University of Geneva

Master Thesis

Model Cheking of Gamma Programs with Algebraic Nets

Author:
Dimitri Racordon

Supervisor:
Dr. Didier Buchs

07.09.13

Abstract

Concurrent programming is still an important challenge of computer science. If the design of parallel or distributed algorithm is a difficult task, the complexity of the mechanisms to manage concurrency over communication and resources sharing may even be a harder piece of work. Among the many programming languages, frameworks and formalisms that have already been proposed to address this issue, an interesting one would be the Γ-Model. Based on a chemical metaphor, this multiset rewriting model entails a high level of parallelism, together with an original programming style. In this master thesis, we study the Γ-Model for the purpose of model checking. Namely, we first translate the model into the Algebraic Nets and observe what properties can be verified, with respect to a state space representation. We use AlPiNA, a symbolic model checker for the Algebraic Petri Nets, to implement our translated models and generate such state spaces. Finally, we propose an extension of the Γ-Model to support stochastic rewriting decisions, and we discuss its properties.

Contents

1 Introduction

Concurrent programming is still an important challenge of computer science. If the design of parallel or distributed algorithm is a difficult task, the complexity of the mechanisms to manage concurrency over communication and resources sharing may even be a harder piece of work. Nevertheless, the constant evolution of hardware and computing techniques, together with the exponential growth of applications size, tend to use more and more parallel or distributed architectures. For instance, according to a recent article from the blog website High Scalability [1], the architecture of Twitter deals with 150 million worldwide users for up to 300 thousands queries per seconds. Such constraints require the use of tremendous computing power, often provided by cloud computing where data and computation power is spread over several volatile virtual computers. Managing such architectures is yet another challenge, where compromises have to be made, trading performances for reliability and scalability. More modest applications have to deal with parallelism too; today's desktop computers or even mobile phones embed multi-core processors.

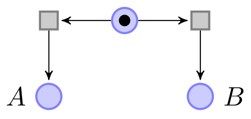

Figure 1.1: modeling of a resource shared between two processes

Many programming languages, frameworks and formalisms have already been proposed in order to address the issue of handling the difficulty of parallel and distributed programs. Among these, an interesting approach may be the use of sets. Well adapted to the manipulation of huge chunks of data, sets allow one to describe a process as a succession of set transformations, thus at a very higher-level than if we had wanted to detail the operations performed on each data, individually. Such paradigms showed a significant popularity in several domains, like search engines for instance. A key advantage is that set programming is generally well suited for massively parallel or distributed machines, such as cloud systems. One notable example is without any doubt MapReduce. This model processes large sets of data by "mapping" the input on several nodes that individually perform some operations on these subsets before they are "reduced" to produce the output. From the side of formal modelization and verification, set programming can be seen as a way to limit the state space combinatorial explosion. Concurrency induced by parallel or distributed computing often produce several paths of execution, accordingly growing the number of possible states. For instance, consider the simple model depicted by a Petri Net in figure 1.1. A resource is shared between two concurrent processes competing to acquire it. Three states are possible, namely 1) neither of the processes acquired the resource yet, 2) process A acquired the resource or 3) process B acquired the resource. The exact same model considered with 100 available resources raises the number of possible states up to 2^{100}, illustrating the problem of combinatorial explosion. One technique to handle this issue is to identify equivalent "paths" of execution. In our example, we could advocate that any path eventu-

[1] http://highscalability.com

ally firing both transitions n times in any order produces the same state, thus drastically reducing the total amount of states to consider. Another technique is the symbolic model checking, which consists of representing the state space symbolically, by the means of clever structures such as data decision diagrams.

In this document, we focus on one model of set programming, namely the Γ-Model, and invistigate on a way to perform model checking on it. We choose the Algebraic Petri Nets, a mathematical modeling language aimed at describing processes, possibly concurrent, as a place/transition system with the use of algebraic terms, to encode the Γ-Model, and rely on previous works in the direction of model checking with Algebraic Petri Nets.

In order to place our work into its context, we will first discuss several approaches involved in sets programming, in particular those that are or can be related to formal modeling. We will then give a deeper introduction to the Γ-model, a multiset rewriting system that relies on a chemical metaphor to express computation in terms of reactions within a set of molecules, and discuss the classes of problems that fit the model. Then we will propose a translation of this model into the Algebraic Petri Nets. Finally we will introduce a possible extension of the Γ-model to handle stochastic decisions, before concluding on several future works directions.

2 Acknowledgement

A lot of people contributed to the achievement of this work. My first thanks go to Prof. Dr. Didier Buchs, who followed me with patience and interest from the beginning, and who provided me with invaluable contribution. I also would like to thank my colleagues from University for their precious advices, especially Dr. Alexis Marechal and Edmundo Lopez.

Great thanks go to my colleague and friend David Lawrence, who spent long hours of discussion on the whiteboard. I also would like to thank David Fisher for his advices on my thesis.

Finally, special thanks go to my parents Yvan and Jessica, my sister Leila, my beloved friend Shiori, and all my other relative and friends for their constant support during my work.

3 State of the Art

3.1 Computation paradigms

In this section, we present a review of some computation paradigms related to the area of set programming. We will classify them into three categories: 1) dataflow programming, 2) coordination programming and 3) multiset rewriting. The choice of these categories is rather arguable; especially one could claim that dataflow and coordination programming share too much concepts to be treated separately. However, to the perspective of set programming, we believe that these categories reflect well enough the different approaches regarding the way data are treated.

One could also wonder why we do not mention Answer Set Programming (ASP) in the above categories. ASP is a paradigm based on a set of rules, the answer set; computation is reduced to seek for a *stable model* complying the answer set for a given problem, generally using SAT solvers. We encourage the reader to consult [1] for a good introduction about ASP. However, we advocate this model does not fit in the field of this document, since we focus on computation paradigms centered around sets of values rather than sets of statements.

3.1.1 Dataflow Programming

Pure Dataflow Model In the 1970s and 1980s, the emergence of massive parallelism motivated researches on an alternative to the dominating "von Neumann" architecture. Major criticisms pointed out that the design of von Neumann hardware, i.e. the use of a global shared memory and a program counter, was the cause of the exponential growth of complexity in parallel programs [2]. Addressing this issue, one popular paradigm is the Dataflow Programming. The definition of the pure dataflow model has been given as early as 1966, by Karp and Miller in [3], who wanted to use a graph representation in order to describe and analyze parallel programs. They gave a model in which computation is a directed graph where instructions correspond to nodes, and arcs represent data dependencies of these instructions. An instruction, i.e. a node is said *fireable* as soon as a data token is available from all its incoming arcs, thus eventually consuming these token and producing new ones in all of its outgoing arcs. As the model makes no assumption neither on the specific time a fireable node will execute, nor on any sort of synchronization between nodes, arcs are defined as *first in first out* (FIFO) queues in order to prevent race conditions among data token. The name *dataflow* is derived from the idea that a *data* may *flow* along the arcs of the graph. This contrasts a lot with the von Neumann execution model in the way that an instruction is scheduled as soon as possible, and not when some program counter reaches it. It also implies that several instructions fireable at the same moment can be executed in parallel, without any concern about concurrent memory access. The absence of any global memory implies a complete locality of the data within a computation graph.

Figure 3.1 shows a short dataflow computation graph, equivalent to a classic

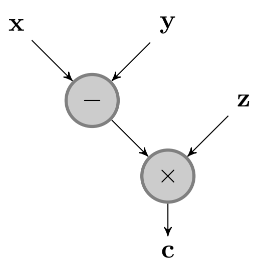

Figure 3.1: Dataflow equivalence of $c = (x - y) \times z$

von Neumann affectation $c = (x - y) \times z$. Nodes of the graph are the variables input x y z, the output c and finally the operators $-$ and $\times$. Arcs represent the dependencies of direct children nodes. Every time a data token is available on the inputs x and y, the node labeled "$-$" is *fireable*, thus eventually executed to produce $x - y$ on its input arc. The node labeled "$\times$" is itself *fireable* as soon as $x - y$ has been produced and a token is available on input z. Recalling that arcs in the dataflow model are FIFO queues, and that nodes can work simultaneously without any form in synchronization, it may be noted that if $x = [x_1, x_2, \ldots, x_n]$, $y = [y_1, y_2, \ldots, y_n]$ and $z = [z_1, z_2, \ldots, z_n]$ are given as sequences of n tokens, the computation will perform $n + 1$ steps to produce a sequence $c = [c_1, c_2, \ldots, c_n]$ of n tokens. As it is free from race any race condition, the model also ensures that $c_1 = (x_1 - y_1) \times z_1$, $c_2 = (x_2 - y_2) \times z_2$, etc. Studied as the *pipelined* dataflow [4], this property allows an important degree of parallelism. Finally, we may emphasize on the functional behavior of computation nodes. Since data are never modified but rather consummed to produce new ones, a dataflow program is completely free from side effects. As a result, given a specific set of inputs, both the behavior and the outputs of a program are completely deterministic.

Hybrid Dataflow In the middle of the 1970s, first dataflow languages started to flourish [5], among with promising hardware architectures [6, 7], contributing to the popularity of the paradigm. In particular, image processing found a huge interest in the pipelined dataflow [8], motivating the development of VLSI microchips, such as the NEC μPD7281 [9]. Despite it was even believed it would replace the von Neumann concepts, further works eventually demonstrated some weaknesses of the dataflow architectures. In his review on the dataflow programming, during the 1970s and 1980s, Veen summarizes most of the objections raised against the dataflow approach, pointing out that the too fine granularity needed to express computation lead to an excessive consumption of resources, compared to the classic sequential alternatives [10]. Two main problems arise from this overhead: in one hand a realistic dataflow machine requires a huge amount of hardware, to store data structure and route it to processors, in the other hand most of the processing power is wasted, either waiting for dependencies, or performing overhead. This issue motivated the development of hybrid architectures, which address the too fine granularity problem by using blocks of sequential processes, interconnected on a dataflow architecture [11, 12]. Starting the 1990s, this concept of hybridity took over the idea of complete orthogonality between the von Neumann and dataflow models and became the dominant area of research in the dataflow community [2]. This broader vision of granularity divided the hybrid dataflow model into two groups [12]. In one hand, the *threaded dataflow* consists of analyzing the computation graph to find sub-graphs that shows a potential sequentiality. Nodes of theses subgraph are then grouped into segments that will be scheduled on only one processing element, avoiding the scheduling overhead. On the other hand, the *large-grain dataflow* consists of replacing the sub-graphs with a high degree of sequentiality by actual sequential processes, potentially expressed with some sequential language. The latter conducted Morrison to introduce the related *flow-based programming* (FBP) paradigm, where sequential "black boxes" exchange information packets by message passing, on a predefined network [13]. Large-grain dataflow has also been studied in the field of the parallelization of sequential programs. In [14] Alves et al. proposed a dataflow execution model to parallelize sequential applications on a dataflow virtual machine. Portions of code identified by the programmer as good candidates for parallelization are translated to *super-instructions* and then fired according to a dataflow model. Balakrishnan and Sohi [15] addressed the issue of finding such parallelizable portions of code by, *speculative parallelization*, using program demultiplexing.

Dataflow Visual Programming Languages Textual languages largely dominated the dataflow community in the 1970s and 1980s, mostly due to the lack of technology for visual programming. However, the advantages of the visual representation of dataflow graphs had already been pointed out [16, 17]. An interesting aspect may that dataflow visual programming languages (DFVPLs) are firstly aimed at software engineering, while their textual counterparts target exploitation of parallelism as a priority. Indeed, the visual meaning of concurrent processes offered by directed-graph representations constitutes a considerable advantage for software engineering [18]. Numbers of DFVPLs were developed during the 1980s [19], including LabView [20, 21], a platform specialized for data analysis tools development, and ProGraph [22, 23], a more general-purpose DFVPL that borrowed a lot of concepts to the Object-Oriented programming. Prograph was eventually withdrawn from the market in the late 1990s, to reborn has Marten in 2003 [24], a visual programming tool for Apple Mac OSX. Nevertheless, both of these products were huge commercial success, proving the interest of visual dataflow approaches for software engineering [25], to the extent that LabView is still used today. Many researches have been conducted on DFVPLs and many languages and platforms have been built for both commercial and academic purposes [21, 24, 26, 27]. However, as this document focuses on set programming modeling, we will not go any further on detailing them.

Dataflow semantics As stated earlier, the dataflow was first modeled to go with a sufficient formal notation so one could reason mathematically about a parallel computation. In [3] Karp and Miller gave the following formal definition to their model:

Definition. *A computation graph G is a directed graph where*

$$\begin{aligned} N = \{n_1, \ldots, n_k\} & \qquad \text{is the set of nodes} \\ D \subseteq N \times N & \qquad \text{is the set of directed arcs} \\ \langle A_p, U_p, W_p, T_p \rangle & \qquad \text{is the parameters of } d_p \in D \end{aligned}$$

We define $in(d_p)$ the node for which the arc d_p is directed out, and $out(d_p)$ the node for which the arc d_p is directed into. The tuple $\langle A_p, U_p, W_p, T_p \rangle$ associates the following parameters to an arc d_p:

i. A_p gives the number of token in the queue of d_p;
ii. U_p gives the number of token added to d_p whenever $in(d_p)$ is fired;
iii. W_p gives the number of token removed from d_p whenever $out(d_p)$ is fired;
iv. T_p is a threshold giving the number of token in d_p which permits the firing of $out(d_p)$.

A node n_i is *fireable* if and only if $A_p \geq T_p$ for each arc d_p directed into n_i. A fireable node n_i will eventually be fired, removing W_p token from each arc d_p directed into it, to produce U_q token in each arc d_q directed out of it. Based on this firing constraint, we give the definition of sequences of firing of a computation graph G. Let ϵ be a sequence of non empty sets $\{S_1, S_2, \ldots, S_t, \ldots\}$ such that each set $S_t \subseteq \{1, \ldots, k\}$, with k the number of nodes in G, denote the firing of nodes n_j for $j \in S_t$ at t. Let $x(j, 0) = 0$ and, for $t > 0$, let $x(j, t)$ denote the number of sets S_m, with $1 \leq m \leq t$, of which j is an element.

Definition. *The sequence ϵ is an execution of G if and only if for all N the following conditions hold:*

i. if $j \in S_{t+1}$ and G has an arc d_p, from n_i to n_j, then $A_p + U_p x(i, t) - W_p x(j, t) \geq T_p$;
ii. if ϵ is finite and of length R, then for each j there exist a node n_i and an arc d_p from n_i to n_j such that $A_p + U_p x(i, R) - W_p x(j, R) < T_p$

Definition. *An execution ϵ of G is called a proper execution if the following implication holds:*

iii. if, for all n_i and for every arc d_p directed from n_i to n_j, $A_p + U_p x(i, t) - W_p x(j, t) \geq T_p$, then $j \in S_R$ for some $R > t$.

Condition *i* ensures that the queue of d_p contains at least T_p token after the successive firings given in $S_1, \ldots, S_t$. Condition *ii* gives the termination condition as the absence of any fireable node. Finally, condition *iii* states that a fireable node will eventually be fired, after a finite number of firings R. An application example of this formalism is given in [3].

Kavi et al. [28] used a simpler model where the FIFO queues associated to the arcs are replaced by *link* nodes, that may contain only one or zero token, while *actor* nodes represent the dataflow instruction. An arc may be directed from a link to an actor, thus representing the actor's inputs, or from an actor to a link, thus representing the actor's outputs. They also observed that, to the perspective of pure theoretical study of the dataflow graph, the meaning of the actors and data token is irrelevent; only the presence of a given token in a given link has an impact on the firing semantic. For this reason, they referred to their model as the *uninterpreted data flow graphs.*

Definition. *An uninterpreted data flow graph* $G = \langle A \cup L, E\rangle$ *is a bipartite labeled graph where the two types of nodes are called actors and links*

$$\begin{array}{ll} A = \{a_1, \dots, a_n\} & \textit{is a set of actors} \\ L = \{l_1, \dots, a_m\} & \textit{is a set of links} \\ E \subseteq (A \times L) \cup (L \times A) & \textit{is a set of arcs} \end{array}$$

A *starting set* $S \subseteq L$ represents the external inputs of the data flow graph.

$$S = \{l \in L \mid (a, l) \notin E, \forall a \in A\}$$

A *terminating set* $T \subseteq L$ represents the external outputs of the data flow graph.

$$T = \{l \in L \mid (l, a) \notin E, \forall a \in A\}$$

The set of input links to an actor a, and the output links from an actor a are denoted by $I(a)$ and $O(a)$.

$$I(a) = \{l \in L \mid (l, a) \in E\}$$
$$O(a) = \{l \in L \mid (a, l) \in E\}$$

$I(l)$ and $O(l)$ are defined similarly for links.

There exists a marking $M : L \to \{0, 1\}$ that maps the absence or presence of a token in a given link. An *initial marking* $M_0 \subseteq S$ gives the input configuration of the system at its initialization. A *final marking* $M_t \subseteq T$ gives the output configuration of the system when it finishes its execution.

Semantic is given in terms of *firing semantic set.* To each actor are attached an *input firing semantic set* F_1 and an *output firing semantic set* F_2.

$$F_1(a, M) \subseteq I(a)$$
$$F_2(a, M) \subseteq O(a)$$

An actor is *fireable* when the following conditions holds.

$$M(l) = 1, \forall l \in F_1(a, M)$$
$$M(l) = 0, \forall l \in F_2(a, M)$$

Firing an actor removes the token from its input firing set $F_1(a, M)$ and produces a token in its output firing set $F_2(a, M)$, thus resulting in a new marking M'.

$$M' = \begin{cases} 0 & \text{if } l \in F_1(a, M) \text{ and } l \notin F_2(a, M) \\ 1 & \text{if } l \in F_2(a, M) \\ M(l) & \text{otherwise} \end{cases}$$

The choice of the elements contained in F_1 and F_2 depend on the firing rule they apply. Kavi et at. distinguished five of types of them: 1) conjunctive $F_1(a, M) = I(a)\ \forall M$; 2) disjunctive $|F_1(a, M)| = 1\ \forall M$; 3) collective $F_1(a, M) \subseteq I(a)\ \forall M$; 4) selective $|F_2(a, M)| = 1\ \forall M$; 5) distributive $F_2(a, M) = O(a)\ \forall M$.

Uninterpreted data flow graphs looses the semantic of the token, thus loosing the ability to use control links as well. In order to model such behaviors, nondeterminism is included into the model, and interpreted as the nondeterminism of F_1 and F_2 for different instances of execution of an actor. The different choices of firing sets can be incorporated directly into F_1 and F_2 by the mean of a probability distribution, associated with each possible choice. Figure 3.2a depicts the a gate t controlled by an actor c_1 while figure 3.2b depicts the same gate using nondeterminist firing set. In the latter case, the firing set $F_2(t, M)$ is defined as

$$F_2(t, M) = \begin{cases} O(t) = l_2 & \text{with probability } p \\ \varnothing & \text{with probability } p - 1 \end{cases}$$

The probability p depends on the frequency of having a *true* value on the control link of the gate.

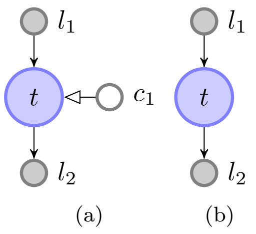

Figure 3.2: A gate t controlled by a control actor or a nondeterminist firing set

Bruza and Weide [29] proposed a semantics for the Dataflow Diagrams (DFD) [30, 31], a slightly different model that the pure dataflow proposed by Karp and Miller. A DFD is a directed graph constructed over *building blocks*, namely *processes* that transform the data, *data stores* that model a collection of data packets at rest and *external entities* that model the communication with the environment. Finally, arcs of the graphs represent the *data flows.* We invite the reader to refer to [31] for further details on the construction of DFDs. Beyond the addition of data stores and external entities, whereas the pure dataflow could be seen as a graph composed by processes and flows only, the main difference lies in the firing condition. In a DFD, a process may be fireable upon several configurations of its input and is not expected to produce a token for each of its input. While this behavior is convenient to describe mechanism of process architecture, it is not sufficient to give a clear definition of what is being described [29]. To address this issue, Bruza and Weide proposed to extend the DFDs by translating them into a variant of Petri Nets [32]. Building blocks are transformed into transitions, and data flows into places for which the precondition and postcondition are respectively the source and destination of the flow. Finally, they add the following properties to their altered version of Petri Net:

(i) a set of *activation possibilities* of the form $m_k \wedge \cdots \wedge m_l \rightarrow z_p \wedge \cdots \wedge z_q$ is attached to each transition, describing the different possible input configurations, together with the corresponding output configurations;

(ii) a capacity is defined for each place, denoting the number of messages a flow can hold;

(iii) a multiplicity is defined on arcs, denoting the number of messages consummed or produced by the firing of a transition;

The reader may note that if the extensions (ii) and (iii) are very common to the definition of Petri Nets [33], the concept of activation possibilities is rather new, and have been the subject of more recent work, unrelated to the dataflow [34, 35]. Based on the refinement of the DFD expressiveness given by their extended model of Petri Net, named *Process Structure* (PS), Bruza and Weide also proposed a method to convert a PS into an algebraic path expression, based on the language COSY [36]. A key advantage of such expressions is the possibility to establish process structure equivalence. They distinguish four types of synchronizations, namely: 1) the *sequence* that corresponds to the execution of several processes in order, written as $p; q; r$; 2) the *selection* that allows the execution of one process among a list of candidates, written as p, q, r; 3) the *repetition* that describes the execution of a process 0 or more times, written as p^*, 1 or more times, written as p^+, or an infinite number of times, written as p^∞; 4) the *concurrency* that models the concurrent execution of two processes P and Q, written as $P||Q$.

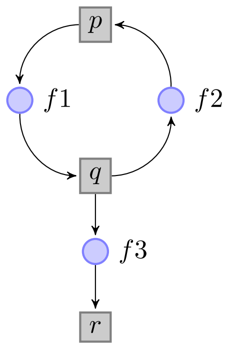

Figure 3.3: Example of process structure

Figure 3.3 depicts an example of process structure. It may be noted that, without defining activation possibilities, the behavior of this PS is not predictable, unlike in a classic Petri Net. For q we give the following behavior $(f_1 \rightarrow f_2 \wedge f_3)$ and assume that others transitions follow a classic Petri Nets firing rule. The process path associated to this PS and activation possibilities are expressed by $p; q; ((p \parallel r); q)^\infty$. Given that the firing of q generates a token in f_3 *and* f_2, there exists an infinite loop in the system, expressed by the repetition $((p \parallel r); q)^\infty$. A different behavior would have been exposed if we had given q the activation possibility $(f_1 \rightarrow f_2 \vee f_3)$. In such a case, the path expression would be $p; q; (p; q)^*; r$, meaning that $p; q$ would be fired infinitely often until r

would eventually be fired. A more complete example of DFD transformation into PS and equivalent process path is given in [37].

MapReduce MapReduce is a prominent tool for parallel data processing that grew a huge popularity these last years in both industrial and academic fields since its popularization by Google and Apache [38, 39]. The model is based on the *map* and *reduce* functions, borrowed from the functional programming [40]. We classify MapReduce as a dataflow paradigm, since it is easy to schematize as a dataflow graph; figure 3.4 depicts such a representation. MapReduce is used over very large input, which first is subdivided into smaller chunks of fixed size. Each chunk is sent to a mapper m_i which parses key/value pairs out of its input and passes them to the user-defined *map* function. Intermediate sets produces by the executions of *map* are equally distributed on the available reducers r_i which sort them in order to group elements with the same key before passing them to the user-defined *reduce* function. Implementations generally take care of the dataflow we sketched above so the programmer must only provide *map* and *reduce* functions. An additional *combiner* function may be provided as well, in order to perform a partial merging of the data before it is sent to the reducers, just after the *map phase*.

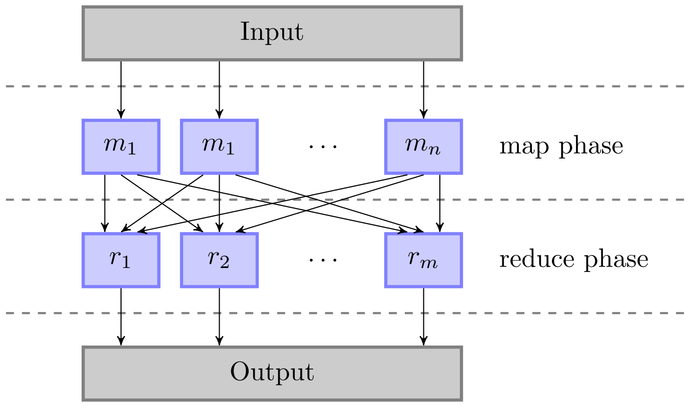

Figure 3.4: Dataflow graph of MapReduce

The success of the MapReduce framework probably lies on its simplicity: a user must only provide two simple and rather intuitive functions to describe a complex parallel operation on a very large input. Another usually advocated advantage is the scalability of MapReduce: if the number of mapper is somehow bound to the size of the input, the number of reducers is totally arbitrary and easy to scale. Finally, MapReduce is highly fault tolerant: intermediate data are *triplicated* on a distributed filesystem before it is sent to the reducers so a task can be reassigned in the event of a failure. On the other hand, MapReduce often suffers from poor efficiency compared to its DBMS counterparts, as Pavlo et al. concluded in a detailed comparison of both models [41]. In addition, MapReduce may be not quite suitable for an important class of algorithms, due to the rigidity of its dataflow. A finer discussion of the advantages and pitfalls of MapReduce is given in [42].

Despite its youth, several important extensions of MapReduce have been studied already. In interesting direction is the definition of more abstract languages on the top of MapReduce. Pig [43] is a high-level dataflow system offering SQL-style data manipulation constructs. Programs are written in Pig Latin, a high-level dataflow language, and compiled into a directed acyclic graph (DAG) that represents its the logical plan of execution. This DAG may be subject to some optimizations and is finally translated into a series of MapReduce jobs. Sawzall [44] is another notable high-level system built on the top of MapReduce. A Sawzall program is a set of statements that fetch a single record of the data, perform operations on it and then emit some values to an external aggregator that processes results. When interpreted, Sawzall statements are executed during the *map phase* and aggregators during the *reduce phase*, letting the MapReduce system manage the distribution of tasks. We may also cite PACT [45], an extension of MapReduce that supports more flexible dataflow architectures. Programs are written using task specific functions, later organized in a flexible workflow.

We encourage the reader to consult [42] for a wider review of MapReduce extensions, especially with respect to its performance issues.

3.1.2 Coordination programming

Data-Driven Coordination Model The 1980s saw the evolution of parallel and distributed architectures, often massively increasing the number of processors involved in their systems. Consequently, applications also grew bigger, targeting complex fields such as information retrieval or air trafic control. Obviously, numbers of parallel programming paradigms flourished, differing in granularity of computation units and communication mechanisms, yet it became apparent that no unique paradigm or language could cover all the needs of complex applications, or to the least not as efficiently as possible [46]. This motivated Gelernter and Carriero to emphasize on the concept of *coordination language* defined as "(...) the glue that binds separate activities into an ensemble" [47]. They claimed that programs should be described in terms of two orthogonal tasks: the actual *computation* involving the manipulation of data, and the *coordination* of processes creation, communication and destruction. Thus, a program should be written in at least two different languages: a computation language such as C to write functional blocks and a coordination language to specify how these blocks may be created and communicate. Furthermore, each block could be written in a different language and execute on a different architecture. This description is strongly related to the large-grain hybrid dataflow we presented in section 3.1.1.

To support their definition of coordination language, Gelernter and Carriero proposed a model articulated around the concept of *generative communication* [48, 49]. This paradigm states that a collection of independent processes is communicating by the mean of a *tuple space* (TS). In order to exchange data, a process A would write *tuples* into the TS and a process B would then withdraw them. The model is said *generative* since a generated tuple remains independant within the TS until it is withdrawn from it. Once in the TS, a tuple may be equally accessed by all the participating processes, but is bound to none. This contrasts with the message-passing approach, in which a message is bound to its sender and receiver at all times. Figure 3.5 illustrates a layout with three processes attached to a common tuple space. A key advantage of this paradigm is that it decouples processes in both time in space, since processes don't need to know each others, nor live at the same time [46]. Such layout is known as the *data-driven* coordination model; the state of computation at any time depends on the values of the data in the TS and the internal configuration of coordinated processes.

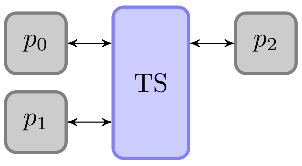

Figure 3.5: A tuple space with three processes attached to

In addition to the tuple space, the model defines three primitive operations:

$out(t_1, \ldots, t_n)$ writes a tuple $t = \langle t_1, \ldots, t_n \rangle$ into the tuple space. Once the statement executed, the process continues immediately.

$in(s)$ randomly withdraws a tuple $t = \langle t_1, \ldots, t_n \rangle$ that matches the template $s = \langle s_1, \ldots, s_n \rangle$ from the tuple space. A template $s_i \in s$ can be either a value, in which case only the tuples for which $t_i = s_i$ will be considered, or a type, in which case only the tuples for which the type of t_i matches s_i will be considered. If there's no tuple satisfying s, the process waits until one does.

$read(s)$ is the same as $in(s)$ except that it only reads the matched tuple t without withdrawing it.

We may precise that this definition slightly differs from the original one given in [48], which assumes that all elements of a tuple have the same type. The notion of arbitrary types was added in [49].

Linda [48, 49] has been the first genuine *coordination language*. In addition to the operations $out()$, $in()$ and $read()$ introduced above, it supports active tuples, evaluating some data before eventually turn into passive tuples. A primitive named $eval(p)$ will generate an active tuple executing p. Over the years, two operations $inp()$ and $readp()$ have also been added, exposing the same behavior than $in()$ and $read()$ except they do not block if no tuple satisfies the given pattern and return $false$ instead. Respecting the concept of orthogonality, Linda is completely independent of the host language it is used with. Several implementations have been proposed, for different paradigms of computation languages. C, Prolog and Java [49, 50, 51] are probably the most famous but constitute by no means an exhaustive list.

Despite some criticisms [52], Linda and so-called *linda-like* or *linda-based* languages dealt a great enthusiasm for a little less than a decade, popularizing the concept of a shared dataspace, i.e a common content-addressable data structure [53]. Notable extensions are multiple first-class citizen tuplespaces [54], objective orientation [55], fully distributed tuple space [56], tuple access control [57] or more recently semantic templates [58]. After a waning in the mid 1990s, a resurgence of interest started in the early 2000s, pushed by some important commercial implementations such as JavaSpaces, GigaSpaces or AutevoSpaces to cite a few [51]. The use of shared dataspaces for web services [59, 60] appears as a notable field domain of application.

The shared dataspace approach is not the only one being used in data-driven coordination models. One could cite *Synchronizers* [61] model as a notable example, which relies on message-passing communication.

Formalization of Data-Driven Models In spite of the huge interest engaged by data-driven, i.e. *tuple space centered* models, few formalisms had been proposed since recently. The semantic of the model is often viewed too straightforward to justify its formalization. However, in their study of scalable tuple spaces, Obreiter and Guntram advocated the use of a formalism to identify structural restrictions, abstracting the different levels of expressiveness induced by proposed extensions of tuple spaces schemes [62]. Their formal model is based on the notion of *fields*, which themselves compose tuples. According to the definition given in [49], a field is either *formal*, i.e. a type to be matched, or *actual*, i.e. constant a value. Finally, templates are not differentiated from tuples, thus sharing the same definition. More formally:

Definition. *F is a set of fields defined as*

$$F = C \cup \bigcup_{c \in C} I_c$$

where C is a set of classes and I is a set of instances labeled by C.

Let $\leq_c \subseteq C^2$ be an antisymetric operator ordering classes such that $c \leq_c c'$ if and only if c is c' or c is a superclass of c'. Multi-inheritance being allowed, $\{C, \leq_c\}$ is a *partially ordered* set assuming the existence of a *minimal element* $\perp_F \in C$ such that $\perp_F \leq_c \forall c \in C$. The reader inclined to reason in terms of object-oriented programming may see $\perp_F$ as the class `object` in a language such as Python. Let the mapping $\tau : F \rightarrow C$ denote the class $c \in C$ of any field $f \in F$.

Definition. *$match_F \subseteq F^2$ is a matching relation on F if and only if*

$$\forall c \in C, \forall f \in F, c \leq_c \tau(f) \Leftrightarrow match_F(c, f)$$

Let $\mathcal{T}(F)$ denote the set of formal and actual tuples to a given set of fields F and d denote the maximal dimension of tuples.

$$\mathcal{T}(F) = \bigcup_{i=1}^{d} F^i$$

Let $\Gamma(F)$ denote the set of semantic tuples.

$$\perp_{\mathcal{J}(F)} = \bigcup_{i=1}^{d} (\perp_F)^i \subseteq \mathcal{P}(\mathcal{T}(F))$$
$$\perp_{\mathcal{J}(F)} \in \Gamma(F) \subseteq \mathcal{P}(\mathcal{T}(F)) \setminus \{\varnothing\}$$

with $\mathcal{J}(F) = \mathcal{T}(F) \cup \Gamma(F)$ denoting the set of tuples and $\mathcal{P}(A)$ depicting the powerset of A. Finally, let $match_{\mathcal{J}}$ denote the matching relation on tuples, with the unique restriction $match_{\mathcal{J}} \subseteq \mathcal{J}(F)^2$.

The above definitions lead to the notion of *tuple space scheme*, given as a 4-tuple $\langle F, match_F, \mathcal{J}(F), match_{\mathcal{J}} \rangle$. Furthermore, if Φ is defined as the set of tuple schemes, then $\Phi_L \subseteq \Phi$ is the set of tuple space schemes complying additional restrictions imposed by a language L. A example of such restrictions on a subset of Φ is given for Linda in [62].

Another approach regarding the formalization of the data-driven coordination model consists of expressing its semantic by the means of process calculi. Ciancarini et al. proposed to embed the generative communication into the the Milner's *Calculus of Communicating Systems* (CCS) [63, 64]. Their model makes the assumption of a completely distributed tuple space, thus considering each message as an autonomous agent $\langle a \rangle$ willing to deliver its contents a to a potential reader. A prefix $\underline{a}$ is added to describe the *read*() operation, while the classic input and output operations, namely a and $\bar{a}$, describe respectively the operations *in*() and *out*(). Table 3.1 shows the operational semantics of the calculus proposed by Ciancarini et al. For the sake of clarity, only the semantics closely related to the generative communication are shown; the reader is encouraged to refer to [63] for a complete description.

$$\begin{array}{cc}
a.P \xrightarrow{a} P & \underline{a}.P \xrightarrow{a} P \\
\tau.P \xrightarrow{\tau} P & \hat{a}.P \xrightarrow{\tau} \langle a \rangle \mid P \\
\langle a \rangle \xrightarrow{\bar{a}} 0 & \langle a \rangle \xrightarrow{\underline{\bar{a}}} \langle a \rangle
\end{array}$$

Table 3.1: Operational semantics of a process calculus for generative communication

In addition to this algebra, Ciancarini et al. studied several observational semantics, namely the bisimulation model, the failure model and the trace model in an attempt to express a congruence of generative communication agents. Busi et al. used this formalism to define a formal model for JavaSpaces [65], including the additional features of JavaSpaces, such as the *event notification*, the *distributed leasing* and the *timeout* on blocking operations.

Yet another interesting approach is given by Semini and Montangero. They used temporal logic [66] to describe the semantics of a tuple space model [67], using a temporal logic language called*Oikos-tl* to produce the specifications of a tuple space model. This language is based on UNITY [68], extended with some non-standard temporal operators, mostly added as syntaxic sugars. They defined an axiomatic and an operational semantics for the prototype language nicknamed *TuSpRel*, where statements take the form

$$guard \mid body \, , \, tell$$

where *guard* are axiomatic expressions that must be verified in order to exececute *body* and *tell*. Tuples are typically read or consummed in *guard* while *tell* writes new tuples, for which values were optionally computed in the *body* expressions. The reader may have noticed that this notation, very closed to the coordination language Swarm [69], shares also a lot of similarities with the statements of Prolog.

Control-Driven Coordination Model Data-driven coordination languages typically offer a set of primitives that abstract the underlying coordination metaphor being used. These primitives are added to a given host language, like in C-Linda to give a straightforward example [49]. Thus, if the conceptual separation between coordination and computation is made clear, the actual program being written usually blends the coordination primitives into the computation code, blurring the distinction between coordination and processes. This lack of distinction motivated Papadopoulos and Arbab to propose an alternative, namely the *control-driven* model [70]. It emphasizes much more on a rigorous separation where the computation parts are treated as black boxes with input/output interfaces while the coordinational parts focus on describing the connections between them. These connections generally are *point-to-point*, but broadcasting can be achieved by setting up *1-to-n* relationships. Additionally, control-driven models allow computational blocks to deliver *control messages*, informing interested other components of their internal state [46]. As a result, coordination is achieved by notifying states changes so other components can react accordingly. Obiously, this model is closely related to the large-grain hybrid dataflow described in section 3.1.1.

Control-driven models languages often advocate the concept of *configuration* [71, 72], expressing the coordination as the overall configuration of computation blocks. This approach proved to be interesting for software development, especially during the design phase [73], but as well as for maintenance, emphasizing on the variability and reusability of building blocks. Another more recent model is based on *channel composition* [74]. In addition to the usual read-/write operations, some composition operators can be used to specify dynamic dataflow topologies. Mobility is accomplished by making channels independent of the location of their source and sink components; moreover components are anonymous for each others so the endpoint of a channel may be replaced freely.

3.1.3 Multiset Rewriting

First Order Chemical Model Banâtre and Métayer introduced the so-called Γ-Model (General Abstract Model for Multiset mAnipulation) [75]. Strongly related to Linda [48, 49] and Associons [76], a programming notation based on tuple queries to express computation. This paradigm relies on a chemical metaphor: computational elements are *molecules*, floating in a solution and eventually reacting with each others according to a set of *reaction rules*. The succession of all reactions expresses the program computation, which consequently stops whenever no further reactions are possible. Clearly, the Γ-Model is a *rule based system* [77] where the working memory is the multiset on which the *matching* operation will find compatible molecules and transform them into new ones. The following is a small intuitive example of a Γ-program applying the *sieves of Eratostheness* in order to identify the prime numbers in given range $[2, \ldots, n]$

$$sieves : x, y \rightarrow y \Leftarrow multiple(x, y)$$

with $multiple : \mathbb{N} \times \mathbb{N} \rightarrow \{true, false\}$ a function returning *true* if x is a multiple of y, $false$ otherwise. The clause $multiple(x, y)$ of the above rule is called the *reaction condition* and specifies the property to be satisfied for the selected x, y to be rewritten y. A key advantage of the Γ-Model is that nothing is said neither about the order of execution, nor about the *process(es)* applicating reaction rules. This high level of abstraction makes possible to express the very *idea* of the algorithm, with almost unless no language idiosyncrasy. In addition, one could say that the model expresses pure parallelism; in our example, any pair of molecules x, y could be simultaneously and nondeterministically considered without any side effect on the overall computation. The reader may be interested in consulting [78, 79] for very good review of the Γ-Model and its evolutions.

Inspired by this model, Andreoli et al. introduced the Interaction Abstract Machine (IAM), which adds the concept of independent agents. Roughly speaking, a reaction rule may create several copies of the multiset, after having consummed reacting molecules, and feed them with the result of the reaction, possibly different for each new multiset. Once this achieved, the newly created agents will perform reactions on their own, independently from the others. Figure 3.6 shows an example of an IAM with the two rules

$$\begin{aligned} & a \parr b \rightarrow (r \parr s) \;\; \& \;\; t \\ & t \parr c \rightarrow \top \end{aligned}$$

Rules are composed of a *head*, the left part, and a *body*, the right part. Resources matching a head are consummed and rewritten with respect to the body. The symbol $\parr$ ("par") allows defining multiple resources in a head, or a body. The symbol $\top$ ("top") in a body expresses the termination of an agent. Inter-agent communication is achieved by *broadcasting*. One could relate this notation to a Prolog statement with multiple clauses. Resources in the head may be prefixed by a symbol $^\wedge$; whenever an agent apply a rule, it will consume the non-prefixed resources, assume the existence of prefixed ones and broadcast them to every other agents, except its direct offspring. IAMs gave birth of Linear Objects (LO) [80], an extension of Prolog that intends to combine object-oriented and logic programming.

Yet another chemical-inspired approach was given by Cervesato et al. They used a multiset rewriting formalism to analyze security protocols [81]. They advocate the use of state rewriting as a simpler method for intrusion detection than the usual analysis of possible traces. Their formalism involves *terms*, *facts*, *states* and *rules*. It shares a lot of similarities with algebraic specifications [82]; terms are well-formed expressions over a signature, facts are first-order atomic

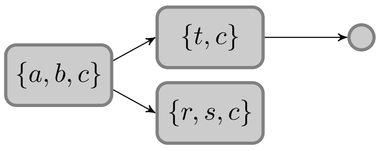

Figure 3.6: A multiagent IAM

formulae over a signature and states are multisets of facts. The left part of a *rule* is a multiset of facts a state should contain in order to be rewritten with the facts of its right part.

$$F_1, \dots, F_k \rightarrow \exists x_1, \dots \exists x_j . G_i, \dots, G_n$$

This rule means that facts $F_1, \dots, F_k$ from a state S would be rewritten as $G_1, \dots, G_n$, where $x_1, \dots, x_j$ would be replaced by new symbols. We can notice that this description matches the definition of the Γ operator, modulo the condition clause. Cervesato carried on a redesign of his work in [83] and introduced the *typed MSR*, which emphasizes much more on the aspects related to security protocols. More recently, Rosa-Velardo introduced the so-called *υ-MSR* [84] which includes name binding and constraints on rewriting rules.

In an attempt to address the issue of combinatorial explosion in Γ-Model implementations, Fradet and Le Métayer proposed the notion of *structured multiset* [85] that basically is a set of pointers, organized according to a problem-specific topology. By example, the list $l = [l_1, l_2, l_3]$ would be written as the following structured mutliset

$$\textbf{next}\ l_1\ l_2,\ \textbf{next}\ l_2\ l_3,\ \textbf{end}\ l_3$$

The main advantage of such structuring is that the programmer is free to describe a structure that could considerably reduce the number of combination for which no reaction condition would ever be satisfied. In addition, it reveals the underlying structure to the compiler that may be able to apply some optimization according to. The continuation of their work lead to the definition of the *shape types* [86], a context free graph grammar based on multiset rewriting rules. The purpose of shape types is to express the properties of data structures, such as double linked lists, binary trees, etc. as classic languages expressiveness is insufficient.

Formalization of First Order Chemical Models A formalization of the Γ-Model is given in [75]. Let M denote a multiset of molecules $m_i \in M$. Let Δ denote a set of reaction rules $\delta_l = \langle r_l, a_l \rangle \in \Delta$. Let r_l denote the *condition* of δ_l, i.e. a function $r_l : \{x_1, \dots, x_n\} \rightarrow \{true, false\}$ and a_l the *action* of δ_l, i.e. a rewriting rule $\{x_1, \dots, x_n\} \rightarrow \{y_1, \dots, y_m\}$.

Definition. *The operator* Γ *applied to a set of reaction rules* Δ *and a multiset*

M is defined as

$$\begin{aligned}\Gamma(\Delta, M) = \exists\,\{x_1, \ldots, x_n\} \in M \mid \forall \delta_l \in \Delta, \exists r_l(x_1, \ldots, x_n) \\ \underline{\textit{then}}\ \Gamma(\Delta, (M - \{x_1, \ldots, x_n\}) \cup a_l(x_1, \ldots, x_n)) \\ \underline{\textit{else}}\ M\end{aligned}$$

Several extensions were proposed to enrich this definition of chemical model. Hankin et al. introduced composition operators in [87] in order to express the semantics of Γ-Programs combination. Namely, two operators $P \circ Q$ and $P + Q$ denote respectively the sequential composition and the parallel composition; $P \circ Q$ means that P will start using the result of Q, i.e. its multiset after it has reached a stable state. $P + Q$ means that reactions of P and Q are executed concurrently on the same multiset until neither can proceed further. For instance, consider the following sorting algorithm

$$\begin{aligned} \mathit{sort} &: \mathit{match} \circ \mathit{init} \\ \mathit{init} &: (x \to (1, x) \Leftarrow \mathit{integer}(x)) \\ \mathit{match} &: ((i, x), (j, y) \to (i, x), (i + 1, y) \Leftarrow (x \leq y \wedge i = j)) \end{aligned}$$

The program *init* will consume every integers from the initial multiset and produce a tuple $\langle \mathit{rank}, \mathit{value} \rangle$ with an rank of one for each tuple. After its completion, the resulting multiset will be used by the program *match* that will eventually produce a list of rank-ordered integers. Note that in the above example, we could write

$$\mathit{sort} : \mathit{match} + \mathit{init}$$

because the reaction rules of *match* and *init* work on disjoint subsets of molecules.

Composition operators opened the lead toward the definition of a *calculus* for Γ-programs; such works were conducted in [88, 89]. Hankin et al. studied the recurring patterns in Γ-Program, called *tropes* for Transmuter, Reducer, OPtimiser, Expander, and Selector [90]. For the sake of simplicity, we'll describe only three of them in this document, since they're the most intuitive.

$$\mathcal{T}(C, f) = x \to f(x) \Leftarrow C(x) \quad \text{(transmute)}$$

$$\mathcal{R}(C, f) = x, y \to f(x, y) \Leftarrow C(x, y) \quad \text{(reducer)}$$

$$\mathcal{E}(C, f_1, f_2) = x \to f_1(x), f_2(x) \Leftarrow C(x) \quad \text{(expander)}$$

The main advantage of the these so-called tropes is that they capture very well the effect a reaction rule has on a multiset. Recalling our example of the sieve of Eratosthenes, clearly our reaction rule is a reducer with $C(x, y) = \mathit{multiple}(x, y)$ and $f(x, y) = y$, intuitively indicating that the size of our multiset will reduce after each successful reaction. Combining tropes with composition operators $P \circ Q$ and $P + Q$, Hankin et al. gave a richer formal semantics of the Γ-Model than in [87], that can be used to compare composition equivalences.

Finally, we may note the use of composition operators in a more general framework called *composed reduction systems* [91], where programs are built upon composition of rewrite relations.

Chemical Abstract Machines The Chemical Abstract Machine (CHAM) [92] is an extension of the chemical model proposed by Banâtre and Métayer firstly aimed at using the chemical metaphor to describe the operational semantics of process calculi. The most important addition to the Γ-Model is the notion of *membrane*, which has the capability to encapsulate a multiset of molecules inside a bigger multiset. In other words, one can define a hierarchical structure with solutions possibly containing subsolutions. Molecules may react freely within membranes, or be extracted by an *airlock* operator that can expose a molecule of a solution to its environment, to seek for possible reactions.

$$\{a_1, a_2, \ldots, a_n\} \rightleftharpoons \{a_1 \triangleleft \{a_2, \ldots, a_n\}\}$$

In the above example, the subsolution $\{a_2, \ldots, a_n\}$ is free to continue its internal reactions, while the molecule a_1 is exposed for potential reactions. CHAM are nondeterministic, and so is the choice of the molecule extracted from its subsolution. The symbol $\rightleftharpoons$ represent a structural equivalence, thus the airlock operator is reversible, preventing any *wrong* extraction of a molecule from its subsolution to prematurely freeze an execution.

The CHAM distinguishes between three kinds of rules, namely the *heating rules* denoted by the symbol $\rightharpoonup$, the *cooling rules* denoted by the symbol *rightharpoondown* and the *reactions rules* denoted by the symbol $\rightarrow$. The two formers are structural manipulations; heating rules decompose complex molecules into smaller ones while cooling rules recompose them. Generally, there exists a cooling rule reversing a given heating rule, and vice versa. On the other hand, reaction rules correspond to a non-reversible modification of molecules. There are two types of reaction rules in a CHAM; proper rewriting rules similar to the rules of the Γ-Model, and reaction rules specific to the algebra of the molecules. For instance, given a CHAM defined over CCS [64], the following reductions may occur

$$\{a.(P + Q) \mid \bar{a}.R \mid S\} \rightharpoonup \{a.(P + Q), \bar{a}.R, S\} \rightarrow \{P + Q, R, S\}$$

assuming $p \mid q \rightleftharpoons p, q$ and $a.p, \bar{a}.p \rightarrow p, q$.

More formally, let $m, m', \ldots$ denote molecules defined over a given algebra, $S, S', \ldots$ denote finite multisets of molecules written $\{m_1, \ldots, m_n\}$ and $r, r', \ldots$ denote specific rewriting rules of the form $x_1, \ldots, x_n \rightarrow y_1, \ldots, y_m$ where x_i and y_j are molecules. Let $\oplus$ denote the multiset union. Let $C[S]$ denote any context of a solution S. A Chemical Abstract Machine obeys the following four general laws

$$x_1, \ldots, x_n \rightarrow y_1, \ldots, y_m \qquad \text{(reaction law)}$$

$$\frac{S \rightarrow S'}{S \oplus S'' \rightarrow S' \oplus S''} \qquad \text{(chemical law)}$$

$$\frac{S \rightarrow S'}{\{C[S]\} \rightarrow \{C[S']\}} \qquad \text{(membrane law)}$$

$$\{m\} \oplus S \leftrightarrow \{m \triangleleft S\} \qquad \text{(airlock law)}$$

CHAMs have been used to describe the operational semantics of several formalisms, such as the Linear Logic [93], the π-calculus [94, 95] or more recently the Interaction Nets [96], a graph rewriting system for classical Linear Logic [97].

Higher Order Chemical Model The Γ-Model as defined above is said first order, since reaction rules cannot be manipulated by the execution of the program itself. In [89], Le Métayer proposed to unify the notion of program, i.e. the collection of reaction conditions, and the multiset into a unique *configuration* object. As a result, a reaction may occur in a configuration itself nested into another configuration where other reactions may occur, simultaneously but at a different hierarchical level. The termination condition is modified according to this new behavior so it is said stable, i.e. terminated, if and only if it does not contain neither reactive elements, neither active, i.e. not terminated, configurations. A configuration is denoted

$$[Prog, Var_1 = Multexp_1, \ldots, Var_n = Multexp_n]$$

and consists of a program *Prog* and a record of named multisets Var_i. A configuration with an empty program is called *passive*, otherwise it is *active*. The semantics stay essentially identical to the first order (see [87]), with the addition of the following rules to capture the higher order features

$$\frac{X \rightarrow X'}{\{X\} \oplus M \rightarrow \{X'\} \oplus M}$$

$$\frac{M_k \rightarrow M'_k}{[P, \ldots, Var_k = M_k, \ldots] \rightarrow [P, \ldots, Var_k = M'_k, \ldots]}$$

where $\oplus$ denote the multiset union. The first rule accounts for the computation of active configurations inside multisets while the second describes the rewriting of a multiset containing active configurations. The reader may note the similarity with the CHAM [92].

More recently, Banâtre et al. introduced a higher order calculus for the Γ-Model, named γ-calculus [98]. Their goal was to identify a minimal calculus from which extensions could be derived, like it had been the case for their first order model. A first minimal calculus γ_0-calculus is presented as a basis for further extensions; their syntax is very close to the λ-calculus [99] so a γ_0-term, i.e. a molecule, is defined as the following

$$\begin{aligned} M ::= {} & x && \text{(variable)} \\ & \mid (\gamma \langle x \rangle .M) && (\gamma\text{-abstraction}) \\ & \mid (M_1, M_2) && \text{(multiset)} \\ & \mid \langle M \rangle && \text{(solution)} \end{aligned}$$

γ-abstractions are one-shot first class citizen reaction rules that consumes a molecule matching its argument and rewrites it as its body. Molecules *float* within a multiset and it is assummed that their neighborhood is constantly and non deterministically changing. Solutions encapsulates molecules so it mimics the hierarchical construction of configurations in [89]. The semantics is given as the following

$$\begin{aligned} (\gamma \langle x \rangle .M), \langle N \rangle &\to_\gamma M[x := N] && \text{if } inert(N) \vee hidden(x, M) \\ \gamma \langle x \rangle .M &\equiv \gamma \langle y \rangle .M[x := y] && \text{with } y \text{ fresh} \\ M_1, M_2 &\equiv M_2, M_2 \\ M_1, (M_2, M_3) &\equiv (M_1, M_2), M_3 \end{aligned}$$

The first rule is the γ-reduction, which applies a γ-abstraction to a solution. A γ-reduction is applied under two possible conditions

i. either $inert(N)$ is verified, meaning that a solution $\langle N \rangle$ is made exclusively of γ-abstractions or exclusively of solutions;

ii. or $hidden(x, M)$ is verified, meaning that the variable x occurs in M only as $\langle x \rangle$.

The intuition behind $hidden(x, M)$ is that if a $\gamma - reduction$ doesn't access the content of a solution, then it is safe to apply it even if the solution content is still active. Two fundamental extensions of this γ_0-calculus are presented in [98]. The first includes reaction condition, modeled as a molecule M_0, added to the γ-abstraction. It is written

$$\gamma \langle x \rangle \lfloor M_i \rfloor .M_1$$

In order to be applied on a solution $\langle N \rangle$, the condition $M_0[x := N]$ of a given γ-abstraction must reduce to *true*. The second extension enriches the γ_0-calculus with n-ary γ-abstraction. It is written

$$\gamma(\langle x_1 \rangle, \ldots, \langle x_n \rangle).M$$

Such a γ-abstraction must atomically capture n solutions $\langle x_1 \rangle, \ldots, \langle x_n \rangle$, otherwise it does not react. Both these extension can be combined and result to a more expressive γ-calculus.

3.2 Formal models

In this section, we briefly introduce some formalisms aimed at describing parallel and concurrent behaviors. We will first discuss Petri Net, a mathematical model with a graphical notation aimed at describing system concurrency. Then we will focus on the process calculi, also known under the term process algebra, another formal model for concurrent systems.

3.2.1 Petri Net

Petri Net, also called plate/transition net (PT-net), is a mathematical model introduced in the early 1960s by C. Petri in his thesis PhD [32]. Particularly well adapted to describe concurrent processes, they are mainly used to study distributed systems [33]. In words, a Petri Net is a bipartite graph where the two types of nodes are called *places* and *transitions*. The state of a system is encoded by a *marking*, that is the number of token in each place of the system. Fireable transitions eventually consume token from their input places and produce new ones to their output places, thus modifying the overall state of the system. [33] contains several detailed examples introducing basic concepts related to Petri Nets. More formally

Definition. *A Petri Net is a 5-tuple* $PN = \langle P, T, F, W, M \rangle$ *where*

$P = \{p_1, \dots, p_n\}$	*is a set of places*
$T = \{t_1, \dots, t_m\}$	*is a set of transitions*
$F \subseteq (P \times T) \cup (T \times P)$	*denotes the flow relations*
$W : F \to \mathbb{N}$	*denotes flow relations multiplicity*
$M : P \to \mathbb{N}$	*is a marking*

Definition. *A transition* $t \in T$ *is fireable in a marking* M *iff the following condition holds*

$$W(t, p) \leq M(p) \forall p \in P$$

Definition. *Firing a transition* t *from a marking* M *produce a new marking* M' *such that*

$$M'(p) = M(p) - W(t, p) + W(p, t) \forall p \in P$$

Over the years, numerous extensions have been proposed to enhance Petri Nets. We do not intend to give a complete review of them; [100] constitutes a good reference in such direction. However, we will briefly introduce two models that related to the family of *High-Level Petri Nets* [101], namely the *Colored Petri Nets* and the *Algebraic Petri Nets*. In these formalisms, token have an internal structure. This allows a model to be constructed upon a much higher level of abstraction.

Colored Petri Nets Colored Petri Nets (CP-nets) were introduced by Jensen in [102]. His model has been strongly influenced by an earlier work of Genrich and Lautenbach who introduced the Predicate/Transision Nets [103]. In CP-nets, a *color*, i.e. a type, is assigned to each token. An important implication of such addition is that it enhances token expressivity so they carry a value, defined over their individual color. Places may contain several token of the same color, and are thus act as multisets. Arcs are labeled by expressions, thus potentially modifying the values of the withdrawn token before new ones are produced. Furthermore, a guard is associated with each transition, evaluating a set of expressions to determine for which token the transition fireable.

Figure 3.7 is an example from [102] modeling a the *dining philosophers* problem. This model is composed of two colors, p for the philosophers $ph_1, \dots, ph_n$ and f for the forks $fk_1, \dots, fk_n$. Places are labeled by their name on the top, and their color on the bottom. It is assummed that the guard is implicitly set to *true* for each transitions. At the initial marking, there is n philosophers token in the place labeled "think" and n forks in the place labeled "forks". The

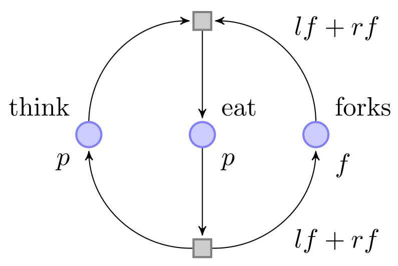

Figure 3.7: Colored Petri Net describing the dining philosophers problem

two functions lf and rf some arcs are labeled with are functions of the form $g : p \to f$ that return respectively the left and right fork of a given philosopher token. The top transition is fireable whenever there exists a philosopher token in “think”, and its corresponding forks in “‘forks”. When it is fired, it will consume these token and put a philosopher in “eat”. Later on, the bottom transition will eventually consume a philosopher from “eat” and produce corresponding token in “think” and “forks”.

Algebraic Petri Nets Algebraic Petri Nets (AP-nets) [104] carry a similar idea than the CP-nets; when token are elements of a color set in a CP-net, they are algebraic terms in AP-net, defined over an algebraic specification [82]. Such specifications have the advantage to propose a solid mathematical reasoning about the defined terms and associated operations. The signature $\Sigma = \langle S, F \rangle$ of a specification is composed of a finite set S of *sorts* and a finite set F of operation on these sorts. Similarly to CP-nets, places in AP-nets are multisets of terms of the same sort. Transition arcs are labeled by algebraic operations, performed on the token the moment they are bind to fire the transition. Furthermore, guards can also be associated to each transition to control its firing possibilities.

A further detailed introduction to algebraic specifications, and their use into the Petri Net formalism is given in section 4.3. We also encourage the reader to consult [104, 105] for an in-depth presentation.

3.2.2 Process calculi

Process calculi, also called process algebra, are another approach to formalize concurrent processes. From [106] we quote “(...) process algebra is the study of the behavior of parallel or distributed systems by algebraic means”. They provide a very high-language with a small collection of primitives, together with a set of operators defined on them. Processes are expressed as the combination of these primitives. Let us say that we have two processes P and Q and that we want to express the way a third process R could be the combination of P and Q, either sequentially, or in parallel. Then we will need some algebraic axioms that can express our model mathematically. Furthermore, we will need such axioms to formally verify whether or not a given model satisfies some properties. One fundamental feature of the process algebra is that it does not take stock of how a process should communicate with another, i.e. by what medium communication is achieved within a given system. Instead, it describes what interface a process exposes to the others.

One pioneer in this context was Robin Milner, who first proposed his Calculus of Communicating Systems (CCS) [64] in the 1980s. The main underlying idea in the CCS is to model processes behavior as states, reached by performing actions. Processes are defined as blocks, possibly identified by a name, with their process interface. The latter describes the communication channels through which a process may interact with its environment. Behavior is described in a CCS language, composed by a small set of five basic operators.

i. The *choice* operator $+$ expresses a choice between two processes as a given time. In other words, $P + Q$ is a process either acting like P, or like Q;

ii. The *parallel composition* operator $|$ denotes a process resulting of the simultaneous execution of two processes P and Q;

iii. The *renaming* operator $[a/b]$ expresses the process result of all actions a renamed as b in a process P;

iv. The *restriction* operator $\backslash a$ denotes the process result of a process P without action a;

v. The *process naming* operator $=^{def}$ allows us to use an identifier A to refer to a process P, which can itself contain the identifier A for recursive definitions.

In addition to this five operators, there are two more important features of the CCS language.

i. The empty process 0, also called nil, denotes a process from which no more action is possible;

ii. The action prefixing $a.P$ expresses a process $P1$ result of performing action a and then acting like P.

By convention, we refer to a as an input channel, and to $\bar{a}$ as an output channel.

Other important works in the field of process calculi are the Communicating Sequential Processes (CSP) [107], proposed by Hoare in the 1980s, which includes the notion of synchronization barrier. We could also cite the π-calculus [95], proposed by Milner, which allows channels name to be communicated as the process evolves. We encourage the reader to consult [106] for a detailed review of the process calculi until the mid 2000s.

3.3 Symbolic model checking

Model checking techniques are, most of the time, based on the systematic exploration of the set of all the possible states a system could fall into [108].

Such exhaustive set is called the state space of the system. Given that it is possible to compute the state space of a system, then we can use a model checker to verify if some property holds for the system, unless it can return a counter example, i.e. the state and/or a path within the state space for which the property is not verified. Let us illustrate this principle with a rather simple example with the PT-net depicted in figure 3.8. This model represents the four seasons with places, while the four transitions correspond to a switch from one season to another. One property to verify could be to check that *it is never possible to be in spring and winter at the same time.* A model checker would then explore the state space, here quite small, and discover that there is no state where both places labeled “spring” and “winter” contains a token. On the other hand, the property *we never reach fall if we were in summer once* would not hold and a model checker would return a path where the place labeled “fall” eventually contains one token, after the place “summer” did. Such properties are often expressed in a temporal logic, such as LTL [66] or CTL.

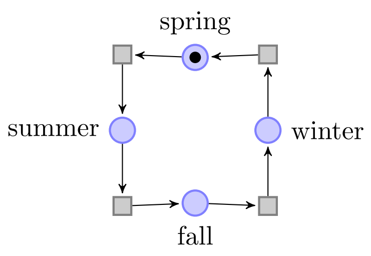

Figure 3.8: Simplistic model of the four seasons

Unfortunately, a problem quickly arises with this approach of model checking: in more complex systems, it is likely that the state space may be very difficult unless impossible to compute, even if it is known to be finite, because it may be exponential in the number of components. This problem is known as the state space explosion. To tackle this problem, Burch et al. proposed to use the Binary Decision Diagrams [109] to encode the state space *symbolically* instead of explicitly, like it is the case in the classic processes we described above [110]. The idea behind this encoding is to represent states as boolean functions. In order to do so, we first need to impose a total ordering on every element composing a state, together with their boolean representation. A characteristic boolean function is made by chaining these elements, with respect to the total ordering, and said to be true if their encoded values corresponds to a reachable state, false otherwise. Such representation drastically reduces the size of the state space, since common characteristics between states are stored only once. In fact, the encoding is logarithmic to the number of states in the best case, linear in the worst [105].

More evolved decision diagrams have been proposed, to enhance the original idea of Burch et al. Probably the most noticeable among these are the Multi-Valued Decision Diagrams [111], the Data Decision Diagrams [112] and the Hierarchical Decision Diagrams [113]. Based on the latter, Buchs and Hostettler proposed more recently the Σ Decision Diagrams [114], mainly targeting the term rewriting systems. The drawback of the Decision Diagrams is that the choice of ordering has a great influence on the size of the symbolic encoding, to the extend that the model designer should build his or her model with consideration to the model checking techniques that will be used, once terminated.

Proposed by Biere et al. another symbolic model checking technique is the *bounded model checking* [115], based on SAT solvers instead of Decision Diagrams. They advocated that the need of a canonical form of the Binary Decision Diagrams is a problem when it comes to encoding big systems. As pointed just above, the variable ordering must be carefully chosen and often requires manual intervention. On the other side, if SAT solvers also rely on a compact binary representation, they do not require such ordering. In order to verify properties, they assume the existence of a counter example of length k and try to generate a formula, i.e. a state, satisfying the counter example, if it exists. The term *bounded model checking* carries the notion of *bound* placed on the maximum length k of the paths to search.

4 Encoding the Γ-Model

4.1 Introduction

One problem that arises when it comes to reasoning about the Γ-Programs is that they lack of a formal definition of their molecules. Indeed, molecules can be everything, from simple integers to complex data structures. If this specificity gives a great level of freedom to write algorithms, it may become problematic when it comes to the verification of the Γ-Model. In [90] Hankin et al. identified program schemes, called *tropes*, and use them to verify program properties; their method is mostly based on reaction rules cardinality and well-founded ordering of the multiset to determine termination and confluence. If their method reveals to be very interesting to determine these properties, it cannot be used to reason about reachable states. For instance, given the following sort program

$$primes : \langle x, y\rangle \rightarrow \langle y\rangle \Leftarrow multiple(x, y)$$

it may be interesting to formally check that a prime number is never removed from the multiset during the entire execution. Moreover, for the case of non-confluent programs, it may be useful to determine what multisets can be reduced from an initial one.

In this section, we propose to encode the Γ-Model into a more formal model, namely the Algebraic Petri Nets. Our goal is to construct the reachability graph [33] of a Γ-Program, given this graph is finite and computable, to analyze its state space. Then we show how we can take advantage of this alternative representation to perform model checking and verify properties that would be hard to verify otherwise, such as invariants. The remaining of this section is composed by a formal definition of the notion of Γ-*Program*, among with some examples. Then we introduce an encoding into AP-nets and study how we can verify the model properties. As a proof of concept, we implement two simple examples on AlPiNA [116], an AP-nets model checker. Finally, we discuss the limitations of our encoding method.

4.2 Γ-Programs

In section 3.1.3 we gave an overview of the multiset rewriting programming, including the Γ-Model [75] and its related works. However, before we go on describing its encoding into AP-nets, we give here a more detailed presentation and formally define what we will call a Γ-*Program*. We will also give two less trivial examples of programs to illustrate this programming style in more elaborated algorithms and familiarize the reader with it.

The idea of the Γ-Model is to express computation by the means of a chemical metaphor. A bag of molecules floating within a solution, move freely according the *Brownian motion* and possibly interact with each other's under a collection of reaction rules. The solution is said stable when no further reaction can occur. This simplistic definition of a chemical solution is used to define the computation as a global evolution of atomic values. Another intuitive definition would be to see the bag of molecules as a multiset and reaction rules as rewritings of it, picking up some values in a non-deterministic way and putting back the result of their reaction.

The multiset is the less constraining data structure. It allows all elements to appear any number of time, as opposed to a set, and it makes no assumption on any order of elements, as opposed to any hierarchical or recursive data structure. Thus, the Γ-Model is free from any constraint on the order in which values are accessed and is by definition purely nondeterministic. Furthermore, it induces a complete parallelism since noting restrain disjoint groups of molecules to interact concurrently, and independently. In fact, all reactions follow a *locality principle* meaning that they rely only on the values of the molecules about to react and never on some global state of the system.

Definition and Notation of Γ-Programs In section 3.1.3, we reported the formalism proposed by Banâtre and Métayer in [75]. Let M denote a multiset of molecules $m_i \in M$. Let Δ denote a set of reaction rules $\delta_l = \langle r_l, a_l \rangle \in \Delta$. Let r_l denote the *condition* of δ_l, i.e. a function $r_l : \{x_1, \ldots, x_n\} \to \{true, false\}$ and a_l the *action* of δ_l, i.e. a rewriting rule $\{x_1, \ldots, x_n\} \to \{y_1, \ldots, y_m\}$.

Definition. *The operator Γ applied to a set of reaction rules Δ and a multiset M is defined as*

$$\begin{aligned}\Gamma(\Delta, M) = \exists\, &\{x_1, \ldots, x_n\} \in M \mid \forall \delta_l \in \Delta, \exists r_l(x_1, \ldots, x_n)\\ &\underline{then}\ \Gamma(\Delta, (M - \{x_1, \ldots, x_n\}) \cup a_l(x_1, \ldots, x_n))\\ &\underline{else}\ M\end{aligned}$$

Using this formalism, we give the following definition of a Γ-Program.

Definition. *A Γ-Program P is a 2-tuple* $\langle \Delta, M \rangle$ *where*

$$\begin{array}{ll}\Delta = \{\delta_1, \ldots, \delta_m\} & \textit{is a set of reaction rules}\\ M = \{m_1, \ldots, m_n\} & \textit{is a multiset of molecules}\end{array}$$

Following the above definition, we could write a program $sieves = \langle \Delta_s, M_s \rangle$ applying the *sieves of Eratostheness* in a range $[2, \ldots, n]$ as

$$\begin{aligned}\Delta_s &= \{\langle multiple(x, y), (\langle x, y \rangle \to \langle y \rangle)\rangle\}\\ M_s &= \{2, \ldots, n\}\end{aligned}$$

If there exists a total ordering of the molecules, we say that a multiset M is in normal form if it is written $M = \{x_1, \ldots, x_n\}$ and $x_i < x_j \implies i < j, \forall i, j$. We say that $\langle \Delta, M \rangle \to_\Gamma M'$ (reduces to) if M' is the normal form of a multiset obtained after one application of a $\delta_l \in \Delta$ on multiset M. For instance

$$\langle \Delta_s, \{2, 3, 4, 5, 6\} \rangle \to_\Gamma \{2, 3, 5, 6\}$$

means that $\{2,3,5,6\}$ is a possible reduction of $\langle \Delta_s, \{2,3,4,5,6\}\rangle$ after one reaction. We write the $\rightarrow_\Gamma^*$ the transitive closure of $\rightarrow_\Gamma$. In words, $\langle \Delta, M\rangle \rightarrow_\Gamma^* M''$ if M'' is a stable solution obtained by successive applications of the reactions rules Δ. For instance

$$\langle \Delta_s, \{2,3,4,5,6\}\rangle \rightarrow_\Gamma^* \{2,3,5\}$$

We use these definitions to formally express the termination and confluence properties.

Definition. *A Γ-Program $P = \langle \Delta, M\rangle$ reaches termination iff*

$$\exists M'' \mid P \rightarrow_\Gamma^* M''$$

Definition. *A Γ-Program $P = \langle \Delta, M\rangle$ is confluent iff*

$$P \rightarrow_\Gamma^* M' \wedge P \rightarrow_\Gamma^* M'' \implies M' = M'', \forall M'$$

Finally, we give the definition of reachable multisets of P, i.e. the set of multisets which can be obtained by an arbitrary number of reduction of P.

Definition. *The set of reachable multisets of $S(P)$ of a program $P = \langle \Delta, M\rangle$ is defined as*

$$S(P) = \{M\} \cup \bigcup_{\forall P \rightarrow_\Gamma M'} S(\langle \Delta, M'\rangle)$$

Γ-programs often contain only one reaction rule; so for the sake of clarity we may use the more compact notation

$$P(M) : \langle x_1, \dots, x_n\rangle \rightarrow \langle y_1, \dots, y_n\rangle \Leftarrow C(x_1, \dots, x_n)$$

where C is the reaction condition. We may omit the multiset M to describe only the program semantic, without any concern about its parameters.

Examples of Γ-Programs In this section we detail two examples of Γ-Programs, in order to illustrate the programming style entailed by the *Gamma*-Model. We start with a path finding program proposed in [117] which returns the shortest path in a directed graph. Let $G = \langle V, E\rangle$ denote a directed graph and $F : E \rightarrow \mathbb{N}$ denote the cost of each edge. We want to find the shortest path from a particular vertex v_r to each other vertices v_i. We encode the graph as a multiset of molecules $\langle v_i, v_j, c, l\rangle$ where

i. u and v are vertices from V and $\langle u, v\rangle \in E$ is an edge;
ii. c is the the cost $F(v_i, v_j)$ associated with the edge;
iii. l is the length of the shortest known path from v_r to v via u.

And we give the following program

$$P_{sp}(M_{init}) : \langle x, y\rangle \rightarrow \langle x, \langle y_u, y_v, y_c, x_l + y_c\rangle\rangle \Leftarrow (x_v = y_u) \wedge (x_l + y_c < y_l))$$

with the initial multiset

$$M_{init} = \{\langle u, v, F(u,v), \infty\rangle \, \forall u, v \in V, u \neq v\} \cup \{\langle v_r, v_r, 0, 0\rangle\}$$

This program computes the shortest path from v_r to v via u. It takes two molecules x, y that share a common vertex and for which the cost to from v_r to y_v is reduced via x_u to update the shortest known path to y_v. Intuitively, we remark that the program eventually terminates because its reaction rule rewrites a shorter path whenever it is applicated on the multiset. It is also confluent since a reaction produces the same molecules it consummed, only modifying the length l of the path being computed. That means a shorter path will always overwrite a previous computation. We add the tuple $\langle v_r, v_r, 0, 0\rangle$ to the initial multiset to enable computation. Once stable, the multiset could contain several tuples $\langle u_1, v, c_1, l_1\rangle, \langle u_2, v, c_2, l_2\rangle$ with $u_1 \neq u_2$, thus depicting several possible paths. As we want to return only the shortest path, we need to remove non optimal solutions; we can use the multiset reduced from the program defined above in another program that would remove the non-optimal solutions.

$$P_{clean} : \langle x, y\rangle \rightarrow \langle y\rangle \Leftarrow (x_v = y_v) \wedge (x_l > y_l)$$

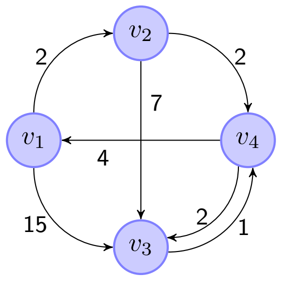

Figure 4.1: A directed graph to illustrate path-finding

We briefly illustrate this program with a concrete example. Let the directed graph represented in figure 4.1; we want to compute the shortest path from v_1 to every other nodes. The resulted encoding is given by the following multiset

$$M_{init} = \left\{ \begin{array}{c} \langle v_1, v_2, 2, \infty\rangle, \langle v_1, v_3, 15, \infty\rangle, \langle v_2, v_3, 7, \infty\rangle, \langle v_2, v_4, 2, \infty\rangle \\ \langle v_3, v_4, 1, \infty\rangle, \langle v_4, v_1, 4, \infty\rangle, \langle v_4, v_3, 2, \infty\rangle, \langle v_1, v_1, 0, 0\rangle \end{array} \right\}$$

Starting with this multiset, two reactions could occur, either computing the length from v_1 to v_2 or from v_1 to v_3. In the first case, the molecule $\langle v_1, v_3, 15, \infty\rangle$ would be rewritten $\langle v_1, v_3, 15, 15\rangle$. This new molecule may now react with $\langle v_3, v_4, 1, \infty\rangle$, rewriting the latter as $\langle v_3, v_4, 1, 16\rangle$. We said before that P_{sp} terminates and is confluent; as a result, we write $\langle P_{sp}, M_{init}\rangle \rightarrow^*_\Gamma M_{int}$ the obtained stable multiset

$$M_{int} = \left\{ \begin{array}{c} \langle v_1, v_2, 2, 2\rangle, \langle v_1, v_3, 15, 15\rangle, \langle v_2, v_3, 7, 9\rangle, \langle v_2, v_4, 2, 4\rangle \\ \langle v_3, v_4, 1, 7\rangle, \langle v_4, v_1, 4, 8\rangle, \langle v_4, v_3, 2, 6\rangle, \langle v_1, v_1, 0, 0\rangle \end{array} \right\}$$

Finally, we remove the non optimal paths with $P_{clean}(M_{int}) \rightarrow^*_\Gamma M_{clean}$ and get

$$M_{final} = \left\{ \begin{array}{c} \langle v_1, v_2, 2, 2\rangle, \langle v_2, v_4, 2, 4\rangle \\ \langle v_4, v_3, 2, 6\rangle, \langle v_1, v_1, 0, 0\rangle \end{array} \right\}$$

which contains only the shortest paths from v_1 to every other vertices.

Our second example is an character recognition program. Given a set of classes and, for each class a collection of samples, the goal is to decide what class matches the best an unknown character. More formally, let C denote the set of classes. Let S denote a set of samples labeled by C. Let x denote the character to classify. Let i denote the unknown character. We want to find $c \in C$ such that

$$\arg\min_{c \in C} \sum_{s_c} |i - s_c|$$

We assume samples and candidate for evaluation are black and white images. Before encoding samples, we split them into pieces of equal sizes, typically to the pixel. This allows us to increase the parallelism with respect to the size of the pieces. Then we encode the molecules as 5-tuples $\langle p, v, t, c, m \rangle$ where

i. p is an identifier of the piece within the sample, typically coordinates;
ii. v is the value of the piece p;
iii. t is the *type* of the molecule;
iv. c is the class $c \in C$ of the piece;
v. m is the match value, i.e. the difference between a sample and the source.

If the size of the pieces is as small as the pixel, then v is a simple boolean denoting whether the pixel at p is black or white. Otherwise, v could be either an array of the pixels value, or even simply the average value of the pixels contained in p. The type t is set to *ukwn* for the source molecules, *smpl* for the samples molecules or *rslt* for comparisons molecules. Initially, m is set as 0 for all molecules. The character recognition is given by the following program

$$P_{rec} = \langle \{\delta_{comp}, \delta_{red}\}, M_{init} \rangle$$

with

$$\delta_{comp} = \left\langle \begin{array}{c} (x_p = y_p) \wedge (x_t = ukwn) \wedge (y_t = smpl) \\ \langle x, y \rangle \to \langle x, \langle y_p, y_v, rslt, y_c, |x_v - y_v| \rangle \rangle \end{array} \right\rangle$$
$$\delta_{red} = \left\langle \begin{array}{c} (x_t = y_t = rslt) \wedge (x_c = y_c) \\ \langle x, y \rangle \to \langle \varnothing, \varnothing, rslt, x_c, x_v + y_v \rangle \end{array} \right\rangle$$

The first rule δ_{comp} computes the difference between two corresponding pieces, from the source and a sample, to produce an intermediate molecule with the result of the comparison. It should be stressed that the source is written back into the multiset for further reaction with other samples, while the sample is definitely removed. The second δ_{red} rule takes two comparison results of the same class and rewrite only one molecule, summing their m values. In other words, this rule reduces the different results of the same class into a single value, similarly to the functional programming *reduce* function, performed on a sequence of data. The first rule terminates, since it eliminates samples after each reaction, and is also confluent because the it rewrites the source, together with a molecule that does not impact further reactions. Obviously, because of its cardinality the second rule terminates. Its confluence can be deducted from the commutatively of the addition $x_v + y_v$ it performs.

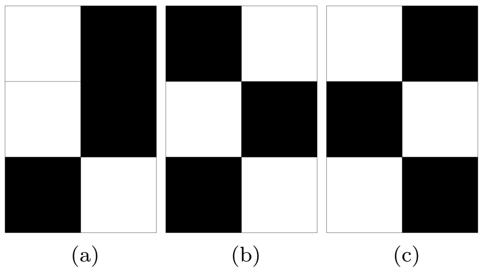

Figure 4.2: A character to recognize (4.2a) and two classes (4.2b, 4.2c)

We now consider this program for the trivial example illustrated in figure 4.2. The character to recognize, i.e. the source, is given in figure 4.2a; the two other figures represent a single sample of two distinct classes. We call i the source and the two classes respectively a and b. For each class we have a sample s_c with $c \in \{a, b\}$. We encode the source with the following collection of molecules

$$i = \left\{ \begin{array}{l} \langle (0,0), 0, ukwn, \varnothing, 0 \rangle, \langle (0,1), 1, ukwn, \varnothing, 0 \rangle \\ \langle (1,0), 0, ukwn, \varnothing, 0 \rangle, \langle (1,1), 1, ukwn, \varnothing, 0 \rangle \\ \langle (2,0), 1, ukwn, \varnothing, 0 \rangle, \langle (2,1), 0, ukwn, \varnothing, 0 \rangle \end{array} \right\}$$

Samples are encoded similarly; the fields t set to *smpl* and c set to the class the sample corresponds to. When the program starts, only the first reaction rule δ_{comp} is enabled, possibly rewriting

$$\begin{array}{l} \langle (0,0), 0, ukwn, \varnothing, 0 \rangle, \langle (0,0), 1, smpl, a, 0 \rangle \\ \rightarrow \langle (0,0), 0, ukwn, \varnothing, 0 \rangle, \langle \varnothing, \varnothing, rslt, a, 1 \rangle \end{array}$$

An interpretation of this intermediate result would be that, so far, a difference of one pixel has been observed between i and s_a. Similarly, the same source could react with $\langle (0,0), 0, smpl, b, 0 \rangle$ and produce $\langle \varnothing, \varnothing, rslt, b, 0 \rangle$, meaning that, so far, no difference has been observed between i to s_b. The second reaction rule δ_{red} is enabled as soon as two intermediates results have been produced for the same class, and reduce them. After it reached a stable state, the multiset would equal to

$$M = \{ \langle \varnothing, \varnothing, rslt, a, 2 \rangle, \langle \varnothing, \varnothing, rslt, b, 4 \rangle \} \cup i$$

By comparing the m values of the two result molecules, we observe that class a matches better the source than class b. In our example, the comparison of the final results is straightforward since we had the same number of samples for both classes. An actual system is likely to have different numbers of samples for its different classes. In such a situation, we should divide the m value by the number of samples of class before proceeding to comparison. It should also be stressed that other kind of operations could be performed by the rule δ_{comp}, largely depending on the encoding used for the molecule values v.

4.3 Algebraic Abstract Data Types and Petri Net

Data types play an essential role in most algorithms. In our study of multiset rewriting techniques, we have seen that computation state entirely depended on the values of the data being computed. Moreover we have seen, particularly in the Γ-Model, that the type of these data was also central to the definition of the algorithms. Any Γ-Program description starts with a definition of the structure, i.e. the type of its molecules. The fact is that data types also embed the operations required to build and updated actual data. Less related to our interest, another advantage of data types is that they imply type checking. Values can be matched against a type, only describing the shape it should have.

In this section, we sketch an introduction on the notion of *Algebraic Abstract Data Types* (AADT) [82], which make possible to formally define data types without any concern about a specific implementation, and their use in conjunction with Petri Net, in a formalism called *Algebraic Petri Nets* (AP-nets) [104]. This will give us the necessary background to study how we can use them to describe Γ-Programs.

Algebraic Specifications In order to describe Algebraic Abstract Data Types, we need to start with a formal definition of their algebra. Hostettler pointed out that such specification should fulfill three requirements [105]

1. p an unambiguous syntax exhaustively defining how the specification should be written;
2. v a semantics describing the behavior of the specification;
3. t an inference system making possible to reason about the specification in terms of deductions.

Algebraic specifications meet these requirements [82]. It is an Order-Sorted Algebra, i.e. one or several sets of values can be defined. Such sets are called *carrier sets*, and the name of a carrier set is called a *sort* [105]. For instance, the booleans are defined over the carrier set $\{0, 1\}$ whose name is *boolean*. The sorts together with their operation names compose the signature of a data type. In other words, this signature represents the exhaustive syntax necessary to describe any algebraic term of its algebra.

Before we go further, we recall the notion of S-Sorted set

Definition. *Let $S \subseteq \mathbb{S}$ be a finite set. A S-Sorted set A is a disjoint union of a family of sets indexed by S*

$$A = \bigcup_{s \in S} A_s \textit{ noted as } A = (A_s)_{s \in S}$$

Based on this definition, we now give the definition of a signature

Definition. *A signature is a 2-tuple* $\Sigma = \langle S, F\rangle$ *where*

$$\begin{array}{ll} S \subseteq \mathbb{S} & \textit{is a finite set of sorts} \\ F = (F_{w,s})_{w\in S^*, s\in S} & \textit{is a } (S^* \times S)\textit{-sorted set of operation names} \end{array}$$

Let ϵ denote the empty word over S so each $f \in F_{\epsilon,S}$ is a constant symbol. Let X denote a S-Sorted set of variables so the set of terms Σ over X is a S-Sorted sorted set $T_{\Sigma,X}$. We give the definition of the axioms as the following

Definition. *The axioms on variables* X *are equational terms* $t = t'$ *such that* $t, t' \in (T_{\Sigma,X})_s$

Finally, we can define the notion of *algebraic specification*

Definition. *An algebraic specification is a Spec* $= \langle \Sigma, X, E\rangle$ *is a signature extended by a collection of axioms* E *on variables* X.

We illustrate the above definitions with an example of the *dining philosophers* problem for four philosophers. Inspired from [104], we start by giving the signature Σ

$$\begin{array}{rl} \textit{sorts}\colon & \textit{phils} \\ & \textit{forks} \\ \\ \textit{opns}\colon & p_0, \ldots, p_4 \colon \epsilon \to \textit{phils} \\ & g_0, \ldots, g_4 \colon \epsilon \to \textit{forks} \\ & lf, rf : \textit{phils} \to \textit{forks} \end{array}$$

This signature is composed by two sorts, *phils* and *forks*, respectively representing the philosophers and the available forks. The operations $p_0, \ldots, p_4$ are constants and denote the four philosophers we consider for the sake this example. Similarly, the constants $g_0, \ldots, g_4$ denote the four available forks. Two additional operations lf and rf return respectively the left and right fork of a given philosopher. We extend this signature with the following axioms to compose a complete specification

$$\textit{axioms}\colon \quad \left.\begin{array}{r} lf(p_i) = g_i \\ rf(p_i) = g_{(i+1)\%4} \end{array}\right\} \text{with } i = 0, \ldots, 4$$

The above axioms describe the semantics of the operations lf and rf such that they return the fork corresponding to a given philosopher. The reader may have noted that this example is very close to the one we gave to illustrate CP-net.

Algebraic Petri Nets To combine AADTs with the PT-net formalism, we first need to give the notion of multiset of terms. In fact, similarity to what was done in CP-net [102], we want places to be associated with a sort so they are actually multisets of sorted terms of a given algebra. Thus we give the definition of extended signature

Definition. *An extended signature* $[\Sigma] = \langle S', F' \rangle$ *where*

$$S' = S \cup \{[s] \mid \forall s \in S\}$$
$$F' = \{\epsilon \to [s], \_\colon s \to [s], \_ + \_\colon [s], [s] \to [s] \mid \forall s \in S\}$$

Now we can give the formal definition of an AP-net specification.

Definition. *An Algebraic Petri Net specification* $NSpec$ *is a 5-tuple* $\langle Spec, P, T, X, AX \rangle$ *where*

$Spec = \langle [\Sigma], X', E \rangle$	*is an extended algebraic specification*
$P = \{p_1, \ldots, p_n\}$	*is a set of places*
$T = \{t_1, \ldots, t_m\}$	*is a set of transitions*
$X = \{x_1, \ldots, x_k\}$	*is a set of variables*
AX	*is a set of axioms*

Let $\tau : P \to S$ denote a function which maps a sort to each place. Let $In_p \in (T_{\Sigma,X})_{[\tau(p)]}$ denote the label of the arcs from place p to transition t, for all places $p \in P$. Let $Out_p \in (T_{\Sigma,X})_{[\tau(p)]}$ denote the label of arcs from transition t to place p for all places $p \in P$.

Definition. *Let* $NSpec = \langle Spec, P, T, X, AX \rangle$ *denote an Algebraic Petri Net specification. An axiom of* AX *is a 4-tuple* $\langle t, C, In, Out \rangle$ *where*

$t \in T$	*is the transition associated to the axiom*
$C \subseteq T_{\Sigma,X} \times T_{\Sigma,X}$	*is a set of equalities*
$In = (In_p)_{p \in P}$	*is a P-sorted set of terms*
$Out = (Out_p)_{p \in P}$	*is a P-sorted set of terms*

Recalling our example algebraic specification to define the dining philosophers problem, we illustrate our definitions with the AP-net depicted by figure 4.3. When the upper transition is fired, it takes a philosopher term from the place labeled "think" and its left and right forks obtained by the operations lf and rf. Similarly, a philosopher term is consummed from the place labeled "eat" and its left and right forks, obtained by the same operations are put back into the place labeled "fork". The reader may have noticed the outstanding similarities with CP-net [102].

4.4 Algebraic Petri Net and Γ-Program

In this section, we present a function to translate a Γ-Program into an AP-net specification. We will use the dining philosophers problem to illustrate are translation and compare the difference between our result and the AP-net we gave in figure 4.3.

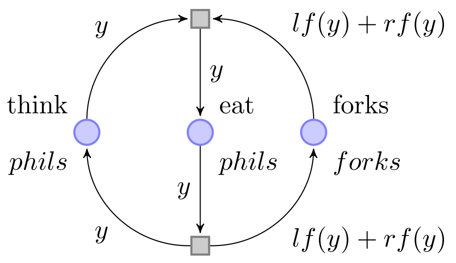

Figure 4.3: Algebraic Petri Net describing the dining philosophers problem

The first idea is to encode the molecules into an algebraic specification. Obviously, this step depends on the problem subject to our modeling, and the data structure we want to use to describe the molecules. Then we give the definition of our translation function

Definition. *Let $\mathcal{P} = \langle \Delta, M \rangle$ be a Γ-Program. Let Spec $= \langle \Sigma, X, E \rangle$ be an algebraic specification composed by at least one sort s which represents the data structure of the molecules $m \in M$. $\Phi(\mathcal{P}, Spec)$ is a $NSpec$ defined by $\langle Spec, P, T, X, AX \rangle$ where*

i. *Spec is an algebraic specification;*
ii. *P is a set of places, composed by one place p of sort s, such that $\tau p = s$;*
iii. *T is a set of transition $\{t_1, \ldots, t_m\}$ where $m = |\Delta|$, and we define a function $\phi : T \rightarrow \Delta$ which associates a reaction rule to a transition;*
iv. *X is a set of variables $\{x_1, \ldots, x_k\}$;*
v. *AX is a set of axioms.*

To give the definition of the axioms AX, we need to translate the reaction conditions of Δ into a set of equalities $T_{\Sigma,X} \times T_{\Sigma,X}$. In order to do so, we assume that *Spec* contains also a sort *boolean* and that every reaction condition r_l is written as an equality of the form $T_{\Sigma,X} \times (T_{\Sigma,X})_{boolean}$.

Definition. *Let $\mathcal{P} = \langle \Delta, M \rangle$ be a Γ-Program. Let Spec $= \langle \Sigma, X, E \rangle$ be an algebraic specification. Let $\Phi(\mathcal{P}, Spec) = \langle Spec, P, T, X, AX \rangle$ be an Algebraic Petri Net specification. An axiom of AX is a 4-tuple $\langle t, C, In, Out \rangle$ where*

i. *t is the transition associated to the axiom;*
ii. *$C \subseteq r_l \times (T_{\Sigma,X})_{boolean}$, with r_l the reaction condition of $\phi(t)$, and we say that $r_l \in T_{\Sigma,X}$ is an operation of Σ;*
iii. *$In = (In_p)_{p \in P}$ is the left part of the rewriting rule $a_l \in (T_{[\Sigma],X})_s$ of $\phi(t)$ and we say that In_p is the label arc from place p to transition t;*
iv. *$Out = (Out_p)_{p \in P}$ is the right part of the rewriting rule $a_l \in (T_{[\Sigma],X})_s$ of $\phi(t)$ and we say that Out_p is the label arc from transition t to place p.*

To illustrate our definitions, we use once again the dining philosophers problem for four philosophers. We start with the following specification $\mathcal{S}_{phils}$

$$\begin{aligned}
sorts\colon\ & molecule \\
& status \\[1em]
opns\colon\ & eat, think, fork : \epsilon \to status \\
& mol : nat, status \to molecule \\
& id : molecule \to nat \\
& statusof : molecule \to status \\
& next : nat \to nat \\
& lf, rf : molecule \to molecule \\[1em]
axioms\colon\ & id(mol(i,s)) = i \\
& statusof(mol(i,s)) = s \\
& next(i) = (i+1)\%4 \\
& \text{if } statusof(m) \in \{eat, think\} \text{ then } lf(m) = mol(id(m), f) \\
& \text{if } statusof(m) \in \{eat, think\} \text{ then } rf(m) = mol(next(id(m)), f)
\end{aligned}$$

Philosophers and forks are encoded as pairs $\langle id, status\rangle$ where $id \in \mathbb{N}$ is a natural and $status \in \{eat, think, fork\}$ distinguishes respectively eating philosophers, thinking philosophers and forks. The operations id and $statusof$ retrieve respectively the identifier m_{id} and the status m_{status} of a molecule m. It must be stressed that we assume the existence of a sort nat, among with some constants and operations such as the addition and the modulo. This allows us to write an operation $next$ which returns the successor of a natural i but bounded to four. Finally we have two operations lf and rf which returns respectively the left and right fork of a philosopher.

Now we give the Γ-Program $\mathcal{P}_{phils}$ such that

$$\mathcal{P}_{phils} = \langle \{\delta_{eat}, \delta_{think}\}, M\rangle$$

with

$$\begin{aligned}
\delta_{eat} &= \langle (x_{status} = think) \wedge (y = lf(x)) \wedge (z = rf(x)), \langle x, y, z\rangle \to \langle x_{id}, eat\rangle\rangle \\
\delta_{think} &= \langle (x_{status} = eat), \langle x\rangle \to \langle \langle x_{id}, think\rangle, lf(x), rf(x)\rangle\rangle \\
M &= \langle i, think\rangle + \langle i, fork\rangle \text{ with } i = \{0, \ldots, 4\}
\end{aligned}$$

In words, the first reaction rule δ_{eat} binds a *thinking* molecule as x and checks that the two other molecules y, z are left and right forks of x. If so, it rewrites only the philosopher x as an eating molecule. Its left and right forks not being rewritten, neighbor philosophers cannot be part of another reaction until the second rule δ_think takes an *eating* philosopher molecule and rewrites it as a thinking one, among with its left and right forks.

Now that we have defined a specification, and a Γ-Program, we can construct the AP-net specification $\Phi(\mathcal{P}_{phils}, \mathcal{S}_{phils})$, describing the net depicted in figure

4.4. The labels next to the transitions represent the guard condition C associated with it. For the sake of clarity, we put wrote those conditions δ_r with δ the reaction rule mapped to the transition.

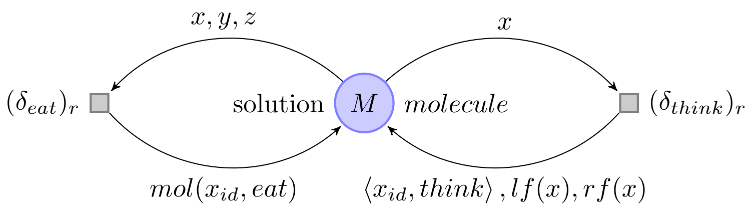

Figure 4.4: Translation of a Γ-Program for the dining philosophers in AP-net

We remark that this network is quite different from the example we gave in figure 4.3. Indeed, in the first case, we emphasize much more on the specification, while in the latter case we take advantage of a particular net topology. Yet their behavior is equivalent and they expose the same properties. In both cases, the net is deadlock free since the capture of left and right forks is atomic. Moreover, as the transition consuming an eating philosopher rewrite the consummed forks, the net can always come back to its initial state; as a result it obviously has liveness too. In both cases, the number of philosophers remains constant. If the number of forks may however vary during the execution of the net, it is necessary to have consummed a pair of forks before it can be rewritten. Thus, both nets are bound to the number of philosophers and forks. Finally, it should be stressed that the number of states generated by the two nets is identical and corresponds to the number of the exhaustive situations which respect the problem constraint.

4.5 A proof of concept with AlPiNA

As a proof of concept, we implemented some simple Γ-Programs translated into AP-nets using AlPiNA [105, 116]. Developed at the University of Geneva, AlPiNA, for Algebraic Petri Nets Analyzer, is a symbolic model checker for the AP-nets, relying on Decision Diagrams (DDs), among with some AP-net specific optimizations to mitigate the state space explosion problem. Properties of a model can be verified with a specialized property language, very close to First Order Logic, with deadlock detection.

The definition of AADTs in AlPiNA is made by the means of a dedicated language. It makes the distinction between the minimal operations necessary to construct the elements of a sort, and the function that operates on sorts. The former are called *generators* and the latter are called *operations*. The semantic of the operations, within the meaning of AlPiNA, is given by a set of axioms in the form of conditional equations. Furthermore, semantic is derived operationally using Term Rewriting: axiom equations constitute a rewriting system in which terms are successively rewritten until it reaches its normal form, i.e. no further rule can be applied on it. Table 4.1 shows the definition of an AADT in AlPiNA. Control flow, i.e. the AP-net places, transitions and arcs, can be defined by the means of a graphical interface, as the form a plugin for Eclipse.

```
Adt naturals

Sorts
  nat;

Generators
  zero: nat;
  suc: nat -> nat;

Operations
  plus: nat, nat -> nat;

Axioms
  plus(zero, $x) = $x;
  plus(suc($x), $y) = suc(plus($x,$y));

Variables
  x : nat;
  y : nat;
```

Table 4.1: Definition of an AADT in AlPiNA

We implemented two problems with AlPiNA: 1) the sieves of Eratosthenes and 2) the dining philosophers problem we used previously. The AADT we wrote for both problems are available in the appendix. We quickly faced the problem of state space explosion as we grew the size of the initial multiset. For the case of the sieves of Eratostheness, the number of possible state is doubled for each non prime number within the range; for the philosophers the exponential growth is even worse. As a result, we could run both problems only for small initial multiset, as a proof of concept.

4.6 Limitations

As our example of the dining philosopher pointed out, one problem of our translation technique is that we cannot take advantage of all the power of AP-nets, since we need to keep all terms in a single place. The main reason of this limitation is due to the fact that a Γ-Program relies on a single data structure, that is the multiset, represented by only one place. Moreover, we may note that a huge limitation of our framework is the assumption that molecules are all of the same type. In fact, nothing prevents a Γ-Program to be defined over an heterogeneous multiset. We already saw such a case with the sort algorithm we used to illustrate composition operators, in section 3.1.3. In this example, the multiset will at some point contain integer molecules, among with pairs of integers.

$$\begin{aligned} sort &: match \circ init \\ init &: (x \rightarrow (1, x) \Leftarrow integer(x)) \\ match &: ((i, x), (j, y) \rightarrow (i, x), (i+1, y) \Leftarrow (x \leq y \wedge i = j)) \end{aligned}$$

A solution to overcome this limitation would be to encapsulate the different sorts within a common *term* sort. Then, using additional operations to extract the actual term, we could distinguish several molecule types. Another solution would be to compose terms such that they contain some values that discriminate their nature. This is what we done with our dining philosopher example, where we used a field *status* that makes the distinction between philosophers and forks. However, both these solutions are not optimal, because they prevent us from using static type checking. Thus, reaction conditions must include some *runtime* type checking to ensure that a variable is bound to the expected data type. Subsorting [105] could be an acceptable solution, preserving the advantage given by static type checking while making possible to write all terms in a single place. Yet another way to preserve static type checking would be to use a place for each different type of molecule. However, we reject this layout for two reasons

1. it is unclear how to identify the sorts directly representing the molecules within a specification among the sorts used to construct them;

2. using several places to represent a single multiset would break the symmetry between a Γ-Program and its AP-net.

One may argue that are translation doesn't take advantage of pattern matching for preconditions, only using terms in closed forms instead. Indeed, many reaction conditions could be encoded as the presence of a certain pattern we would seek when binding the transition. This is what we did in the AP-net of figure 4.3 where one precondition is written $lf(y) + rf(y)$, which stands for *any term that match* $lf(y) + rf(y)$ with y being withdrawn from the place "think". Thus we could avoid to attach a guard to the transition. Even if it doesn't affect the class of algorithms we can encode, our choice represents a limitation with respect to the conciseness of AP-net produced.

Another limitation, however mostly related to the Γ-Model itself, is the inability to handle data structure other than a multiset. This means that molecules must be enriched with information about the global data structure, such as indexes. This becomes a problem particularly when it comes to program verification because it prevents reaction rules to make assumption on the molecules neighborhood; typically, algorithms involving graphs often consider linked vertices only. As a result, our models are easily subject to state space explosion because of the number of molecules to test for possible reaction. Fradet and Le Métayer investigated this issue and proposed the *structured multisets* [85]. They gave a rather intuive introductory example with a sorting algorithm. Let the following structued multiset denote a list

$$\textbf{next}\ l_1\ l_2,\ \textbf{next}\ l_2\ l_3,\ \textbf{end}\ l_3$$

Then we can express a sort algorithm like

$$sort : \textbf{next}\ x\ y, \bar{x} > \bar{y} \implies \textbf{next}\ x\ y, x := \bar{y}, y := \bar{x}$$

The above program states that the addresses x, y should exchange their pointed values if, for a relation **next** $x\ y$, the values $\bar{x}, \bar{y}$ are not well ordered. For a more familiar notation, let say that we want to sort the list $[5, 2, 7]$, then we would write a structured the multiset

$$\{\langle \textbf{next}, l_1, l_2\rangle, \langle \textbf{next}, l_2, l_4\rangle, \langle \textbf{end}, l_3\rangle, \langle \textbf{val}, l_1, 5\rangle, \langle \textbf{val}, l_2, 2\rangle, \langle \textbf{val}, l_3, 7\rangle\}$$

where **next**, **end** and **val** are keywords of the list structure. Yet interesting, we rejected this solution because one important problem of that approach is that data structure may be modified by the application of a reaction rule, possibly compromising the integrity of the entire multiset. As a result, either we should prove that no reaction rule can compromise the multiset integrity, or the correctness of the multiset should be verified after each reaction. In addition, it should be stressed that without addressing the previously explained issue, state space explosion would reveal to be even worse.

5 Stochastic Γ-Model

5.1 Introduction

In the previous section, we introduced the Γ-Model and illustrated its style with a couple of examples. We notices that several features of the model were particularly interesting the with respect to the concurrent programming and set programming. We also saw that the Γ-Model entails a huge level of parallelism, due to the total locality of its rewritings, among with a complete nondeterminism of execution, mimicking the Brownian motion. However, if nondeterminism over the order execution is implied by the model, it is difficult to describe nondeterministic rewritings. The closest way to achieve such behavior is to use multiple reaction rules defined over non-disjoint input molecules. For instance, let two reaction rules δ_a and δ_b such that

$$\delta_a : \langle x, y \rangle \to x - y \Leftarrow x > y$$
$$\delta_b : \langle x, y \rangle \to x + y \Leftarrow x > y$$

then the multiset $M = \{4, 8\}$ could be reduced to either $\{4\}$ or $\{12\}$, depending on the rule that would have been executed. The choice of the rule being applied is fair; in other words, if there exists a pair x, y such that $x > y$, then δ_a and δ_b have both one chance over two of being executed. Now let us consider a system where we want δ_a to have more chance to be executed than δ_b. As the model ensures that every rule whose condition is satisfied have the same probability to be executed, we could add more instances of δ_a to simulate a bigger probability.

Obviously, the main problem with that approach is that it adds a lot of complexity to the program. Moreover, the more probabilities are asymmetric, the more complexity grows. For instance, if we want δ_a to statistically happen ten times more often than δ_b, we need eleven rewriting rules. Yet another problem is that this approach limits us to discrete probabilities. Let us consider a system where the probability of execution of δ_a is not constant, but depends on the value of its arguments x, y. In other words, let $p(x, y)$ be a function which returns the probability P_a of δ_a to be executed, then we want $P_b = 1 - p(x, y)$ denote the probability of δ_b to be executed. With the above approach, the only solution we would be to discretize $p(x, y)$ and assign the adequate number of instances of δ_a and δ_b to each step of $p(x, y)$. Clearly, this solution is not acceptable because of its additional complexity.

In this section we introduce an extension of the Γ-model to address this problem. Namely, we propose to assign a probability function to each reaction rule so the choice of execution can be unfair when a tuple $\langle x_1, \ldots, x_n \rangle$ satisfies several rules. We give the syntax and semantics of our extension and show how we can apply our translation to AP-net with almost no modification.

5.2 Intuitive approach

We will start with an intuitive approach, based on a simple example. Our goal is to model a blog notation system, where blogs quality is evaluated according to the comments let by their readers. We represent each blog by its popularity $p \in \mathbb{R}$, initially set to 1. We say that the more a blog is popular, the less its popularity is affected by new comments. This gives us the following pair of reaction rules

$$\delta_+ : x \to x + \frac{1}{x+1}$$
$$\delta_- : x \to x - \frac{1}{x+1}$$

The two reaction rules denote respectively a positive and a negative commenting. Considering that when there exist two reaction rules whose condition are satisfied the choice of the one be executed is fair, we can assume that positive and negative comments are let with the same probability. Obviously, this program is not confluent, and never reaches termination. Indeed, we cannot define a well-founded ordering on the multiset that would demonstrate that the application of either rule somehow reduces the multiset, which means we can prove the termination, according to [90].

In a realistic system, it is likely that a blog with a good popularity receives more positive comments, while a blog with bad popularity receives negative comments. This behavior is really hard to model with the classic Γ-Model since we normally can only give deterministic choices based on molecule values. As we stated in the introduction, in order to achieve this, we would need discretize the probability distribution of our different outcomes and define an according number of reaction rule for each ceil. For instance, we could refine our above program to say that blogs with a popularity greater than 1 receive positive comments with a probability $p = 2/3$, while blogs with popularity of 1 or below receive negative comments with a probability $p = 2/3$. Then we would to redefine our reaction rules like

$$\delta_{p+} : x \to x + \frac{1}{x+1} \Leftarrow x > 1 \qquad \delta_{n+} : x \to x - \frac{1}{x+1} \Leftarrow x \leq 1$$
$$\delta_{p-} : x \to x + \frac{1}{x-1} \Leftarrow x > 1 \qquad \delta_{n-} : x \to x - \frac{1}{x+1} \Leftarrow x \leq 1$$

and put an according number of instance in our program, i.e.

$$2 \times \delta_{p+} + 1 \times \delta_{p-} + 1 \times \delta_{p+} + 2 \times \delta_{p-}$$

Obviously, this solution is not acceptable: in addition to the outstanding number of reaction rules we needed to define, it is far from scalable to any modification of the behavior, even small. Each new discretization of the probability distribution requires to redefine all reaction rules.

As an alternative, we propose to explicitly add the notion of frequency of the rule execution directly within the language. As a result, when a multiset of molecules $\{x_1, \ldots, x_n\}$ satisfies the reaction condition of several rules, the choice of the one being executed may be unfair; Furthermore it may depend on the value of the molecules; we call this extension Stochastic Γ-Model. Coming back to our example, this would give us the following reaction rules

$$\delta_+ : x \rightarrow x + \frac{1}{x+1}, f(x)$$
$$\delta_- : x \rightarrow x - \frac{1}{x+1}, 1 - f(x)$$

where $f(x)$ denotes the frequency of giving a positive comment

$$f(x) = \begin{cases} {}^2/_3 & \text{if } x > 1 \\ {}^1/_3 & \text{if } x \leq 1 \end{cases}$$

Whenever both rules are satisfied, which is always true in our situation, the choice of the one to be executed is weighted according to the frequency function $f(x)$. When a blog has a good popularity, then δ_+ is more likely to be executed and we use the complementary frequency for δ_-. Note that in this case, rule frequencies are normalized to one, but it may not be the case in every situation. This is why we talk about frequency and not probability. In fact, the probability p of a rule to be executed is its frequency f, normalized for all other fireable rules at the same time. For instance, consider replacing the frequency function f we used above with the following

$$f'(x) = \begin{cases} 2 & \text{if } x > 1 \\ 1 & \text{if } x \leq 1 \end{cases}$$

Of course, the complement is now given by $3 - f'(x)$. Once normalized, the probabilities of execution of our two reaction rules remain identical.

5.3 Formal definition

Now we introduce the formal definition of our Stochastic Γ-Model. Let M denote a multiset of molecules $m_i \in M$. Let Δ^{sto} denote a set of reaction rules $\delta_l = \langle r_l, a_l, f_l \rangle \in \Delta^{sto}$. Let r_l denote the *condition* of δ_l, i.e. a function $r_l : \{x_1, \ldots, x_n\} \rightarrow \{true, false\}$. Let a_l the *action* of δ_l, i.e. a rewriting rule $\{x_1, \ldots, x_n\} \rightarrow \{y_1, \ldots, y_m\}$, and let $f_l : \{x_1, \ldots, x_n\} \rightarrow \mathbb{R}$ denote the frequency of the reaction rule. Let $R_{x_1,\ldots,x_n} \subseteq \Delta^{sto}$ denote the set of reaction rules whose condition is satisfied by the multiset $x_1, \ldots, x_n \in M$.

Definition. *$D_{x_1,\ldots,x_n}$ is a random variable such that*

$$Pr(D_{x_1,\ldots,x_n} = \delta_l) = \frac{f_l(x_1, \ldots, x_n)}{\sum_{\delta_r \in R_{x_1,\ldots,x_n}} f_r(x_1, \ldots, x_n)}$$

$Pr(D_{x_1,\ldots,x_n} = \delta_l)$ denote the probability that $D_{x_1,\ldots,x_n}$ equals δ_l; it is computed after the normalization of the frequencies of the reaction rules satisfied by $\{x_1, \ldots, x_n\}$. We deduce that $D_{x_1,\ldots,x_n} \in R_{x_1,\ldots,x_n}$. For the sake of conciseness, from now on we write only R and D when the multiset $\{x_1, \ldots, x_n\}$ they refer to is obvious or irrelevant. It should be stressed that this definition is still valid in normal Γ-Model if we assume that $f_l(x_1, \ldots, x_n) = 1$ for all reaction rules. Because of normalization, we guarantee that

$$\sum_{\delta_l \in R} Pr(D = \delta_l) = 1$$

and we give the semantics of our extended operator Γ^{sto}

Definition. *The operator Γ^{sto} applied to a set of reaction rules Δ^{sto} and a multiset M is defined as*

$$\begin{aligned}\Gamma^{sto}(\Delta^{sto}, M) = {} & \exists \{x_1, \ldots, x_n\} \in M \mid R \neq \varnothing \\ & \underline{\textit{then}}\ \Gamma^{sto}(\Delta^{sto}, (M - \{x_1, \ldots, x_n\}) \cup a_l(x_1, \ldots, x_n)), \delta_l = D \\ & \underline{\textit{else}}\ M\end{aligned}$$

Finally we give the definition of stochastic Γ-Programs

Definition. *A sotchastic Γ-Program P is a 2-tuple $\langle \Delta^{sto}, M \rangle$ where*

$\Delta^{sto} = \{\delta_1, \ldots, \delta_m\}$	*is a set of reaction rules with frequencies*
$M = \{m_1, \ldots, m_n\}$	*is a multiset of molecules*

5.4 Properties of the Stochastic Γ-Model

In this section, we discuss the properties of our Stochastic Γ-Model. In particular, we show how we can compute state transitions, i.e. the evolution of the multiset. Then, based these properties, we show how we can use Discrete Time Markov Chains [118, 119] to express termination and confluence.

In section 4.2 we introduced the reduction operator $\rightarrow_\Gamma$ which, given a Γ-Program $P = \langle \Delta, M \rangle$, returns a multiset M' obtained after one application of a single rule of Δ on the multiset M. We write $\rightarrow_{\Gamma^{sto}}$ the same operator, extended to our stochastic model, and we introduce the notion of probability of reachability; that is the probability of a multiset M' to be obtained after a single reduction of M, given the program P. More formally, let ϵ denote a sequence of molecules $x_1, \ldots, x_n$ where $x_i \in M$ and $\delta_l \in \Delta^{sto}$ a reaction rule such that $(M - \epsilon) \cup a_l(\epsilon) = M'$.

Definition. *The probability of reachability of M', knowing $P = \langle M, \Delta^{sto} \rangle$ is given by*

$$Pr(\langle \Delta^{sto}, M \rangle \rightarrow_{\Gamma^{sto}} M') = \begin{cases} Pr(\epsilon = x_1, \ldots, x_n \mid M) \times Pr(D = \delta_l) & \textit{if } \exists \epsilon \\ 0 & \textit{otherwise} \end{cases}$$

In words, we first need to compute the probability of choosing the correct sequence ϵ out of M, and multiply it by the probability of choosing δ_l out of Δ^{sto}, knowing ϵ. As we can consider ϵ as one n-permutation of M, then we can compute the probability of choosing it with

$$Pr(\epsilon = x_1, \dots, x_n \mid M) = \frac{(|M| - n)!}{n!}$$

In the last section, we gave the definition of the probability to choose δ_l, knowing a multiset of molecules.

The reader may have noticed the similarities between our Stochastic Γ-Model and a Markov Chain. This may appear even clearer if we consider every multiset configuration as a state. Indeed, all multisets obtained by reduction of the initial one are in fact successor states of the initial state of the program, i.e. its initial multiset. Furthermore, one can predict expectations about the future behavior of the program, based only on the its current multiset configuration. This allows us to write the probability of reachability of a Stochastic Γ-Program as single step transition of a Markov Chain

$$Pr(\langle \Delta^{sto}, M \rangle \rightarrow_{\Gamma^{sto}} M') \equiv Pr(M_n = M' \mid M_{n-1} = M)$$

We extend our definition of reachable multisets $S(P)$ from section 4.2 to our stochastic model, and we can now define the probability of the n-steps transition from the multiset M to the multiset M', if $M' \in S(\langle \Delta, M \rangle)$. From now on we use the terms state or multiset interchangeably to denote the multiset of a program. We write the n-steps transition from a state s to a state t like

$$p^n_{s,t} = \sum_{r \in S(P)} p^k_{s,r} p^{n-k}_{r,t}$$

where $0 < k < n$, and

$$p^1_{s,t} = Pr(\langle \Delta^{sto}, s \rangle \rightarrow_{\Gamma^{sto}} t)$$

Now we may express termination and confluence of stochastic Γ-programs by the means of the n-steps stransition.

Definition. *A stochastic Γ-program $P = \langle \Delta^{sto}, M \rangle$ reaches termination iff*

$$\forall s, t \in S(P), p^1_{s,t} = 0, \forall t \in S(P) \backslash s$$

Definition. *A stochastic Γ-program $P = \langle \Delta^{sto}, M \rangle$ is confluent iff*

$$\forall s, t \in S(P), p^n_{s,t} = 1, \forall t \in S(P)$$

with n a finite number.

We may also rely on Markov Chains to identify some other interesting properties about the evolution of the multiset in a stochastic Γ-programs. From Levin et al. we emphasis on the definition of accessible state [119], which stands that a state t is accessible from s if there exists $n > 0$ such that $p^n_{s,t} > 0$. From

Norris, we emphasis on the notions of recurrent and transient states [118]: a state s is *recurrent* if

$$\sum_{n=1}^{\infty} p_{s,s}^{n} > 0$$

or *transient* if

$$\sum_{n=1}^{\infty} p_{s,s}^{n} = 0$$

In other words, a state is recurrent if the program keeps reach it, no matter the number of reaction rules applied; a state is transient if we eventually cannot return to it.

5.5 Limitations

Our Sotchastic Γ-Model proposes to explicitly define the frequency of the reaction rules execution. We saw that it reveals to be very useful to describe systems where some particular events should happen more often that others, or where the probability that en event occur depends on the values of the terms to be rewritten. However, to the point of view of model checking, it has no effect on the space of the possible outcomes of a computation. Indeed, if we consider a system where one rule has twice the frequency that another, we will remark that the state space remains identical to the one of another system, where the same two rules would happen with the same frequency. Actually, the state space of a sotchastic Γ-program may be influenced by the its rules frequencies only if one rule $\delta_l \in \Delta^{sto}$ can return a frequency $f_l(x_1, \ldots, x_n) = 0$ for a particular tuple of molecules $\langle x_1, \ldots, x_n \rangle$. But we remark that in such case, the frequency f_l acts as the reaction condition r_l; this means we could also define that $r_l(x_1, \ldots, x_n) = false$ for the same tuple of molecules to produce an equivalent state space.

Another limitation of our model is that rules frequencies are not absolutes. In fact, the actual probability of choosing one rule doesn't depend only on its frequency, but also on the frequency of the other rules whose condition is satisfied at the same time. This feature is required for the cases where the reaction rules operats on different subset of molecules which still overlaps. Consider the following rules

$$\delta_1 = \left\langle x > 0 \rightarrow x^2 \right\rangle$$
$$\delta_2 = \left\langle x < 5 \rightarrow \sqrt{x} \right\rangle$$

The frequencies are not relevant to our example, so we discard them for the sake of simplicity. We remark that both rules are satisfied for $x \in [0, 5]$ while only $delta_1$ is when $x \geq 5$ and only $delta_2$ is when $x \leq 0$. If δ_1 has a greater frequency than δ_2, the latter still needs a probability $Pr(D = \delta_2) = 1$ for all $x \leq 0$.

6 Conclusion

6.1 Contribution of this work

In this work, we studied the Γ-Model, a multiset rewriting model based on a chemical metaphor. Computation is expressed in term of reaction between molecules, representing the program data, floating within a common solution. We investigated on a way to formalize this model in order to perform model checking; namely we used Algebraic Petri Nets and proposed a method to translate any Γ-program into its equivalent Algebraic Petri Net. Using the model checking tool AlPiNA, we could give a proof of concept of our translation by constructing the state space of small problems and study some basic properties. Finally we proposed an extension of the Γ-Model, called Stochastic Γ-Model, to support stochastic decision for non discrete probability distributions.

We think that our main contribution was to enrich the Γ-Model with a new approach, i.e. the formal verification. By the means of our translation into the AP-nets, we could show that the Γ-Model could be exploited not only to describe algorithms to be implemented, but also to formally very these algorithms. Our second contribution is our extension: the Stochastic Γ-Model. We showed that the Γ-Model could not model a certain class of programs, were rewritings should be done with unfair frequencies.

6.2 Observations

We observed that the Γ-Model often performed poorly when it went to its implementation. Some researches have shown with more success that the Γ-Model could be implemented efficiently enough on specialized architectures [120]. But as pointed out in [121], the main issue lies in the fact that a lot of useless comparisons must be done against every permutation of molecules. Moreover, a single change in the multiset invalidates most of the comparison already computed. If implementations optimized for a particular program can perform reasonable performances, a general purpose implementation is likely to be unrealistic. We could confirm those observations with our own implementation, which is based on the work of Gladitz et al. [121]. We also observed that we faced access concurrency issues as we increased the processes.

But in contrast with this observation, we believe that the Γ-Model may well be used as a model checking model. In [78] Banâtre told about using the Γ-Model as a “bridge between specifications and implementations”; we think that we could use it *as* a specification language.

Finally, we may emphasis on our use of Markov Chains to express some properties of the Stochastic Γ-Model. We observed that our reasoning could also be used in the classic Γ-Model. From the view of program verification, an interesting advantage of this approach is the possibility the express the state space in terms of reachability probability.

6.3 Future works

Our work was based exclusively on the first order Γ-Model. An interesting step forward would be to explore the translation of higher-order models, such as the γ-calculus [98]. Such models have the ability to encapsulate sub-solutions, in which computation is independent of the parent solutions until it reaches a stable states. Such behavior is hard to reproduce in Algebraic Petri Nets, since transitions should be fireable only as soon as complete sub-system has reached termination, but not before. A lead in this direction may be LLAMAS [122], proposed by Marechal, where transitions can be synchronized.

In the direction of the observations we made above, an interesting future work would be to build a genuine model checker fot the Γ-Model. We saw that the Algebraic Petri Nets and the Γ-Model shared a lot of characteristics, both being rewriting systems. This means that underlying theory of Algebraic Petri Nets model checking could be a good start for this purpose. In particular, we think that Σ Decision Diagrams may be a good candidate to encode the state space of Γ-Programs.

References

[1] V. Lifschitz, "What is answer set programming?," in *AAAI* (D. Fox and C. P. Gomes, eds.), pp. 1594–1597, AAAI Press, 2008.

[2] W. M. Johnston, J. R. P. Hanna, and R. J. Millar, "Advances in dataflow programming languages," *ACM Comput. Surv.*, vol. 36, pp. 1–34, Mar. 2004.

[3] R. M. Karp and R. E. Miller, "Properties of a model for parallel computations: determinacy, termination, queueing," *Siam Journal on Applied Mathematics*, vol. 14, pp. 1390–1411, Nov. 1966.

[4] W. W. Wadge and E. A. Ashcroft, *LUCID, the dataflow programming language.* San Diego, CA, USA: Academic Press Professional, Inc., 1985.

[5] P. R. Kosinski, "A data flow language for operating systems programming," in *Proceeding of ACM SIGPLAN - SIGOPS interface meeting on Programming languages - operating systems*, (New York, NY, USA), pp. 89–94, ACM, 1973.

[6] J. B. Dennis and D. P. Misunas, "A preliminary architecture for a basic data-flow processor," in *Proceedings of the 2nd annual symposium on Computer architecture*, ISCA '75, (New York, NY, USA), pp. 126–132, ACM, 1975.

[7] A. L. Davis, "The architecture and system method of ddm1: A recursively structured data driven machine," in *Proceedings of the 5th annual symposium on Computer architecture*, ISCA '78, (New York, NY, USA), pp. 210–215, ACM, 1978.

[8] T. Temma, M. Iwashita, K. Matsumoto, H. Kurokawa, and T. Nukiyama, "Data flow processor chip for image processing," *Electron Devices, IEEE Transactions on*, vol. 32, pp. 1784–1791, Sept. 1985.

[9] T. Jeffery, "The upd7281 processor," *Byte*, vol. 10, pp. 237–246, Nov. 1985.

[10] A. H. Veen, "Dataflow machine architecture," *ACM Comput. Surv.*, vol. 18, pp. 365–396, Dec. 1986.

[11] L. Bic, "A process-oriented model for efficient execution of dataflow programs," *J. Parallel Distrib. Comput.*, vol. 8, pp. 42–51, Jan. 1990.

[12] J. Silc, B. Robic, and T. Ungerer, "Asynchrony in parallel computing: From dataflow to multithreading," *Journal of Parallel and Distributed Computing Practices*, vol. 1, no. 1, pp. 1–33, 1998.

[13] J. P. Morrison, *Flow-Based Programming: A New Approach to Application Development.* New York, NY, USA: Van Nostrand Reinhold, 1994.

[14] T. A. Alves, L. A. Marzula, F. M. Franca, and V. S. Costa, "Trebuchet: exploring tlp with dataflow virtualisation," *Int. J. High Perform. Syst. Archit.*, vol. 3, pp. 137–148, May 2011.

[15] S. Balakrishnan and G. S. Sohi, "Program demultiplexing: Data-flow based speculative parallelization of methods in sequential programs," *SIGARCH Comput. Archit. News*, vol. 34, pp. 302–313, May 2006.

[16] A. L. Davis, "Data driven nets - a class of maximally parallel, output-functional program schemata," *Technical report*, 1974.

[17] A. Davis and R. Keller, "Data flow program graphs," *Computer*, vol. 15, pp. 26–41, Feb. 1982.

[18] J. Browne, S. Hyder, J. Dongarra, K. Moore, and P. Newton, "Visual programming and debugging for parallel computing," *Parallel Distributed Technology: Systems Applications, IEEE*, vol. 3, no. 1, pp. 75–83, 1995.

[19] D. D. Hils, "Visual languages and computing survey: Data flow visual programming languages," *Journal of Visual Languages & Computing*, vol. 3, pp. 69–101, Mar. 1992.

[20] M. G. Vose and G. Williams, "Laboratory virtual instrument engineering workbench," *BYTE*, vol. 11, pp. 84–92, Sept. 1986.

[21] *LabVIEW - User Manual.* Austin, TX, USA: National Instruments Corporation, Apr. 2003.

[22] S. Matwin and T. Pietrzykowski, "Prograph: A preliminary report," *Computer Languages*, vol. 10, no. 2, pp. 91–126, 1985.

[23] P. Cox, F. R. Giles, and T. Pietrzykowski, "Prograph: a step towards liberating programming from textual conditioning," in *Visual Languages, 1989., IEEE Workshop on*, pp. 150–156, 1989.

[24] *Marten - Quick Start.* Bozeman, MT, USA: Andescotia LLC, Dec. 2005.

[25] T. Green and M. Petre, "Usability analysis of visual programming environments: A cognitive dimensions framework," *Journal of Visual Languages & Computing*, vol. 7, pp. 131–174, June 1996.

[26] G. Berry, "Scade: Synchronous design and validation of embedded control software," in *Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems* (S. Ramesh and P. Sampath, eds.), pp. 19–33, Springer Netherlands, 2007.

[27] J. Rizzo, *Cinema 4D Beginner's Guide.* Birmingham, UK: Packt Publishing, Nov. 2012.

[28] K. M. Kavi, B. P. Buckles, and U. N. Bhat, "A formal definition of data flow graph models," *IEEE Trans. Comput.*, vol. 35, pp. 940–948, Nov. 1986.

[29] P. D. Bruza and T. van der Weide, "The semantics of data flow diagrams," in *In Proceedings of the International Conference on Management of Data*, pp. 66–78, McGraw-Hill Publishing Company, 1993.

[30] W. P. Stevens, G. J. Myers, and L. L. Constantine, "Structured design," *IBM Systems Journal*, vol. 13, no. 2, pp. 115–139, 1974.

[31] E. Yourdon, “Just enough structured analysis.” Accessed October 14, 2013.

[32] C. Petri, *Kommunikation mit Automaten.* Schriften des Rheinisch-Westfälischen Instituts für Instrumentelle Mathematik an der Universität Bonn, Rhein.-Westfäl. Inst. f. Instrumentelle Mathematik an der Univ. Bonn, 1962.

[33] T. Murata, “Petri nets: Properties, analysis and applications,” *Proceedings of the IEEE*, vol. 77, pp. 541–580, Apr. 1989.

[34] A. Pinl, *Probabiliy Propagation Nets.* PhD thesis, Univesität Koblenz Landau, Nov. 2007.

[35] Y. Liu, H.-K. Miao, H.-W. Zeng, Y. Ma, and P. Liu, “Nondeterministic probabilistic petri net a new method to study qualitative and quantitative behaviors of system,” *Journal of Computer Science and Technology*, vol. 28, pp. 203–216, Jan. 2013.

[36] P. Lauer, P. Torrigiani, and M. Shields, “Cosy a system specification language based on paths and processes,” *Acta Informatica*, vol. 12, no. 2, pp. 109–158, 1979.

[37] E. Falkenberg, R. van der Pols, and T. van der Weide, “Understanding process structure diagrams,” *Information Systems*, vol. 16, no. 4, pp. 417–428, 1991.

[38] J. Dean and S. Ghemawat, “Mapreduce: simplified data processing on large clusters,” *Commun. ACM*, vol. 51, pp. 107–113, Jan. 2008.

[39] T. White, *Hadoop: The Definitive Guide.* USA: Yahoo! Press, 2010.

[40] R. Bird and P. Wadler, *An introduction to functional programming.* Hertfordshire, UK, UK: Prentice Hall International (UK) Ltd., 1988.

[41] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker, “A comparison of approaches to large-scale data analysis,” in *SIGMOD ’09: Proceedings of the 35th SIGMOD international conference on Management of data*, (New York, NY, USA), pp. 165–178, ACM, 2009.

[42] K.-H. Lee, Y.-J. Lee, H. Choi, Y. D. Chung, and B. Moon, “Parallel data processing with mapreduce: a survey,” *SIGMOD Rec.*, vol. 40, pp. 11–20, Jan. 2012.

[43] A. F. Gates, O. Natkovich, S. Chopra, P. Kamath, S. M. Narayanamurthy, C. Olston, B. Reed, S. Srinivasan, and U. Srivastava, “Building a high-level dataflow system on top of map-reduce: the pig experience,” *Proc. VLDB Endow.*, vol. 2, pp. 1414–1425, Aug. 2009.

[44] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan, “Interpreting the data: Parallel analysis with sawzall,” *Sci. Program.*, vol. 13, pp. 277–298, Oct. 2005.

[45] D. Battré, S. Ewen, F. Hueske, O. Kao, V. Markl, and D. Warneke, "Nephele/pacts: a programming model and execution framework for web-scale analytical processing," in *Proceedings of the 1st ACM symposium on Cloud computing*, SoCC '10, (New York, NY, USA), pp. 119–130, ACM, 2010.

[46] G. A. Papadopoulos and F. Arbab, "Coordination models and languages," *Advances in computers*, vol. 46, pp. 329–400, 1998.

[47] D. Gelernter and N. Carriero, "Coordination languages and their significance," *Commun. ACM*, vol. 35, pp. 97–107, Feb. 1992.

[48] D. Gelernter, "Generative communication in linda," *ACM Trans. Program. Lang. Syst.*, vol. 7, pp. 80–112, Jan. 1985.

[49] S. Ahuja, N. Carriero, and D. Gelernter, "Linda and friends," *Computer*, vol. 19, no. 8, pp. 26–34, 1986.

[50] G. Sutcliffe, "Prolog-d-linda v2 : A new embedding of linda in sicstus prolog," in *Proc. Workshop on Blackboard-based Logic Programming*, pp. 105–117, 1993.

[51] G. C. Wells, "New and improved: Linda in java," *Science of Computer Programming*, vol. 59, pp. 82–96, Jan. 2006.

[52] C. Tech Correspondence, "Technical correspondence," *Commun. ACM*, vol. 32, pp. 1241–1258, Oct. 1989.

[53] G. Roman and H. Cunningham, "Mixed programming metaphors in a shared dataspace model of concurrency," *Software Engineering, IEEE Transactions on*, vol. 16, pp. 1361–1373, Jan. 1990.

[54] D. Gelernter, "Multiple tuple spaces in linda," in *PARLE '89 Parallel Architectures and Languages Europe* (E. Odijk, M. Rem, and J.-C. Syre, eds.), vol. 366 of *Lecture Notes in Computer Science*, pp. 20–27, Springer Berlin Heidelberg, 1989.

[55] S. Castellani, P. Ciancarini, and D. Rossi, "The shape of shade: a coordination system," tech. rep., 1996.

[56] G. C. Wells, *A Programmable Matching Engine for Application Development in Linda.* PhD thesis, University of Bristol, July 2001.

[57] N. Busi, R. Gorrieri, R. Lucchi, and G. Zavattaro, "Secspaces: a data-driven coordination model for environments open to untrusted agent," *Electronic Notes in Theoretical Computer Science*, vol. 68, pp. 310–27, Jan. 2003.

[58] L. j. b. Nixon, E. Simperl, R. Krummenacher, and F. Martin-recuerda, "Tuplespace-based computing for the semantic web: A survey of the state-of-the-art," *Knowl. Eng. Rev.*, vol. 23, pp. 181–212, June 2008.

[59] P. Álvarez, J. A. Bañares, P. R. Muro-Medrano, J. Nogueras, and F. J. Zarazaga, "A java coordination tool for web-service architectures: The location-based service context," in *Revised Papers from the International Workshop on Scientific Engineering for Distributed Java Applications*, FIDJI '01, (London, UK, UK), pp. 1–14, Springer-Verlag, 2003.

[60] R. Lucchi and G. Zavattaro, "Wssecspaces: a secure data-driven coordination service for web services applications," in *Proceedings of the 2004 ACM symposium on Applied computing*, SAC '04, (New York, NY, USA), pp. 487–491, ACM, 2004.

[61] S. Frølund and G. Agha, "A language framework for multi-object coordination," in *Proceedings of the 7th European Conference on Object-Oriented Programming*, ECOOP '93, (London, UK, UK), pp. 346–360, Springer-Verlag, 1993.

[62] P. Obreiter and G. Gräf, "Towards scalability in tuple spaces," in *Proceedings of the 2002 ACM symposium on Applied computing*, SAC '02, (New York, NY, USA), pp. 344–350, ACM, 2002.

[63] P. Ciancarini, R. Gorrieri, and G. Zavattaro, "Generative communication in process algebra," tech. rep., 1995.

[64] R. Milner, *Communication and concurrency.* Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1989.

[65] N. Busi, R. Gorrieri, and G. Zavattaro, "On the semantics of javaspaces," in *Formal Methods for Open Object-Based Distributed Systems IV* (S. Smith and C. Talcott, eds.), vol. 49 of *IFIP Advances in Information and Communication Technology*, pp. 3–19, Springer US, 2000.

[66] A. Pnueli, "The temporal logic of programs," in *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, SFCS '77, (Washington, DC, USA), pp. 46–57, IEEE Computer Society, 1977.

[67] L. Semini and C. Montangero, "A refinement calculus for tuple spaces," *Science of Computer Programming*, vol. 34, no. 2, pp. 79 – 140, 1999.

[68] K. Chandy and J. Misra, *Parallel program design: a foundation.* Computer Science Series, Addison-Wesley Pub. Co., 1988.

[69] H. C. Cunningham, *The shared dataspace approach to concurrent computation: the swarm programming model, notation, and logic.* PhD thesis, St. Louis, MO, USA, 1989.

[70] G. Papadopoulos and F. Arbab, "Control-driven coordination programming in shared dataspace," in *Parallel Computing Technologies* (V. Malyshkin, ed.), vol. 1277 of *Lecture Notes in Computer Science*, pp. 247–261, Springer Berlin Heidelberg, 1997.

[71] E. Tryggeseth, B. Gulla, and R. Conradi, "Modelling systems with variability using the proteus configuration language," in *Selected papers from the ICSE SCM-4 and SCM-5 Workshops, on Software Configuration Management*, (London, UK, UK), pp. 216–240, Springer-Verlag, 1995.

[72] J. Magee and M. Sloman, “Constructing distributed systems in conic,” *IEEE Trans. Softw. Eng.*, vol. 15, pp. 663–675, June 1989.

[73] M. Buffo, “Experiences in coordination programming,” in *Database and Expert Systems Applications, 1998. Proceedings. Ninth International Workshop on*, pp. 536–541, 1998.

[74] F. Arbab, “Coordination of mobile components,” *Electronic Notes in Theoretical Computer Science*, vol. 54, pp. 1–16, Aug. 2001.

[75] J.-P. Banâtre and D. Le Métayer, “A new computational model and its discipline of programming,” Tech. Rep. RR-0566, INRIA, 1986.

[76] M. Rem, “Associons: A program notation with tuples instead of variables,” *ACM Trans. Program. Lang. Syst.*, vol. 3, pp. 251–262, July 1981.

[77] A. Gupta, C. Forgy, A. Newell, and R. Wedig, “Parallel algorithms and architectures for rule-based systems,” *SIGARCH Comput. Archit. News*, vol. 14, pp. 28–37, May 1986.

[78] J.-P. Banâtre, P. Fradet, and D. L. Métayer, “Gamma and the chemical reaction model: Fifteen years after,” in *Proceedings of the Workshop on Multiset Processing: Multiset Processing, Mathematical, Computer Science, and Molecular Computing Points of View*, WMP ’00, (London, UK, UK), pp. 17–44, Springer-Verlag, 2001.

[79] J.-P. Banâtre, P. Fradet, and Y. Radenac, “Software-intensive systems and new computing paradigms,” ch. The Chemical Reaction Model Recent Developments and Prospects, pp. 209–234, Berlin, Heidelberg: Springer-Verlag, 2008.

[80] J.-M. Andreoli and R. Pareschi, “Logic programming,” ch. Linear objects: logical processes with built-in inheritance, pp. 495–510, Cambridge, MA, USA: MIT Press, 1990.

[81] I. Cervesato, N. A. Durgin, P. D. Lincoln, J. C. Mitchell, and A. Scedrov, “A meta-notation for protocol analysis,” in *Proceedings of the 12th IEEE workshop on Computer Security Foundations*, CSFW ’99, (Washington, DC, USA), pp. 55–, IEEE Computer Society, 1999.

[82] H. Ehrig and B. Mahr, *Fundamentals of algebraic specification: Equations and initial semantics.* EATCS monographs on theoretical computer science, Springer-Verlag, 1985.

[83] I. Cervesato, “Typed msr: Syntax and examples,” in *Proceedings of the International Workshop on Information Assurance in Computer Networks: Methods, Models, and Architectures for Network Security*, MMM-ACNS ’01, (London, UK, UK), pp. 159–177, Springer-Verlag, 2001.

[84] F. Rosa-Velardo, “Multiset rewriting: a semantic framework for concurrency with name binding,” in *Proceedings of the 8th international conference on Rewriting logic and its applications*, WRLA’10, (Berlin, Heidelberg), pp. 191–207, Springer-Verlag, 2010.

[85] P. Fradet and D. L. Mtayer, “Structured gamma,” *Science of Computer Programming*, vol. 31, no. 23, pp. 263 – 289, 1998.

[86] P. Fradet and D. Le Métayer, “Shape types,” in *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL ’97, (New York, NY, USA), pp. 27–39, ACM, 1997.

[87] C. Hankin, D. Le Métayer, and D. Sands, “A calculus of gamma programs,” in *Languages and Compilers for Parallel Computing* (U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, eds.), vol. 757 of *Lecture Notes in Computer Science*, pp. 342–355, Springer Berlin Heidelberg, 1993.

[88] D. Sands, “A compositional semantics of combining forms for gramma programs,” in *Proceedings of the International Conference on Formal Methods in Programming and Their Applications*, (London, UK, UK), pp. 43–56, Springer-Verlag, 1993.

[89] D. Le Métayer, “Higher-order multiset programming,” 1994.

[90] C. Hankin, D. Le Métayer, and D. Sands, “A parallel programming style and its algebra of programs,” in *PARLE ’93 Parallel Architectures and Languages Europe* (A. Bode, M. Reeve, and G. Wolf, eds.), vol. 694 of *Lecture Notes in Computer Science*, pp. 367–378, Springer Berlin Heidelberg, 1993.

[91] D. Sands, “Coordination programming,” ch. Composed reduction systems, pp. 211–231, London, UK, UK: Imperial College Press, 1996.

[92] G. Berry and G. Boudol, “The chemical abstract machine,” in *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL ’90, (New York, NY, USA), pp. 81–94, ACM, 1990.

[93] A. Samson, “Computational interpretations of linear logic,” *Theoretical Computer Science*, vol. 111, pp. 3–57, Apr. 1993.

[94] R. Milner, “Functions as processes,” in *Automata, Languages and Programming* (M. Paterson, ed.), vol. 443 of *Lecture Notes in Computer Science*, pp. 167–180, Springer Berlin Heidelberg, 1990.

[95] R. Milner, *Communicating and mobile systems: the pi-calculus.* New York, NY, USA: Cambridge University Press, 1999.

[96] I. Mackie and S. Sato, “A calculus for interaction nets based on the linear chemical abstract machine,” *Electron. Notes Theor. Comput. Sci.*, vol. 192, pp. 59–70, Nov. 2008.

[97] Y. Lafont, “Interaction nets,” in *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL ’90, (New York, NY, USA), pp. 95–108, ACM, 1990.

[98] J.-P. Banâtre, P. Fradet, and Y. Radenac, “Principles of chemical programming,” *Electronic Notes in Theoretical Computer Science*, vol. 124, pp. 133 – 147, Mar. 2005.

[99] G. Revesz, *Lambda-calculus combinators and functional programming.* New York, NY, USA: Oxford University Press, Inc., 1988.

[100] H. Ehrig, J. Padberg, G. Juhás, and G. Rozenberg, *Unifying Petri Nets*, vol. 2128 of *Lecture Notes in Computer Science.* 2001.

[101] K. Jensen, “High-level petri nets,” in *Applications and Theory of Petri Nets* (A. Pagnoni and G. Rozenberg, eds.), vol. 66 of *Informatik-Fachberichte*, pp. 166–180, Springer Berlin Heidelberg, 1983.

[102] K. Jensen, “Coloured petri nets and the invariant-method,” *Theoretical Computer Science*, vol. 14, no. 3, pp. 317 – 336, 1981.

[103] H. Genrich and K. Lautenbach, “System modelling with high-level petri nets,” *Theoretical Computer Science*, vol. 13, no. 1, pp. 109 – 135, 1981.

[104] W. Reisig, “Petri nets and algebraic specifications,” *Theoretical Computer Science*, vol. 80, no. 1, pp. 1–34, 1991.

[105] S. P. Hostettler, *High-Level Petri Net Model Checking.* PhD thesis, University of Geneva, Nov. 2011.

[106] J. Baeten, “A brief history of process algebra,” *Theoretical Computer Science*, vol. 335, no. 23, pp. 131 – 146, 2005.

[107] C. A. R. Hoare, *Communicating Sequential Processes.* Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1985.

[108] E. M. Clarke and E. A. Emerson, “Design and synthesis of synchronization skeletons using branching-time temporal logic,” in *Logic of Programs, Workshop*, (London, UK, UK), pp. 52–71, Springer-Verlag, 1982.

[109] R. E. Bryant, “Graph-based algorithms for boolean function manipulation,” *IEEE Trans. Comput.*, vol. 35, pp. 677–691, Aug. 1986.

[110] J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang, “Symbolic model checking: 1020 states and beyond,” *Information and Computation*, vol. 98, no. 2, pp. 142 – 170, 1992.

[111] T. Kam, T. Villa, R. K. Brayton, and A. L. Sangiovanni-Vincentelli, “Multi-valued decision diagrams for logic synthesis and verification,” Tech. Rep. UCB/ERL M96/75, EECS Department, University of California, Berkeley, 1996.

[112] J.-M. Couvreur, E. Encrenaz, E. Paviot-Adet, D. Poitrenaud, and P.-A. Wacrenier, “Data decision diagrams for petri net analysis,” in *Application and Theory of Petri Nets 2002* (J. Esparza and C. Lakos, eds.), vol. 2360 of *Lecture Notes in Computer Science*, pp. 101–120, Springer Berlin Heidelberg, 2002.

[113] J.-M. Couvreur and Y. Thierry-Mieg, “Hierarchical decision diagrams to exploit model structure,” in *Formal Techniques for Networked and Distributed Systems - FORTE 2005* (F. Wang, ed.), vol. 3731 of *Lecture Notes in Computer Science*, pp. 443–457, Springer Berlin Heidelberg, 2005.

[114] D. Buchs and S. P. Hostettler, *Sigma Decision Diagrams.* TERMGRAPH 2009 : Preliminary proceedings of the 5th International Workshop on Computing with Terms and Graphs, 2009.

[115] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, "Symbolic model checking without bdds," in *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '99, (London, UK, UK), pp. 193–207, Springer-Verlag, 1999.

[116] S. Hostettler, A. Marechal, A. Linard, M. Risoldi, and D. Buchs, "High-level petri net model checking with alpina," *Fundamenta Informaticae*, vol. 113, pp. 229–264, Aug. 2011.

[117] J.-P. Banâtre, A. Coutant, and D. Le Metayer, "A parallel machine for multiset transformation and its programming style," *Future Gener. Comput. Syst.*, vol. 4, pp. 133–144, Sept. 1988.

[118] J. Norris, *Markov Chains.* No. No. 2008 in Cambridge Series in Statistical and Probabilistic Mathematics, Cambridge University Press, 1998.

[119] D. Levin, Y. Peres, and E. Wilmer, *Markov Chains and Mixing Times.* American Mathematical Society, 2009.

[120] M. Viellot, "Gamma program synthesis in reconfigurable logics," *TSI. Technique et science informatiques*, vol. 14, no. 5, pp. 567–583.

[121] K. Gladitz and H. Kuchen, "Shared memory implementation of the gamma-operation," *J. Symb. Comput.*, vol. 21, pp. 577–591, June 1996.

[122] A. Marechal and D. Buchs, "Unifying the semantics of modular extensions of petri nets," in *Application and Theory of Petri Nets and Concurrency* (J.-M. Colom and J. Desel, eds.), vol. 7927 of *Lecture Notes in Computer Science*, pp. 349–368, Springer Berlin Heidelberg, 2013.

[123] J.-M. Andreoli, P. Ciancarini, and R. Pareschi, "Research directions in concurrent object-oriented programming," ch. Interaction abstract machines, pp. 257–280, Cambridge, MA, USA: MIT Press, 1993.

[124] P. Ciancarini, D. Fogli, and M. Gaspari, "A logic language based on gamma-like multiset rewriting," in *Proceedings of the 5th International Workshop on Extensions of Logic Programming*, ELP '96, (London, UK, UK), pp. 83–101, Springer-Verlag, 1996.

[125] P. Inverardi and A. L. Wolf, "Formal specification and analysis of software architectures using the chemical abstract machine model," *IEEE Trans. Softw. Eng.*, vol. 21, pp. 373–386, Apr. 1995.

[126] C. Marché, "Rewriting computation and proof," ch. Towards Modular Algebraic Specifications for Pointer Programs: A Case Study, pp. 235–258, Berlin, Heidelberg: Springer-Verlag, 2007.

[127] B. Möller, "Towards pointer algebra," *Science of Computer Programming*, vol. 21, no. 1, pp. 57 – 90, 1993.