surv.*, 5(4):223–245, 1973.

[39] Maurice Herlihy and J. Eliot B. Moss. Transactional Memory: Architectural Support for Lock-free Data Structures. *SIGARCH Comput. Archit. News*, 21(2):289–300, 1993.

[40] Butler W Lampson. Protection. *ACM SIGOPS Operating Systems Review*, 8(1):18–24, 1974.

[41] Louis Mandel and Marc Pouzet. ReactiveML: A Reactive Extension to ML. In *Proceedings of the 7th ACM SIGPLAN international conference on Principles and practice of declarative programming*, PPDP '05, pages 82–93, New York, NY, USA, 2005. ACM.

[42] A Almeida Matos. Typing secure information flow: declassification and mobility. *These de doctorat, École Nationale Supérieure des Mines de Paris*, 2006.

[43] Ana Almeida Matos and Gérard Boudol. On declassification and the non-disclosure policy. In *Proceedings of the 18th IEEE Workshop on Computer Security Foundations*. IEEE, 2005.

[44] Jayadev Misra and William R. Cook. Computation Orchestration. *Software & Systems Modeling*, 6(1):83–110, 2007.

[45] Jay Munro. Antivirus research and detection techniques. *Antivirus Research and Detection Techniques, ExtremeTech*, 2002.

[46] Aaftab Munshi et al. The OpenCL Specification. *Khronos OpenCL Working Group*, 1:l1–15, 2009.

[47] T. Murata. Petri Nets: Properties, Analysis and Applications. *Proceedings of the IEEE*, 77(4):541 –580, 1989.

[48] Jens Muttersbach, Thomas Villiger, and Wolfgang Fichtner. Practical Design of Globally-Asynchronous Locally-Synchronous Systems. In *Advanced Research in Asynchronous Circuits and Systems, 2000. (ASYNC 2000)*, pages 52 –59, 2000.

[49] Andrew C Myers. JFlow: Practical Mostly-static Information Flow Control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 228–241. ACM, 1999.

[50] Andrew C Myers and Barbara Liskov. A decentralized model for information flow control. In *ACM SIGOPS Operating Systems Review*, volume 31, pages 129–142. ACM, 1997.

[51] Andrew C Myers, Lantian Zheng, Steve Zdancewic, Stephen Chong, and Nathaniel Nystrom. Jif: Java Information Flow. *Software release. Located at http://www. cs. cornell. edu/jif*, 2005, 2001.

[52] Robert H. B. Netzer and Barton P. Miller. What Are Race Conditions?: Some Issues and Formalizations. *ACM Lett. Program. Lang. Syst.*, 1(1):74–88, 1992.

[53] Bradford Nichols, Dick Buttlar, and Jacqueline Proulx Farrell. *Pthreads Programming*. O'Reilly, 1996.

[54] CUDA Nvidia. Compute Unified Device Architecture Programming Guide. 2007.

[55] Scott Oaks and Henry Wong. *Java Threads*. O'Reilly Media, Inc., 2004.

[56] Rolf Oppliger. Internet security: Firewalls and beyond. *Communications of the ACM*, 40(5):92–102, 1997.

[57] Rob Pike. Concurrency is not Parallelism (It's Better). http://concur.rspace.googlecode.com/hg/talk/concur.html.

[58] Kuchi VS Prasad. A Calculus of Broadcasting Systems. *Science of Computer Programming*, 25(2):285–327, 1995.

[59] Alejandro Russo and Andrei Sabelfeld. Security for multithreaded programs under cooperative scheduling. In *Perspectives of Systems Informatics*, pages 474–480. Springer, 2007.

[60] Andrei Sabelfeld. The Impact of Synchronization on Secure Information Flow in Concurrent Programs. In *Proceedings of Andrei Ershov 4th International Conference on Perspectives of System Informatics*, 2001.

[61] Andrei Sabelfeld and David Sands. Probabilistic Noninterference for Multi-threaded Programs. In IEEE, editor, *13th Computer Security Foundations Workshop*, 2000.

[62] Andrei Sabelfeld and David Sands. Declassification: Dimensions and Principles. *Journal of Computer Security*, 17(5):517–548, 2009.

[63] Davide Sangiorgi and David Walker. *The pi-calculus: a Theory of Mobile Processes*. Cambridge University Press, 2003.

[64] Behrooz A. Shirazi, Krishna M. Kavi, and Ali R. Hurson. *Scheduling and Load-balancing in Parallel and Distributed Systems*. IEEE Computer Society Press, 1995.

[65] Vincent Simonet. Flow Caml in a Nutshell. In *Proceedings of the first APPSEM-II workshop*, pages 152–165. Nottingham, United Kingdom, 2003.

[66] Geoffrey Smith. A New Type System for Secure Information Flow. In *Proceedings of the 14th IEEE Computer Security Foundations Workshop*. IEEE, 2001.

[67] Geoffrey Smith and Dennis Volpano. Secure Information Flow in a Multi-threaded Imperative Language. In ACM, editor, *Proceedings POPL '98*, pages 355–364. ACM Press, 1998.

[68] Jean-Ferdinand Susini. A Very Experimental Release of the SugarCubes: SugarCubes v5. http://jeanferdysusini.free.fr/index.php?action=SC.

[69] Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. A Sound Type System for Secure Flow Analysis. *Journal of Computer Security*, 4(3):167–187, 1996.

[70] David A Watt, Brian A Wichmann, and William Findlay. *Ada Language and Methodology*. Prentice Hall International (UK) Ltd., 1987.