THESIS / THÈSE

MASTER IN COMPUTER SCIENCE

Towards distributed model checking: a network memory storage mechanism

Miche, Geoffrey

Award date:
2004

Institute of Informatics
University of Namur
Namur, Belgium

Towards distributed model checking: A network memory storage mechanism

Geoffrey Miche

Thesis presented in order to obtain a Master's degree in Computer Science
Academic year 2003-2004

Résumé

Aujourd'hui, les systèmes hardware et logiciels continuent de prendre une place importante dans notre vie quotidienne. Pensons, par exemple, aux ascenseurs, aux systèmes de contrôle aérien et aux trains. Ces exemples montrent l'importance de la fiabilité de tels systèmes. En effet, il est facile d'imaginer le désastre qu'occasionnerait un système de contrôle aérien défectueux. Une des conséquences pourrait être que deux avions entrent en collision à cause d'une erreur dans la partie du système qui donne les positions des avions. Vérifier la correction de tels systèmes est donc d'une importance cruciale.

Des méthodes existent pour vérifier de tels systèmes. L'une d'entre elles s'appelle *model checking* et a souvent été utilisée avec succès pour vérifier des systèmes complexes. Le défit principal que le model checking doit relever est de faire face au problème bien connu de l'explosion de l'espace d'états.

Des techniques pour éviter ce problème existent. Plusieurs d'entre elles telles que la *partial order reduction* ou le *model checking on-the-fly* sont largement répandues. De nouvelles techniques continuent cependant d'être développées et expérimentées. Leur but étant de repousser la limite des systèmes sujets à ce problème.

Nous proposons, dans ce mémoire, une méthode pour éviter le problème de l'explosion de l'espace d'états. L'idée est d'augmenter la puissance de calcul et plus particulièrement la mémoire principale en utilisant une grappe d'ordinateurs connectés entre eux par un réseau. L'objectif est de pouvoir allouer des quantités de mémoire très importantes pour des systèmes complexes tout en étant performant.

Abstract

Nowadays, hardware and software systems continue to take an important place in our daily life. Let us think of elevators, air traffic control systems and trains. These examples show the major importance for such systems to be reliable. Indeed, they often contain bugs and it is crucial to verify their correctness in order to avoid problems. We can easily imagine the disaster if an air traffic control system was not verified. One of the consequences could be that two planes collide with each other because of a bug in the part of the system that gives the plane positions.

Methods exist for verifying such systems. One of them is called *model checking* and has been used successfully in practice to verify complex systems. The main challenge with model checking is to deal with the well-known problem of the *state space explosion.*

Techniques to avoid this problem exist as well. Some of them such as *partial order reduction* or *on-the-fly model checking* are widely spread. New techniques however continue to be developed and experimented. The goal is to push away the limit of systems subject to state explosion.

We propose in this master thesis a method to avoid state space explosion. The idea is to increase the computational power and more specifically random-access memory by using a cluster with computers connected between them by a network. The objective is to allow the allocation of huge amounts of memory for complex systems and to be efficient.

I want to thank more particularly my instigator, Mr. Jean-Marie Jacquet for all his precious advice during the writing of this thesis.

I also want to thank the research team of ParaDiSe Laboratory and Professor Luboš Brim for their welcome during my stay and especially my supervisor Jiří Barnat for his precious advice during the development of the prototype and the writing of my thesis. A special thank to Office for International Studies of Masaryk University which allowed me to spend a pleasant stay in Czech Republic thanks to the activities they organize for Erasmus students.

I also think of my sister Magali, Mic, M-L, Sven, Jean-Bernard and Ged for their precious help.

Special thanks to my girlfriend and my parents who supported me during the writing of my thesis.

Contents

List of Figures

List of Tables

Table of acronyms

LTL Linear Temporal Logic

CTL Computational Tree Logic

FIFO First In First Out

MPI Message Passing Interface

ISO International Standards Organization

ANSI American National Standards Institute

IEEE Institute of Electrical and Electronics Engineers

PVM Parallel Virtual Machine

TID Task Identifier

CORBA Common Object Request Broker Architecture

OMG Object Management Group

IDL Interface Description Language

ORB Object Request Broker

SII Static Invocation Interface

DII Dynamic Invocation Interface

IR Interface Repository

SSI Skeleton Static Interface

DSI Dynamic Skeleton Interface

OA Object Adapter

ImpIR Implementation Repository

DFS Depth-First Search

BFS Breadth-First Search

SSSP Single Source Shortest Path Problem

SCC Strongly Connected Component

LRU Last Recently Used

TCP Transmission Control Protocol

UDP User Datagram Protocol

Introduction

Nowadays, hardware and software systems are widely used in applications where failure is unacceptable. Let us think of air traffic control systems, elevator control systems and trains. These examples illustrate that it is very important to have reliable systems. A well-known example of such a failure system is the Ariane 5 rocket which exploded less than forty seconds after being launched. We can also imagine that two planes collide with each other because of a bug in the part of the air traffic control system in charge of the plane positions.

The need for reliable hardware and software systems is clearly critical in view of the involvement of such systems in our lives. Techniques to verify the correctness of those systems exist. One of the purposes of this master thesis is to introduce the reader to the verification technique called *model checking*. It is a well-known method for automatic verification of software and concurrent systems. It has been successfully used to discover well-hidden bugs in substantial industrial systems.

Model checking consists in verifying properties that a system must satisfy. This check is done on a model which represents the system in question. Unfortunately, everything is not perfect and this method can encounter a major problem. It is the *state space explosion problem*. It appears when the model of a system contains a huge number of states. As a consequence, the verification of systems encountering this problem is not really efficient. A reason is that the resources of computers are limited. Methods to face up to this problem exist and are quite efficient. Some of them consist in reducing the size of the memory used. The aim of one of these methods is to take advantage of the resources of several computers linked by a network. It is called *distributed model checking* and algorithms for model checking are distributed. However, the latter suffers from *revisiting*[1] when the graph modeling the system is browsed.

The main purpose of this master thesis is the design and the experimentation of a prototype which takes advantage of the increase of the computational power (especially random-access memory) and which does not suffer from revisiting. Another goal of this prototype is to allow to allocate more than 4 Gigabytes of memory to a program which is the maximal amount of memory actually allocated to one process. The prototype was implemented in the framework of the DiVinE project[2] during three months.

The thesis is composed of four chapters. The first one introduces the theoretical concepts of model checking necessary to understand our work. It describes the tasks to do in order to apply model checking to systems. The first task is *modeling* and uses *automata*. The second

[1]Revisiting is a word used to express the re-exploring of states many times.

[2]DiVinE is currently developed in the *ParaDiSe laboratory* of the Faculty of Informatics, Masaryk University, Brno (Czech Republic).

one is *specification* which states the properties that the model of a system must satisfy. It uses *temporal logics*. The last task is the *verification* which is generally done by a model checker and which is described in a theoretical manner in the chapter. Examples are often given to allow the reader to understand better the theoretical basis.

The second chapter is devoted to the description of the *state space explosion problem* and solutions to avoid it.

The third chapter describes the design of the prototype from the analysis of the available communication technologies between computers in a cluster to the results of the experimentation. It also describes the working of the prototype.

The fourth chapter concludes this thesis by introducing possible improvements or methods to implement the prototype from the beginning.

Chapter 1

Model checking

We are going to introduce in this chapter some theoretical concepts of model checking. Automata and temporal logics are basic elements of model checking. Actually the algorithms used for model checking check if a given automaton satisfies a given temporal formula. This chapter also tackles some important properties and finishes with the description of a tool for model checking.

1.1 Automata

An automaton is just a machine evolving from one state to another one thanks to the action of transitions. Automaton is a general concept which draws characteristics from the *finite automata* in language theory, from *Kripke structures* or from *transition systems* in other areas. Kripke structures, transition system and Büchi automata for automata theory are defined in this section. The purpose of their use is to model systems in order to apply model checking. This section is a summary of [Sch04], [BBF+01] and [JGP99].

1.1.1 An automaton example

Example 1.1. This example from [Galps] introduces the section about automaton. The automaton in figure 1.1 represents the model of a traffic light. The behavior of the automaton is the following one. The color of the traffic light is *red* at the *initial state*. The light turns green if the light is red and a car is present. If the light is orange, then the light turns red. When the light is already green and a car is present, the light remains green. On the other hand, if the light is green and there is no car twice in a row, then the light turns orange.

1.1.2 Definition of an automaton

Definition 1.1. We first define a set $Prop = \{P_1, P_2, \ldots\}$ which gives us the set of elementary properties. Now we can define an automaton as a tuple $M = \langle S, E, T, S_0, l \rangle$.

- S is a finite set of states;
- E is the finite set of transition labels;

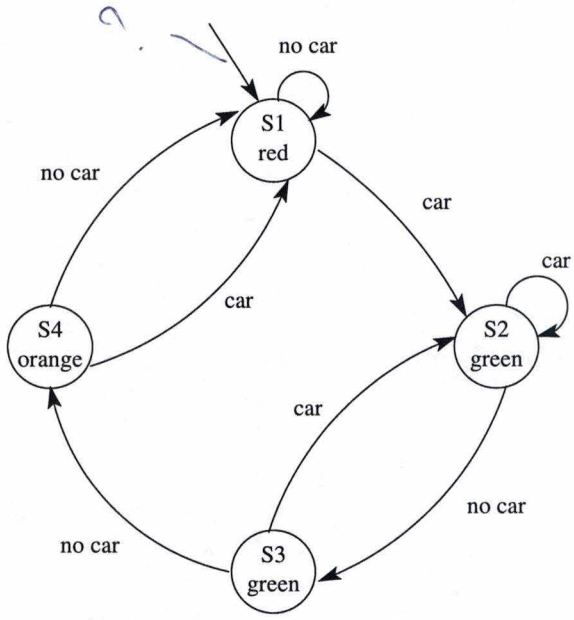

Figure 1.1: Automaton of a traffic light

- $T \subseteq S \times E \times S$ is the set of transitions;
- $S_0 \subseteq S$ is the set of initial states;
- l is the mapping which associates with each state of S the finite set of elementary properties which hold in that state.

If we take the first example in figure 1.1, we obtain :

$S = \{S_1, S_2, S_3, S_4\}$
$E = \{car, no\ car\}$
$S_0 = \{S_1\}$;
$T = \{(S_1, no\ car, S_1), (S_1, car, S_2), (S_2, car, S_2), (S_2, no\ car, S_3), (S_3, car, S_2), (S_3, no\ car, S_4), (S_4, no\ car, S_1), (S_4, car, S_1)\}$

$$l = \begin{cases} S_1 \rightarrowtail \{red\} \\ S_2 \rightarrowtail \{green, car\} \\ S_3 \rightarrowtail \{green, \neg car\} \\ S_4 \rightarrowtail \{orange\} \end{cases}$$

Other useful definitions are as follows. The first concerns the path in an automaton. The second is relative to the execution and the last concerns states with the property of being reachable.

Definition 1.2. A *path* in an automaton can be defined as a sequence σ, finite or infinite, of transitions (s_i, e_i, s'_i) of M which follow each other.

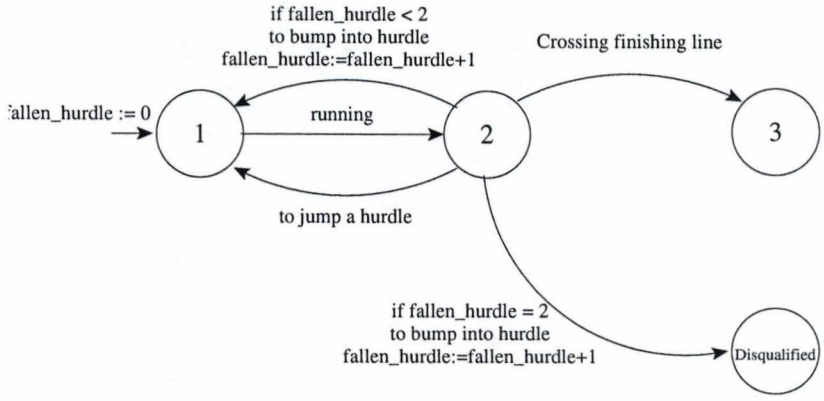

Figure 1.2: Example of an automaton with state variables

Definition 1.3. A *complete execution* or *execution* is an execution which is *maximal.* That is to say it cannot be extended.

Definition 1.4. A *reachable state* is a state which appears in the execution tree of an automaton.

1.1.3 Automata and state variables

It is also possible to add what is called *state variables* in an automaton. A state variable can be considered as a *guard* for transitions. This means that a transition cannot occur if the condition on the variables does not hold. A feature is added to transitions: they can modify the value of the variable(s). The example of automaton with state variables shown in figure 1.2 defines a hurdle race. If the runner bumps into more than two hurdles during the race, he is disqualified otherwise he finishes the race. It happens that we must unfold an automaton with state variables in order to apply model checking methods. This kind of automaton is called *transition systems.* The states are called *global states* and they have a component which identifies the state like in a simple automaton. This first component is called *control state.* The other component gives the value of each variable (see figure 1.3). There are no more guards with transitions seeing that the value of *fallen_hurdle* is known in each state.

1.1.4 Synchronization

In order to model some big systems, it often happens that systems are split into several models or subsystems. These models are also called system components. To obtain the global automaton of the system, the system components have to be *synchronized.* The result is called *synchronized product.* Here is an example of synchronized automaton: we have two models which represent the incrementation by one of two integers. When both models are combined, a new model where each state is the combination of both integers is obtained. But in the global model, we want them to evolve together. So the synchronized transition is *(increment by one, increment by one)* instead of *(-, increment by one), (increment by one, -)* which means that they can evolve independently.

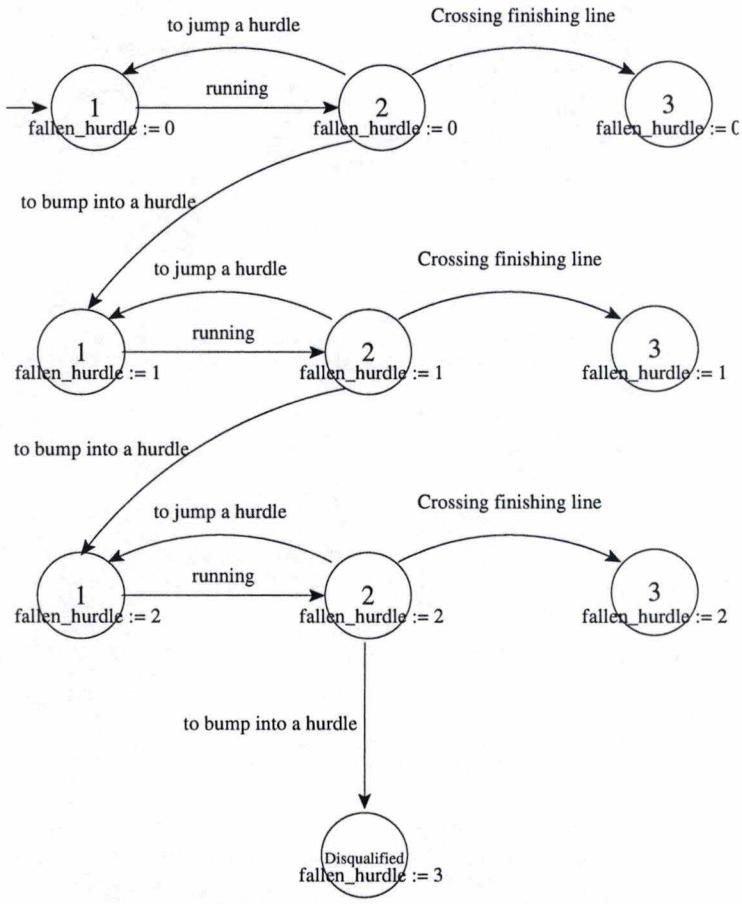

Figure 1.3: Unfolded automaton

Definition

Before defining the synchronized automaton, let us define the *Cartesian product* of submodels M_i.

Definition 1.5. Let $M = \langle S, E, T, S_0, l \rangle$ be an automaton where :

- $S = S_1 \times \ldots \times S_n$;
- $E = \prod_{1 \leq i \leq n} (E_i \cup \{-\})$;
- $S_0 = (S_{0,1}, \ldots, S_{0,n})$;
- $T = \left\{ \begin{array}{l} ((s_1, \ldots, s_n), (e_1, \ldots, e_n), (s'_1, \ldots, s'_n)) | \forall i, \\ e_i =' -' \; and \; s'_i = s_i, \; or \; e_i \neq' -' \; and \; (s_i, e_i, s'_i) \in T_i \end{array} \right\}$;
- $l((s_1, \ldots, s_n)) = \bigcup_{1 \leq i \leq n} l_i(s_i)$

The definition of the *Cartesian product* alone does not allow the model to be synchronized. Indeed, the fact that the symbol '-' can be used in transitions allows components to evolve separately. The creation of a set $sync \subseteq \prod_{1 \leq i \leq n}(E_i \cup \{-\})$ allows the synchronization. In the example, *sync* equals to $\{(increment\ by\ one,\ increment\ by\ one)\}$. To obtain the definition of synchronized product, the set of transitions has to be modified.

$$T = \left\{ \begin{array}{l} ((s_1, \ldots, s_n), (e_1, \ldots, e_n), (s'_1, \ldots, s'_n)) | (e_1, \ldots, e_n) \in sync \\ and\ \forall i, e_i =' -'\ and\ s'_i = s_i,\ or\ e_i \neq' -'\ and\ (s_i, e_i, s'_i) \in T_i \end{array} \right\}$$

Properties

It is possible to operate a *relabelling* of the synchronized automaton. For example, *(increment by one, increment by one)* can be relabelled in *(increment by one)*. *Reachable states* are states which are still reachable after synchronization. In the example, only states where both integers are equal are reachable. Thus the *reachability graph* is the automaton where all non-reachable states are deleted. An important problem associated with reachability graph is the size of the graph. It is related to what is called *the state explosion.* This problem is often met and comes from the fact that we are confronted with a product (see definition 1.5). An example is the unfolded automaton with variables when the global automaton is infinite.

Other types of synchronization

There exist other types of synchronization like synchronization by message passing, by shared variables. In the first one, a distinction is made in transition labels. They are emitting labels and reception labels. Emitting labels are denoted !m and reception labels ?m where m is a message. Let us note that there also exist models which use *asynchronous messages.* They are better for describing communication protocols whereas synchronous ones are better for describing control or command systems(i.e.: an elevator). Concerning synchronization by shared variables, it consists in sharing variables between several submodels of a system.

Synchronization automata are often used in *Petri Nets* which are suited to express parallel systems. Indeed, they can be considered as synchronized automata permitting the dynamic creation of parallel components or as automata juggling with integer counters through the use of restricted set of primitives.

1.1.5 Kripke structures

Kripke structures are often used to model concurrent systems. They are considered as the models of *temporal logics* which are defined in section 1.2. This is a kind of automata where state labels are fundamental and the transition labels have less importance. The definition of these structures is very similar to the one of automaton:

Definition 1.6. $Prop = \{P_1, P_2, \ldots\}$ is the set of elementary properties. Kripke structure is a tuple of four elements $M = \langle S, T, S_0, l \rangle$.

- S is a finite set of state

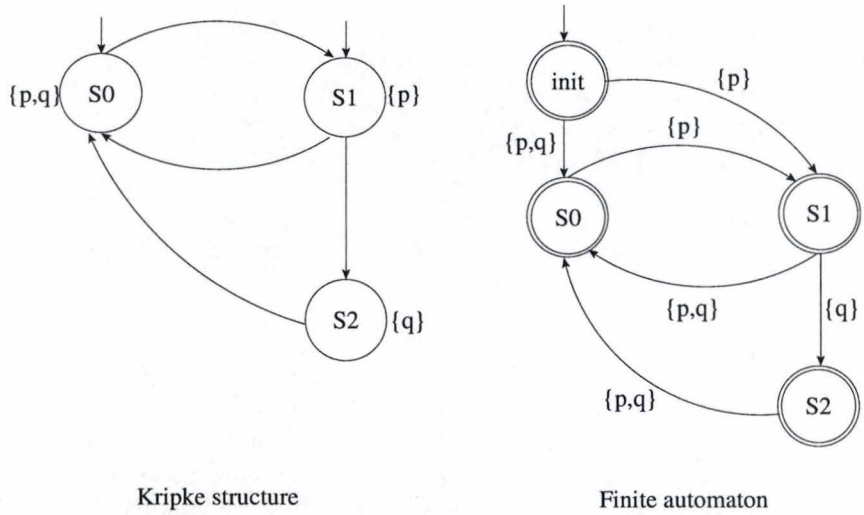

Figure 1.4: Transforming a Kripke structure into a finite automaton

- $T \subseteq S \times S$ is a transition relation that must be total. For every state $s \in S$, there is a state $s' \in S$ such that $T(s, s')$;
- $S_0 \subseteq S$ is the set of initial states;
- l is the mapping which associates with each state of S the finite set of elementary properties which hold in that state. That are state labels.

It is easy to see that the main difference resides in the set of transitions where labels disappear. In order to have a better view of what a Kripke structure is, the example in figure 1.4 [JGP99] shows the transformation of a Kripke structure into a finite automaton. The finite automaton obtained is an automaton where all states are accepting. The other change is the creation of a new state because labels move to transitions. The new state also permits to express the transition to S_0 with {p,q} and to S_1 with {p}.

1.1.6 Büchi automata

Büchi automata can almost be considered as automata over finite words. The difference resides in the fact that Büchi automata are used with infinite words instead of finite words. The reason is that most concurrent systems are designed not to halt during normal execution. So computation is modeled as infinite sequences of states. In other words, they are not designed to end in an accepting state but rather to go through accepting states infinitely often during an execution.

Definition 1.7. *Büchi automata* can be defined as a tuple $M = (S, E, S_0, T, F)$. A particular characteristic is for F (the set of final states for finite automata over finite words). F is called in this case the set of *accepting states* and not *final states*.

An execution now corresponds to an *infinite path* in the graph of the Büchi automaton. It is also said that the execution is *accepting* when one or several accepting states appear infinitely often during an execution. So it is possible that the Büchi automaton loops. When a Büchi automaton has several accepting sets, it is commonly called *generalized Büchi automaton.*

Definition 1.8. A *generalized Büchi automaton* is a tuple $M = (S, E, S_0, T, F_1, \ldots, F_k)$. $F_1, \ldots, F_k$ are the accepting sets.

Let us note that Büchi automata are used in LTL temporal logic which is described below.

1.2 Temporal logic

It is not easy to express properties related to the behaviors of a system. There exist several possibilities to express it. Natural language is one of them but it is too ambiguous. So it does not suit well. Another possibility is *first order logic.* This one is better but it is too mathematic for our goals. Moreover it can only express properties of states but not properties of behaviors. Another problem of these formalisms is the problem of time which is very important in the construction of properties. First order formulas leave the nature of time implicit. These problems are the reasons for the creation of *temporal logics.* Temporal logics can be defined as a form of logic specially tailored for statements and reasoning which involve the notion of order in time[BBF+01]. They permit to obtain a clearer and simpler notation. The properties in temporal logics include the notion of time and they are very closed to natural language. For example, temporal adverbs of natural language such as "always", "until", ... are expressed in temporal logics. In this section three of the most used temporal logics are detailed. This section is a summary of [BBF+01], [JGP99], [Roypt], [Sch04], and [Lerpt].

1.2.1 The language of temporal logic

The basic component of temporal logic is a set of *atomic propositions.* It corresponds to the set $Prop = \{P_1, P_2, ...\}$ previously defined. A proposition P is thus defined as true in a state q if and only if $P \in l(q)$.

The second component is the well-known *boolean combinators.* These are the constants *true* and *false*, the negation $\neg$, and the boolean connectives $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$. A formula in which propositions and boolean combinators are combined is called *propositional formula.* For example, $\neg licence \Rightarrow \neg car$ which means if not licence then not car.

Then there are *temporal combinators* composed of five elements:

- G means *always in the future* (in all future state of a sequence of states).
  Mathematically, it can be expressed as: $\exists i, \forall j : 0 \leq i < j : (j \wedge p)$. $\Leftrightarrow G\,p$?
- F means *eventually* (in some future state of a sequence of states).
  Mathematically : $\exists i,\ \forall j : 0 \leq j < i : (j \wedge \neg p) \wedge (i \wedge p)$.
- X means *at the next time* (in the next state of a sequence of states).
  Mathematically : $\exists i, j : i \geq 0 \wedge j = i + 1 : (i \wedge \neg p) \wedge (j \wedge p)$.

- U means *until*. This element is more complicated than the other. It is used to combine two properties. $P_1 U P_2$ states that P_1 is verified *until* P_2 is verified.
  Mathematically : $\exists i : \forall j : 0 \leq j < i : (i \wedge P_2) \wedge (j \wedge P_1)$.

- R means *release*. This one can be considered as the dual of U combinator. $P_1\ R\ P_2$ means that P_2 is verified along the sequence of states up to and including the first state where P_1 is verified. Let us note that P_1 is not required to be verified eventually.
  Mathematically : $\forall i, \nexists j : j > i : (i \wedge P_1) \wedge (j \wedge P_1) \Rightarrow (i \wedge P_2)$.

G and F can give two special combinations if they follow each other. GFp can be read as *always in a future state p* will be verified. It means that p is infinitely often satisfied along the considered execution. Infinitely often is noted $\overset{\infty}{F}$. The dual FG is also possible and it means *all the time from a certain time onwards*. It is noted $\overset{\infty}{G}$.

Example 1.2.

- $G(\neg gas \Rightarrow X car\ stops)$ states that "always in the future if there is no more gas in the car then at the next state the car stops".

- $G(start\ race \Rightarrow (run\ U\ cross\ final\ line))$ states that "always in the future if the race is started then runners run until they cross the final line".

And finally, there are *path quantifiers*. They permit to quantify over a set of executions. It is only possible to quantify over one execution without them. There are two path quantifiers:

- A means *all the executions*.

- E means *there exists an execution*.

Example 1.3.

- $AGFp$ states that along every execution, at any time, there is a future state where formula p is verified.

- $AGEFp$ states that at any time of any execution, there exists an execution in which in a future state, formula p is satisfied.

The difference between these examples is that the second formula can be satisfied even if there exists an execution in which p is never verified.

1.2.2 CTL*

CTL* formulas are used to describe properties of *computation trees*. This is why CTL means *computation tree logic*. The tree is formed by choosing a state from a Kripke structure. This state is the *initial state* and then the structure is unfold into an infinite tree with the chosen initial state as root. The obtained tree states all of the possible executions starting from the initial state. In CTL*, both *path quantifiers* and *temporal combinators* are used. This temporal logic has also two types of formulas. The first ones are *state formulas* and they

state *a property of a state* (it refers to a set of executions). The second ones are *path formulas* which state *a property of a computation path* (it refers to one execution).

The syntax of CTL* formulas is given by the following rules:

State formulas

- If $p \in Prop$, then p is a state formula.
- If f and g are state formulas, then $\neg f, f \vee g$, and $f \wedge g$ are state formulas.
- If f is a path formula, then Ef and Af are state formulas.

Path formulas

- If f is a state formula, then f is also a path formula.
- If f and g are path formulas, then $\neg f, f \vee g, f \wedge g, Xf, Ff, Gf, fUg$, and fRg are path formulas.

Before providing the semantics, the following notations are important to understand. $M, s \models f$ means that f is true at state s in the Kripke structure[1] M. $M, \pi \models f$ means that f is true along path π in the Kripke structure M. f and g are state formulas and k and j are path formulas. π^i defines the suffix of π starting at state s_i.

The semantics of CTL* formulas is defined as follows:

- $M, s \models p \iff p \in l(s)$.
  p is true at the state s of the Kripke structure M if and only if p belongs to the set of state labels which contains the properties which are true in that state.
- $M, s \models \neg f \iff M, s \not\models f$.
  The negation of f is true at the state s of the Kripke structure M if and only if the state s of the Kripke structure M does not satisfy f.
- $M, s \models f \vee g \iff M, s \models f$ or $M, s \models g$.
  Formula $f \vee g$ is true at the state s of the Kripke structure M if and only if the state s of the Kripke structure M satisfies f or satisfies g.
- $M, s \models f \wedge g \iff M, s \models f$ and $M, s \models g$.
  Formula $f \wedge g$ is true at the state s of the Kripke structure M if and only if f and g are true together at the state s of the Kripke structure M.
- $M, s \models Ek \iff \exists \pi$ from $s \mid M, \pi \models k$.
  Formula Ek which means that there exists an execution which satisfies k is true if and only if there is a path π starting at state s such that k is true along this path in the Kripke structure M.
- $M, s \models Ak \iff \forall \pi$ starting from s, $M, \pi \models k$.
  Formula Ak which means that all executions satisfy k is true if and only if for every path π starting at state s, k is true along this path in the Kripke structure M.

[1] To remember what a Kripke structure is, see definition 1.6.

- $M, \pi \models f \iff s$ is the first state of π and $M, s \models f$.
  Formula f is true along a path π in the Kripke structure M if and only if state s is the first one of the path in question and f is true at state s of the Kripke structure M.

- $M, \pi \models \neg k \iff M, \pi \not\models k$.
  The negation of formula k is true along a path π in the Kripke structure M if and only if k is false along path π in the Kripke structure M.

- $M, \pi \models k \vee j \iff M, \pi \models k$ or $M, \pi \models j$.
  Formula $k \vee j$ is true along a path π in the Kripke structure M if and only if k is true or j is true along path π in the Kripke structure M.

- $M, \pi \models k \wedge j \iff M, \pi \models k$ and $M, \pi \models j$.
  Formula $k \wedge j$ is true along a path π in the Kripke structure M if and only if k and j are true together along path π in the Kripke structure M.

- $M, \pi \models Xk \iff M, \pi^1 \models k$.
  Formula Xk is true along a path π in the Kripke structure M if and only if k is true along a suffix of π starting with the state s_1 which is the state following the first state of π.

- $M, \pi \models Fk \iff \exists y \geq 0 \mid M, \pi^y \models k$.
  Formula Fk is true along a path π in the Kripke structure M if and only if there exists a suffix of π starting with the state s_y where k is true along the suffix.

- $M, \pi \models Gk \iff \forall i \geq 0, M, \pi^i \models k$.
  Formula Gk is true along a path π in the Kripke structure M if and only if k is true along all suffixes of π starting with a state s_i.

- $M, \pi \models kUj \iff \exists y \geq 0 \mid M, \pi^y \models j$ and $\forall 0 \leq x < y, M, \pi^x \models k$.
  Formula kUj is true along a path π in the Kripke structure M if and only if there exists a suffix of π starting with the state s_y where j is true along the suffix and k is true along suffix of π starting with state s_x where x is between 0 and y but not equal to y.

- $M, \pi \models kRj \iff \forall x \geq 0$, if for every $i < x$ $M, \pi^i \not\models k$ then $M, \pi^x \models j$.
  Formula kRj is true along a path π in the Kripke structure M if and only if for all x greater than or equal to 0, if for every x strictly greater than i such k is not true along the suffix of π starting from state s_i then j is true along the suffix of π starting from state s_x.

Example 1.4.

- $A(G\neg sleep \Rightarrow Xtired)$ which means along every execution, at every moment, if "not sleep" then the next state is "tired".

- $AG(\neg sleep \Rightarrow EFtired)$ which means at any time of any execution, if "not sleep" there exists an execution in which a future state is "tired". But it can be possible that if "not sleep", there are no states which satisfy "tired" (if you take medications in order not to be tired for example).

- The combination of two previous examples with a boolean combinator is possible. $A(G\neg sleep \Rightarrow Xtired) \vee AG(\neg sleep \Rightarrow EFtired)$. The meaning is obvious.

1.2.3 CTL

CTL is a sublogic of CTL*. It is commonly defined as a *branching time logic*. Both *state formulas* and *path formulas* are allowed. However, a restriction exists. Each of the temporal combinators X, F, G, U, and R must be absolutely preceded by a path quantifier A, E. So it is not possible to have formulas with a temporal combinator followed by another temporal combinator (i.e.: $A(G\neg sleep \Rightarrow Xtired)$). But the second example 1.4 given for CTL* is allowed in CTL. On the other hand, the last example is not allowed. The reason is obvious. Formulas which are boolean combinations of path formulas containing temporal operators (i.e.: $pUq \vee pUt$) are also not allowed.

The syntax of CTL formulas is given by the following rules:

State formulas

- If $p \in Prop$, then p is a state formula.
- If f and g are state formulas, then $\neg f, f \vee g$, and $f \wedge g$ are state formulas.
- If f is a path formula, then Ef and Af are state formulas.

Path formulas

- If f is a state formula, then f is also a path formula.
- If f and g are state formulas, then Xf, Ff, Gf, fUg, and fRg are path formulas.

The main difference between syntax of CTL and CTL* resides in the fact that the syntax of path formulas is restricted. Indeed, we can observe that *path formulas* have been changed in *state formulas* in the second line of the *path formulas* syntax.

The semantics of CTL formulas is the same as for CTL*.

There are ten basic CTL operators in CTL and the most common of them are represented in figure 1.5:

- AX and EX
- AF and EF
- AG and EG
- AU and EU
- AR and ER

It is to be noted that each of the ten operators can be expressed with EX, EG, and EU.

- $AXf = \neg EX(\neg f)$
- $EFf = E(TrueUf)$
- $AGf = \neg EF(\neg f)$

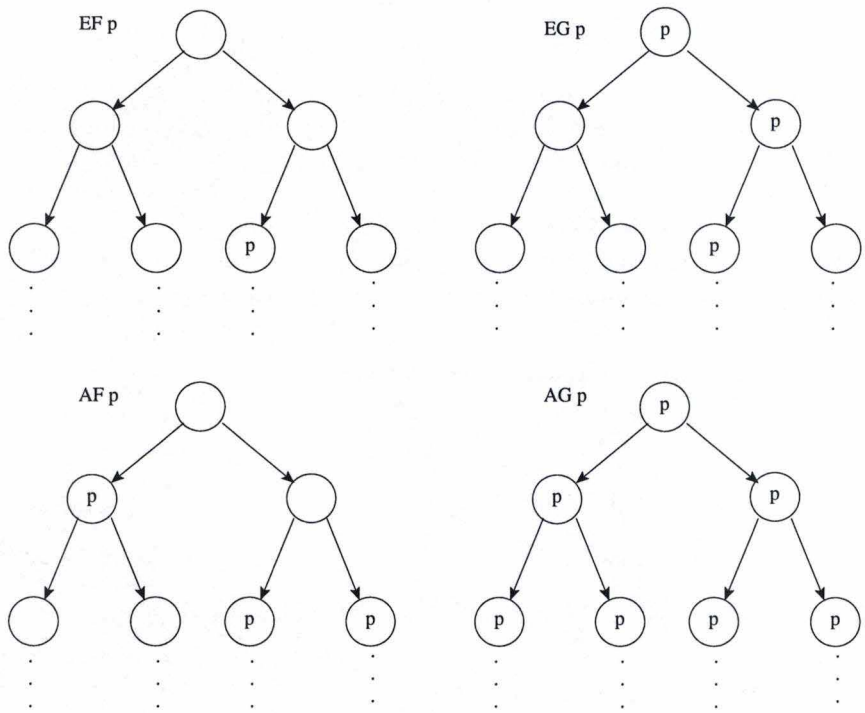

Figure 1.5: Most common operators in CTL

- $AFf = \neg EG(\neg f)$
- $A(fUG) \equiv \neg E(\neg gU(\neg f \wedge \neg g)) \wedge \neg EG(\neg g)$
- $A(fRg) \equiv \neg E(\neg fU\neg g)$
- $E(fRg) \equiv \neg A(\neg fU\neg f)$

Example 1.5.

- EF *sun* means that it is possible to have sun one day.
- AG *4seasons* means that the statement *there are four seasons in a year* is always true.

1.2.4 LTL

Linear Temporal Logic (LTL) is also a subset of CTL* like CTL and is commonly defined as *linear-time logic.* As for CTL, there are formulas which are not allowed in LTL. If we consider again, examples 1.4 given in CTL*. The first one $A(G\neg sleep \Rightarrow Xtired)$, which is not allowed in CTL, is allowed here. The second one $AG(\neg sleep \Rightarrow EFtired)$ is not allowed in LTL but well in CTL. And the last one is not allowed as for CTL. Here the reason is also obvious. It can be observed from these examples that formulas with *state formulas* inside cannot be expressed in LTL. The conclusion of this is LTL only treats *path formulas.* Hence,

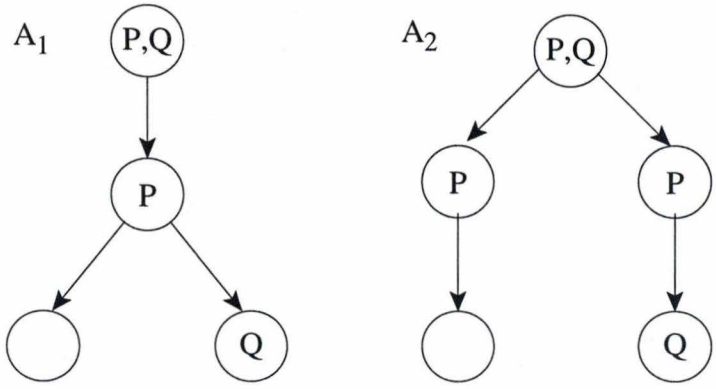

Figure 1.6: Two indistinguishable automata for LTL

a consequence is that LTL does not use *path quantifiers* A and E. Another consequence is that LTL is not able to distinguish certain automata (see figure 1.6). If a formula is true for one of the automata then it is true for the other automaton. Indeed, LTL sees these two automata as the same set of paths. Execution 1 is seen as $\{P, Q\}.\{P\}.\{-\}$ ... Execution 2 is seen as $\{P, Q\}.\{P\}.\{Q\}$ ... This is not the case for CTL. It is possible to find a formula which is true for one and false for the other one. $AX(EX\ Q\ \wedge\ EX\ \neg Q)$ is true in A_1 and false in A_2. LTL formulas are generally noted Af where f is obviously a path formula but it often happens that the A is not written.

The syntax of LTL formulas is given by the following rules:

Path formulas

- If $p \in Prop$, then p is a path formula.
- If f and g are path formulas, then $\neg f, f \vee g, f \wedge g, Xf, Ff, Gf, fUg$, and fRg are path formulas.

Before providing semantics, notations are defined as follows: f and g are path formulas, $M, \pi \models f$ means that f is true along path π in the Kripke structure M. π^i defines the suffix of π starting at state s_i.

The semantics of LTL formulas is defined as follows:

- $M, \pi \models f \iff s$ is the first state of π and $M, s \models f$.
  Formula f is true along a path π in the Kripke structure M if and only if state s is the first one of the path in question and f is true at state s of the Kripke structure M.
- $M, \pi \models \neg k \iff M, \pi \not\models k$.
  The negation of formula k is true along a path π in the Kripke structure M if and only if k is false along path π in the Kripke structure M.
- $M, \pi \models k \vee j \iff M, \pi \models k$ or $M, \pi \models j$.
  Formula $k \vee j$ is true along a path π in the Kripke structure M if and only if k is true or j is true along path π in the Kripke structure M.

- $M, \pi \models k \wedge j \iff M, \pi \models k$ and $M, \pi \models j$.
  Formula $k \wedge j$ is true along a path π in the Kripke structure M if and only if k and j are true together along path π in the Kripke structure M.

- $M, \pi \models Xk \iff M, \pi^1 \models k$.
  Formula Xk is true along a path π in the Kripke structure M if and only if k is true along a suffix of π starting with the state s_1 which is the state following the first state of π.

- $M, \pi \models Fk \iff \exists y \geq 0 \mid M, \pi^y \models k$.
  Formula Fk is true along a path π in the Kripke structure M if and only if there exists a suffix of π starting with the state s_y where k is true along the suffix.

- $M, \pi \models Gk \iff \forall i \geq 0, M, \pi^i \models k$.
  Formula Gk is true along a path π in the Kripke structure M if and only if k is true along all suffixes of π starting with a state s_i.

- $M, \pi \models kUj \iff \exists y \geq 0 \mid M, \pi^y \models j$ and $\forall 0 \leq x < y, M, \pi^x \models k$.
  Formula kUj is true along a path π in the Kripke structure M if and only if there exists a suffix of π starting with the state s_y where j is true along the suffix and k is true along suffix of π starting with state s_x where x is between 0 and y but not equal to y.

- $M, \pi \models kRj \iff \forall x \geq 0$, if for every $i < x$ $M, \pi^i \not\models k$ then $M, \pi^x \models j$.
  Formula kRj is true along a path π in the Kripke structure M if and only if for all x greater than or equal to 0, if for every x strictly greater than i such k is not true along the suffix of π starting from state s_i then j is true along the suffix of π starting from state s_x.

In view of the semantics, it is clear that LTL deals with paths and not with states. The examples 1.2 of the car and the race are good examples of LTL temporal logic.

1.3 Model checking

The *model checking problem* is very simple. It consists in finding out whether a given automaton satisfies a given temporal formula or not. Considering Kripke structures as the models of temporal logics, it is normal to meet them in CTL and LTL model checking. Language theory approach is also used in model checking and it is generally preferred for LTL model checking. This section is a summary of [BBF+01], [JGP99], and [Lerpt].

1.3.1 CTL model checking

In this part, we develop the way of checking model with CTL temporal formulas. As CTL can only express states, we can reason in terms of which states satisfy which formulas. Therefore we do not consider the executions. The algorithms for CTL model checking operate by labelling each state where subformulas of the main formula are true. This is the main goal.

The model is represented by the Kripke structure $M = (S, T, L)$ and the labelling is done with the set $label(s)$ of subformulas of f which are true in state s. $label(s)$ is also equal to

```
procedure EU(f, g)
    V := {s | g ∈ label(s)};
    for all s ∈ V do label(s) ∪ {E(fUg)};
    while V ≠ ∅ do
        choose s ∈ V
        V := V\{s};
        for all t such that T(t, s) do
            if E(fUg) ∉ label(t) and f ∈ label(t) then
                label(t) := label(t) ∪ {E(fUg)};
                V := V ∪ {t};
            end if;
        end for all;
    end while;
end procedure
```

Figure 1.7: Procedure for labelling states satisfying formula $E(fUg)$

$l(s)$ at the initial phase. The algorithm then goes through a series of stages. During the *ith* stage, subformulas with $i-1$ nested CTL operators are processed. In other words, the state-labelling algorithm is successively applied to the subformulas of a CTL formula f. This is done by starting with the shortest subformula (most deeply nested) and works outwards to include all subformulas of f. Proceeding like this guarantees that when a subformula of f is processed, all its subformulas have already been processed. When a subformula is processed, it is also added to the labelling of each state in which it is true. At the end of the algorithm, state s of Kripke structure M satisfies f if and only if $f \in label(s)$.

Seeing as any CTL formula can be expressed in terms of $\neg, \vee, EX, EU$, and EG (see CTL temporal logic), it is sufficient to handle them. So for formulas of the form $\neg f$, states that are not labeled by f are labeled. For $f \vee g$, states that are labeled either by f or by g are labelled. For EXf, every state which has some successors labeled by f is labeled. It is more complicated for the last two formulas. For formulas of the form $E(fUg)$, the goal is to find all states which are first labelled with g. The work is then done backwards using the converse of the transition relation T and all states that can be reached by a path in which each state is labelled with f are found. All such states should be labelled with $E(fUg)$. A procedure for the latter formula is given in figure 1.7. This procedure adds the formula to $label(s)$ for every s that satisfies it. The labelling of subformulas f and g have already been processed. The other formula EGf is based on the decomposition of the graph into *Strongly Connected Component (SCC).*

Definition 1.9. A *strongly connected component* C is a *maximal* subgraph such that every node in C is reachable from every other node in C along a directed path entirely contained within C.

C is considered *nontrivial* if and only if either it has more than one node or it contains one node with a self-loop. The developed algorithm works with a *restricted* Kripke structure $M' = (S', T', l')$ obtained from M by deleting from the set S all of these states in which f does not hold and by restricting T and l accordingly. The graph composed of S' and T' is then divided into strongly connected components. The following stage is to find states belonging

```
procedure EG(f)
    S' := {s | f ∈ label(s)};
    SCC := {C | C is nontrivial SCC of S'};
    V := ⋃_{C∈SCC} {s | s ∈ C};
    for all s ∈ V do label(s) ∪ {EGf};
    while V ≠ ∅ do
        choose s ∈ V
        V := V\{s};
        for all t such that t ∈ S' and T(t, s) do
            if EGf ∉ label(t) then
                label(t) := label(t) ∪ {EGf};
                V := V ∪ {t};
            end if;
        end for all;
    end while;
end procedure
```

Figure 1.8: Procedure for labelling states satisfying formula EGf

to nontrivial components. After that, backward work is done using the converse of T' to find all the states that can be reached by a path in which each state is labeled with f. Such a procedure is given in figure 1.8. This procedure adds EGf to $label(s)$ for every s. As for the algorithm for $E(fUg)$, the inside formula f has already been processed.

In order to have a better understanding of the CTL model checking and theoretical concepts given before, here is an example.

Example 1.6. This example from [JGP99] describes the behavior of a microwave oven. Figure 1.9 gives the Kripke structure of the system. For a better understanding, each state is labeled with both atomic propositions which are true and the negations of the propositions that are false in the state. The CTL formula to check is the following : $AG(Start \Rightarrow AFHeat)$. The formula can be transformed into an equivalent formula to use the algorithms of CTL given previously. So we obtain $\neg E(true\ U\ (Start \wedge EG\neg Heat))$. The first thing to do is to compute the set of states which satisfy the atomic formulas and then proceed to more complicated subformulas.

$S(Start) = \{2, 5, 6, 7\}$ is the set of states which satisfy the atomic proposition $Start$.
$S(\neg Heat) = \{1, 2, 3, 5, 6\}$ is the set of states which do not satisfy the atomic proposition $Heat$.

To compute $S(EG\neg Heat)$, the set of nontrivial strongly connected components has to be found in $S' = S(\neg Heat)$. The set obtained is $SCC = \{\{1, 2, 3, 5\}\}$. $\{2, 5\} \notin SCC$ because it is not the maximal subgraph. State 6 does not belong to the strongly connected component $\{1, 2, 3, 5\}$ because state 6 can not reach another state where $\neg Heat$ is true. There is no other strongly connected components in SCC because there is no more subgraph with strongly connected states where $\neg Heat$ is verified in the different states and where the states are not connected to state 1, 2, 3 or 5. We then proceed by setting V, the set of all states which should be labeled by $EG\neg Heat$, to be the union over the elements of SCC. Thus, initially V equals to $\{1, 2, 3, 5\}$. Seeing that no other state in S' can reach a state in V along a path

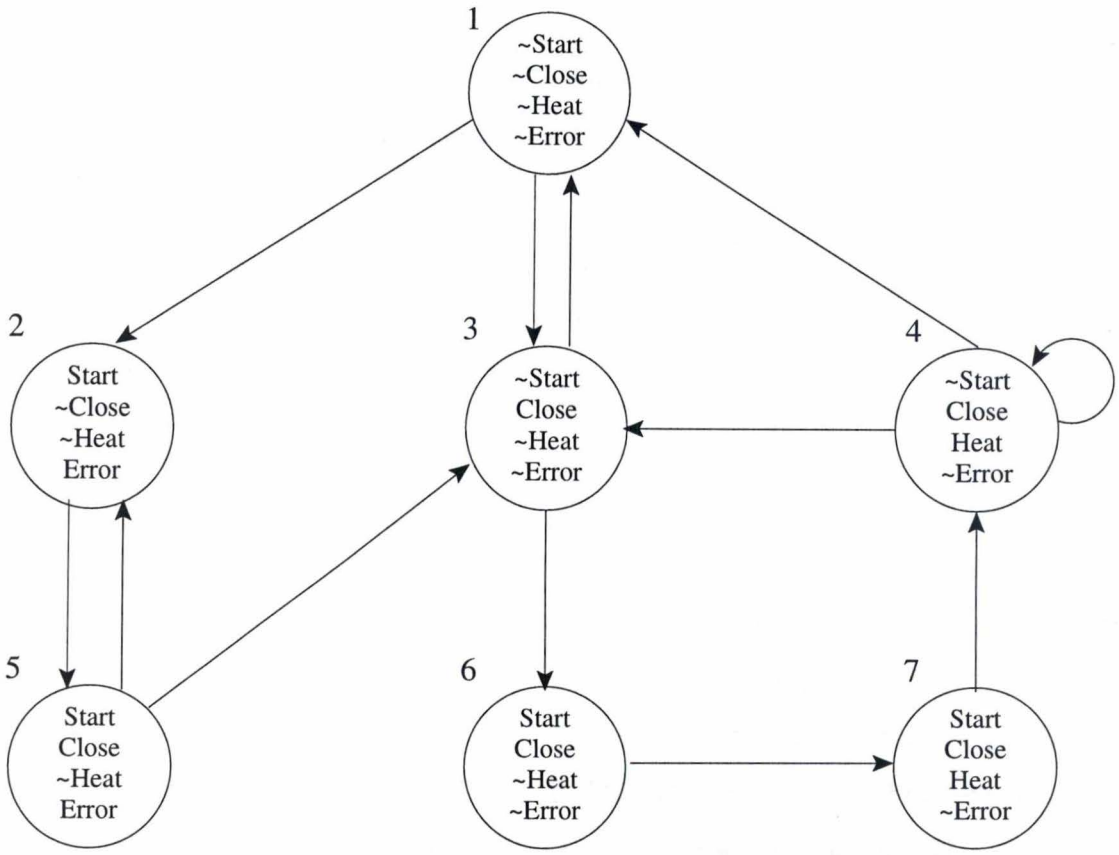

Figure 1.9: Kripke structure of the microwave oven

in S', the computation terminates with: $S(EG\neg Heat) = \{1, 2, 3, 5\}$.
The next formula to compute is $S(Start \wedge EG\neg Heat)$ and the result set is $\{2, 5\}$.
In order to compute $S(E[True\ U\ (Start \wedge EG\neg Heat)])$, V is set to $S(Start \wedge EG\neg Heat)$.
The next stage is to use the converse of the transition relation to label all states in which the formula is true. So, this is the result:

$$S(E[True\ U\ (Start \wedge EG\neg Heat)]) = \{1, 2, 3, 4, 5, 6, 7\}.$$

The last stage is to use the negation of the last subformula.

$$S(\neg E[True\ U\ (Start \wedge EG\neg Heat)]) = \emptyset$$

The last result is empty and thus does not contain initial state 1. The conclusion is that the system described by the Kripke structure does not satisfy the given formula.

1.3.2 LTL model checking

In this subsection, two ways of LTL model checking are described. One uses Kripke structure and is called LTL model checking by tableau. The second uses language theory and particularly Büchi automata.

Example 1.7. To illustrate all these concepts in an example [JGP99], we can reconsider the example 1.6 of the microwave oven. The automaton is the same than the one in figure 1.9. The formula to check is $A((\neg Heat)UClose)$ which will be true in the model if it is not possible for the oven to heat when the door is opened. It is easier to check the negation of the formula. So we check if formula $E\neg((\neg Heat)UClose)$ is not satisfied.

The first stage is to compute the closure of $\neg((\neg Heat)UClose)$. This computation is done by using definition 1.11. To be clearer, let f be $((\neg Heat)UClose)$. Thus, we obtain:

$$CL(\neg f) = \{\neg f, f, Xf, \neg Xf, X\neg f, Heat, \neg Heat, Close, \neg Close\}$$

The next stage is to compute the set of atoms. As for the first stage, computation is done using the definition (see definition 1.12) . $(\neg Heat)UClose$ is in K_A if and only if either Close is in K_A or both $\neg Heat$ and $X((\neg Heat)UClose)$ are in K_A (see definition 1.12). K_A has also to be consistent with $l(s_A)$. It means that what is in K_A must be in $l(s_A)$ for state s_A. Two states contain formulas $\neg Close$ and $\neg Heat$: state 1 and state 2. The set of formulas associated with these states can be represented as following:

$$K_1' = \{\neg Close, \neg Heat, f, Xf\} \text{ or } K_1'' = \{\neg Close, \neg Heat, \neg f, X\neg f, \neg Xf\}$$

Therefore, the atoms are $(1, K_1'), (2, K_1'), (1, K_1''), (2, K_1'')$. The consistency between K_A and $l(s_A)$ is respected seeing that $\neg Close$ and $\neg Heat$ are in $l(s_1)$ and $l(s_2)$. This can be done for other states. We obtain the following sets for states 3, 5 and 6 which contain $\neg Heat$ and $Close$:

$$K_2' = \{Close, \neg Heat, f, Xf\} \text{ or } K_2'' = \{Close, \neg Heat, \neg f, X\neg f, \neg Xf\}.$$

The atoms are $(3, K_2'), (5, K_2'), (6, K_2'), (3, K_2''), (5, K_2''), (6, K_2'')$.

And for states 4 and 7:

$$K_3' = \{Close, Heat, f, Xf\} \text{ or } K_3'' = \{Close, Heat, f, X\neg f, \neg Xf\}.$$

$(4, K_3'), (7, K_3'), (4, K_3''), (7, K_3'')$ are the atoms.

Now, we can go to next stage. In this stage, the transition relations between atoms will be given. There is a transition relation between an atom A and B if there is a transition from s_A to s_B in M and for every formula of the form $Xf \in K_A, f \in K_B$. So there is a transition from $(1, K_1')$ to $(2, K_1')$ seeing that there exists a transition from state 1 to state 2 and that $Xf \in K_1'$ and $f \in K_1'$. It is the same for $(1, K_1'')$ and $(2, K_1'')$ but here $X\neg f \in K_1''$ and $\neg f \in K_1''$. There is no transition from $(1, K_1')$ to $(2, K_1'')$ because we have $Xf \in K_1'$ and we do not have $f \in K_1''$. The remaining transitions are computed in the same way.

For the last stage, we know that a state s satisfies $\neg f$ if there is an atom (s, K) such that $\neg f \in K$ and there is a path from (s, K) in the graph that leads to a self-fulfilling strongly connected component. The latter is $SCC = \{1, 2, 3, 5\}$. When we look at the full graph, we observe that no such atom leads to SCC. That is why the conclusion is no state satisfies $E\neg((\neg Heat)UClose)$. This means that each state satisfies $A((\neg Heat)UClose)$.

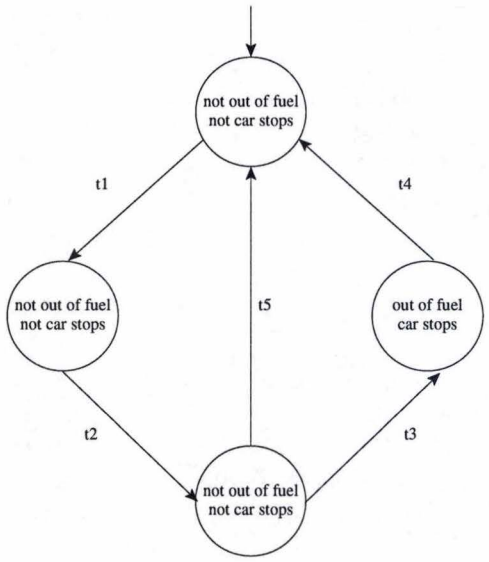

Figure 1.10: Automaton A

LTL model checking with Büchi automata

We already know that LTL model checking deals with path formulas and not state formulas. A finite automaton will generally give rise to infinitely many different executions. These ones are often infinite in length. These are the reasons why language theory is generally used instead of Kripke structures. The algorithms in LTL reason on Büchi automata[3]. The constructed automaton describes the executions which do not satisfy formula f. It is noted $B_{\neg f}$. The Büchi automaton is usually associated with an automaton A representing a system for which we want to check if it satisfies f. The result of this association is a strongly synchronized automaton noted $A \otimes B_{\neg f}$. The new automaton represents the behaviors of A accepted by $B_{\neg f}$. In other words, it represents the executions of A which do not satisfy formula f. This automaton is thus the intersection between behaviors of A which have to satisfy f and behaviors of $B_{\neg f}$ which satisfies $\neg f$. If this intersection is empty, the formula f holds for A. Otherwise it corresponds to a counterexample. Checking the emptiness of a Büchi automaton is simple. It consists in finding a strongly connected component which is reachable from an initial state and which contains an accepting state.

Example 1.8. The example from [BBF+01] shows how LTL model checking works with Büchi automata. The LTL formula which has to be satisfied by the automaton A in figure 1.10 is $G(out\ of\ fuel \Rightarrow XFCar\ stops)$. This formula states that at any time the fact that a car is out of fuel must be followed later by the fact that the car stops. The negation of the formula states that there exists a state in which the car is out of fuel and after which we will never encounter again a state in which the car stops. The Büchi automaton obtained and shown in figure 1.11 is not deterministic. It means that it can choose to move from state q_0 to q_1 each time *out of fuel* is verified, knowing that *Car stops* will never hold again. We have

[3]See definitions 1.7 and 1.8 for reminder.

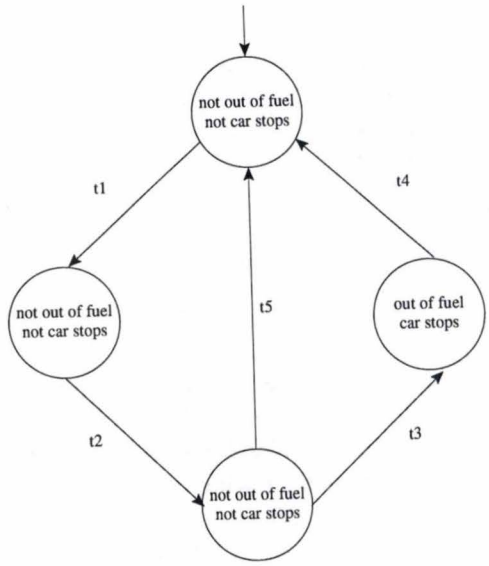

Figure 1.10: Automaton A

LTL model checking with Büchi automata

We already know that LTL model checking deals with path formulas and not state formulas. A finite automaton will generally give rise to infinitely many different executions. These ones are often infinite in length. These are the reasons why language theory is generally used instead of Kripke structures. The algorithms in LTL reason on Büchi automata[3]. The constructed automaton describes the executions which do not satisfy formula f. It is noted $B_{\neg f}$. The Büchi automaton is usually associated with an automaton A representing a system for which we want to check if it satisfies f. The result of this association is a strongly synchronized automaton noted $A \otimes B_{\neg f}$. The new automaton represents the behaviors of A accepted by $B_{\neg f}$. In other words, it represents the executions of A which do not satisfy formula f. This automaton is thus the intersection between behaviors of A which have to satisfy f and behaviors of $B_{\neg f}$ which satisfies $\neg f$. If this intersection is empty, the formula f holds for A. Otherwise it corresponds to a counterexample. Checking the emptiness of a Büchi automaton is simple. It consists in finding a strongly connected component which is reachable from an initial state and which contains an accepting state.

Example 1.8. The example from [BBF+01] shows how LTL model checking works with Büchi automata. The LTL formula which has to be satisfied by the automaton A in figure 1.10 is $G(out\ of\ fuel \Rightarrow XFCar\ stops)$. This formula states that at any time the fact that a car is out of fuel must be followed later by the fact that the car stops. The negation of the formula states that there exists a state in which the car is out of fuel and after which we will never encounter again a state in which the car stops. The Büchi automaton obtained and shown in figure 1.11 is not deterministic. It means that it can choose to move from state q_0 to q_1 each time *out of fuel* is verified, knowing that *Car stops* will never hold again. We have

[3]See definitions 1.7 and 1.8 for reminder.

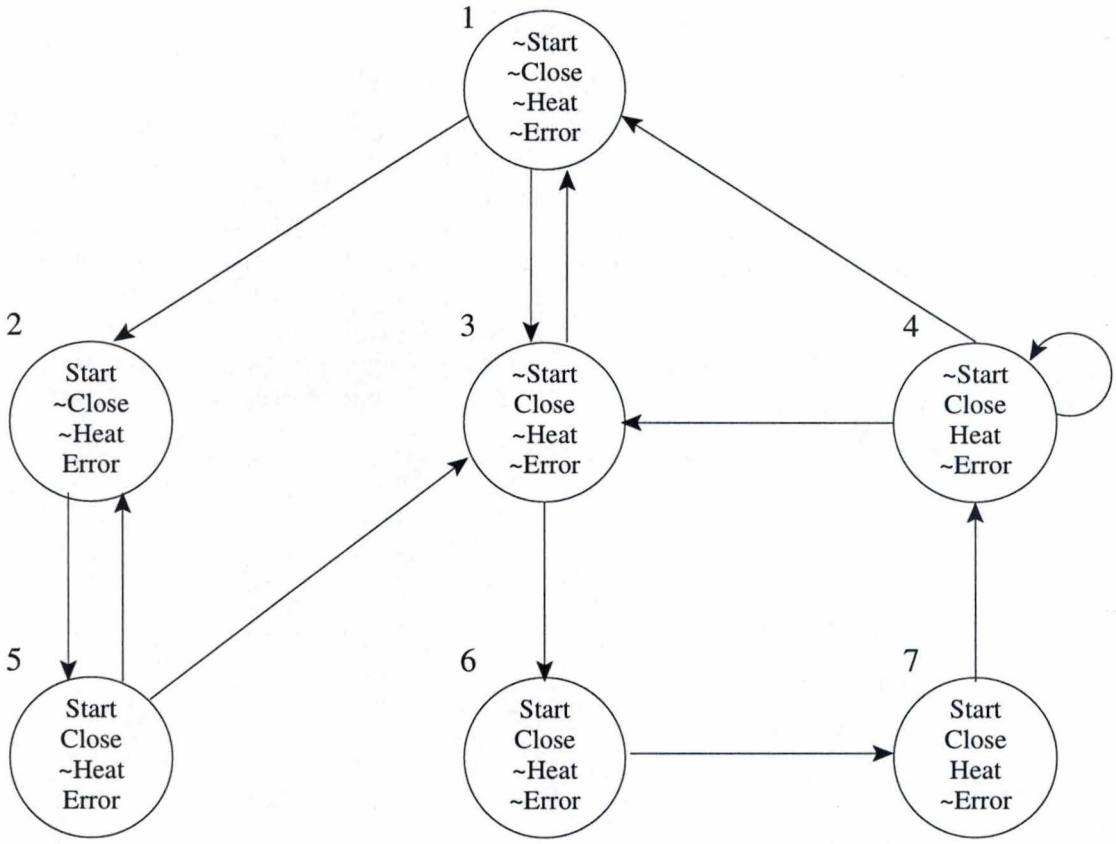

Figure 1.9: Kripke structure of the microwave oven

in S', the computation terminates with: $S(EG\neg Heat) = \{1, 2, 3, 5\}$.
The next formula to compute is $S(Start \wedge EG\neg Heat)$ and the result set is $\{2, 5\}$.
In order to compute $S(E[True\ U\ (Start \wedge EG\neg Heat)])$, V is set to $S(Start \wedge EG\neg Heat)$.
The next stage is to use the converse of the transition relation to label all states in which the formula is true. So, this is the result:

$$S(E[True\ U\ (Start \wedge EG\neg Heat)]) = \{1, 2, 3, 4, 5, 6, 7\}.$$

The last stage is to use the negation of the last subformula.

$$S(\neg E[True\ U\ (Start \wedge EG\neg Heat)]) = \emptyset$$

The last result is empty and thus does not contain initial state 1. The conclusion is that the system described by the Kripke structure does not satisfy the given formula.

1.3.2 LTL model checking

In this subsection, two ways of LTL model checking are described. One uses Kripke structure and is called LTL model checking by tableau. The second uses language theory and particularly Büchi automata.

LTL model checking by tableau

The idea here is to use an algorithm which involves a *tableau construction.*

Definition 1.10. A *tableau* is a graph derived from a formula from which a model for the formula can be extracted if and only if the formula is satisfiable.

In this case, we use Kripke structure $M = (S, T, l)$ and formulas f conformed to syntax of LTL temporal logic[2]. This means that we only consider path formulas. As for CTL model checking, all operators do not have to be handled. It is sufficient to consider only the temporal operators X and U. Indeed, $Ff = True\ U\ f$, $Gf = \neg F \neg f$ and $fRg = \neg(\neg f U \neg g)$. And formula like Af can be also transformed into an equivalent formula $\neg E \neg f$. To be able to say that the state s of the Kripke structure M satisfies Ef, three concepts have to be defined:

Definition 1.11. The *closure of* f noted $CL(f)$ is the smallest set of formulas containing f and satisfying:

- $\neg f \in CL(f) \iff f \in CL(f)$.
- if $f \vee g \in CL(f)$, then $f, g \in CL(f)$
- if $Xf \in CL(f)$, then $f \in CL(f)$.
- if $\neg Xf \in CL(f)$, then $X \neg f \in CL(f)$.
- if $fUg \in CL(f)$, then $f, g, X(fUg) \in CL(f)$.

Definition 1.12. An *atom* can be defined as a pair $A = (s_A, K_A)$ where $s_A \in S$ and $K_A \subseteq CL(f) \cup Prop$ such that:

- for each proposition $p \in Prop, p \in K_A \iff p \in l(s_A)$.
- for every $f \in CL(f), f \in K_A \iff \neg f \notin K_A$.
- for every $f \vee g \in CL(f), f \vee g \in K_A \iff f \in K_A$ or $g \in K_A$.
- for every $\neg Xf \in CL(f), \neg Xf \in K_A \iff X \neg f \in K_A$.
- for every $fUg \in CL(f), fUg \in K_A \iff g \in K_A$ or f, and $X(fUg) \in K_A$.

In an intuitive way, an atom $A = (s_A, K_A)$ is defined in such a way that K_A is a maximal consistent set of formulas that is also consistent with the labelling of s_A.

Definition 1.13. The last definition is about nontrivial strongly connected components. It is said that a SCC noted C of a graph is *self-fulfilling* if and only if for every atom A in C and for every $fUg \in K_A$ there exists an atom B in C such that $g \in K_B$.

Now, with these concepts, we can say that a state s of a Kripke structure M satisfies Ef if and only if there exists an atom $A = (s, K)$ in a graph such that $f \in K$ and there exists a path in that graph from A to a self-fulfilling strongly connected component.

[2]See section 1.2.4 for more details.

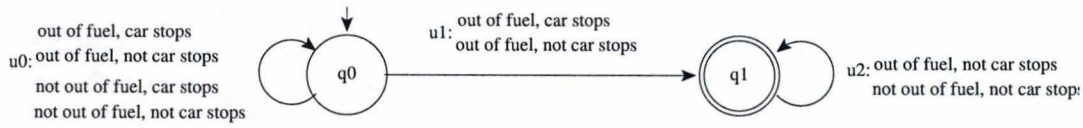

Figure 1.11: Büchi automaton $B_{\neg f}$

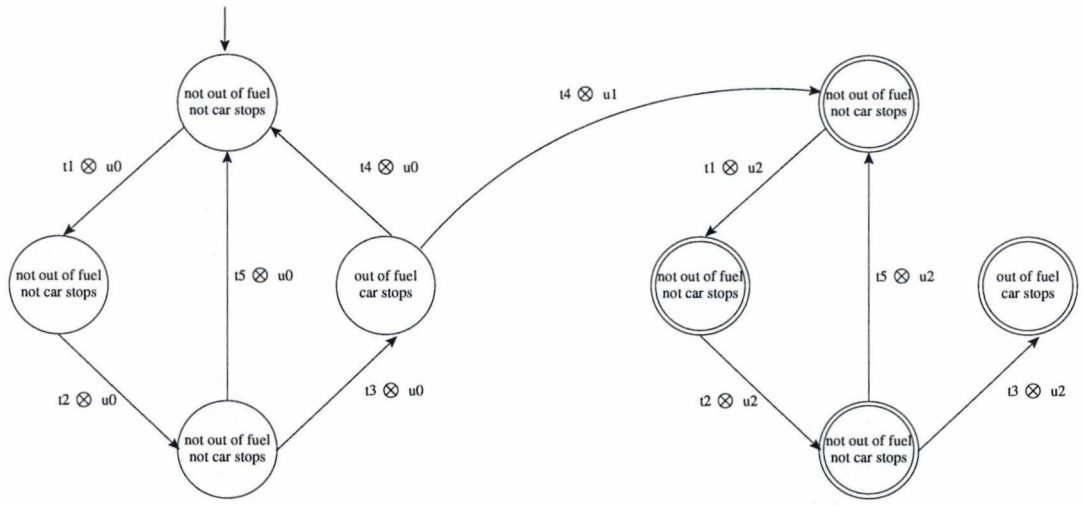

Figure 1.12: Synchronized automaton $A \otimes B_{\neg f}$

a counterexample and thus a bad behavior (not satisfying $G(out\ of\ fuel \Rightarrow XFCar\ stops)$ if the automaton does not leave anymore state q_1 without blocking. A particularity of the Büchi automaton is that the transitions are labelled by valuations of the atomic propositions *out of fuel* and *Car stops*. This allows the automaton to observe the propositions appearing along an execution. The automaton representing the synchronization of A and $B_{\neg f}$ is given in figure 1.12. In this latter, a transition $t \otimes u_0$ is possible if t leaves a state where *out of fuel* is true or false. Transitions $t \otimes u_1$ are possible if t leaves a state where *out of fuel* is true and transitions $t \otimes u_2$ are possible if t leaves a state where *car stops* is false. A consequence of the latter kind of transitions is that there is no transition from a state where *out of fuel* and *car stops* are true to a state where both are false.

In view of the synchronized automaton, we can say that the intersection between A and $B_{\neg f}$ where $f = G(out\ of\ fuel \Rightarrow XFcar\ stops)$ is not empty. Behaviors of A are therefore accepted by $A \otimes B_{\neg f}$. It occurs when we reach the right part of $A \otimes B_{\neg f}$ and when avoiding to block. Indeed, we are in a strongly connected component containing accepting states. The conclusion is that A does not satisfy formula $G(out\ of\ fuel \Rightarrow XFcar\ stops)$.

1.3.3 CTL* model checking

The idea for CTL* model checking is very simple. It consists in combining both techniques described before. This means to use the state-labelling technique from CTL model checking

with LTL model checking. The latter handles formulas which are path formulas but it is possible to extend the algorithm of LTL. This extension permits to handle formulas Ef in which f contains arbitrary state formulas. In order to achieve that, we consider that subformulas of f have already been processed and the labelling of states has been correctly done. The next stage is to replace each state subformula with a fresh atomic proposition in the formula f and also in the labelling of the model. So a new formula is obtained and is noted Ef'. If the analysis of the formula shows us that it is in CTL, then it is logical that the CTL model checking algorithm is chosen in order to be applied on the formula. If it appears that the formula is not in CTL but it is a LTL path formula, then the algorithm for LTL model checking is logically used. In both cases, the formula is added to the label set of all of those states that satisfy it. A last case has to be treated. It concerns subformula Ef of a more complex CTL* formula. In this case, the procedure is repeated with Ef replaced by a fresh atomic proposition and this is continued until the whole formula is processed.

More technically, the algorithm for CTL* works in stages like the one of CTL. So when the algorithm is in stage i, formulas of level i are processed. The state formulas of level i are defined as follows:

- Level 0 is obviously the level of atomic propositions.
- Level $i+1$ contains all state subformulas g such that all state subformulas of g are of level i or less and g is not contained in any lower level.

To understand better how the mechanism of levels works, we can reconsider the example 1.6 of the microwave oven from [JGP99]. The CTL* formula to verify is the following:

$$AG((\neg Close \wedge Start) \Rightarrow A(G\neg Heat \vee F\neg Error))$$

This formula states that if the illegal sequence $(\neg Close \wedge Start)$ occurs, then the microwave oven will never heat or it will eventually arrive in a state where error is not satisfied. In other words, it will eventually be reset. The illegal sequence occurs when the start button is pressed before the door is closed. It has to be noticed that the formula can not be expressed in CTL because there are some temporal operators which follow each other. It is also impossible to express it in LTL because there are path quantifiers used in the formula.

This formula can be transformed into an equivalent formula as seen previously to simplify model checking. So if we take the negation of the formula, we obtain:

$$\neg EF(\neg Close \wedge Start \wedge E(F\ Heat \wedge G\ Error))$$

Now that we have the formula, we can give the levels of the subformulas of this formula :

- Level 0 subformulas are $Close$, $Start$, $Heat$, and $Error$. These are the atomic propositions;
- Level 1 subformulas are $E(F\ Heat \wedge G\ Error)$ and $\neg Close$;
- Level 2 subformula is $EF(\neg Close \wedge Start \wedge E(F\ Heat \wedge G\ Error)$;
- Level 3 contains the entire formula. Hence, $\neg EF(\neg Close \wedge Start \wedge E(F\ Heat \wedge G\ Error))$.

In order to continue to describe the algorithm for CTL* model checking, the following definition is necessary:

Definition 1.14. A subformula Eh_1 of g which is a CTL* formula is said to be *maximal* if and only if Eh_1 is not a strict subformula of any strict subformula Eh of g.

As an example, let us take the following formula : $E(a \vee E(b \wedge EFc))$. In this formula, EFc is a maximal subformula of $E(b \wedge EFc)$ but not of $E(a \vee E(b \wedge EFc))$.

As said before, model checking consists in checking if a given automaton satisfies a given formula. Thus, we consider here the Kripke structure $M = (S, T, l)$ and the CTL* formula f. We also have to consider the state subformula g of f which is of level i. The stage i of the algorithm leads g to be added to all states where g is true. Obviously, at this stage, states of M have already been labelled with all state subformulas of level $i-1$ or less. The manner to treat g depends on its form. These ones are detailed as follows:

- If g is an atomic proposition, then g is in $label(s)$ if and only if it is in $l(s)$.
- If $g = \neg k$, then g is added to $label(s)$ if and only if k is not in $label(s)$.
- If $g = k \vee j$, then g is added to $label(s)$ if and only if either k or j are in $label(s)$.
- If $g = Eg_1$, then the procedure $E(g)$ is applied to add g to the label of all states which satisfy the formula. In this latter, $Eh_1, \ldots, Eh_v$ are the maximal subformulas of g and $a_1, \ldots, a_v$ are the fresh atomic propositions. In this procedure, we can observe a formula noted g'. This is the formula where each subformula Eh_i is replaced by their respective proposition a_i. The result of that is a formula Eg'_1 where g'_1 is a LTL formula. All is done supposing that the LTL model checker updates $label(s)$ to obtain that state s of M satisfies g' if and only if $g' \in label(s)$. The procedure for $E(g)$ is given in figure 1.13.

Example 1.9. The example 1.6 of the microwave oven is, once again, the reference. The Kripke structure is thus the same that the one in figure 1.9. The formula to be treated is

$$\neg EF(\neg Close \wedge Start \wedge E(F\ Heat \wedge G\ Error))$$

The verification proceeds as follows. At the lowest level (level 0), atomic propositions are treated. The next level, the formula $\neg Close$ is added to states 1 and 2. The other formula $E(F\ Heat \wedge G\ Error)$ is handled by an LTL model checking procedure seeing that this formula is a LTL formula. The result is that the formula is false in all states. Therefore, it is is not added to any state label. In level 2, the upper subformula $EF(\neg Close \wedge Start \wedge E(F\ Heat \wedge G\ Error))$ is handled. It is not a CTL formula. Hence the first step is to replace the subformula of level 1 $E(F\ Heat \wedge G\ Error)$ by the atomic proposition a. An LTL model checking procedure is then applied to the new LTL formula obtained $EF(\neg Close \wedge Start \wedge a)$. We observe that the formula is false in any state. Thus, in the last level, all states are labeled with

$$\neg EF(\neg Close \wedge Start \wedge E(F\ Heat \wedge G\ Error))$$

The conclusion is that the microwave oven respects that specification.

```
procedure E(g)
    if g is a CTL formula then
        apply CTL model checking for g;
        return;
    end if;
    g' := g[a_1/Eh_1, ..., a_k/Eh_k];
    for all s ∈ S
        for i = 1, ..., k do
            if Eh_i ∈ label(s) then
                label(s) := label(s) ∪ {a_i};
            end if;
    end for all;
    apply LTL model checking for g';
    for all s ∈ S do
        if g' ∈ label(s) then
            label(s) := label(s) ∪ {g};
        end if;
    end for all;
end procedure
```

Figure 1.13: Procedure to treat CTL* formula of the form $E(fUg)$

1.4 Properties

It can be sometimes useful to add properties to model checking techniques. Such a property can lead to a more efficient model checking according to the goal purchased. As an example, if the verification goal is to check that something *never* occurs, then *safety* property is used. In this section, five properties are detailed: *reachability* property, *safety* property, *liveness* property, *deadlock-freeness* property and *fairness* property. It is to be noticed that other kinds of properties such *progress* property or *response* property exist. This section is a summary of [BBF+01].

1.4.1 Reachability

Definition 1.15. A *reachability property* states that some particular situation can be *reached.*

As examples, it is possible "to enter in a critical section" or the negation is also possible "not to enter in a critical section". This property is one of the most successful strategies for analysing and validating computer protocols.

In temporal logic, reachability property are naturally expressed with the following beginning EF. The formula which follows that beginning must be constructed without temporal combinators. Such a kind of formula is called *present tense formula.* $EF critical\_section$ belongs to this family of formulas. It is also possible to express reachability with the EU construction. It gives us $E\neg car\_stops\ U\ out\_of\_fuel$. All these formulas use CTL. LTL logic is not the best suited logic to express reachability but the latter can be used to express reachability in a negative way.

Reachability is easy to verify for a model checker. Indeed, a model checker is generally able to construct the reachability graph and in some cases, a simple look at it allows to answer any reachability question. The set of reachable states cannot however be easy to see, especially when several automata are synchronized. There exist two major kinds of algorithms for reachability: *forward chaining* and *backward chaining* algorithms.

The *forward chaining* algorithm constructs the set of reachable states by starting from the initial states. Their successors are then added along the computation until the moment where no more states can be added to the set of reachable states.

The basic idea for the *backward chaining* is to construct the set of states which can lead to the states for which the reachability property is in question. The latter states are called *target states.* Once it has been done, the next step is to add the direct predecessors until the moment where no more states can be added to the set of target states. Finally, a test is performed on the set to check if some initial state belongs to it.

This method has two drawbacks. The first one is that a set of target states is necessary before executing the backward search. The second is that it is often more complicated to compute the direct predecessors than the successors of the states. A good example for this is automata with state variables. The successors are computed directly by evaluating expressions whereas the computation of predecessors requires solving equations and complex evaluations.

Another problem of these algorithms, and especially the forward chaining, is the state explosion problem[4].

1.4.2 Safety

Definition 1.16. A *safety property* expresses that, under certain conditions, an event *never* occurs.

An example of the expression of a safety property is as long as the checked flag is not waved, the F1 race is not finished".

The elements of temporal logics which express the property are temporal combinator G and path quantifier A in CTL logic and only the temporal combinator G in LTL. Thus, we can have formulas such

$$AG\neg(out\_of\_fuel \wedge \neg car\_stops)$$

The same formula has an equivalent in LTL when the A is removed. It is to be noticed that safety property is the negation of reachability property. Indeed, $\neg EF$ is equivalent to AG.

However, it is generally easier to express safety properties with *past temporal formula.* It is called *syntactic characterization.* The basic idea of the latter is that when a safety property is violated, it should be possible to notice it immediately. The formula can be written as before AGf or Gf but the difference is that f is a past formula. The combinators used to construct such formulas are logically the opposite of the future temporal combinators. So there are F^{-1}, X^{-1} and S which is the opposite of U. F^{-1} means that a formula was verified at some past instant. X^{-1} means that a formula was verified in the state immediately preceding the current state. fSg means that g was satisfied at a past instant, and that, since then, f is satisfied. As the path to the current state is fixed, the path quantifiers A and E are not used

[4]The problem is discussed in chapter 2.

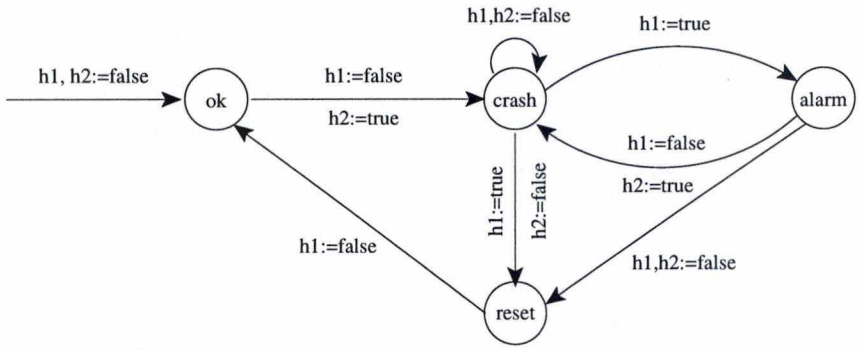

Figure 1.14: An automaton with history variables

together with past combinators. If the example above is used with past combinators, the formula is $AG(car\_stops \Rightarrow F^{-1}out\_of\_fuel)$. It thus means that it is always true that if the car stops then we were out of fuel earlier.

The problem which appears is that past formulas are generally not handled by model checkers. Two solutions are proposed. The first one is to eliminate the past and to obtain a future formula. This solution is not easy at all. The second one is to use *history variables method.* The idea is to transform a formula AGf where f is a past formula in a formula with reachability property. This is possible by using *history variables.* Their goal is to store the occurrences of some past events without modifying the future behavior of the system. In practice, it is often sufficient to associate one history variable with each subformula which has a past combinator at its root.

Example 1.10. As an example, three variables are coupled with the following formula:

$$F^{-1}((X^{-1}P)SQ)$$

The first one h_1 for $X^{-1}P$, the second one h_2 for h_1SQ and the last one h_3 for $F^{-1}h_2$. All of these variables are initially false and they are modified along an automaton.

Example 1.11. This second example from [BBF+01] illustrates an automaton with history variables in figure 1.14. We can observe that there are two variables. h_1 is for formula $X^{-1}crash$ and h_2 is for $(\neg reset)\ S\ crash$. Therefore, safety formulas such as $AG(alarm \Rightarrow X^{-1}crash)$ or $AG(alarm \Rightarrow (\neg reset)\ S\ crash)$ can be respectively expressed as $AG(alarm \Rightarrow h_1)$ and $AG(alarm \Rightarrow h_2)$. There are no more past operators and they are thus handled by model checkers. It now means "at any time of any execution, if *alarm* is satisfied then h_1 is true" and "at any time of any execution, if *alarm* is satisfied then h_2 is true".

1.4.3 Liveness

Definition 1.17. A *liveness property* states that, under certain conditions, some event *will ultimately* occur.

An example of liveness property is "the program will terminate".

There exist two large families of liveness properties: simple liveness also called *progress* and repeated liveness also called *fairness*. Fairness property is detailed in subsection 1.4.5.

The best suited operator of temporal logic for liveness property is the F operator. $AG(req \Rightarrow AFsat)$ which means that "any request will ultimately be satisfied"is a formula with liveness. Another operator is also able to express liveness. This is the U operator. However, these formulas are a bit special. Let us consider the formula $fuel\_light\ U\ car\_stops$ which means that the fuel warning light of a dashboard is switched on until the car stops. The proposition *car_stops* is true in a state of a path and for all states encountered before this state, *fuel_light* is true. The conclusion of this is that *car_stops* is a liveness property because it will eventually hold and *fuel_light* is a safety property because it always holds beforehand. The set is nevertheless considered as a liveness property.

Liveness properties can play two roles in the verification process. They can appear either as *liveness properties* which have to be verified or as *liveness hypotheses* which are made on the system model. The liveness hypothesis generally made is that the system under consideration does not terminate or does not stay inactive without reasons. However, the hypotheses on a model can sometimes be subtle and can lead to errors. It is possible that the model seems correct for several aspects of the real system whereas the behaviors are not the same ones.

Despite this problem between real system and model, it remains possible to verify the behaviors with a model checker. Two conditions must be met for this. The first condition is that the liveness hypotheses of the model must be less restrictive than the desired ones. In other words it means that the behaviors modeled are more general than those of the real system. The second condition is that the temporal logic used must be able to express the liveness hypotheses which are not handled by the model. This has as a consequence that the satisfaction of a given property is only verified for the behaviors for which the liveness hypotheses hold. It is possible to express the behaviors of a liveness hypothesis in temporal logics. The formula $f_{liv} \Rightarrow g$ can be written and it means that g is true for the behaviors of the liveness hypothesis. This formula is verified along paths. This is thus a LTL formula. If g was written in CTL, then f_{liv} would have to be inserted into each occurrence of a path quantifier in g. For example, formula $AFE\ fuel\_lightUcar\_stops$ is transformed into the following formula:

$$A(f_{liv} \Rightarrow FE(f_{liv} \wedge fuel\_lightUcar\_stops))$$

Another problem of liveness properties is that they are sometimes not precise enough. The following example is a good illustration: the fact that the button to call an elevator is pressed does not guarantee that the request will be realised in reasonable delays. A solution is to have a maximal delay, that is to say to have a bound. This is called *bounded liveness*. As an example, if the button of an elevator is pressed, the request will be realised within 3 minutes. However, adding this constraint leads to a *safety* property. Indeed, the given example of *req* and *sat* can be written as a safety formula with past operators :

$$AG(sat \Rightarrow F^{-1}req)$$

which means that at any time *sat* is true if a request was expressed at a past instant.
As a consequence, all the methods for safety properties can be applied to bounded liveness properties. In the other hand, these properties can be useful for specification of timed systems.

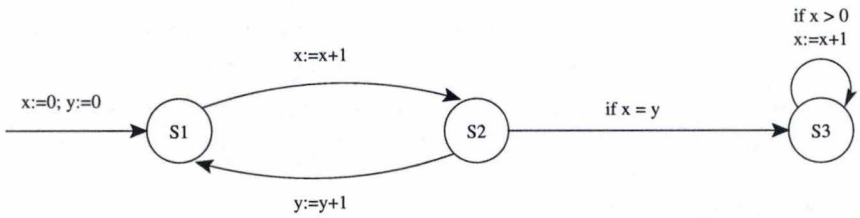

Figure 1.15: A deadlock-free system

1.4.4 Deadlock-freeness

Definition 1.18. A *deadlock-freeness* property states that the system can never be in a situation in which no progress is possible.

In other words, some undesirable event will never occur. This is considered as a correctness property relevant for systems that are supposed to run indefinitely.

This property is generally written $AGEXtrue$ in CTL and can be read "whatever the state reached may be, there will exist an immediate successor state". This property can be considered as a particular case of safety property. The reason of this is that the deadlocked states can often be described explicitly. It is sufficient after this to verify that these states cannot be reached.

Example 1.12. This example from [BBF+01] illustrates deadlock-freeness. Let us consider the automaton A in figure 1.15. This is an automaton with two integer variables. We can say that the automaton is deadlock-free because x and y are equal in S_1 and not in another state. As a consequence, the automaton is never in S_3. As said before, deadlock-freeness can be expressed in a safety property. So instead of using $AGEX\ true$, the safety formula $AG\neg(s_3 \wedge x \leq 0)$ expresses deadlock-freeness.

1.4.5 Fairness properties

Definition 1.19. A *fairness property* expresses that, under certain conditions, an event will occur (or will fail to occur) *infinitely often.*

Examples of this property are "the gate will be raised infinitely often", "if a die is infinitely often thrown, then the die will infinitely often roll". As said in subsection 1.4.3, this property is also called *repeated liveness.*

As regards temporal logics, the best suited combinators to express this property are $\overset{\infty}{F}$ and $\overset{\infty}{G}$ described in section 1.2 on temporal logics. The first one means infinitely often or an infinite number of times. So if we have $\overset{\infty}{F}\ P$, it is not possible to meet a final state in which P holds but well an infinite number of states where P holds. The examples above can be written as $A\ \overset{\infty}{F}\ gate\_raised$ and $A(\overset{\infty}{F}\ die\_thrown \Rightarrow \overset{\infty}{F}\ die\_rolls)$. The second combinator which means *all the time from a certain time onwards* is as well useful. An execution satisfying $\overset{\infty}{G}\ P$ means that P is true for all the states encountered, except possibly for a finite number of them. The

example of the die can be also written here and the result is $A(\overset{\infty}{F}\, die\_rolls \vee \overset{\infty}{G}\, \neg die\_thrown)$. However, everything is not fine. Fairness properties cannot be expressed in CTL seeing that $\overset{\infty}{F}$ or $\overset{\infty}{G}$ are a combination of two temporal combinators. To respond to this problem, an extension of CTL was created and it is called *CTL+Fairness*. The $\overset{\infty}{F}$ and $\overset{\infty}{G}$ are allowed in this logic.

Fairness is often used to describe nondeterministic sequences. That is to say when a nondeterministic choice is done, the choice is assumed to be fair. The example of the die is a good example. The behavior of a die when it is thrown is fair if the property

$$A(\overset{\infty}{F}\, 1 \wedge \overset{\infty}{F}\, 2 \wedge \overset{\infty}{F}\, 3 \wedge \overset{\infty}{F}\, 4 \wedge \overset{\infty}{F}\, 5 \wedge \overset{\infty}{F}\, 6)$$

is satisfied.

As for liveness, it is useful to make a distinction between *property* and *hypothesis*. It is to be noticed that hypothesis is very often used.

Example 1.13. This example from [BBF+01] uses a model of alternating bit protocol. The components of this model are a transmitter A, a receiver B, a line AB for the messages, and a line BA for acknowledgements. The model allows *lost* messages. It is represented by the nondeterministic behavior of the lines AB and BA.

The safety property to check here is *any message received is actually a message that was emitted earlier.*

Liveness properties such as *any emitted message is eventually received* fail because the model allows the unreliable lines to lose all the messages.

However, if we restrict to occasionally lost messages, the nondeterministic choices have to be restricted too. This is done in such a way that all the messages are not lost.

Let us now consider a protocol of transmission in which *emitted* message corresponds to several *received* messages. The liveness property to verify $G(emitted \Rightarrow Freceived)$ is satisfied by all the fair behaviors, which are expressed as

$$A(\overset{\infty}{F}\, \neg loss \Rightarrow G(emitted \Rightarrow Freceived))$$

The situation described shows that a liveness property *subject to verification* depends on a fairness *hypothesis*. The liveness property is thus not anymore applied on unreliable lines but well on a fair model which guarantees that all the messages are not lost. This model is fair thanks to the fairness formula given above.

There exist two kinds of fairness: *strong* and *weak* fairness. These apply to fairness properties of the form *if P is continually requested, then P will be granted infinitely often.* Weak fairness is used when P is requested without interruption. Thence, the formulas $(\overset{\infty}{G}\, requests\_P) \Rightarrow FP$ or $(\overset{\infty}{G}\, requests\_P) \Rightarrow \overset{\infty}{F}\, P$ are obtained. Strong fairness interprets P *is continually requested* as P is requested in an infinitely repeated way but possibly with interruptions. The obtained formulas are $(\overset{\infty}{F}\, requests\_P) \Rightarrow FP$ or $(\overset{\infty}{F}\, requests\_P) \Rightarrow \overset{\infty}{F}\, P$.

1.5 A tool: SPIN

A model checker is described in this section. This is a practical application of the theory. It gives a view of how to do model checking in real life. This section is a summary of [BBF+01].

SPIN is a tool designed by G. J. Holzmann at Bell Labs, Murray Hill, New Jersey, USA. SPIN was developed with the goal of simulation and verification of distributed algorithms.

The first step in the process of verification is to describe the system under study in the SPIN's specification language. The latter is called *Promela*. For communication, the processes can use communication channels of the type First In First Out (FIFO), shared variables or rendez-vous. Two operation modes are used in SPIN. The first one allows the user to get familiar with the behavior of his/her system by simulating its execution. The other one checks if the system satisfies properties written in LTL temporal logic.

The basic idea for SPIN comes from the model of automata communicating via *bounded* channels. A consequence is that SPIN is unable to verify systems with infinite state such as Petri Nets. The main feature is the use of techniques to avoid state explosion such state compression, on-the-fly model checking and hashing techniques.

Description of the processes

The description is thus done with the Promela specification language. The first stage is to define constants and global variables.

Example 1.14. To illustrate this, we consider the example from [BBF+01] of an elevator which serves three floors. The declaration is thus the following :

```
bit doorisopen[3];
chan openclosedoor=[0] of {byte, bit};
```

The bits array *doorisopen* shows if the door of a floor is in the state *open* or *closed*. 1 represents open and 0 corresponds to closed. The *openclosedoor* channel is the tool used to communicate between the elevator and the doors. The length of the buffer associated with the channel is 0. The channel accepts messages of the form {byte,bit} where byte is for the floor to which the operation applies and a bit for the order sent to the door of the floor.

A process is described by the word *proctype* followed by its name and its arguments.

Example 1.15.

```
proctype door(byte i){
do
:: openclosedoor?eval(i),1;
   doorisopen[i-1]=1;
   doorisopen[i-1]=0;
   openclosedoor!i,0
od
}
```

The example of the process *door* described shows that the process takes a floor as a parameter. It indicates that the door is open, and then that it is closed. Finally, it signals the closing to the elevator. The *do* is a loop in which a nondeterministic choice is made between all instruction sequences starting with :: (there is only one in the door process).

In a more general way, the *do* instruction permits *guards*. These ones restrict the set of sequences that can be chosen.

Example 1.16. The mechanism of guards are used to control the process of the *elevator*.

```
proctype elevator(){
show byte floor = 1;

do
:: (floor != 3) -> floor++
:: (floor != 1) -> floor--
:: openclosedoor!floor,1;
   openclosedoor?eval(floor),0
od
}
```

The elevator process is constructed with two guards (floor != 3) and (floor != 1). Their goal is to check that the elevator is not at the third floor when a request for going up is submitted and that it is not at level 1 when a request for going down is submitted. The third operation is to send an order to the door of the current floor. It then has to wait for the same door to close before moving.

The next step consists in joining all these parts.

Example 1.17. The system execution starts with an initialization process *init*

```
init{
atomic{
run door(1); run door(2); run door(3);
run elevator()}
}
```

The instructions *run door(1)* and the following ones instantiate the processes corresponding to the three doors and the elevator. Each instance of the processes then runs in parallel with the ones that already exist.

Simulation

This mode permits to try out some executions with a graphical interface. There exist three modes in the simulation mode. *Random* mode leaves the nondeterministic choice to SPIN when we are confronted to one of it. The *interactive* mode leaves the choice to the user. The last one which is the *guided* mode is used with the verification. When SPIN finds an error, the execution leading to the latter is stored in order to replay the execution to find the cause. This error is obviously a counterexample of the good behavior of the system. Simulation only permits to view the working of the system but does not carry out the verification.

Verification

The verification techniques in SPIN concern the analysis of the complete system. It allows to check that some property is satisfied by all reachable states or all the possible executions of the system.

There exists a primitive to indicate invariants which must be satisfied when the system is in a given state.

Example 1.18.

```
assert(doorisopen[i-1] &&
!doorisopen[i%3] &&
!doorisopen[(i+1)%3]);
```

This specifies that when a door is open, the other ones must be closed. To be complete, this assertion is added between the instructions to open and close door number i in example 1.15.

As regards the properties to check, there are transformed from LTL into a formula understandable by SPIN as the following example.

Example 1.19. $G(open_1 \Rightarrow Xclosed_1)$ is written [ ] (open1 -> X closed1).
The atomic propositions of this formula are defined as follows:

```
#define open1 doorisopen[0]
#define closed1 !doorisopen[0]
```

$F(open_1 \vee open_2 \vee open_3)$ is written <> (open1 || open2 || open3)

```
#define open1 doorisopen[0]
#define open2 doorisopen[1]
#define open3 doorisopen[2]
```

The latter is a property which is not satisfied because the model of the elevator described can go up and down infinitely without stopping on floors.

Chapter 2

The state explosion problem

The goal of this chapter is to introduce to the state space explosion problem and to propose solutions to it. The exposition of the first three solutions comes from [BBF+01] and [JGP99]. The distributed solution comes from [BBv02], [BB03], [Bar02a], and [Bar02b]. The information used to describe the DiVinE project comes from [divml].

2.1 The state explosion problem

In chapter 1, several methods for verification of concurrent systems have been described. The problem which has been also raised is when a model represented by an automaton explodes in the number of states generated. This is the state explosion problem. The latter occurs in systems with many components which can interact with each other or systems which have data structures that can assume many different values.

Example 2.1. This example comes from [expgn]. Let us consider four queues with a capacity of 64 bytes. A queue can be modeled as a field of the type byte and the size 64. If we consider that each byte can have 256 values and that in each queue there are 64 different values of the type byte, then one queue can acquire 256^{64} different values. And for the four queues, it gives 256^{256} different states.

We can see that it grows rapidly. This is thus a problem because computer resources are not sufficient to be able to handle it. This is an obstacle to efficient model checking. Several solutions are described in the rest of the chapter.

2.2 On-the-Fly model checking

Two ways for checking models have been described in section 1.3 of chapter 1. One using Kripke structures to represent the model of the system. The construction of this structure can lead to a graph with an exponential number of states. The second way constructs a synchronized graph from an automaton A representing the concurrent system and an automaton $B_{\neg f}$ which represents the bad behaviors of this system. The automata used are Büchi automata. After this construction, the emptiness of the intersection between these two automata is checked. The latter tells if a counterexample is found. The problem which can

appear is that the construction of the graph modelling the system can lead to a huge number of states. For the synchronized automaton, it is often worst.

The technique called *On-the-Fly model checking* uses the automata theory approach. It permits to avoid to construct the entire state space of the system. The reason for this is that only the Büchi automaton $B_{\neg f}$ is constructed in the first stage instead of constructing the automaton $B_{\neg f}$ and the entire automaton A in the first step. The states of the automaton A are generated only when needed while checking the emptiness of its intersection with automaton $B_{\neg f}$. One of the advantages of this technique is that when the intersection is computed, some states of A may never be created. Another advantage is that a counterexample may be found before the end of the creation of the two automata. Hence, when a counterexample is found, there is no more need to continue the construction up to the completion of the intersection of the two automata. To conclude the section about the *On-the-Fly model checking* technique, we can say that the latter leads to a reduction in the number of states generated. A considerable space may be saved but a saving of time may also be made.

2.3 Abstraction by State Merging

Abstraction by State Merging is a method to reduce the number of states in an automaton and comes from [BBF+01]. The first question is "what the word *abstraction* means". It actually refers to the nature of the simplifications performed. These simplifications generally consist in ignoring some aspects of the automaton involved. Now that this explanation is done, the method can be described.

The basic idea underlying this method consists in viewing certain states of the concerned automaton as identical. This is a kind of factorization of states. So the merged states are put together to form a *super-state* and all transitions which lead out of one of the merged states now lead out of the unique state.

Example 2.2. We can take the unfolding automaton of figure 1.3. The merged automaton is shown in figure 2.1. The merged automaton is more readable and the number of states goes from 10 to 5. This is an interesting result. And if the rule was to be disqualified after three fallen hurdles instead of two, the automaton should go from 13 to 6 states. So the saving is really significant. The other advantage is the reducing of the number of transitions. However, problems appear for the correctness of the system. There are properties which hold in the new automaton and not in the unfolded automaton. As an example, a runner can reach the state 3 without running.

The merged automaton thus handles more behaviors than the non-merged automaton but the behaviors of the non-merged automaton are nevertheless preserved. The use of state merging is useful to verify *safety* properties. The reason is the following. Seeing that a merged automaton handles more behaviors, it fulfills less safety properties. Hence, if it satisfies a safety property then *a fortiori* the non-merged automaton also satisfies the property. However, if the merged automaton does not satisfy a safety property, then it does not imply that the non-merged automaton does not satisfy the property. No conclusions can be made.

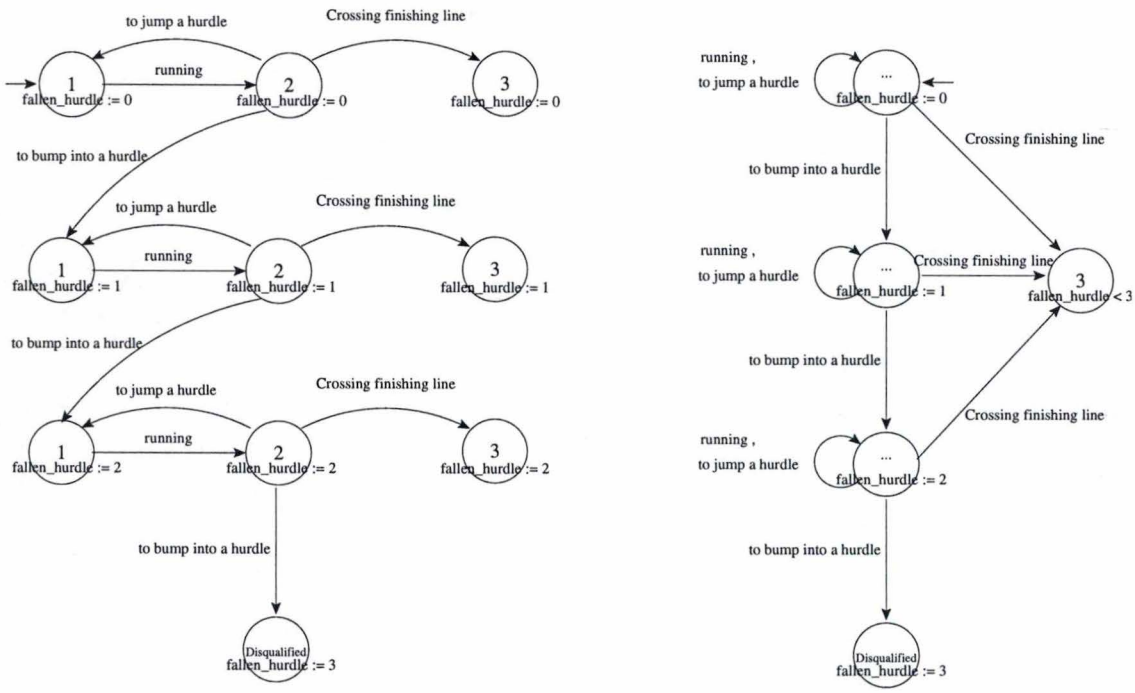

Figure 2.1: The unfolding automaton after merging

2.4 Partial Order Reduction

The concerned systems in this section are the *asynchronous* ones. They allow to treat all possible orderings which lead to the same state. This generally finishes with state explosion. If we imagine, as an example, a model with n transitions which can be executed concurrently, then $n!$ different orderings are possible and there are 2^n different states. If the temporal logic formula does not make a distinction between these sequences, then it is very interesting to consider only one sequence. In this case, we obtain $n + 1$ states.

The aim is to reduce the number of states in the graph while preserving the correctness of the checked property. It conducts us to find a technique which allows this reduction. This technique is *partial order reduction.* It is thus another technique to reduce the size of a graph. The latter exploits the commutativity of concurrently executed transitions which result in the same state when they are executed in different orders. This is why asynchronous systems are concerned. Another point to underline is that the behaviors of the reduced graph are a subset of the behaviors of the full state graph. It is also interesting to note that on-the-fly model checking can be combined with partial order reduction.

Transitions are very important in partial order reduction. Indeed, it specifies which transitions should be included in the reduced model and which should not. However, the definition 1.6 of Kripke structure has to be modified in order to draw a distinction between different transitions in a system. The definition of T is changed and concerns a set of relations instead of one transition relation. We obtain thus a *state transition system.* A transition

```
hash(s0);
set on_stack(s0)
expand_state(s0)

procedure expand_state(s)
    work_set(s):=ample(s);
    while work_set(s) is not empty do
        let α ∈ work_set(s);
        work_set(s):=work_set(s) {α};
        s':=α(s)
        if new(s') then
            hash(s');
            set on_stack(s');
            expand_state(s')
        end if;
        create_edge(s, α, s');
    end while;
    set completed(s);
end procedure
```

Figure 2.2: Depth-first search with partial order reduction

$\alpha \in T, \alpha \subseteq S \times S$ is said *enabled* in a state s if there is a state s' such that $\alpha(s, s')$ and the set is *enabled*(s). If it is not the case, α is said *disabled*.

The reduction is performed by a Depth-First Search (DFS) algorithm used to construct the graph. The algorithm is detailed in figure 2.2.

The search starts with an initial state s_0 and proceeds recursively. For each state s it selects only a subset *ample*(s) of the enabled transitions *enabled*(s), rather than the full set of enabled transitions which is used to construct the full graph. The depth-first search explores only successors generated by these transitions. When a state is encountered for the first time, it is labeled as *on_stack* and as *completed* when all of its successors have been reached. A state marked *on_stack* is thus a useful information for computing *ample*(s). Indeed, it is not necessary to keep transitions which lead to already encountered states. The calculation of the latter needs to satisfy three goals :

- Many behaviors must be sufficiently present in the reduced graph to obtain correct results from the model checking.
- The graph obtained when using *ample*(s) should be smaller than the one using *enabled*(s), otherwise it is useless.
- The overhead in calculating *ample*(s) must be reasonably small.

$\alpha(s, s')$ is noted $s' = \alpha(s)$ when α is deterministic.

The model checking algorithm is then applied to the reduced state graph obtained. This one terminates with a positive answer if the property described by the formula holds for the full state graph. If it is not the case, then a counterexample is given. It is possible that the counterexample differs from the one which can be obtained when using the full graph.

2.5 Distributed LTL model checking

The goal of this technique is to solve LTL model checking but not by reducing the graph such as the previously described techniques. It can be nonetheless combined with on-the-fly model checking. The idea is to use only main memory instead of disk swapping when main memory is entirely used. The problem actually resides in the limitation of memory available. A solution is to increase it by building a parallel computer as a network of workstations commonly called *cluster*. The sequential LTL model checking[1] is based on depth-first search. The idea is to reduce the model checking to the non-emptiness problem of Büchi automata. This can be further transformed to a problem of detecting a reachable cycle in the graph with an accepting state. An important point of it is the *postorder* which is used for cycle detection in the graph. *Postorder* means that each node in a graph is visited after its children. The problem when adapting it to distributed approach is that the DFS order is not preserved. The reason of this is the different computer speeds in the cluster. Three approaches to deal with this problem are possible. The first one is to use additional data structures to maintain the global DFS order. The second one is to use a different search procedure which does not care about DFS order. The last one is to distribute the state space in such a way that cycles are not split up among the cluster. It allows not to care about the global DFS order.

2.5.1 Using additional structures

This approach of distribution uses the *Nested DFS* algorithm and allows parallel execution of the algorithm on each workstation. States are stored randomly on workstations. A *Nested DFS* algorithm is divided in two searches. The goal of the first one (primary) is to find reachable accepting states while the second one (nested) tries to detect accepting cycles. It is thus used to check the non-emptiness. The problem is that it leads to an incorrect result because the order is not preserved. The postorder is important for the nested-DFS. The reason is that it must be started from the accepting states in the postorder defined by the primary DFS. If it was not the case, then the cycle could be missed. The solution is to use a special data structure called *dependency structure* which keeps the proper order of accepting states. The latter is built in such a way that a nested DFS can start only if all the accepting states before the starting one have finished their nested DFS. Each computer has its own dependency structure. Then nested DFS to find an accepting cycle is done.

2.5.2 Negative Cycles

The problem of detecting accepting cycles is reduced to a problem of detecting negative length cycles in this approach. The link with Büchi automata is the following. A Büchi automaton corresponds to a graph in which transitions are ordered by pairs of states. If we assign lengths to transitions in such a way that transitions from an accepting state are set to -1 and all others set to 0, then negative cycles coincide with accepting cycles. It allows to reduce the problem of non-emptiness to the negative cycle problem. The latter problem is closely linked to the Single Source Shortest Path Problem (SSSP). It consists in finding the shortest paths from a specific source state to every other state in a directed graph which has weight associated with each transition. The sequential method to solve this problem is

[1] To remember sequential LTL model checking, see section 1.3.2.

called *scanning method*. It maintains the distance label $d(s)$ and its parent state $p(s)$ for every state s. This method must be modified in order to be able to detect negative cycles. For the distributed algorithm, *walk to root* cycle detection strategy is chosen. The test is done by starting a *walk* in the parent graph from a state being updated *back* to the root. This approach can be used with additional structures to allow several walks to be performed in parallel. If the result of the walk leads to the root, then we have a cycle. Moreover, if the length of the cycle is negative, then we have detected a negative cycle. Hence, an accepting cycle and a counterexample.

2.5.3 Property based distribution

In order to have efficient algorithms for cycle detection in a distributed way, it is interesting to partition the graph in such a way that no accepting cycles are divided among workstations. It also allows to limit the nested DFS to the paths that can really form a cycle in the graph. It is not possible to have a state which can be visited by two different nested DFS coming from two different computers. If we apply this approach to LTL model checking, then the automaton representing the bad behaviors is decomposed into maximal SCCs. They are considered as heuristic to partition state space. The main idea is that the partition function checks which SCC the formula part of the synchronized automaton belongs to. It then places the state on the same computer as all the other states whose formula part is in the same SCC in the decomposition of the automaton with bad behaviors. The nested DFS is now done on one computer. So other nested DFS procedures can be executed on different computers in a simultaneous way. Each SCC are examined and if there is a result given by any nested DFS from any computer then we have a counterexample. Improvements can be done to this approach. Indeed, there are three types of SCCs in the negative automaton.

- Type F (*Fully accepting*): any cycle within the component contains at least one accepting cycle.
- Type P (*Partially accepting*): there is at least one accepting cycle and one non-accepting cycle within the component.
- Type N (*Non-accepting*): there are no accepting cycles within the component.

The improvements are the following ones. States belonging to type N can be randomly distributed among computers. Cycles of type F component can be detected sequentially without using nested DFS seeing that each cycle contains an accepting cycle. As regards type P components, they can be either placed on a single computer or distributed among the cluster. Afterwards they can be checked for cycles.

2.5.4 DiVinE

The DiVinE project or Distributed Verification Environment for the long name is a library developed in *ParaDiSe laboratory* of the Faculty of Informatics of Masaryk University in Brno (Czech Republic). It is the practical application of the solution described in the previous section. Its goal is to make developments of distributed verification algorithms easier. It also comprises an implementation of some model checking algorithms and modules for easy making

comparisons and statistics of verification algorithms. It is at a development stage at this time and it wants to become in long term a tool like SPIN instead of being a library. The main features of DiVinE are in support for the distributed generation of the state space, dynamic load balancing, distributed generation of counterexamples, fault tolerance and re-partitioning.

DiVinE has its own language to write models. The idea is that the system that we want to model is composed of processes. These processes can transit from one process state to another through transitions. Those transitions can be guarded[2] and can be synchronized through channels. The word *effect* represents the assignment to variables. The elements of this language are the following: processes, global or local variables, constants, process states, transitions, channels and the type of system which can be synchronous or asynchronous.

Example 2.3. This is an example of a model written in DVE language.

```
byte x;                          //Variables declaration

process A {                      //Process declaration
state r1,r2;                     //Process states
init r1;                         //Process states - initial state declaration
trans                            //Transitions
 r1 -> r2 {},
 r2 -> r1 { guard x==1; };
}

process B {                      //Process declaration
state r1,r2;                     //Process states
init r1;                         //Process states - initial state declaration
trans                            //Transitions
 r1 -> r2 { effect x=1; },
 r2 -> r1 { effect x=0; };
}

system async;                    //Type of system
```

It has already been said that the state space generation of complex system can lead to a huge number of states and transitions. To be able to handle this, it can be interesting to develop a kind of memory mechanism which uses main memory of several computers. Indeed, with only one computer, it is not possible to allocate more than 4 Gigabytes of memory for one process. It is thus not possible to carry out verification if the graph requires more than 4 Gigabytes. The distributed mechanism allows it. This is a reason for developing a prototype based on this mechanism and to experiment it. It can be also interesting to check if an application level network swapping is faster than the operating system disk swapping. As regards the distributed algorithms to do LTL model checking and the ones integrated in DiVinE, they suffer from the problem of *revisiting*[3]. When algorithms meet the problem, they take a lot of time to terminate. It is another important reason to try a non-distributed algorithm which does not suffer from revisiting and which takes advantage of a bigger main

[2]See subsection 1.1.3 for details about *guards*.

[3]Revisiting is a word used to express the re-exploring of states many times.

memory built with different main memories of several computers. The design of this prototype and the experimentation is described in the next chapter.

Chapter 3

A network memory storage mechanism

This chapter is devoted to the description of the prototype and the experimentations that are carried out. The reasons to develop such a prototype have been given at the end of chapter 2. If the experiments are conclusive, then it will be reimplemented by the DiVinE team in order to be fully compatible with DiVinE. This prototype considers only state storing seeing that standard solutions to LTL model checking produced a counterexample without having stored the generated transitions.

3.1 Programming language

There exists a lot of programming languages in computer science. However, all of these are not well suited to develop a network memory storage mechanism. The best candidates for this are Java and C/C++. Java is object oriented and is easier to take in hand but it has a major drawback. This is the virtual machine which reduces the performance of programs. Nevertheless, this virtual machine is also an advantage for portability. A program normally runs on all platforms where the good virtual machine is installed. As regards C/C++, it is harder to take in hand and it can represent a problem for portability, except when the program respects the ANSI norm. The major advantage of C/C++ is performance. This is the reason why this language has been chosen for the development of the prototype. Moreover, the DiVinE library for which the prototype is developed is written in C/C++.

3.2 Communication Technologies

3.2.1 Shared memory

Shared memory[1] is one of the two techniques for communicating between parallel processes. It allows memory to be accessed by more than one processor and this is done via a shared bus or a communication network. Computers using shared memory usually have some kind

[1]It comes from [defry] and [wikrg].

of local cache on each processor to reduce the number of accesses to shared memory. The problem is that they need a cache consistency protocol to ensure that one processor's cached copy of a shared memory location is considered as erroneous when another processor writes to that location.

3.2.2 Message Passing Interface

Message Passing Interface (MPI)[2] is a de facto standard for communication among nodes[3] running a parallel program on a distributed memory system. MPI is a library of routines that can be called from programs written in several programming languages. MPI's advantage is that it is both portable (because MPI has been implemented for almost every distributed memory architecture) and fast (because each implementation is optimised for the hardware it runs on) [defry]. MPI is called *de facto standard* because the standard groups that defined it are not official like International Standards Organization (ISO), American National Standards Institute (ANSI), or Institute of Electrical and Electronics Engineers (IEEE). An important notion with MPI is the notion of *master* process and *slave* process. The master is represented by the integer 0. In practical, MPI launches the given program on the different nodes including the master and thus creates a process on each processor. This is why the term *process* is used instead of computer. The process is thus the unit of parallelism. It is not possible to add or delete a process during an execution. No mechanism for loading onto processors or assigning processes to processors are provided.

Message passing is actually the second technique for communicating between parallel processes. It is slower than shared memory but it permits to avoid the problems of contention for memory. A system like that provides primitives for sending and receiving messages. The latter can be *asynchronous* or *synchronous*. A synchronous send will not complete until the receiving process has received the message. This allows the sender to have a confirmation that the message has been successfully received. The asynchronous manner sends the message and does not wait for a confirmation. As regards the receive primitive, it will wait until the moment a message is received if it is synchronous whereas the asynchronous manner will return immediately and this, either with a message or to say that no message has arrived.

Point-to-point communication

This is one of the major feature of MPI. Point-to-point communication is a way to communicate between pairs of processes.

When a send operation is done, a message is sent. This one contains the data to send, the type of the data and an *envelope*. The latter allows the good receiver to catch the message. It contains some information such as *destination, tag* and *communicator*. The first two are obvious. The *tag* field allows the program to make a distinction between different types of messages. For example, if two messages are sent with a different data. This is possible that the second message is the first to arrive whereas the first receive operation in the code is planned for the first sent message. Thus, if there was no tag the message should be received and the program could crash or block. So the tag is a kind of security to be sure that a

[2]The description of this standard is inspired by [abopi], [lamrg], [MPIrg], [MPIpi] and [Walml].
[3]A *node* is the name generally given to a computer in a cluster.

sent message has been received by the right receive operation. The *communicator* specifies the *communication context.* It allows a message to be received in the context within it was sent. The communicator also specifies the set of processes which shares this communication context. It is called *process group.* The latter is ordered and processes are defined by their rank inside this group. So the valid values for destination field are from 0 to $n-1$ where n is the number of processes in the group. As an example, the predefined communicator MPI_COMM_WORLD allows communication with all processes that are created by MPI when the program is executed.

So that a message is received, a receive operation has to be encountered and then the envelope of the message is examined. If the destination field corresponds to the process which analyzes the message, the tag field matches the tag field of the receive operation and the process belongs to the process group, then the message is received. A *status* is also present in the receive operation. Its goal is to store information in case of error.

There exist two kinds of operation modes. One is blocking and the other is non-blocking. The blocking operations allow to be sure that the buffer containing the data is copied out before returning from the send operation and that the buffer was filled before returning from the receive operation. In the other hand, the non-blocking can return from the send or receive operation before the data buffer is copied out or filled. It allows to continue the process without completing the communication. However, MPI provides functions to complete the communication. The completion of a send operation thus indicates that the sender is free to update the locations of the send buffer. The completion of a receive operation indicates that the receive buffer is filled. So the receiver can access it and the status object for errors is set. The completion of an operation taken individually is said *locally complete.* When all processes concerned with the operation in question are in a state of a locally complete operation, the operation is said *globally complete.*

Four kinds of communication modes are possible: *standard, ready, synchronous* and *buffered.* These modes are combined only with the send operation. The *standard* mode allows a send operation to be initiated even if a matching receive has not been initiated. This mode can be started whether a matching receive has been posted or not. It can complete before a matching receive is posted but it is not always the case. The standard mode is said to be *non-local.* The *ready* mode allows a send to be initiated only if a matching receive has been initiated. If it is not the case the operation is erroneous and its result is undefined. The completion of the send operation does not depend on the status of a matching receive and thus the send buffer can be reused. As regards the *synchronous* mode, it is the same than the standard mode. However, there is a constraint which is that the send will not complete until the message delivery is guaranteed. In other words, the send can be started whether a matching receive was posted or not but it will complete only if a matching receive is posted. The last mode which is the *buffered* one is also similar to the standard mode but completion is always independent of a matching receive, and the message may be buffered to be sure of this. This mode is said to be *local* seeing that it does not care about what happens for another process. There is only one mode for the receive operation which is the *standard* one.

Collective communication

A *collective communication* requires coordinated communication within a group of processes. The *tag* field is not used here. The reason is that a communication involves all processes

in a group. All collective routines block until they are *locally* completed. There exist two kinds of routines. One for data movement routines and the other for global computation routines. As regards the *data movement routines*, there are three types. *broadcast* allows to transmit data from one member to all members of a group. *Gather* transmits data from all group members to one member. The last one *scatter* sends data from one member to all. Collective routines such as broadcast or gather have a single originating or receiving process. Such a process is called the *root*. It is thus possible to have arguments in the collective functions which are ignored for all participants except the root. There exists a variant of gather where all members can receive the data and also one for scatter/gather which sends from all to all. The figure 3.1 illustrates these three types and their variants called respectively *allgather* and *alltoall*.

The *global computation routines* are *reduce* and *scan*. Reduce allows each process in a group to compute a part of the global computation. The result of each process is then combined on the root process to obtain the global result. As an example, we can say that each process calculates the maximum tree height for its region and process 0 computes the global maximum tree height. An illustration is also given in figure 3.2. Different versions of the reduction routine are provided depending on whether the results are made available to all processes in the group, just one process, or are scattered cyclically across the group. Common reduction operations are the evaluation of the maximum, minimum, or sum of a set of values distributed across a group of processes. The scan routines perform partial reductions in which process i receives data from processes 0 to i. This is also called *prefix-reduction*. This allows to perform a reduction but processes which take part are not necessary all the processes of the group.

3.2.3 Parallel Virtual Machine

Parallel Virtual Machine (PVM)[4] also uses the technique of message passing. This system is designed to allow a network of heterogeneous machines to be used as a single distributed parallel processor. It thus coordinates different computers to execute concurrent or parallel computation.

Principles

PVM is based on several principles which are detailed below.

The first one is *user-configured host pool*. It leaves the selection of a set of computers to the user. Both single-processor or multiprocessor hardwares can be in the host pool. Moreover, the host pool can be modified during an execution by adding or deleting computers.

Translucent access to hardware means that application programs can consider the hardware environment as a collection of virtual processing elements. It can also choose to exploit the capabilities of specific computers in the host pool by attributing computational tasks on the most appropriate computers.

Process-based computation: the unit of parallelism is a task, an independent sequential thread of control that alternates between communication and computation. PVM does not

[4]The description of PVM comes from [pvmml] and [GBD+94].

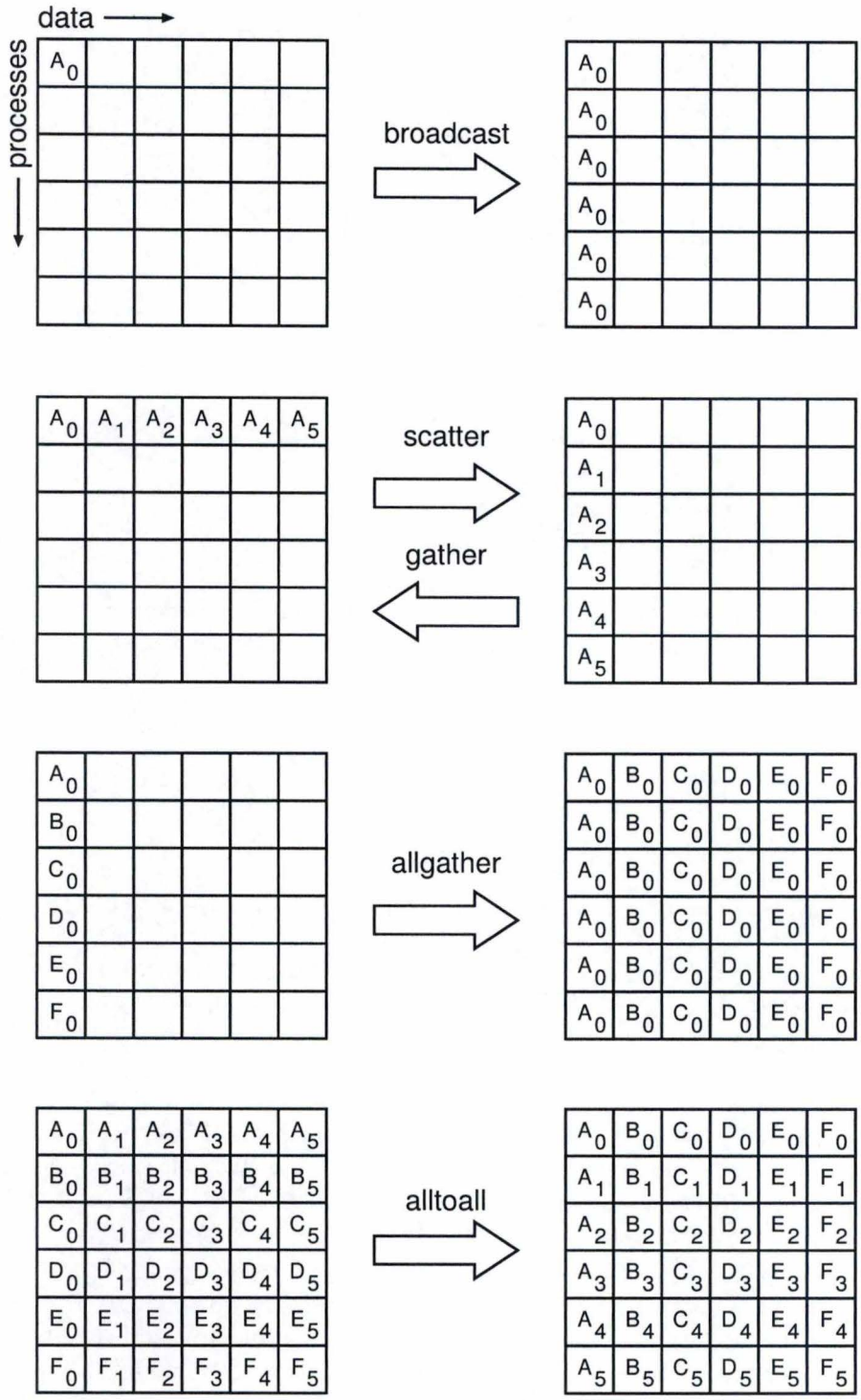

Figure 3.1: Collective data movement

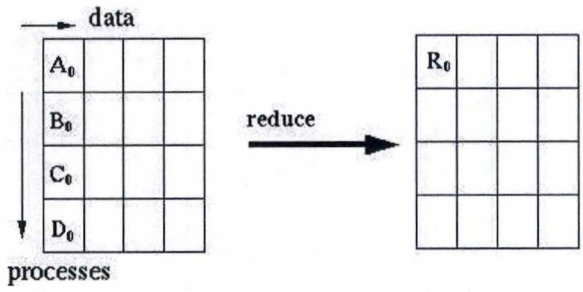

Figure 3.2: Reduce operation

enforce process-to-processor mapping. Multiple tasks can thus be executed on a single processor.

PVM uses *message-passing model.* Tasks performing a part of an application workload communicate by sending and receiving messages. Message size is limited only by the amount of available memory.

The system supports *heterogeneity* in terms of computers, networks, and applications. Message passing in PVM permits messages composed with more than one data type to be exchanged between computers having different data representations.

The last one is the *multiprocessor support.* The latter permits to take advantages of multiprocessor hardwares.

Working

The PVM system is composed of two parts. The first one consists in launching a daemon that is situated on all computers constituting the virtual machine. A daemon is a program which is not invoked explicitly, but which lies dormant waiting for one or several conditions to occur. An example of a daemon is ftpd under UNIX systems. The second part is a library of PVM interface routines. It contains primitives that are needed for the cooperation between the tasks associated with the execution of a program.

PVM can be used at several levels. At the highest level, the transparent mode, tasks are automatically executed on the most appropriate computer. In the architecture-dependent mode, the user specifies which type of computer executes a particular task. In low-level mode, the user may specify a particular computer to execute a task. In all of these modes, PVM takes care of necessary data conversions from computer to computer as well as low-level communication issues.

PVM can be executed in three different paradigms in order to communicate. The first called *crowd computing* allows processes to execute the same code and to perform computations on different portions of the workload. This mode can be subdivided into two categories. The first one is the *master-slave* like in MPI. The master is responsible for process spawning, initialization, collection and display of results, and perhaps timing of functions. The slave programs perform the actual computation involved. Their workloads are allocated by the master. They can also perform the allocations themselves. The second subdivision is

the *node-only* mode where multiple instances of a single program execute, with one process responsible for the non-computational tasks in addition to contributing to the computation itself. The second approach is called *tree computation.* Processes are spawned in a tree-like manner. This mode can be used for applications using algorithms such branch-and-bound or divide-and-conquer. The last approach called *hybrid* can be considered as a combination of the tree model and crowd approach.

Computation model

The model is based on the concept that a program consists of several tasks. Each one is responsible for a part of the program computational workload. It happens that an application is parallelized along its functions. In other words, each task performs a different function. As an example, we can imagine a program divided in the following functions: input, problem setup, solution, output, and display. This method is often called *functional parallelism.* The other method is *data parallelism.* It consists in having all the same tasks but each of them knows and solves a small part of the data. It is similar to the reduce method in MPI. It is possible to use previous methods independently or together.

Send and receive operations

A task is identified by an integer Task Identifier (TID). Messages are sent to and received from TIDs. They are generated by the daemon and are not chosen by the user. The reason is that they must be unique in the entire virtual machine. Both send and receive operations are identified by a TID and a *message tag.* It thus allows to be sure that a send operation is received by the good receive operation.

The sender of a message does not wait for an acknowledgment from the receiver. This one continues as soon as the message has been released on the network and the message buffer can be safely deleted or reused. This corresponds to the blocking mode in MPI. PVM correctly delivers messages if the destination exists. Message order from each sender to each receiver in the system is preserved. Both blocking and nonblocking receive primitives are provided, so a task can wait for a message without consuming processor time. A receive with timeout is also provided, which returns after a specified period of time if no message has arrived.

Load balancing

Another feature of PVM is load balancing. It can be very useful for applications. It allows to be sure that each host is doing its fair share of work. It can conduct to an improvement of the performance. The simplest method is *static* load balancing. In this method, the problem is divided and tasks assigned to processors only once. The size and number of tasks can vary according to the processing power of a given machine. When computational loads vary, a more sophisticated dynamic method of load balancing is required. It works as a master-slaves program where the master manages a set of tasks. It sends jobs to do to the slaves when they become idle.

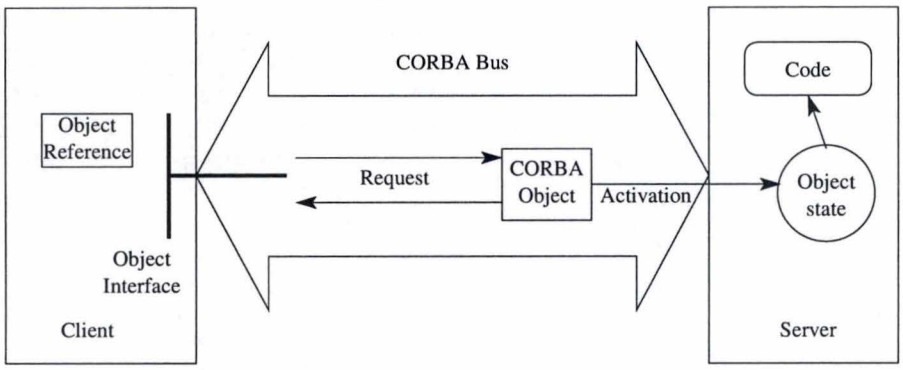

Figure 3.3: The client/server model for CORBA

3.2.4 Common Object Request Broker Architecture

Common Object Request Broker Architecture (CORBA)[5] is one of the technology in fashion for the moment in distributed architecture. It was created and is controlled by Object Management Group (OMG). It is an *object oriented middelware*[6]. It manages the interaction between disparate applications among the heterogeneous computing platforms. The key concepts are reuse, interoperability and portability of software components. The CORBA bus also permits to hide the technical layers such as the operating system, the processor or the network.

Object client/server model

The bus provides an object oriented client/server model of abstraction and cooperation between the applications. Each application can provide services represented by CORBA objects. This is the abstraction part of the model. Interactions between applications are represented by distance calls of the objects methods. This is the cooperation part. The notion of client/server is only useful when an object is used. The server is the one which provides the object and the client is the one which uses this object. The model is illustrated in figure 3.3. In this figure, the *client* is a program which calls the object methods through the CORBA bus. The *object reference* is a structure referencing the CORBA object and containing information needed to find the bus. The *object interface* is the abstract type of the object and it defines its operations and attributes. The latter is described with a special language called Interface Description Language (IDL). IDL is a computer language for describing the interface of a software component. It is essentially a common language for writing the "manual" on how to use a piece of software from another piece of software. IDLs are used in situations where the software on either side may not share common *call semantics*. *Request* is the mechanism to call an operation or to access an object attribute. *CORBA bus* transmits requests. *CORBA object* is the targeted software component. *Activation* associates a CORBA object with the implantation object. *Implantation object* is the component which codes the CORBA object at

[5]The description of CORBA comes from [Eng04] and [GGM04].

[6]A middelware is a software agent that mediates between different components such as an application program and a network.

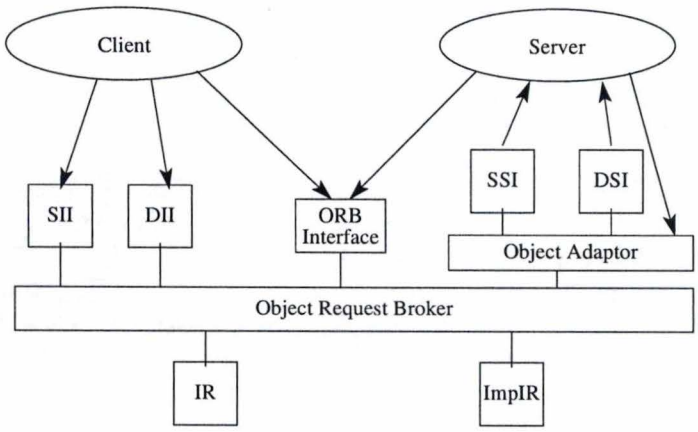

Figure 3.4: CORBA bus

a given time. The *implantation code* contains the methods associated with the implantation operations of CORBA object. The *server* is the container for implantation objects and for operation executions.

CORBA bus

The bus provides interesting features. It provides a *link* between a few programming languages. *Call transparency* makes the request sender believe that it is done locally. *Static and dynamic calls* allow to send requests to objects. Static calls are controlled at compilation and dynamic calls are controlled at execution. It provides *interoperability* between buses as well. The composition of the bus is given in figure 3.4. In this figure, Object Request Broker (ORB) is the layer to transmit requests to objects. Static Invocation Interface (SII) is the static interface which allows to send requests. It is generated by IDL. Dynamic Invocation Interface (DII) has the same goal as SII but it is done dynamically. It is to be noted that no SII is generated in this case. Interface Repository (IR) provides a service with objects which represents information given by IDL and which are available during execution. Skeleton Static Interface (SSI) is the static skeleton interfaces which allow the object implantation to receive requests which are for the implantation. This interface is generated by SII. Dynamic Skeleton Interface (DSI) is the interface for dynamic skeletons. It is generated by a DII and SII are not needed. Object Adapter (OA) is the object adaptor which aims at creating CORBA objects and handling them. Implementation Repository (ImpIR) is the referential of implantation which contains needed information to the activation.

Different approaches can be considered for the bus. With the *process bus* approach, clients and objects are in the same memory space. This is generally used for embedded systems. The second one, *operating system bus*, allows communication to be done between processes which are on the same computer. An example of this is the desktop manager *GNOME*. The next one is *server bus*. Requests are delivered to one or several servers which are responsible for delivering requests to distant objects. This allows a centralized management of the bus. In the last approach, *network bus*, processes are situated on different computers and requests

are transmitted via the network.

3.2.5 RAW TCP/UDP

RAW is not really a technology but it rather means a concept of making the communication protocol ourselves using TCP or UDP. With this technique, we just develop what we need in order to implement the network memory storage mechanism. It can lead to maximum efficiency for communication operations, hence better performance. The major drawback of this technique is the fact that everything has to be done from the beginning and by the programmer. It implies a very good knowledge at a low level of network programming and a longer development time than another technology.

3.2.6 Choice

Now that technologies which are potentially usable in our prototype have been described, one of them has to be chosen. The prototype we will develop only needs basic communications and guarantees on the data transfer. Shared memory is an interesting technology for its rapidity but it remains the problem of memory consistency. The other problem is that it does not use memory of other computers. As regards CORBA, it is a widespread technology very useful for reuse and portability. It is sometimes useful to have a high-level programming but we do not need such a level for our prototype. Moreover, data to exchange are states of a graph which are just a sequence of bytes. An object-oriented middelware appears to be not appropriated for our goals. Another reason is that we just need basic operations to send and receive data when CORBA provides heavy operations to exchange objects. The last argument against CORBA is that it is not well suited for performance. Indeed, it takes a lot of time to communicate. The best suited technology seems to be RAW TCP/UDP seeing that just what we need is developed. Thus, it leads to high performance. This technique can nevertheless take more time to develop than using an existing technique or more problems may be met if it is not well implemented. This is the reason why we do not choose it. The last two technologies, MPI and PVM, are also well suited for our goals. Both provide basic and more complicated send and receive operations for data. It is thus easy to transmit states between processes. Both also allow to develop efficient programs. It appears that PVM is better in some points such as load balancing, dynamic process management which are not implemented in MPI. It thus seems that PVM is the right choice. However, MPI is the technology already used in the development of DiVinE project. Since the prototype to develop and to experiment is destined for DiVinE, the logical choice is to choose MPI instead of PVM.

3.3 Design of a prototype

3.3.1 Principles

The principles of the prototype are quite simple. It has three main principles. As model checking which is applied in the context of the prototype is based on LTL temporal logic and

the automata theory, the first one is thus to go through a given synchronized automaton[7]. The next goal is to store a state and the last one is to find if a state has already been stored. The reason for treating only states is the possibility to generate a counterexample without storing transitions and has previously been explained. Therefore, transitions will not be treated in the prototype. When a state is encountered since we go through the automaton, the prototype calls a function to check if the given state has already been stored. If it is not the case, then a function is called to store the state in memory. A positive result to the check function means that the state was previously encountered and that we are in a cycle.

3.3.2 Characteristics

The prototype has several important characteristics. The two major ones are the use of a hashtable and memory pages.

The hashtable is responsible for keeping information about the place where a state is stored. The place is a page. The existence of a hashtable implies the existence of a hashing function. The latter comes from DiVinE. The reason is that a hashing function was implemented in a first time seeing that the one used in DiVinE was not really good. However, it appears that the developed one leads to very bad results. Indeed, information about almost all states of a graph were stored in only one line of the hashtable instead of being split up among the elements of the hashtable. A new hashing function was later implemented in DiVinE and this one leads to better results. This is the reason for using the hashing function from DiVinE. The hashtable is a table in which each element contains a list called *collision list*. It is possible when hashing two different states that the result of the hashing leads to the same element in the hashtable. It is thus important to have a solution to avoid the crushing of a previously stored information about a state in this element. In order words, it is important to protect the information already stored and to allow new information about another state to be stored in the same element of the hashtable. This solution is the collision list. The collision list is simply a chained list in which each element contains information about a state. All cells relate to a different state. It means that each element of information in the hashtable is different. It is due to the fact that a state and its information are stored only once respectively in the page and in the hashtable. A cell of the collision list is composed of three smaller cells. One for the identification number of the page where the state is stored. The second contains the position of the state in the page. The last one points to the next cell of the list. The hashtable and its collision lists are illustrated in figure 3.5. Let us note that the size of the hashtable is given by the user in a first time and it is modified to obtain the first prime number next to the given one. Using a prime number permits a better hashing than with a non prime number. The elements of the hashtable are set to *null pointer* at initial time. The hashtable is on the master[8] and not on the slaves[9]. In the figure, the size of the hashtable is equal to m elements with $m > 0$. The collision list of element i with $1 \leq i \leq m$ have a size n_i where $n_i \geq 0$.

A page is simply a sequence of bytes in the prototype. Its goal is to contain states which are sequences of bytes too. The fact that we use bytes to represent a page is a will to have

[7]As reminder a synchronized automaton is the product of synchronization between the automaton modelling the system and the one illustrating the bad behaviors of this system.

[8]The master is the node on which the prototype is launched.

[9]A slave is a node of the cluster but not the master.

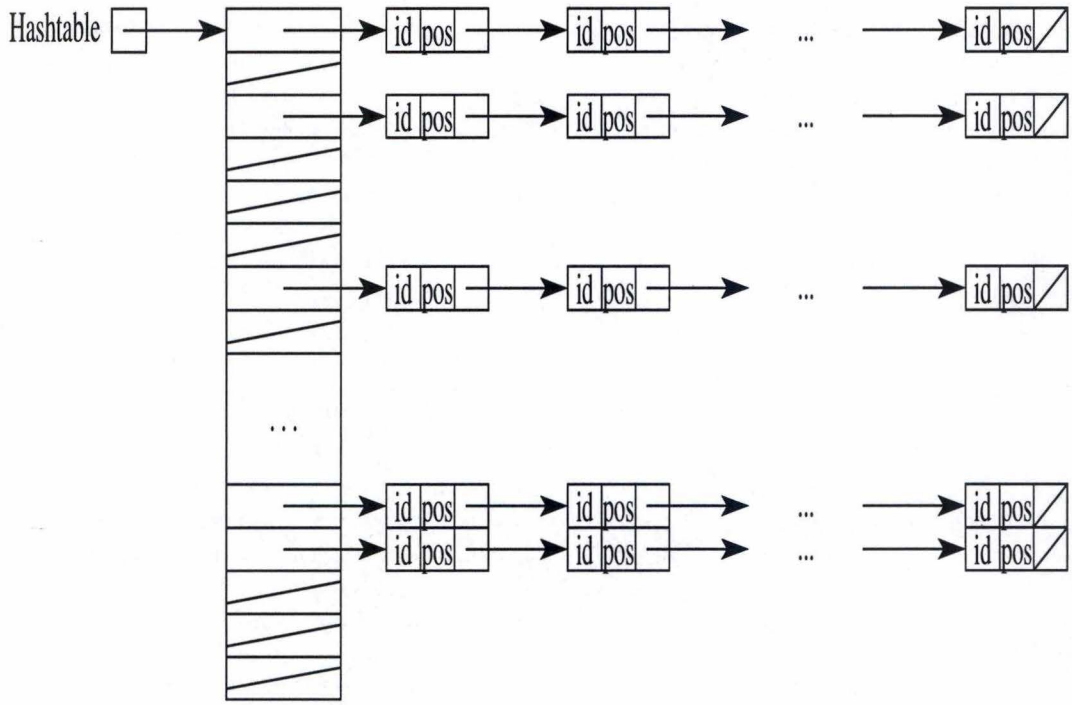

Figure 3.5: Graphic representation of the hashtable and its collision lists

low-level representations for data. It means that high-level functions such as the ones which operate on strings are not used. However, a page can be imagined as a table made of two columns. It is illustrated in figure 3.6. The first row of the table is not used to store a state. It is actually used to identify a page. So the left column of this row contains the identification number of the page. This number is used in each element of collision lists and allows to know in which page the state associated with the element of the list is stored. The right column of this first row is used to know the number of the first free row in the page. It means that if states are already stored, the right column indicates the row which follows the last stored state in the page and which is free. This row is obviously free because states are stored sequentially in the page. In other words, if states are stored in the page, the rows following the last stored state until the end of the page are free and previous rows are filled. Having zero in the last row permits to put this zero in the first row when a state is stored and to know that the page is full for the next storage. Thus, to obtain the real size of the page, the wished size given by the user is incremented by one. All pages are set to this size. The right column of the first row and the last row are set to zero at the initial phase. The remaining rows indicate the next row free of states. Each row where a state is stored has the right column set to zero. Each row free of states are set to the next free row.

Example 3.1. The size given by the user for pages is 10. So the page is composed of 11 rows which are not filled with a state. The first row thus contains the identification number of the page and the first row which does not contain a state. Let us imagine that the identification number is 20. The page at initialization is shown in figure 3.7. The first free row is thus

identification number	first free row
state	next free row
state	next free row
state	next free row
state	next free row
state	page full

Figure 3.6: Virtual representation of a page

Initialization

20	1
free	2
free	3
free	4
free	5
free	6
free	7
free	8
free	9
free	10
free	0

Figure 3.7: Example of a page at initialization

During execution

20	5
State 1	0
State 2	0
State 3	0
State 4	0
free	6
free	7
free	8
free	9
free	10
free	0

Figure 3.8: Example of a page during execution

row 1. Let us now consider that we are at a given step of the execution. The right column of the first row is set to 5. It means that four states have been already stored and that the next state will be stored at row 5. The right column of this latter row is thus set to 6, 7 for row 6, 8 for row 7, 9 for row 8, 10 for row 9 and 0 for row 10. This is shown in figure 3.8.

3.3.3 Approach

The chosen approach is a static approach. It consists in creating all the memory storage at the beginning of the execution of the prototype. All the pages and the hashtable are thus created at the initialization phase of the prototype. Another reason to talk about static approach is that no pages, rows of a page or elements of the hashtable are deleted or added during an execution. The size of the pages and the hashtable is fixed at the beginning of the prototype and are given by the user. The number of pages per node[10] is also given by the user and does not change. It is due to the fact that it is not possible to add or delete processes with MPI. The collision lists are the only ones which evolve during execution. Indeed, cells containing information about states can be added to them. It is however not possible to delete cells. It is coherent not to be able to delete a cell because deleting this information means that the state concerned is not stored anymore. Moreover, allowing deleting cells without deleting these ones in a page is a problem because it means that the state will be stored several times if it is in a cycle of the graph. To be able to do it, a function to delete the state in the page must be implemented and this is not the goal of the prototype.

[10]A node is a computer belonging to the cluster created with MPI.

```
procedure BFS()
    queue := {initstate}
    while queue is not empty do
        state := Head(queue)
        queue := Tail(queue)
        foreach s ∈ Succ(state) do
            queue := queue ∪ {s}
        end foreach;
    end while;
end procedure
```

Figure 3.9: Breadth-first search algorithm for the prototype

3.3.4 Graph browsing

The algorithm to browse the graph is based on Breadth-First Search (BFS). The first test to do with the prototype is to check if it works fine and if it gives the right number of states in the graph. It permits to see if a state is stored several times or only one as we want. DFS and nested DFS algorithms stop when a cycle or an accepting cycle is detected. It does not allow to check if all the states are stored only once. This is the reason for using a BFS algorithm despite the latter may lead to bad performance. The pseudo-code for the BFS algorithm is given in figure 3.9. This algorithm is only used to browse the graph and does not include the treatment for storing or checking states. The algorithm puts the initial state of the graph in a queue. It then enters in a loop if the queue is not empty. The first state of the queue is popped and for each successors of this state, we put them at the end of the queue. It means that all direct successors of a state are treated before to go deeper into the graph.

3.3.5 Evolution of the prototype

The goals of this section are to describe the different evolution steps of the prototype and their working. The prototype uses blocking MPI functions to send and receive data in the cluster in all steps. However the used functions are asynchronous. It means that a sender must not wait that the matching receive operation is published in the MPI environment. The sequence diagrams illustrating the treatment of a state are detailed in appendix A. The reader is invited to consult appendix C to obtain more details about the steps of the prototype.

Step 1: swapping randomly

The first step in the development of a prototype is to swap two pages between the master and a slave. It works as follows. The BFS algorithm is thus used to browse the graph.

When a state is popped from the queue[11], this one is treated as follows. The state is checked in order to see whether it has already been stored in a page or not. To proceed, the state is hashed by the hashing function. The result is the position in the hashtable. Afterwards, the collision list is browsed and for each cell encountered, the information is used to check if it corresponds to our state. It is to be noted that if the state has already been

[11]For more details on the BFS working, see section *Graph browsing*.

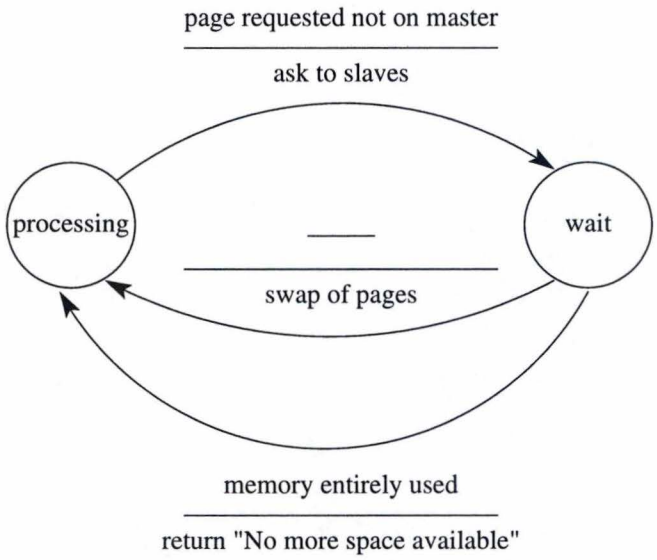

Figure 3.10: Statechart for the master in step 1

stored, the browsing of the collision list has obviously stopped. The identification number is used to find the page where the state is supposed to be stored. If the page is not on the master, then a search is done on slaves to find the page. When it is found, the slave containing the page swaps it with a page on the master. The master one is chosen *randomly* among the pages existing on it. When the right page is on the master, the information from the cell is used to compare our state with the state already contained in the page. If the result is positive, then the state is not stored and the BFS algorithm continues. For a negative result, the prototype continues with a storing function.

The storing part of this step consists in finding a page with free space. The search for free space is done on the master. If all the pages are full, slaves are sequentially called to find a page with free space. It means that the first node is called, then the second one if pages from the first node are full and so on. When a page with free space is found on a slave, the latter swaps the page as explained above. After the swap, the state is stored in the page by the master. States are sequentially stored in pages[12]. The information concerning our state is also put in the hashtable. The position in the hashtable is given by the hashing function. The state is put in a new cell at the beginning of the collision list. We thus take advantage of locality by putting information into a cell at the beginning of the list. Indeed, if the browsing of the graph is in a cycle, then a state has already been stored in the first cells of the list and we can stop earlier the browsing of the list.

The statechart of the master is shown in figure 3.10 and in figure 3.11 for the slaves. In the figures, the word *page* is used for both the page in which a state is checked and a page with free space for the storage.

[12]See section 3.3.2 for details.

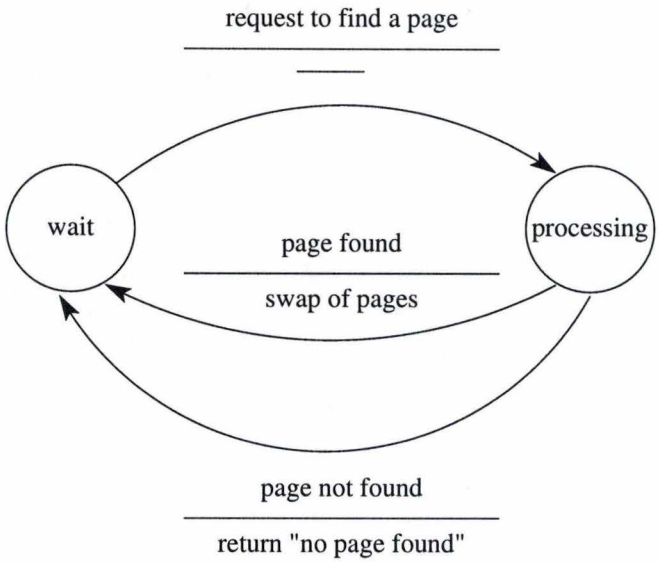

Figure 3.11: Statechart for the slave in step 1

Step 2: swap of full pages

The working of this step is the same as the one explained in step 1. The difference resides in the page which is chosen on the master when a swap must be made when storing or when checking if a state is stored. Indeed, in this case, the prototype calls a function to find a page on the master which is full. The idea is to keep on the master the page which contains free space and to move full pages to the slaves. This step is a transition to step 3.

The statecharts are the same as in step 1.

Step 3: swap of full pages and checking job for the slaves

This step provides a new important feature. It keeps the features of step 2 but the new one concerns the verification of a previously stored state. When a state is popped from the queue, it is checked to see if it has already been stored in a page. The state is hashed and the collision list matching with the result of the hashing is browsed. The identification of the page is thus taken from cell in the list. If the page is on the master, then the working is the same as in step 1. But if the page is not on the master, then the information (both identification number and position of the state in the page) is sent to the slaves. All the slaves search for the page. The one on which the page is checks the state stored at the position number sent by the master with the state sent at the same time. If there is a match, a positive result is sent to the master. In the other case, a negative result is sent. Afterwards the master calls the function to store the state if the state has not already been stored. The use of blocking MPI functions is important here because the master must wait the answer of slaves before storing the state. If non-blocking functions were used, states could be stored several times.

The storing function works as in step 2. If no page with free space is found on the master,

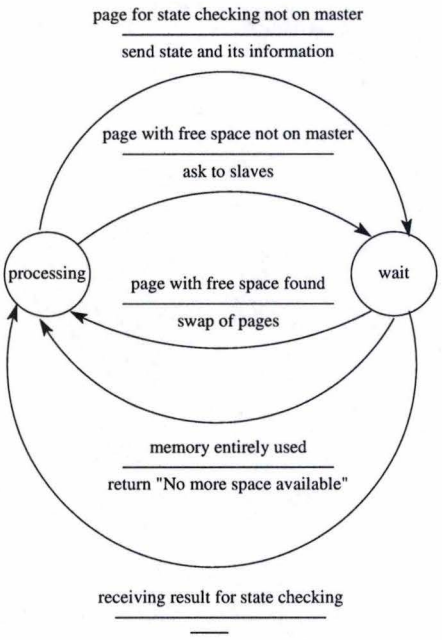

Figure 3.12: Statechart for the master in step 3

then a swap is made. When a page with free space is found on a slave, the latter page and the full page chosen on the master are exchanged. The advantage of moving full pages is that we keep pages with free space on the master. This is interesting seeing that the function to store a state can swap pages and that the function to check a state does not need to swap. A gain of performance can thus be made because the number of swaps are lesser.

The statecharts for master and slave are respectively given in figures 3.12 and 3.13.

Step 4: Last Recently Used

This step is similar to step 2. It does not use the changes made in step 3. The modification resides in a *Last Recently Used (LRU)* which is added to the prototype. The latter contains the last used pages on the master. When a swap is made, instead of choosing a full page on the master, a page which is not in the LRU list is chosen. This allows to keep on the master pages which are often used.

For example, it is possible that states which follow each other can often be encountered. Seeing that the page is sequentially filled, some of them or all of them can be in the same page. This page is subject to be used a few times. It can thus be interesting to keep it on the master and take advantage of spatial locality.

The statecharts are the same as in step 1.

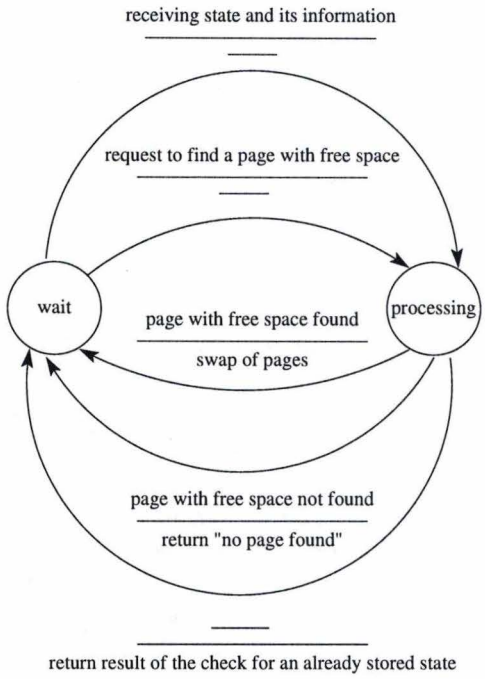

Figure 3.13: Statechart for the slave in step 3

Step 5: LRU and checking job for the slaves

The last step is a mix of steps 3 and 4. The working is the same as in step 3. When a state is checked to see whether it is stored or not, the job is done by the slaves if the page supposed to contain the state is not on the master.

As regards storage job, it is almost done in the same way. The difference is that when a swap must be done, the page from the master is chosen according to the LRU list as in step 4.

The statecharts are the same as in step 3.

3.4 Tests

A series of tests have been done on the different steps of the prototype. All the examples chosen for the tests come from DiVinE and are thus written in DVE language. The first objective of the test was obviously to check if the number of stored states matches the number of states in the graph of the example. The second objective is to see which step gives the best performance and the last one is to change parameters such as the size of the hashtable in order to see how the prototype responds to these changes in terms of performance. Tests have been executed on computers equipped with hyperthreaded Pentium4 2.06GHz and 1 Gigabyte of RAM.

3.4.1 Elevator

The example chosen here is an elevator which serves 14 floors. The graph is composed of 165424 states and 495096 transitions. If the prototype works fine, every step has to store 165424 states. The size of states is constant and is equal to 15 bytes. Results are given in table 3.1. In the table, the parameters are *np* for the number of nodes in the cluster, *hashsize* is the size of the hashtable, *pagesize* is the number of states per page, and *nbr_pages* is the number of pages per node. The column *Common info* is some information which is common to the execution of each step. *s* is the number of states which were stored during an execution. *used* is the percentage of use for the hashtable. *max_list* is the maximum length for collision lists. *w* is the number of times that the prototype swapped pages between the master and a slave. *avg* is the average of cells per collision lists. The collision lists considered by *avg* are the ones corresponding to the use of elements of the hashtable. For example, if the hashtable is used at 80%, the average is calculated on the basis of these 80%. The 20 remaining percents are not taken into consideration. As regards time results, a double point separates hours, minutes, and seconds. A simple point separates seconds and thousandth seconds. *System* is the implementation of the prototype for only one computer. It uses the resources of one computer. For *system*, the number of pages must be multiplied by *np* in order to obtain the number of pages on the computer. This is due to the fact that the total amount of memory allocated is split up between the nodes of the cluster and *nbr_pages* multiplied *pagesize* corresponds to the amount of memory for one node.

Analysis

The first observation is that all the different steps store the right number of states which is 165424. This observation is also true when parameters are changed. We can already conclude that the first aim is achieved and that the prototype works fine.

System release is obviously faster than the prototype because it uses only one computer and the graph fits in main memory. The fastest step for the elevator example is *step 1*. It can be explained by the fact that a *random function* is used to choose the page. This is quite rapid. The other steps search for a full page or a full page which has not recently been used (LRU). Time gained from swapping full pages or non recently used pages instead of randomly chosen pages is lost while the search in order to find these full pages is done. Another explanation which is complementary with the previous one is that swapping randomly allows to swap pages with free space and we can sometimes keep full pages which are more often used on the master. In other words, the random function works fine with the elevator example. It also means that the states in a page do not follow the obtained order while the graph is being browsed. Indeed, a page which is not full can be swapped.
As regards steps 3 and 5, they are faster than steps 2 and 4 because the check for an already stored state is done by the slaves. So the number of times that a swap is done between the master and a slave goes down and it takes less time. However they are slower than step 1 because of the explanation given in the first paragraph but also because of the use of blocking MPI functions. Indeed, the master waits and does nothing while slaves are checking. It represents an important waste of time. Step 3 is faster than step 5. Indeed, handling a LRU list is quite heavy. The list is often browsed several times in order to choose the page on the master which must be swapped and each time that a page is used, the list is updated. We

Parameters	System	Step 1	Step 2	Step 3	Step 4	Step 5	Common info
hashsize=165437 pagesize=2000 nbr_pages=30 np=3	0:8.54 w=0	49:08.16 w=26875	1:35:10 w=197090	1:08:12 w=53	1:36:45 w=165437	1:08:46 w=53	s=165424 used=63% max_list=8 avg=1.57
hashsize=80021 pagesize=2000 nbr_pages=30 np=3	0:08.73 w=0	56:43.57 w=60366	1:57:27 w=259762	1:15:26 w=53	1:57:15 w=281082	1:15:51 w=53	s=165424 used=86% max_list=11 avg=2.41
hashsize=300007 pagesize=2000 nbr_pages=30 np=3	0:08.42 w=0	46:46.28 w=14148	1:27:49 w=168483	1:05:00 w=53	1:28:59 w=175048	1:06:00 w=53	s=165424 used=41% max_list=7 avg=1.34
hashsize=165437 pagesize=600 nbr_pages=100 np=3	0:11.57 w=0	48:33.95 w=22772	1:28:12 w=201247	1:08:56 w=176	1:27:19 w=203343	1:09:38 w=176	s=165424 used=63% max_list=8 avg=1.57
hashsize=165437 pagesize=3000 nbr_pages=20 np=3	0:08:01 w=0	50:47.21 w=29726	1:43:36 w=195003	1:07:31 w=36	1:57:28 w=22349	1:08:11 w=36	s=165424 used=63% max_list=8 avg=1.57
hashsize=165437 pagesize=3000 nbr_pages=30 np=2	0:08.01 w=0	42:01.14 w=12592	1:28:31 w=129990	45:39.12 w=26	1:26:32 w=131463	45:41.50 w=26	s=165424 used=63% max_list=8 avg=1.57
hashsize=165437 pagesize=3000 nbr_pages=15 np=4	0:08.01 w=0	56:17.80 w=45627	1:56:21 w=235798	1:16:47 w=41	2:17:39 w=274437	1:19:02 w=41	s=165424 used=63% max_list=8 avg=1.57

Table 3.1: Table of results for the elevator example

thus waste, as explained previously, more time to find the page to swap using a LRU list than simply searching for a full page to swap.

The observation resulting from the changes operated on parameters is that the hashtable is not well filled. For a size close to the number of states in the graph, the theoretical result is maximum one cell per elements of the hashtable. In practice, the percentage of used elements is equal to 63%. It is not very good. The maximum length for a collision list is 8 cells. It thus takes more time to check a state (because the list has to be visited several times until the end) than if the maximum length was 1 cell. A reduction of the size leads to a slower performance because each time a state is checked, the collision list is visited. In this case, the maximum length of a list is 11 cells and all the lists are generally longer. Indeed, the average number of cells per used list is 2.41 cells when it is 1.57 for the size close to the number of states to store. It is an important difference and it explains the longer execution time. The used percentage goes up to 86%. An increase of the size leads to better performance. Indeed, if we look at the results, the average goes from 1.57 up to 1.34 and the maximum length decreases by one. The used percentage falls to 41%. However, the gained time is not significant. Thus, having a big hashtable is not very useful.
We can also observe that changing the number of pages per node has a slight influence on the performance. Increasing the number of pages to 100 per nodes improves slightly the

performance of step 1. Step 3 and 5 encounter the opposite effect. In the case of a decreasing from 30 to 20 pages per nodes, the slight changes are observed and the opposite effect is observed. Step 1 shows an increasing of the execution time and for steps 3 and 5 a decreasing is observed. All these changes can be explained by the variations in the number of swaps when comparing with the first test. The variation of these parameters is not a key point to speed up the prototype for this example.
Another interesting factor is the number of nodes in the cluster. The observation is that when the number of computers is decreased, the prototype is more rapid. The opposite effect is obviously observed for an increase of the number of computers in the cluster. The gain or waste of time are relatively important and apply to all steps of the prototype. It is thus a parameter to take into account. The explanation is that the number of swaps decreases when the number of computers decreases and vice-versa. Indeed the amount of pages on the master is bigger and the master can thus process more states before working with the slaves. In this case, step 3 and 5 are closer to step 2. Step 5 is also closer to step 3 because the number of times the LRU list is browsed when a page must be swapped is smaller.

3.4.2 Firewire link

The second example is the model of layer link protocol of the IEEE-1394. In other words, it is the protocol for the firewire bus. The number of states in the graph is 188420 and the number of transitions is 462598. States have a size of 59 bytes. The results of the tests are shown in table 3.2. The concepts used in the table of the elevator are also used here.

Analysis

As for the elevator example, the first objective is achieved. All steps have stored 188420 states. This is also true when executing with different parameters.

System remains the fastest and it is normal seeing that it fits in main memory. We can observe that it takes more time to execute. The main difference with the previous example is that *step 1* is not faster than the other steps. It can be explained by the fact we are not lucky with the random function which chooses the page to swap on the master. If we look closer, then we can observe that the number of time the prototype swapped is greater than for the elevator example. This number is also closer to the number obtained for steps 2 and 4. That is not the case for the elevator. All the different executions of the prototype on the firewire protocol example show the same behavior for step 1. That is to say a number of swaps closer to the one of the other steps, excepted for steps where the checking job has been given to the slaves.
The step which gives the best performance is step 3. The reason is the same as for the elevator example. It takes more time to handle the LRU list than choosing simply a full page if we compare with step 5. If the comparison is done with steps 2 and 4, then it is explained by the fact that the number of times the prototype swapped is lower. However, the blocking functions play again a key role in the performance decrease.
As regards the changes operated on parameters, the hashtable also encounters filling problems. The percentage is not really high and the observation is the same as the one described in the elevator example. For a size close to the number of states to store, the ideal percentage is not met and is equal to 63%. The maximum length of collision lists is 7 cells. When the size

Parameters	System	Step 1	Step 2	Step 3	Step 4	Step 5	Common info
hashsize=188431 pagesize=1000 nbr_pages=63 np=3	0:23.46 w=0	52:38.30 w=44846	54:14.48 w=49686	42:57.44 w=126	55:01.35 w=52410	43:24.86 w=126	s=188420 used=63% max_list=8 avg=1.58
hashsize=50021 pagesize=1000 nbr_pages=63 np=3	0:24.72 w=0	1:36:58 w=169278	1:38:27 w=182865	53:45.73 w=126	1:45:22 w=188198	52:43.29 w=126	s=188420 used=97% max_list=16 avg=3.86
hashsize=360007 pagesize=1000 nbr_pages=63 np=3	0:23.27 w=0	47:38:27 w=23366	48:26.27 w=26847	44:43.40 w=126	47:57.04 w=28602	43:14.33 w=126	s=188420 used=41% max_list=7 avg=1.28
hashsize=188431 pagesize=500 nbr_pages=126 np=3	0:26.44 w=0	47:10.62 w=44221	46:58.19 w=50151	43:59.24 w=251	50:33.82 w=50409	46:52.43 w=251	s=188420 used=63% max_list=8 avg=1.58
hashsize=188431 pagesize=3000 nbr_pages=21 np=3	0:22.45 w=0	1:12:46 w=47610	1:10:44 w=49622	46:19.19 w=42	1:16:46 w=56028	48:59.23 w=42	s=188420 used=63% max_list=8 avg=1.58
hashsize=188431 pagesize=945 nbr_pages=100 np=2	0:23.82 w=0	42:42.42 w=25053	50:34.07 w=29099	39:54.80 w=100	50:42.87 w=29132	39:36.26 w=100	s=188420 used=63% max_list=8 avg=1.58
hashsize=188431 pagesize=378 nbr_pages=100 np=5	0:28.37 w=0	58:04.98 w=64019	1:00:06 w=74808	50:31.55 w=399	1:00.15 w=74791	53:20.30 w=399	s=188420 used=63% max_list=8 avg=1.58

Table 3.2: Table of results for the firewire link example

is smaller, the percentage is higher. For our example, a size of 50021 implies a use of 97%. However, the maximum length is of 16 cells and the average number of states per used lists goes from 1.58 to 3.86. The difference is significant and it explains the results obtained in terms of execution time. When the size is bigger, the percentage of use decreases to 41% in our example. It only allows to decrease the maximum length of a list by one. The average number goes from 1.58 to 1.28. This is not a huge difference and it explains the slight decrease of execution time.

The changes operated on a page are interesting. We can observe it is more rapid when the number of pages is greater, except for steps with the checking job given to the slaves. It is slower when the number of pages is lower. It is due to the fact that it takes less time to swap small pages than big pages.

The number of nodes is also very interesting. When the number of nodes decreases, the number of swaps decreases. It speeds up the prototype because the master works for a longer time. In case of an increase of the number of nodes, the performance from step 2 to step 5 are not good if we compare with the first test. However, the interesting point is that step 1 becomes more efficient than the other steps. It even seems more efficient than the results obtained for the first test.

Parameters	System	Step 1	Step 2	Step 3	Step 4	Step 5	Common info
hashsize=177167 pagesize=900 nbr_pages=100 np=2	0:21.10 w=0	2:02:45 w=37609	4:27:31 w=249568	2:23:07 w=97	4:29:51 w=252909	2:42:47 w=97	s=177147 used=63% max_list=8 avg=1.58

Table 3.3: Table of results for the dining philosophers example

3.4.3 Dining philosophers

The third example is based on the dining philosophers. Eleven philosophers have dinner and one of them is left handed. The graph of the model is composed of 177147 states and 1299078 transitions. States have a size of 33 bytes. In view of the number of transitions, the execution time will be greater if we compare with the previous examples. The results of the tests done are shown in table 3.3. The concepts used in the table of the elevator are also used here.

Analysis

First of all, states are stored only once and the right number has been stored by the prototype. The first objective is also achieved for this example.

System remains as usual the most rapid. We can also observe that the execution time for all steps is very high. It is explained by the huge number of transitions and the number of states which is nearly the same as the ones met in the other examples. A major point is that as for the elevator example the most efficient step is the number 1. The reasons for that are the same as in the elevator example. The number of swaps between step 1, step 2 and 4 is great. There are around 7 times less states in step 1 than in step 2 and step 4. It explains the big difference in terms of execution time which is approximately two hours. The difference between step 1 and steps 3 or 5 which is around 20 minutes. It is explained by the fact that it takes more time to find a full page or a page not recently used for swapping than a page chosen by the random function.

Tests in which parameters vary have not been done since the access to computers of the laboratory have been restricted. The other reason is that the prototype has the same behavior as in the previous examples[13].

3.5 Encountered problems

Several problems have been encountered when developing the prototype. One is a misunderstanding with the principles that the prototype must follow[14]. Indeed, the prototype which was under development did not match the vision of the Czech supervisor. The other ones are technical. Release problems between DiVinE and MPI were encountered. It took a

[13]See previous examples to see how it evolves.

[14]The idea of the program developed when there was the misunderstanding is given in chapter 4, dynamic approach.

lot of time to resolve them. Because of that, the working tests concerning the communication between nodes were delayed. There were also problems when executing the different steps. Computers in the lab were changed, computers were switched off while executing, processes were killed for obscure reasons. We encountered power cuts, script failures as well. Let us note that the tests were mostly executed in ParaDiSe laboratory from Belgium.

3.6 Conclusion

In view of the results given by the different examples, we can say that the prototype achieves its first goal which is to be a network memory mechanism. It is possible to use it for large models which need more than 4 Gigabytes of memory.

The drawback is that the prototype is not fast. The changes operated in order to improve the prototype by adding a LRU list, swapping only full pages or giving the checking job to the slaves are not efficient when we compare them with our expectations. The performance problem probably comes from the use of blocking functions, especially when a check for an already stored state is carried out by the slaves. It also seems that the performance decreases with an increase of the number of nodes. Despite the fact that the fastest step is not step 1 in the firewire example (except when the number of nodes is greater), it is the most efficient step. It is possible that the bad performance is due to the fact we are not lucky with the random function in the firewire example. However it may be useful to find an alternative to step 1. Indeed, if the performance depends on the example, then it is not interesting to execute step 1 to see whether it works well or not with this example. The size of the hashtable is closer to our expectations when it is closed to the number of states to store. As regards the page size, it is slightly better when pages are not too big.

Thus the improvements to do concern the performance of the prototype. These improvements are the topic discussed in the next and last chapter.

Chapter 4

Perspectives

It has been shown in the previous chapter that the prototype achieves its main goal which is to be sure that the states of a graph are stored only once. However, the prototype suffers from a lack of performance. The different sections of this chapter concern the improvements and the other possible manners to implement the prototype from the beginning. The solutions proposed in this chapter do not necessary lead to better performance. The prototype is experimental and experimentations must be carried out to verify whether the solutions speed up the prototype or not.

4.1 Improvements to the prototype

4.1.1 Optimal value for parameters

The first point developed is the optimization of the value given to the different parameters of the program. The results from the tests have shown that better results are obtained when the number of nodes is small. It is thus interesting to evaluate how many computers are needed to execute the program.

The other value which is interesting to optimize is the size of the hashtable. The previous chapter has shown that it is more interesting to have a hashtable size closed to the number of states to store. However, the hashtable has a use of 63% in the examples. The latter can be improved to be closer to the ideal value of 100%, which means maximum one cell per collision lists. In order to put a cell into a collision list, the state is hashed to know the position of this list in the hashtable. Seeing as the hashing function is responsible for the place where the information about a state is placed in the hashtable, a solution is to improve this hashing function. It increases the percentage upper than 63% and reduce the maximum length of the collision lists. It means a faster prototype. Indeed, when a state is checked to see whether it has already been stored or not, the state is hashed and the collision list is browsed. If the distribution of the states in the hashtable is better and collision lists are smaller, then the time needed to check a state is lower too. However, such a hashing function is not easy to obtain and it can take a lot of time. It can even be impossible to find a better function since it is experimentation.

4.1.2 Use of non-blocking functions

An interesting improvement to do is to change the prototype in order to support non-blocking functions. As explained before, the prototype currently uses blocking functions. It means that when a checked operation on a state is carried out and the page given by the information cell in the collision list is not on the master, the master waits for the result of the check done by the slaves. The master is thus idle for that time. This is the case of steps 3 and 5. The other steps are also concerned but it is less important. It is therefore interesting to modify the prototype in order to allow the master to continue the execution while the check is done by the slaves. It is not necessary to use non-blocking functions for the storage job of the prototype seeing that the page with free space chosen is *always swapped* on the master and that the master needs this page to store the state. In other words, this is only the master which carries out the storage job.

Let us note that it is not an easy task to do. The coordination between the master and the slaves to optimize the workload must be correctly done in order to avoid a deadlock of the prototype or errors. For example, it is possible that a slave or the master catch the wrong send or receive operations and that leads to deadlock most of the time. If the master continues to work while the check is carried out by the slaves, then the master can not start storing a state before obtaining the result of the check. However the master can swap a page to prepare the storage if there is no more free space on it. These examples show the difficulty of coordinating the operations inside the cluster. It is obvious that the use of non-blocking functions speeds up the prototype. This decrease of execution time can be significant for steps 3 and 5.

4.1.3 Giving the storage job to the slaves

Another solution to improve the performance of the prototype is to allow slaves to store states when there are no more pages with free space on the master. The tests have shown that when the slaves carry out the checking job the steps are more rapid than the steps which swap all the time. Time can be also gained if the storage job is done by the slaves. It can be however less significant when we compare with the time gained in steps 3 and 5. This is normal seeing that the number of times the prototype checks if a state is stored can be significantly greater than the number of times the prototype stores a state. This improvement permits to avoid swapping.

Despite the time gained with this solution, it will be a pity if the operations between nodes are blocking. Hence the other advantage related to the previous subsection is to take advantage of non-blocking functions. Indeed, we can imagine that when a check is carried out, all the slaves send information about free space on it at the same time. So when a state must be stored, the master sends the information to the node with free space without waiting for a response. The node can be the one with the smallest identification number if we want that the stored states follow the order in which we encountered them while browsing the graph. If it is not the case, any node with free space can be chosen. The slaves are responsible for the storage and the master can store information in the hashtable and goes on with the execution.

Let us notice that, with this solution, there are no more swaps. The steps 2 to 5 which search for an ideal page to swap are not useful anymore. The time spent when searching for a full page or working on the LRU list can be spared. It is logical that the execution time will decrease and the prototype will be then more efficient.

```
state := initstate;
DFS(state);

procedure DFS(state)
    if (state, 0) ∉ visited then
        visited := visited ∪ {(state, 0)}
        in_stack := in_stack ∪ {state}
        foreach s ∈ Succ(state) do
            DFS(s)
        end foreach;
        if Accepting(state) then
            NDFS(state)
        endif
        in_stack := in_stack \ {state}
    end if
endprocedure;

procedure NDFS(state)
    if (state, 1) ∉ visited then
        visited := visited ∪ {(state, 1)}
        foreach s ∈ Succ(state) do
            if s ∈ in_stack then
              Report("Cycle")
            else NDFS(s)
            endif
        end foreach;
    end if
endprocedure;
```

Figure 4.1: Nested depth-first search algorithm

4.1.4 Nested DFS

Nested DFS is the last solution proposed to improve the prototype. It is closer to model checking and especially to LTL model checking[1]. The idea is to use here other algorithms to browse the graph of a model in order to verify a model instead of using the BFS algorithm. The latter is nonetheless slow. The nested DFS allows to find *an accepting cycle* in the graph of a synchronized product[2]. The latter corresponds to a counterexample and thus a bad behavior of the system. It means that the model of the system is not good. At this time, it is not necessary to continue since a counterexample has been found. The use of these algorithms leads to better performance seeing that if the model is not good, we stop. Moreover, it allows the prototype to do model checking and verify the correctness of a system. It is the major improvement to do in order to have a more efficient prototype. The pseudo-code for nested DFS algorithm from [BBv02] is given in figure 4.1.

[1]The one using automata theory.

[2]The synchronized product is the product of the automaton modeling the system and the automaton representing the bad behaviors of the system.

In this figure, a state is first checked if it has not already been discovered by the DFS procedure. Number 0 and 1 are used to indicate whether the state has been encountered by the DFS procedure or by the NDFS procedure. It allows to goes through every reachable state only once. The state is then put on a stack and its successors are generated. The algorithm calls itself until no more successors can be generated. The next step is to check if the current state is accepting. If it is the case the nested DFS procedure is called. In other words, the nested DFS is started when the first DFS backtracks from an accepting state. The state is popped from the stack if it is not accepting.

The nested DFS first checks if the state given as a parameter is not a previously visited state by the procedure. If it is not, then the state is added to the set of visited ones. Afterwards all the successors of the state are generated and for each of them, the algorithm checks if they are on the stack handled by the DFS procedure. If it is on the stack, then it means that a cycle has been detected and the algorithm stops. The states met in the nested DFS procedure compose the counterexample. These states are the ones labelled with integer 1. If it is false, then the algorithm calls itself. When the nested DFS is done without success, the DFS procedure resumes.

4.2 Re-implementing the prototype

4.2.1 A more dynamic approach

The *dynamic approach* is another approach which the prototype can use. The static approach[3] allocates the memory at the beginning of the execution of the prototype. The problem which can be encountered is that the memory allocated is exhausted and no more space is available. With the dynamic approach, there are no more pages in which states are stored. This approach actually allocates the memory needed for the hashtable at the beginning of the execution. This is quite normal because a reallocation of tables during an execution is not a rapid and reliable operation. It is not possible as for the static approach to change the size of the hashtable. States are stored directly in the hashtable. The collision list thus contains cells in which states are stored. The list evolves as the graph is browsed. State space is thus not limited by the amount allocated to pages. This is the reason why this approach is said to be *dynamic*. Another special feature is that the hashtable is split up among the nodes of the cluster. It is considered as a *distributed hashtable*. It allows to use the memory of all the computers composing the cluster. It is the network memory mechanism. If the size of the hashtable is m, then it is split up among the nodes. Each hashtable of a node has a size of m/n where n is the number of nodes in the cluster. The size is an integer. Hence, the size of each hashtable can not be the same if the result of m/n is a real. The different collision lists have a size of $k \geq 0$. Cells contain different states. i is the ith node in the cluster and $0 \leq i < n$. An illustration of the split hashtable is given in figure 4.2. The working is the following one. When the graph is browsed and a state is encountered, the state is hashed and the result gives the position of its collision list in the hashtable. It means that for the hashing function all the parts of the different hashtables are put together and it appears to be only one big hashtable. With this position, we can know to which node the state must be sent. It is obvious seeing that the position is in a part of the hashtable and

[3]For more details, see chapter 3, subsection 3.3.3.

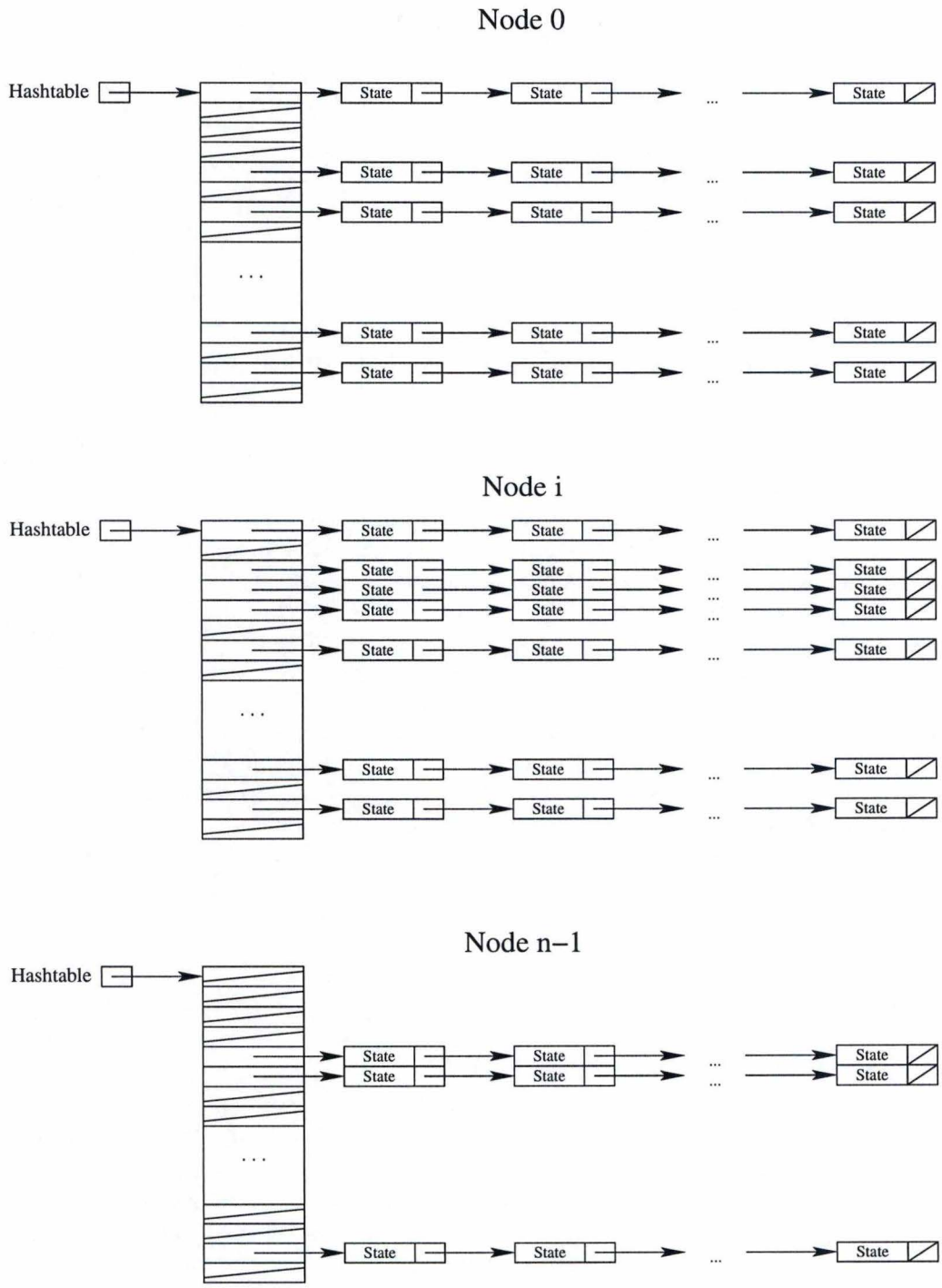

Figure 4.2: Hashtable and the dynamic approach

that this part corresponds to a certain node. When the information is received by the right node, the collision list corresponding to the given position in the part of the hashtable is then browsed and the state is checked with the one contained in the cells. If there is a match, the browsing of the graph goes on and the same procedure with the new state is started. If the state has not already been stored, then the state is added at the beginning of the list. Afterwards the browsing of the graph can go on.

The idea can however be improved by using non-blocking functions. It is possible because when a state is hashed, it is transmitted to the node which owns the part of the hashtable relating to the result of the hashing and this node is responsible for the check and the storage. We are thus sure that if the state has already been stored, it is on this node and not another one. Moreover, no messages or results must be sent to master at the end of the treatment of the state. When the part of hashtable owned by the master is not concerned by the treatment of a state, it can simply send the state and the result of the hashing to the slave and afterwards continue the browsing of the graph while a slave is treating the state. Hence, it seems that time is gained and the performance are improved.

Nothing is perfect and this method has thus a drawback. It is possible that the physical memory of a node is more used than the one of another node. It depends on the quality of the hashing function. When the latter does not give good results, the collision lists are bigger and the percentage of use of the complete hashtable is weak. If we are not lucky, then a node will contain more states than another node and for a very large graph it can be a problem. However, we can be luckier and despite the bad quality of the hashing, the sharing out among nodes is more efficient. In the other hand, if the hashing is efficient (close to 100% of use of the hashtable), then the sharing out is almost perfect. When we look at the tests of chapter 3, we can observe that a size of the hashtable greater than the number of states in the graph is not ideal for this method. The smaller size is better for the sharing out but it may not be perfect because of the increase of the maximum length of collision lists. The ideal should be that the size of the hashtable is equal to the number of states to store but on condition that the actual hashing function is improved. With the actual function, it seems that a size slightly smaller than the number of states remains the best. This method seems to be very interesting to experiment.

4.2.2 PVM

When PVM was described in chapter 3, it seemed to be the technology which suits best for the design of the prototype. However, MPI was chosen because it has already been used in the DiVinE project. It is not impossible to use both technologies in the project on condition that the prototype is not linked with another part of DiVinE which uses MPI. In that way, the prototype can take advantage of PVM and the features that it provides. An interesting feature is load balancing. It allows to share out the workload between the nodes. So each node are not idle for a certain time. Another paradigm such as the tree computation, hybrid or node-only can also be used instead of the *master-slaves* method[4].

[4]For more details about PVM paradigms, see PVM section in chapter3.

4.2.3 RAW TCP/UDP

This method is possible to design if we want to improve the prototype by using communication operations which are the most efficient. As said before, this method probably requires more time to develop. Indeed, all the communication operations are manually implemented by the designer of the prototype.

An idea is to develop a system with a main node and where daemons are launched on the computers. All daemons must be launched manually on each computer. It can be done with "ssh"command. This daemon is listening for requests. The connection will be done through sockets between computers. On each computer, one socket is devoted to the listening of requests and another one is used for sending or receiving data. It is also possible to add a node in the cluster if it is needed. Indeed, if the daemon is launched on a computer and the latter sends information to the main node, then it will be registered on the main node and this one can use the new added node. It works more or less like an architecture Client/Server. The main node sends the information, after nodes have been registered to it, in order to allocate the memory needed for the execution. We can also imagine that the allocation of memory is done when the node received its first job. It allows not to block resources when the node does not work. Methods used to do load balancing are also possible to implement.

The protocol used can be Transmission Control Protocol (TCP) if we want to be sure that data are received. If the prototype tolerates losses, it can be envisaged to use User Datagram Protocol (UDP). It even permits to speed up the execution.

4.3 Origin of the ideas

The origin of the ideas which have been proposed in this chapter in order to improve the prototype is discussed in this section.

Most improvements of the first part of the chapter were suggested at the end of the training period in Czech Republic: optimal value for the size of the hashtable, the storage job for the slaves, and nested DFS algorithm for browsing the graph. However, the idea to modify the hashing function was not planned. It comes from the observation of the last obtained results as well as the use of non-blocking functions and the research to find an optimal value of the number of nodes.

As regards the second part concerning the re-implementation of the prototype, all the ideas that have been developed have not been suggested and also result from the observation of the performance of the prototype.

Conclusion

Hardware and software systems are present everywhere in our daily life. The correctness of these systems is very important in order to avoid problems which can cost a lot of money in some cases. Methods have been developed to verify all these systems and *model checking* is one of them. It is probably the one which is the best suited for verifying systems.

Model checking is based on the model of systems and the verification of properties which must be satisfied by these systems. This model is a graph with states and transitions. We have seen that the biggest fear is to encounter the *state space explosion problem* which happens when the number of states is huge. When this problem is met, the program can not be efficient. The main challenge for model checking is to deal with this problem. Methods to avoid the state explosion problem have been described. They suggest solutions to reduce the number of states that are generated. Some of them have nevertheless drawbacks. Indeed, the methods to reduce the number of states generated such as partial order reduction or abstraction apply to a subset of the behaviors of the full graph. It is often sufficient to verify the most important properties. Another technique has been described. It uses the computation power of several computers grouped in a cluster. This is the *distributed model checking* method. We have also seen that this method may suffer from the revisiting problem. Another way is to use non-distributed algorithms and to take advantage of a large memory composed with the memories of several computers.

It is therefore in this context that we try to develop a prototype of a network memory mechanism. Different difficulties have been encountered since the beginning of the development of the program. Most of them are technical. The main goal of the prototype has however been achieved. The mechanism works fine and the states belonging to the model of a system are correctly stored. It is even possible to use it when the model requires more than 4 Gigabytes of memory. We have thus shown that it was possible to build a big random-access memory in order to handle huge industrial systems.

The prototype evolved for three months of development. In addition, we have shown that a reflexion on techniques in order to improve the performance was done. They have been implemented. Unfortunately, the prototype remains quite slow. Thus, it is not really efficient at this moment. It is probably due to the use of non-blocking functions for communicating inside the cluster and models which contain a big number of cycles.

In view of the poor performance, we can conclude that the development of the prototype is not a complete success. That is why perspectives for future work have been defined.

We hope that the reader has realized that model checking is an interesting domain of research and, above all, very useful to verify the correctness of huge industrial systems. Finding new methods to allow model checking to deal with the state explosion problem as well as to

develop a program using parallel and distributed concepts is not an easy task to do. Since the main goal is achieved and considering the poor performance of the prototype, we can finally say that we have won a battle but not the war.

Bibliography

[abopi] http://www.mpi-softtech.com/products/cluster/about_mpi/. (Last visit on July 28th 2004).

[Bar02a] J. Barnat. How to distribute ltl model-checking using decomposition of negative claim automaton. In *SOFSEM 2002 Student Research Forum Proceedings, Milovy, Czech Republic*, 2002.

[Bar02b] J. Barnat. Using verified property to partition the state space in ltl model-checking. In *Proceedings of the Summer School on Modelling and Verifying Parallel Processes (MOVEP'2002), Nantes, France*, 2002.

[BB03] L. Brim and J. Barnat. Distribution of explicit-state ltl model-checking. In Thomas Arts and Wan Fokkink, editors, *8th International Workshop on Formal Methods for Industrial Critical Systems (FMICS 03)*, volume 80 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2003.

[BBF+01] B. Bérard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, Ph. Schnoebelen, and P. McKenzie. *Systems and Software Verification: model-cheking techniques and tools*. Springer, 2001. ISBN 3-540-41523-8.

[BBv02] J. Barnat, L. Brim, and I. Černá. Property Driven Distribution of Nested DFS. In *VCL 2002: The Third International Workshop on Verification and Computational Logic, Pittsburgh PA, October 5, 2002 (held at the PLI 2002 Symposium)*, 2002.

[defry] http://www.hyperdictionary.com/dictionary/. (Last visit on August 12th 2004).

[divml] http://anna.fi.muni.cz/divine/index.html. (Last visit on August 4th 2004).

[Eng04] V. Englebert. Systèmes coopératifs. Namur, Belgium, 2004.

[expgn] http://www.cesnet.cz/doc/techzpravy/2002/ipv6hwdesign/. (Last visit on July 30th 2004).

[Galps] Stéphane Galland. Les circuits séquentiels. EURISE department, Jean-Monnet University, http://set.utbm.fr/membres/galland/enseignement/archi/seq2.ps. (Last visit on July 26th 2004).

[GBD+94] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine. A User's Guide and Tutorial for Networked Parallel Computing*. MIT Press, 1994. ISBN 0-262-57108-0.

[GGM04] J-M. Geib, C. Gransart, and P. Merle. Corba: des concepts à la pratique. http://www.rennes.supelec.fr/rennes/si/equipe/lme/ENSEIGNEMENT/TD_CORBA/cours_CORBA.pdf, (Last visit on August 2nd 2004).

[JGP99] E. Clarke Jr, O. Grumberg, and D. Peled. *Model checking.* The MIT Press, 1999. ISBN 0-262-03270-8.

[lamrg] http://www.lam-mpi.org/. (Last visit on August 12th 2004).

[Lerpt] Flavio Lerda. Ltl model checking. Carnegie Mellon University, www-2.cs.cmu.edu/~emc/15-820A/reading/ltl_model_checking.ppt. (Last visit on July 5th 2004).

[MPIrg] http://www.mpi-forum.org/. (Last visit on August 12th 2004).

[MPIpi] http://www-unix.mcs.anl.gov/mpi/. (Last visit on August 12th 2004).

[pvmml] http://www.csm.ornl.gov/pvm/pvm_home.html. (Last visit on August 1st 2004).

[Roypt] Abhik Roychoudhury. Temporal logics. National University of Singapore, www.comp.nus.edu.sg/~cs4271/lectures/lec3-4.ppt. (Last visit on July 5th 2004).

[Sch04] P-Y. Schobbens. Preuves automatiques et preuves de programmes. Namur, Belgium, 2004.

[Walml] David W. Walker. Mpi : From fundamentals to applications, http://www.cs.utk.edu/~dongarra/WEB-PAGES/cs594-2001.html. (Last visit July 28th 2004).

[wikrg] http://wikipedia.org. (Last visit on August 12th 2004).

[worMC] http://www.fi.muni.cz/concur2002/PDMC/. (Last visit on August 12th 2004).

Appendix A

Diagrams

The sequence diagrams for the treatment of a state by the different steps of the prototype are presented in this appendix.

A.1 Step 1: swapping randomly

The sequence diagrams of step 1 for the treatment of a state are shown in figure A.1 for the master and in figure A.2 for the slaves.

A.2 Step 2: swap of full pages

The sequence diagrams of step 2 are illustrated in figure A.3 for the master and in figure A.4 for the slaves.

A.3 Step 3: swap of full pages and checking job for the slaves

The following figures illustrate sequence diagrams for the treatment of a state in step 3. Figure A.5 concerns the master and figure A.6 concerns the slaves.

A.4 Step 4: Last Recently Used

The sequence diagrams are the same as in step 2.

A.5 Step 5: LRU and checking job for the slaves

The sequence diagrams are the same as in step 3.

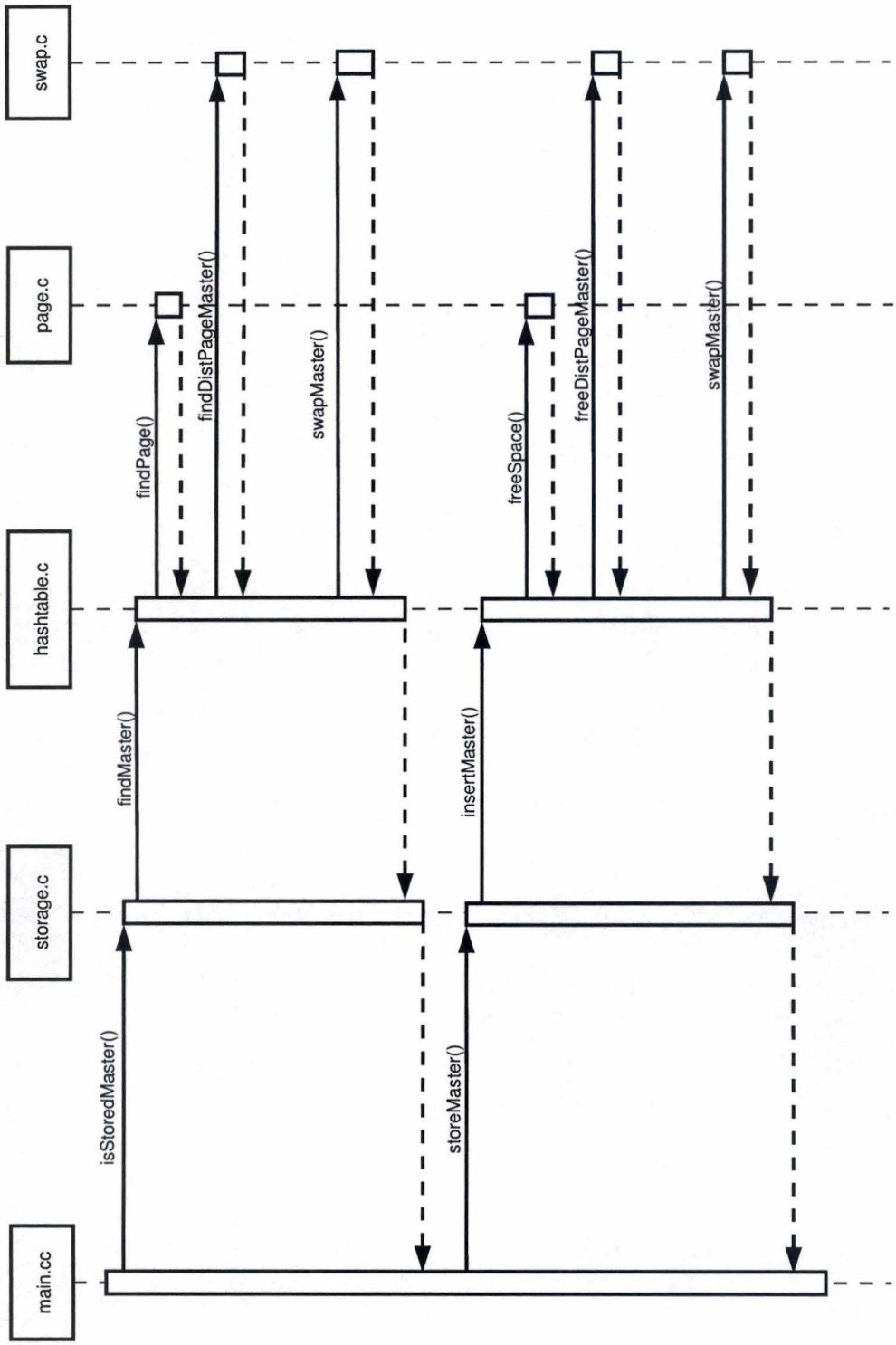

Figure A.1: Sequence diagram for the master in step 1

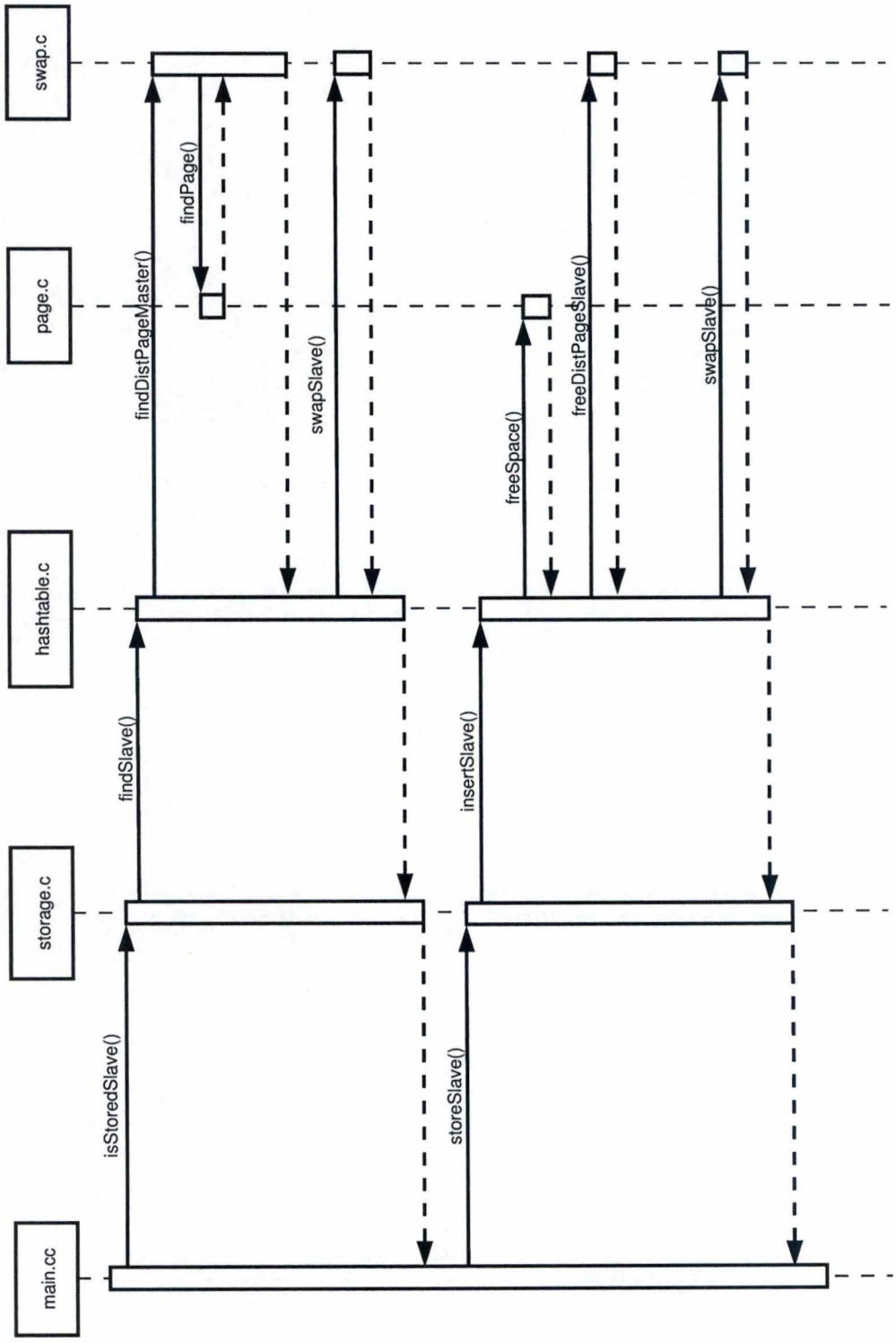

Figure A.2: Sequence diagram for the slave in step 1

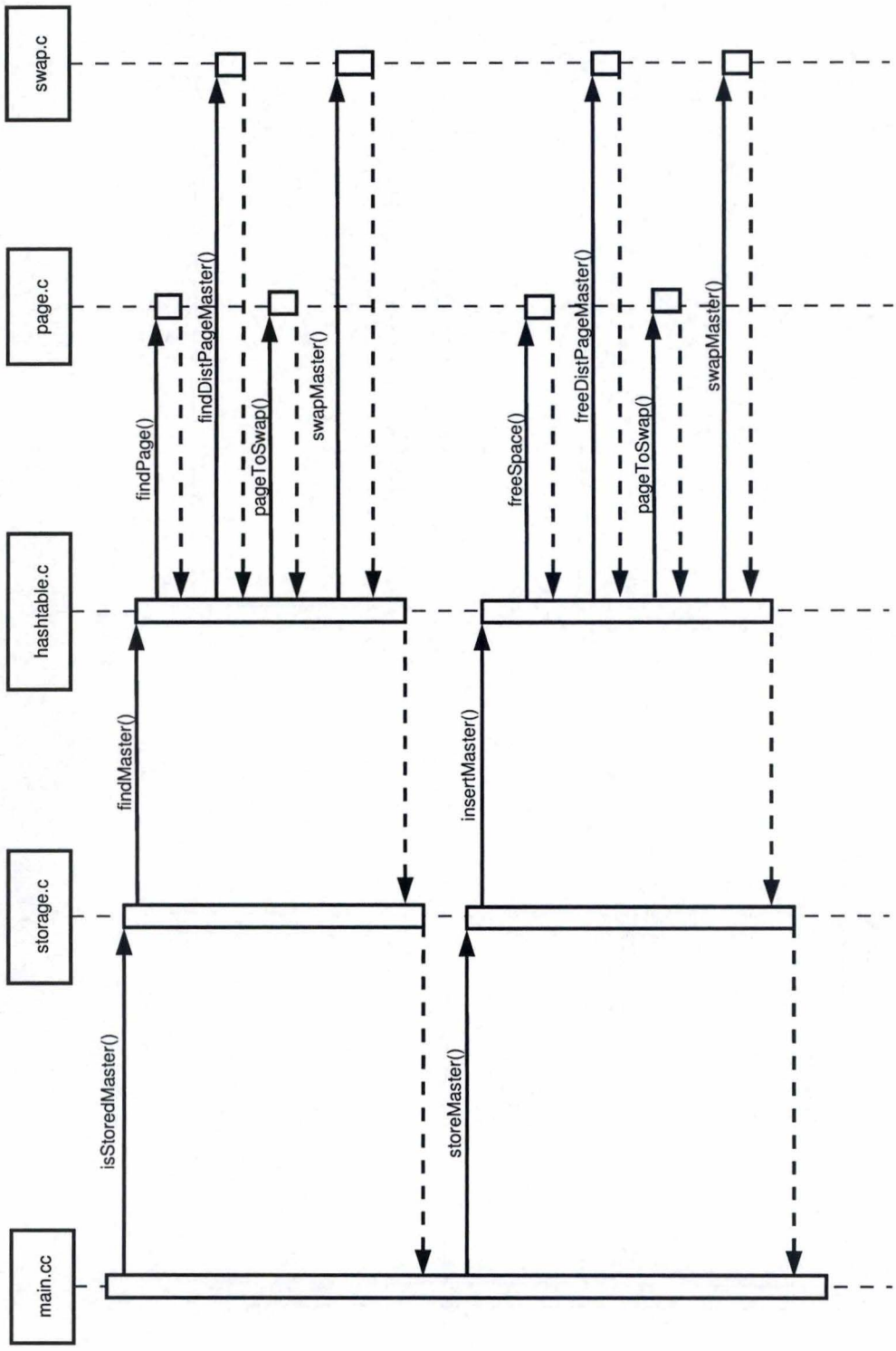

Figure A.3: Sequence diagram for the master in step 2

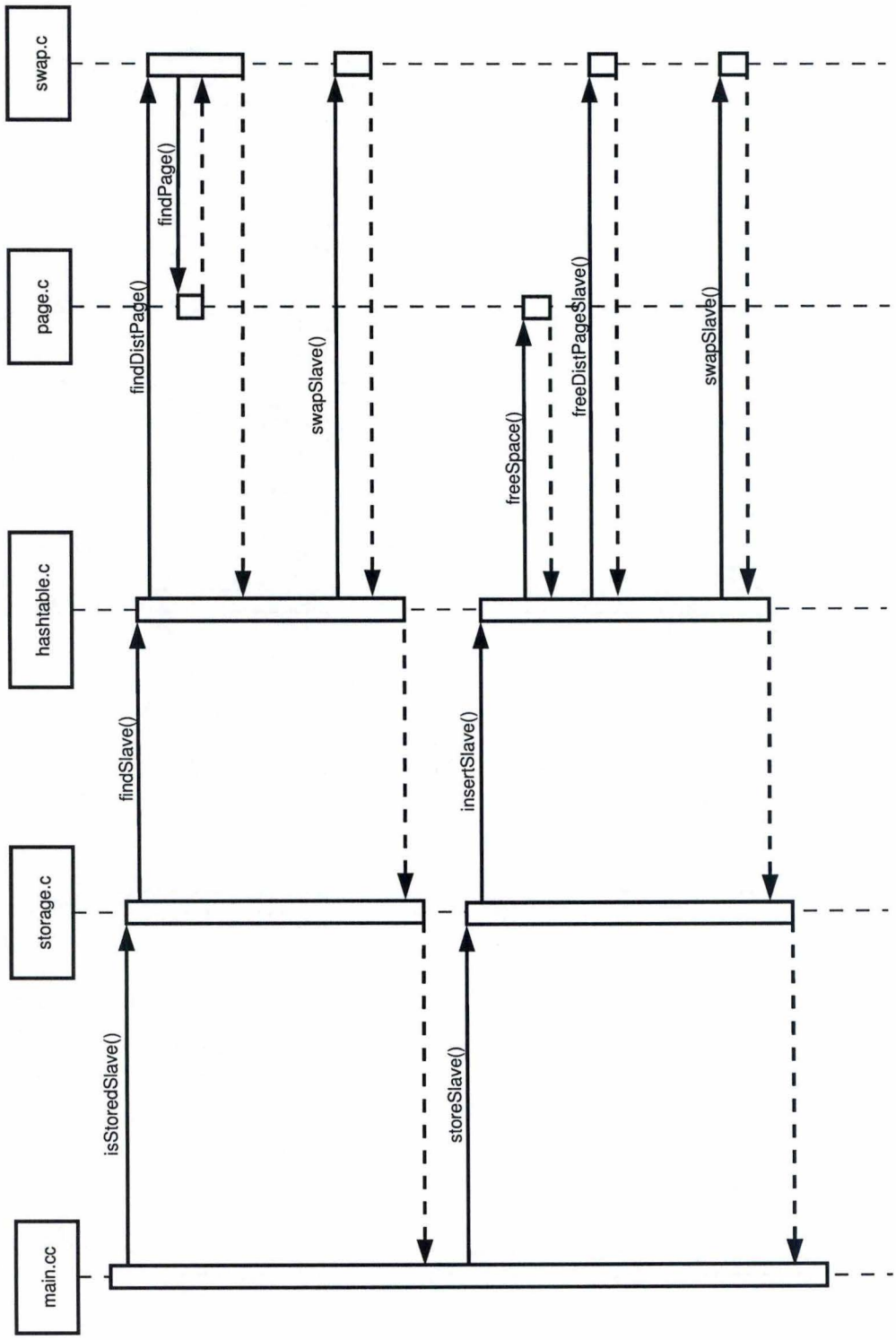

Figure A.4: Sequence diagram for the slave in step 2

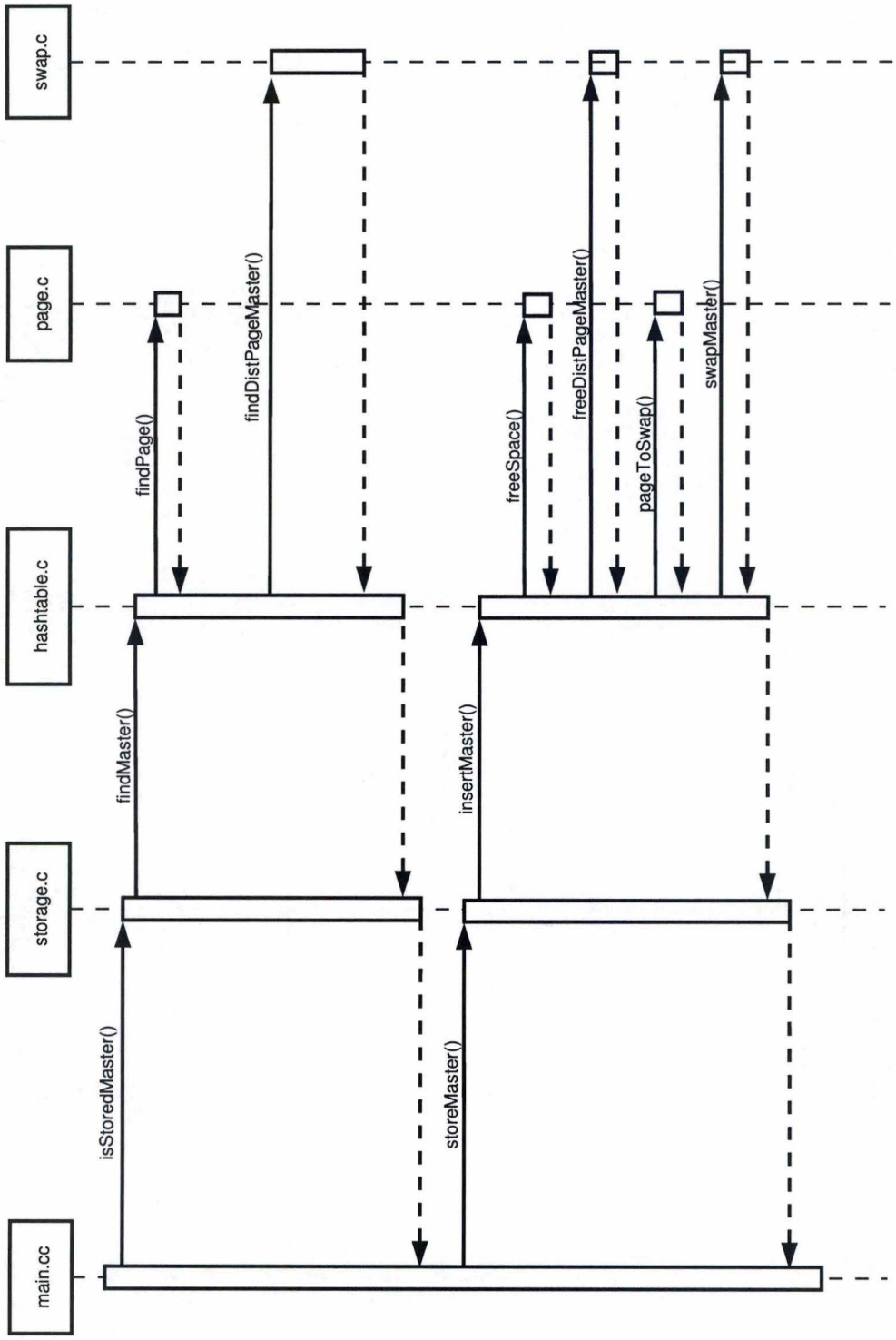

Figure A.5: Sequence diagram for the master in step 3

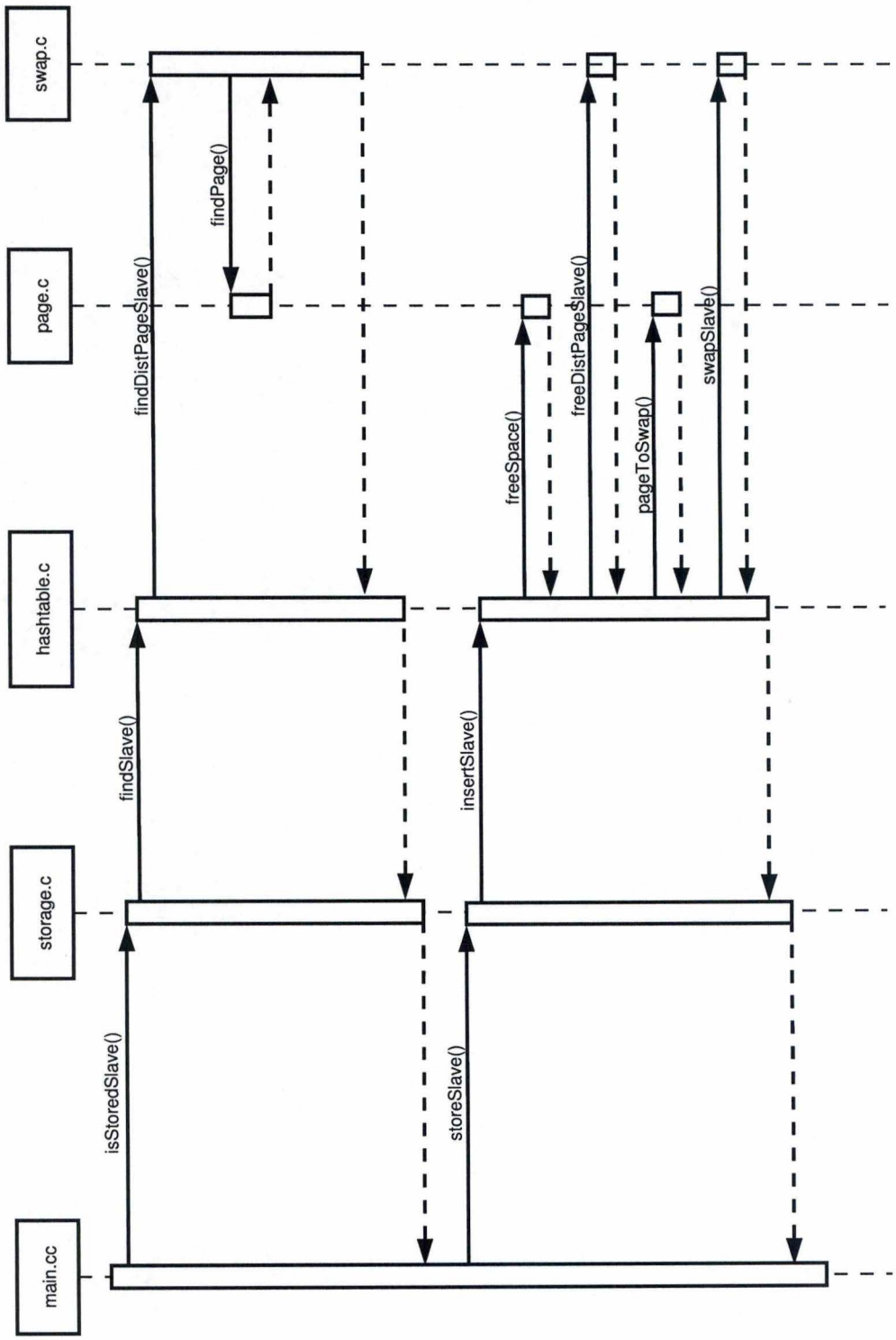

Figure A.6: Sequence diagram for the slave in step 3

Appendix B

MPI bindings

The appendix B contains a detailed description of the MPI functions, MPI constants and MPI communicators used in the prototype. All descriptions are for C programming language. The description comes from [MPIpi].

B.1 Constants

MPI_CHAR: char
MPI_INT: int
MPI_UNSIGNED: unsigned int
MPI_ANY_SOURCE: it allows to accept a message from anyone in a receive operation.

B.2 Communicators

The type of *communicators* are MPI_Comm in C.
MPI_COMM_WORLD: contains all the processes created on each node of the cluster.

B.3 Functions

MPI_init

This function initializes the MPI execution environment

Synopsis

```
#include "mpi.h"
int MPI_Init(int *argc, char ***argv)
```

Input Parameters

argc Pointer to the number of arguments
argv Pointer to the argument vector

Errors

MPI_SUCCESS

No error; MPI routine completed successfully.

MPI_ERR_OTHER

This error class is associated with an error code that indicates that an attempt was made to call MPI_INIT twice. MPI_INIT may only be called once in a program.

MPI_Abort

This function terminates MPI execution environment

Synopsis

```
#include "mpi.h"
int MPI_Abort( MPI_Comm comm, int errorcode )
```

Input Parameters

comm Communicator for tasks to abort
errorcode Error code to return to the calling environment

Notes

It terminates all MPI processes associated with the communicator *comm*. If it is *MPI_COMM_WORLD*, then all the processes created by MPI terminates.

MPI_Comm_size

This function determines the size of the group associated with a communicator

Synopsis

```
#include "mpi.h"
int MPI_Comm_size ( MPI_Comm comm, int *size )
```

Input Parameters

comm Communicator.

Output Parameters

size returns an integer which contains the number of processes in the group of *comm*.

MPI_Comm_rank

This function determines the rank of the calling process in the communicator

Synopsis

```
#include "mpi.h"
int MPI_Comm_rank ( MPI_Comm comm, int *rank )
```

Input Parameters

comm Communicator.

Output Parameters

rank returns an integer which contains the rank of the calling process in the group of *comm*.

MPI_Send

This function performs a basic send

Synopsis

```
#include "mpi.h"
int MPI_Send( void *buf, int count, MPI_Datatype datatype, int dest, int tag,
              MPI_Comm comm)
```

Input Parameters

buf initial address of sending buffer (choice)
count number of elements in sending buffer (positive integer)
datatype datatype of each element of the sending buffer (handle)
dest rank of destination (integer)
tag message tag (integer)
comm Communicator (handle).

Notes

The function may block until the message has been received. It is one of the blocking function used in the prototype.

MPI_Recv

This function performs a basic receive

Synopsis

```
#include "mpi.h"
int MPI_Recv( void *buf, int count, MPI_Datatype datatype, int source, int tag,
              MPI_Comm comm, MPI_Status *status)
```

Input Parameters

count maximum number of elements in receiving buffer (positive integer)
datatype datatype of each element of the receiving buffer (handle)
dest rank of source (integer)
tag message tag (integer)
comm communicator (handle).

Output Parameters

buf initial address of the receiving buffer (choice)
status status object

Notes

The function may block until a message has been received. It is the other blocking function used in the prototype.

MPI_Finalize

This function terminates the MPI execution environment

Synopsis

```
#include "mpi.h"
int MPI_Finalize()
```

Notes

All processes must call this routine before exiting. The number of processes running after this routine has been called is undefined; it is recommended not to perform more than a return *rc* after calling *MPI_Finalize*.

Index