Université de Montréal

Compilation de scénarios dans un environnement d'intégration

par

Dhafer Ben Khedher

Département d'Informatique et de Recherche Opérationnelle
Faculté des Arts et des Sciences

Mémoire présenté à la Faculté des Études Supérieures
en vue de l'obtention du grade de
Maître ès Sciences (M.Sc.)
en Informatique

Avril, 2002

Université de Montréal
Faculté des études supérieures

Ce mémoire intitulé :

Compilation de scénarios dans un environnement d'intégration

présenté par :

Dhafer Ben Khedher

a été évalué par un jury composé des personnes suivantes :

Guy Lapalme
Président-rapporteur

Rachida Dssouli
Directrice de recherche

Julie Vachon
Membre du jury

Mémoire accepté le 28 mai 2002

Résumé

La spécification des systèmes réactifs temps-réel est une tâche lourde, mais permettant la détection d'erreurs et diminuant les coûts. L'automatisation de certaines activités de spécification par l'approche de scénarios développée dans [Sal 01] permet de construire un prototype du système, exprimé sous forme d'un automate temporisé [Alu 94], qui peut être utilisé dans la phase de vérification, de simulation ou encore de test. Ce prototype est synthétisé à partir d'un ensemble de scénarios, où chacun modélise un comportement partiel du système sous forme d'une séquence d'interactions entre le système et l'environnement.

Dans ce projet SCENA, nous mettons en œuvre un environnement pour automatiser la génération de spécifications formelles sous forme d'un automate temporisé à partir des scénarios de comportement du système. Nous définissons les techniques, les structures de données et les algorithmes nécessaires à l'implémentation de l'environnement.

Nous apportons des éléments de réponse aux problèmes d'architecture et de conception par la proposition des solutions orientées objets tout en tenant compte de l'aspect performance. Nous décrivons aussi, les grammaires des langages d'acquisition des entrées du système: les attributs, les variables, les scénarios et les directives. Nous fournissons un exemple d'application complet.

Enfin, nous implémentons l'environnement automatisant la génération de spécification formelle à partir d'un ensemble de scénarios selon les deux types d'intégration : implicite et explicite. L'intégration explicite, au contraire de l'implicite, prend en compte l'ordonnancement des scénarios par l'intermédiaire d'un ensemble de directives.

Mots-clés : système réactif temps-réel, spécification formelle, automate temporisé, scénario, intégration de scénarios, compilation.

Abstract

The specification of the real-time reactive systems is a heavy task, but allowing the detection of errors and a reduction of the development costs. The automation of certain activities in the specification process by the approach of scenarios developed in [Sal 01] permits to build a system prototype expressed in the form of a timed automaton [Alu 94]. This prototype is synthesized from a set of scenarios, where each one represents a partial description of a system behavior. The prototype of the system can be used in other development phases such as verification, simulation and test.

In this project SCENA, we setup an environment to automate formal specification synthesis by integrating real-time scenarios into a timed automaton. We define techniques, data structures and algorithms of the development environment.

We address the problems of architecture and design by means of oriented object solutions while taking into account some performance aspects. We also describe grammars of acquisition languages of the system inputs: attributes, variables, scenarios and directives.

Finally, we implement the environment automating the synthesis of formal specification from a set of scenarios according to two types of integration: implicit and explicit. Explicit integration, contrary to the implicit one, takes into account an order for certain scenarios execution through a set of directives.

Keywords: real-time reactive system, formal specification, timed automata, scenario, scenarios integration, compilation.

Remerciements

Je tiens à exprimer ma profonde reconnaissance à Madame *Rachida Dssouli* pour m'avoir encadré tout au long de ce projet, pour les précieuses suggestions qu'elle n'a cessé de me prodiguer, pour la documentation qu'elle m'a fournie et pour la patience et la disponibilité dont elle a fait preuve à mon égard.

Je me tiens à cœur de remercier plus profondément mes chers parents qui ont fait de moi ce que je suis aujourd'hui. Que dieu, le tout puissant, leur donne santé et longue vie.

Je dédie ce mémoire à ma femme *Ibtissem* qui n'a cessé de m'encourager tout au long de mes études.

Mes remerciements aussi à mes frères, mes sœurs, mes beaux-parents et tous les membres de ma famille, qu'ils trouvent ici la reconnaissance des soutiens qu'ils m'ont toujours apportés.

Finalement, je remercie mes coéquipiers *Aziz Salah* et *Yassir Bellout* pour leur coopération et leurs critiques constructives sur notre travail.

Table de Matières

Liste des figures

Liste des abréviations

GDI Graphe de Directives d'Intégration

TA Timed Automata

UML Unified Modeling Language

Introduction

Les systèmes réactifs temps-réel sont des systèmes critiques et présentent des difficultés dans la phase de spécification et de conception. La tâche de spécification de ces systèmes est lourde et elle oblige souvent les concepteurs à passer directement des spécifications informelles à l'implémentation, réduisant ainsi la détection d'erreurs et augmentant les coûts.

L'automatisation de certaines activités de spécification par l'approche de scénarios développée dans [Sal 01] permet de construire un prototype du système qui peut être utilisé dans la phase de vérification, de simulation ou de test. Ce prototype est synthétisé à partir d'un ensemble de scénarios exprimé sous forme d'un automate temporisé [Alu 94]. Chaque scénario modélise un comportement partiel du système sous forme d'une séquence d'interactions entre le système et l'environnement.

Ce travail s'insère dans le cadre d'un projet qui vise à mettre en œuvre un environnement pour automatiser la génération de spécification formelle sous forme d'un automate temporisé à partir des scénarios de comportement du système. Ce projet, intitulé SCENA, est réalisé dans le cadre d'un contrat de recherche externe entre France Télécom et le groupe téléinformatique de l'Université de Montréal.

L'environnement SCENA fournit un processus de spécification formelle à partir des scénarios. Il permet une construction incrémentale d'un modèle de spécification d'un système par l'ajout des scénarios au prototype courant. La spécification est générée sous forme d'automate temporisé. L'approche consiste à transformer les scénarios en actions-règles, puis, construire les places et les transitions de l'automate résultat. Les actions-règles sont extraites des actions des scénarios. Elles comportent

toute l'information sur l'état du système et sa condition d'activation avant et après l'exécution. L'environnement SCENA offre deux types d'intégration: implicite et explicite. L'intégration implicite reproduit tous les comportements des scénarios. Ce type d'intégration ne permet aucun contrôle sur la façon dont les scénarios seront intégrés dans la spécification générée. L'intégration explicite de scénarios, au contraire de l'implicite, utilise des directives qui vont contraindre le système par des contraintes d'ordre d'exécution.

Dans ce travail nous contribuons à la définition de l'architecture du système, la modélisation du système via la conception orientée objet et la définition des structures de données du système. Nous contribuons aussi au développement des algorithmes nécessaires à la manipulation des différents éléments de données du système.

Ensuite, nous raffinons l'architecture du système en terme de fonctionnalités et de modules. Puis, nous développons les algorithmes dérivés des formules présentées dans [Sal 01] nécessaires à la génération des actions-règles. Ainsi, nous implémentons ces algorithmes dans l'environnement SCENA selon les deux types d'intégration. Nous développons l'interface de l'outil et nous intégrons tous les modules du système. Nous écrivons aussi, les grammaires des langages d'acquisition des entrées du système et nous implémentons les compilateurs correspondants. Les algorithmes de construction des places et des transitions de l'automate sont développés et implémentés par une autre équipe.

1.1 Les scénarios

De nos jours, l'utilisation des scénarios se manifeste dans plusieurs domaines de génie logiciel. Les scénarios ont été utilisés dans diverses activités du cycle de vie de développement des systèmes informatiques notamment dans l'acquisition des besoins [Hol 82], la génération des spécifications et des prototypes [Hsi 94] et la construction des interfaces usagers graphiques [Elk 00].

Il n'y a pas une définition universelle pour le terme scénario. Le dénominateur commun de plusieurs définitions décrit un scénario comme une séquence

d'interactions usager-système nécessaires pour accomplir une tâche que l'usager désire réaliser [Lus 97]. Un scénario est donc une description partielle ou complète d'une tâche. Une autre définition considère un scénario comme une description partielle d'interactions entre l'environnement et le système [Des 98]. C'est cette dernière définition que nous avons adoptée dans notre projet. En supposant que les interactions sont abstraites et prennent la forme d'actions observables.

L'approche scénarios [Sal 01] utilisée dans l'environnement SCENA se distingue des autres approches par la considération des variables ainsi que le temps comme information véhiculée dans les scénarios, et aussi par sa méthode d'intégration incrémentale des scénarios. La modélisation du temps dans notre approche est formulée par des contraintes temporelles sur les horloges du système. Nous pouvons comparer les horloges comme les temporisateurs dans la plupart des systèmes à temps réels. La prise en compte des variables est formulée par des contraintes sur les variables du système.

1.1.1 Représentation d'un scénario

Un scénario est une description partielle du comportement du système à spécifier. Chaque scénario *sc*, est décrit par une liste de nœuds $sc = [N_1, \ldots, N_{n+1}]$ où chaque nœud N_i (pour $1 \leq i \leq n$) est à son tour décrit par une liste d'actions $a_i = [a_{i1}, \ldots, a_{iki}]$. Nous représentons un scénario sous forme d'un arbre (Figure 1.1). Les arcs de l'arbre représentent les actions du scénario. Chaque action décrit les changements de l'état du système après son exécution. Une action admet la structure suivante :

- Une contrainte d'horloges appelée l'invariant de l'action,
- Une étiquette de l'action,
- Une contrainte sur les variables,
- Une contrainte d'horloges appelée la garde de l'action,
- Une affectation de variables,
- Une affectation d'horloges.

Nous distinguons deux catégories de nœuds, les nœuds principaux et les nœuds secondaires. Les nœuds principaux sont les nœuds internes de l'arbre. Ils forment l'axe principal de l'arbre représentant le chemin normal d'exécution du scénario. Les nœuds secondaires sont par contre les feuilles de l'arbre et représentent les exceptions d'exécution du scénario. Dans la figure 1.1, un nœud N_i est composé par une action principale a_{i1} et un ensemble d'actions secondaires a_{ij} tel que $1<j\leq k_i$. Ces derniers sont des alternatives de l'action principale a_{i1}. Ces alternatives peuvent représenter toute interruption d'exécution suite à diverses raisons.

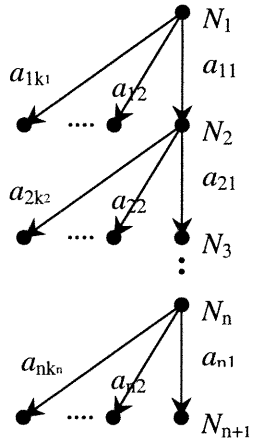


Figure 1.1 Représentation d'un scénario

1.2 Algorithmes d'intégration

L'intégration de scénarios est une phase cruciale pour spécifier tout le système dans sa globalité. Pour accomplir cette tâche, il faut donner un formalisme aux scénarios. Plusieurs approches ont été explorées. Certaines approches représentent les scénarios sous forme de graphes (réseau de pétri) [Dan 97], d'autres utilisent les arbres et les techniques de la théorie de langage [Hsi 94]. Une autre approche consiste à modéliser les scénarios par des automates temporisés [Som 96]. Cette dernière approche peut traiter non seulement la phase d'acquisition des besoins dans le cycle de vie du système mais peut aussi accompagner les concepteurs durant des

phases plus avancées des activités de développement. C'est cette dernière approche d'intégration qui a été choisie pour l'implémentation de l'environnement SCENA. Elle permet une spécification de manière incrémentale en intégrant les scénarios au fur et à mesure grâce à l'insensibilité de l'intégration à l'ordre d'ajout des scénarios.

1.3 Organisation du document

Ce mémoire est structuré comme suit. Le chapitre 2 présente l'environnement SCENA ainsi que ses différentes composantes. Le chapitre 3 présente les automates temporisés ainsi que leurs formalismes. Le chapitre 4 présente le formalisme de l'approche utilisée pour la compilation de scénarios en automates temporisés. Le chapitre 5 décrit l'architecture générale en terme de fonctionnalités et de modules adoptée pour le système. Le chapitre 6 décrit le module d'horloges et le module de variables. Nous détaillons les structures de données et les algorithmes développés pour manipuler les différents types de données. Dans le chapitre 7, nous décrivons les structures de données et les algorithmes développés pour générer les actions-règles. Le chapitre 8 décrit l'interface et les fonctionnalités de l'application ainsi que les grammaires développées pour l'acquisition des entrées du système. Nous fournissons un exemple d'application complet traitant le cas d'un commutateur téléphonique. Enfin, nous proposons des améliorations à apporter à notre outil en présentant les fonctionnalités manquantes. L'annexe présente les grammaires des différents langages d'acquisition des entrées du système.

L'environnement SCENA

2.1 Introduction

De nos jours, avec le développement technologique et l'augmentation de la complexité des systèmes informatiques, il est devenu plus approprié de partitionner le cycle de vie du système et d'automatiser chaque partie. Les différents fragments deviennent plus contrôlables et facilement testables pour repérer les anomalies, y remédier et réintégrer le tout dans un seul ensemble. L'approche scénarios permet de synthétiser la spécification d'un système en intégrant plusieurs spécifications partielles qui sont les scénarios. L'environnement SCENA utilisant l'approche scénarios permet une construction incrémentale d'un modèle de spécification d'un système par l'ajout des scénarios au prototype courant. La spécification est générée sous forme d'automate temporisé.

L'environnement SCENA offre deux modes d'intégration: le mode implicite et le mode explicite. Le mode d'intégration explicite ne met pas en cause le mode d'intégration implicite et la coexistence de ces deux modes d'intégration dans l'environnement SCENA est tout à fait compatible. L'intégration implicite dans l'environnement SCENA se base sur un algorithme qui permet de générer une spécification qui comporte toutes les interactions possibles entre les scénarios à intégrer. Ainsi, il reproduit tous leurs comportements mais il ne permet aucun contrôle sur la façon dont les scénarios fournis seront intégrés dans la spécification générée. L'intégration explicite, quant à elle, permet d'imposer certaines contraintes sur l'enchaînement des scénarios. Les contraintes sont de type séquentiel ou alternatif, par exemple, exécuter un scénario après un autre ou alternativement.

2.2 Processus de spécification dans l'environnement SCENA

L'outil SCENA propose de générer une spécification formelle d'un système à partir d'un ensemble de scénarios. Les scénarios fournissent la description du comportement du système tandis que la description du domaine d'application permet la spécification des données. Le processus de spécification formelle à partir des scénarios dans l'environnement SCENA est composé de six étapes principales qui sont les suivantes :

1. Acquisition du domaine d'application,
2. Acquisition des scénarios et des directives,
3. Extraction des actions-règles,
4. Compilation des actions-règles en automate temporisé,
5. Simulation,
6. Vérification de la spécification.

Dans la suite, nous allons détailler les différentes étapes du processus de spécification dans l'environnement SCENA et les liens entre elles (Figure 2.1).

2.2.1 Acquisition du domaine d'application

La description du domaine d'application est une modélisation statique du système. Elle inclut un ensemble d'horloges, un ensemble d'étiquettes d'événements et un ensemble de variables discrètes à domaines finis. Les variables décrivent les propriétés non temporelles du système. Les étiquettes représentent les événements et permettent la synchronisation avec l'environnement comme dans les systèmes de transitions étiquetées.

Nous définissons la caractérisation de l'état basé sur la description du domaine d'application. Les occurrences des événements permettent la mise à jour de l'état du système. Les événements peuvent être de deux types différents, la réception ou l'envoi des messages qui représentent les actions observées.

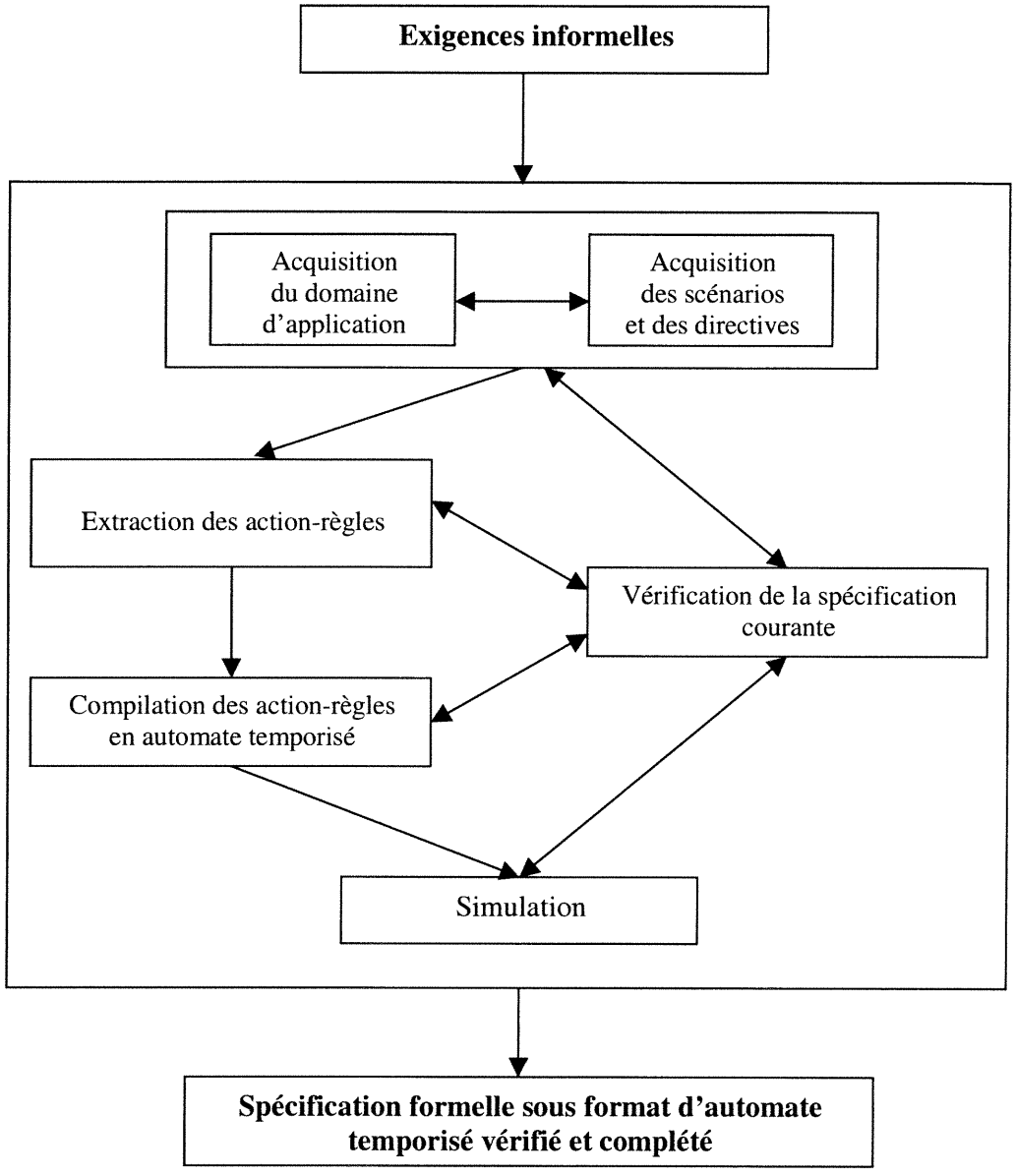


Figure 2.1 Vue générale sur le processus de spécification dans l'environnement SCENA

2.2.2 Acquisition des scénarios et des directives

Dans l'approche SCENA, le comportement du système est exprimé par des scénarios dont l'ordonnancement peut être contraint par des directives d'intégration explicite. Les scénarios modélisent les différents fragments d'exécution du système tandis que les directives décrivent l'ordre d'exécution des scénarios.

2.2.2.1 Acquisition des scénarios

Le comportement du système est décrit par un ensemble de scénarios. Un scénario permet de décrire, durant son exécution, le changement de l'état du système. Il est une suite de spécifications décrivant un ensemble de traces d'exécution. L'extraction des scénarios se fait à partir des exigences informelles. La description des scénarios est formulée dans un langage que nous avons défini pour exprimer le comportement du système. Ainsi, les scénarios sont passés au système par l'intermédiaire des fichiers textes qui sont compilés par un compilateur qui fait l'analyse syntaxique et lexicale et exécute les actions sémantiques qui remplissent les structures de données des scénarios.

2.2.2.2 Acquisition des directives

Dans la méthode d'intégration explicite, nous utilisons des directives pour contraindre le système par des contraintes d'ordre d'exécution. Par exemple, exécuter un scénario avant un autre ou exécuter alternativement deux ou plusieurs scénarios. Ceci est très utile dans le cas où nous voudrions produire la spécification d'une composante existante pour laquelle nous connaissons la façon dont les scénarios sont séquencés. Un compilateur de directives interprète les directives écrites dans un fichier et remplit les structures de données correspondantes.

2.2.3 Extraction des actions-règles

Cette étape permet d'extraire les actions-règles à partir des actions des scénarios et des contextes des nœuds. Le contexte est un couple de contraintes attaché à un nœud d'un scénario. Il est composé d'une contrainte temporelle et d'une contrainte sur variables. Chaque action-règle est indépendante du contexte et comporte toute

l'information sur l'état du système et sa condition d'activation avant et après l'exécution. A chaque génération d'une action-règle nous vérifions si elle est nouvelle ou redondante. Dans le cas de redondance nous conservons seulement l'information sur l'action de provenance.

2.2.4 Compilation des actions-règles en automate temporisé

Cette étape génère un automate temporisé à partir des actions-règles obtenues dans l'étape précédente. La construction de l'automate nécessite le partitionnement des places issues des actions-règles et d'ajouter les transitions entre les places de l'automate. Le partitionnement des places utilise un algorithme qui assure un partitionnement minimal. L'aspect de minimalité s'exprime par le fait que le partitionnement d'une place n'est effectué que s'il est nécessaire.

2.2.5 Simulation

La simulation du fonctionnement du système est l'étape finale de spécification de l'approche SCENA. Nous modélisons l'automate temporisé pour qu'il soit exécuté par un simulateur. Le principe de simulation est simple, à partir d'une transition et d'une condition initiale, il faut déterminer à chaque fois la prochaine transition à franchir. Cette étape nous permet de détecter les anomalies de la spécification.

2.2.6 Vérification de la spécification

Les scénarios, les directives et les actions-règles peuvent contenir des contradictions ou des oublis qui peuvent se propager jusqu'aux spécifications générées sous format d'automate temporisé. Cette activité vise à éliminer les incohérences et retracer leur cause afin de faciliter leur correction. Elle est répartie en plusieurs étapes. Nous commençons d'abord par découvrir les incohérences liées aux scénarios en tant qu'entités isolées. Nous nous assurons aussi que chaque scénario forme bien une entité exécutable. Ensuite, nous vérifions la cohérence de tous les scénarios en les disposant selon les directives. Puis, nous vérifions la cohérence des actions-règles en tant qu'entités indépendantes. Enfin, nous vérifions que l'automate de la spécification courante est cohérent. Cette étape s'assure donc

de la cohérence de l'ensemble de tous les scénarios et leurs interactions les uns par rapport aux autres.

La vérification de la spécification s'étend à la simulation pour vérifier les cas de blocage. Nous visons dans cette étape la détection de certaines formes d'erreurs comme, par exemple, le cas de réception d'un message que le système ne peut pas exécuter puisqu'il n'a aucune transition étiquetée par la réception de ce message à partir de son état actuel.

3.3 Conclusion

Dans ce chapitre, nous avons décrit l'environnement SCENA en globalité. Ce dernier intègre toutes les phases de spécification formelle d'un système temps-réel par l'approche d'intégration de scénarios. Le résultat d'une telle spécification est un automate temporisé. Nous avons détaillé les différentes étapes du processus de spécification dans l'environnement SCENA et les liens entre elles. Dans le chapitre suivant, nous présentons la théorie des automates temporisés ainsi que le formalisme de contraintes temporelles et d'affectations d'horloges nécessaires pour la manipulation des horloges dans l'environnement SCENA.

Les automates temporisés

3.1 Introduction

Dans notre approche, le résultat d'une spécification d'un système réactif temps-réel est un prototype synthétisé, exprimé sous-forme d'un automate temporisé [Alu 94]. Cet automate est facilement exploitable dans les autres phases de cycle de vie de développement d'un système tels que la simulation et la détection d'erreurs. Les automates temporisés permettent aussi, par leur formalisme, la modélisation du contrôle temporisé selon le modèle dense du temps, ce qui est primordiale pour les systèmes temps-réel. Dans ce chapitre, nous présentons le formalisme des systèmes de transitions et des automates temporisés. Nous décrivons aussi le formalisme de contraintes temporelles et d'affectation d'horloges nécessaires pour la manipulation des horloges dans l'environnement SCENA.

3.2 Modélisation des automates temporisés

Un automate temporisé ''TA'' pour Timed Automata [Alu 94] est un système de transitions étiquetées étendu par un ensemble de variables réelles appelées horloges. Les horloges sont des temporisateurs qui ont des valeurs positives et croissent uniformément avec le temps. Dans les sections suivantes, nous allons présenter le modèle des automates temporisés. Nous décrivons aussi le formalisme de contraintes temporelles et d'affectation d'horloges.

3.2.1 Les systèmes de transitions

Nous modélisons un système discret par un graphe composé d'états et de transitions. Les transitions sont étiquetées par des symboles d'événements. Un système de transitions S est un tuple $(Q, Q^\circ, \Sigma, \rightarrow)$ où :

- Q est un ensemble d'états,
- $Q^\circ \subseteq Q$ est un ensemble d'états initiaux,
- Σ est un ensemble d'étiquettes (ou d'événements),
- $\rightarrow \subseteq Q \times \Sigma \times Q$ est un ensemble de transitions.

Le système débute avec un état initial, et si $q \xrightarrow{a} q'$ alors le système peut transiter de l'état q vers l'état q' à l'événement a.

3.2.2 Les systèmes de transitions avec les contraintes temporelles

Pour exprimer le comportement du système avec des contraintes temporelles, nous considérons un graphe fini augmenté par un ensemble de variables réelles, appelées horloges, dont les valeurs croissent uniformément avec le passage du temps. Le passage à travers les arcs du graphe est instantané au contraire aux sommets ou le temps peut s'écouler. Une horloge peut être mise à zéro par un arc du graphe. Avec chaque arc nous associons une contrainte portant sur les horloges. Un arc ne peut être franchi que seulement si les valeurs courantes des horloges satisfont la condition associée à l'arc. Avec chaque sommet nous associons une contrainte portant sur les horloges appelée son invariant, et le système peut rester dans le sommet tant que la contrainte associée au sommet est vérifiée par les valeurs des horloges.

3.2.2.1 Les contraintes temporelles

Pour définir les automates temporisés formellement, nous devons définir les contraintes temporelles qui sont admissibles comme invariants et conditions sur les arcs. Soit H l'ensemble des horloges et $\Phi(H)$ l'ensemble des contraintes temporelles. Une contrainte temporelle ϕ est définie par la grammaire suivante:
$\phi := x \# c \mid \phi_1 \wedge \phi_2 \mid$ Vrai, où $\# \in \{\leq, <, =, \geq, >\}$, x est une horloge dans H et c est une constante entière.

Pour des raisons de commodité nous allons ajouter le type de contrainte $x\text{-}y \# c$ qui peut être exprimé par les autres contraintes. Remarquons que nous n'utilisons

pas l'opérateur $\vee$, sous forme de $\phi_1 \vee \phi_2$ car celui-ci revient à diviser l'arc en deux de façon à ce que chacun satisfasse une contrainte.

3.2.2.2 Interprétation d'horloges

Une interprétation d'horloges θ est une application qui associe à chaque horloge de H une valeur réelle positive. $\Theta(H)$ représente l'ensemble des interprétations d'horloges de H et $\phi(\theta)$ représente la valeur de vérité de la contrainte temporelle ϕ évaluée en θ.

3.2.2.3 Affectation d'horloges

Une affectation d'horloge λ permet la remise à zéro de certaines ou de toutes les horloges de H. Par exemple, $\lambda = \{x := 0\}$ où $x \in H$. Soit $\theta \in \Theta(H)$, $\theta[\lambda]$ désigne l'affectation d'horloges définie par $\theta[\lambda](x) = 0$ pour toute horloge x évoqué dans λ et $\theta[\lambda](y) = \theta(y)$ pour les autres horloges y de H. $\phi[\lambda]$ représente la contrainte satisfaite par toutes les interprétations $\theta[\lambda]$ tel que θ satisfait ϕ.

3.2.3 Syntaxe et sémantique d'un automate temporisé

Un automate temporisé A est un tuplet (L, L°, Σ, H, I, E) tels que :

- L est un ensemble fini de sommets,
- $L \subseteq L^\circ$ est un ensemble initial de sommets,
- Σ est un ensemble fini d'étiquettes,
- H est un ensemble fini d'horloges,
- I est une association qui relie à chaque sommet s un ensemble de contraintes temporelles dans $\Phi(H)$,
- $E \subseteq L \times \Sigma \times 2^H \times \Phi(H) \times L$ est un ensemble d'arcs.

La sémantique de l'automate temporisé A est définie en lui associant un système de transitions S_A. Un état de S_A est une paire (s,v) tels que s est un sommet de A, v est une interprétation d'horloges de H et v satisfait l'invariant $I(s)$. L'ensemble de tous les états de A est noté Q_A et l'ensemble de tous les étiquettes de A est noté Σ_A.

Un état (s,v) est un état initial, si s est un sommet initial de A et $v(x) = 0$ pour toute horloge $x \in H$. Il y a deux types de transitions dans un S_A:

- Changement d'état dû à l'écoulement du temps : pour un état (s,v) et une variable réelle du temps en incrémentation $\delta \geq 0$, $(s,v) \xrightarrow{\delta} (s, v+\delta)$ si pour tout $\delta' / 0 \leq \delta' \leq \delta$, $v+\delta'$ satisfaisant l'invariant $I(s)$.

- Changement d'état dû à un sommet-arc : pour un état (s,v) et un arc $\langle s, a, \varphi, \lambda, s' \rangle$ tel que v satisfait ϕ, $(s,v) \xrightarrow{a} (s', v[\lambda := 0])$.

Notons que la notion de traces pour représenter une suite d'exécution dans les systèmes de transitions discrets a été remplacé par la notion de pas tel que définie dans [Alu 93] et [Yov 93].

3.2.4 Analyse de l'accessibilité

Un sommet s d'un automate temporisé A est dit accessible si quelques états Q' qui ont comme sommet s sont accessibles par le système de transition S_A. Le problème de l'accessibilité est de déterminer si tous les sommets cibles sont accessibles ou non. L'entrée du problème de l'accessibilité est l'automate temporisé A et l'ensemble $L_F \subseteq L$ des sommets cibles. L'analyse de la sécurité dans les systèmes temps-réel peut être formulée comme un problème d'accessibilité dans les automates temporisés.

Définition : Soit A un TA et S_A son système de transitions, et soit e et e' deux états de S_A, nous disons que e' est accessible en un pas à partir de e s'il existe un réel $d \geq 0$ tel que : $e \xrightarrow{d} e + d$ et $e + d \xrightarrow{m} e'$ avec $m \in \sum_A \cup \{0\}$.

3.2.5 Les systèmes de transitions à abstraction du temps

Un système de transitions S_A d'un automate temporisé A, a infiniment plusieurs états et infiniment plusieurs symboles. En première étape, nous définissons un autre système de transitions, appelé système de transitions à abstraction du temps noté U_A,

dont les transitions sont étiquetées seulement par des symboles dans $\sum_A$ en cachant les étiquettes exprimant l'écoulement du temps. L'ensemble d'états initiaux de U_A est égal à l'ensemble d'états initial de S_A et l'ensemble d'étiquettes de U_A est le même que l'ensemble $\sum$ de A. La relation de transitions de U_A est la relation notée '$\Rightarrow$' définie comme suit:

Soit q et q' deux états et a une étiquette, $q \overset{a}{\Rightarrow} q$' ssi il existe un état q'' et un réel δ tel que : $q \xrightarrow{\delta} q'' \xrightarrow{a} q$' maintient dans le système de transitions S_A.

Pour résoudre le problème d'accessibilité dans les automates temporisés, nous pouvons considérer les systèmes de transitions à abstraction de temps U_A au lieu de S_A.

3.2.6 Région d'équivalence

Pour chaque réel δ, fr(δ) signifie la partie fractionnelle de δ, et $\lfloor\delta\rfloor$ signifie la partie intégrale de δ; c'est à dire $\delta = \lfloor\delta\rfloor + fr(\delta)$. Nous définissons une relation d'équivalence dans l'espace des états d'un automate qui égalise deux états avec le même sommet si les valeurs de toutes les horloges sont égaux dans la partie intégrale et dans l'ordre dans la partie fractionnelle. La partie intégrale est nécessaire pour déterminer si une contrainte temporelle est vraie ou pas, tandis que la partie fractionnelle est nécessaire pour décider quelle contrainte temporelle changera la première sa partie intégrale. La relation d'équivalence notée $\cong$, appelée région d'équivalence, est définie sur l'ensemble de toutes les interprétations d'horloges de H.

3.2.7 Automate de régions

La relation de région d'équivalence sur les interprétations d'horloges est prolongée à la relation d'équivalence sur l'espace des états par nécessité des états équivalents pour avoir des sommets et des interprétations d'horloges identiques:

$$(s, v) \cong (s', v') \text{ ssi } s = s' \text{ et } v \cong v'$$

Le système de transitions d'un automate temporisé A qui respecte la région d'équivalence est appelé automate de régions de A. Dans ce cas, le problème d'accessibilité peut être résolu linéairement avec le nombre de sommets et exponentiellement avec le nombre d'horloges.

3.2.8 Zone d'horloges

Une zone d'horloges ϕ est un ensemble d'interprétations d'horloges décrit par la conjonction d'un ensemble de contraintes temporelles élémentaires. Une contrainte temporelle élémentaire est constituée d'une seule horloge ou de la différence de deux horloges.

3.2.9 Matrices de différence de bornes

Nous pouvons utiliser les matrices pour représenter efficacement les zones d'horloges. Supposant un automate temporisé qui a k horloges et soit une horloge fictive nommée **0** dont la valeur est toujours zéro, alors une zone d'horloges est représentée par une matrice de dimension $(k+1) \times (k+1)$. Chaque entrée de la matrice représente la borne supérieure de la différence entre deux horloges. Nous utilisons la constante ∞ dans le cas d'absence de bornes.

3.3 Conclusion

Dans ce chapitre, nous avons présenté le modèle d'automates temporisés qui permet la modélisation du contrôle temporisé selon le modèle dense du temps. Nous avons décrit aussi, le problème de l'accessibilité dans les automates temporisés qui est relié au sujet de l'analyse de la sécurité dans les systèmes temps-réel. Dans le chapitre suivant, nous allons nous baser sur ce modèle pour représenter la théorie de compilation de scénarios en automates temporisés. Nous décrivons aussi le formalisme de contraintes sur variables et d'affectations de variables nécessaires pour la manipulation des variables dans l'environnement SCENA. Ensuite, nous présentons le formalisme des scénarios et les formules récurrentes pour la génération des actions-règles. Enfin, nous présentons l'approche d'intégration explicite de scénarios en décrivant le formalisme des directives.

Compilation de scénarios en automates temporisés

4.1 Introduction

Ce chapitre présente le fondement théorique de la compilation de scénarios en automates temporisés [Sal 01]. Dans cette approche, la méthode d'intégration utilisée est insensible à l'ordre d'ajout des scénarios. Cela permet la spécification du système de manière incrémentale par ajout de scénarios au prototype courant. Nous distinguons deux modes d'intégration : le mode d'intégration implicite qui n'utilise aucune directive sur l'ordre d'exécution des scénarios et le mode explicite qui impose certaines contraintes sur l'enchaînement des scénarios par l'intermédiaire d'un ensemble de directives. Dans le présent chapitre, nous présentons les formalismes de contraintes sur variables et d'affectations de variables. Ensuite, nous détaillons la représentation formelle de scénarios et nous décrivons les formules récurrentes de génération des contextes et des actions-règles. Enfin, nous détaillons le principe d'intégration explicite de scénarios.

4.2 Spécification d'un système réactif temps-réel

Un système réactif temps-réel est un processus qui communique avec son environnement selon des contraintes temporelles strictes. Ce processus peut être constitué de plusieurs composantes. Nous nous intéressons seulement aux processus séquentiels. Le comportement d'une composante est décrit par un ensemble de scénarios qui constituent une spécification de son comportement. Dans le premier chapitre, nous avons vu qu'un scénario est composé d'un ensemble d'actions. Chaque action indique comment l'état de la composante change après son exécution.

Par la suite, tous les scénarios sont intégrés en un TA modélisant le système réactif temps-réel.

La description de la spécification du comportement du processus nécessite d'abord la déclaration des attributs, des variables avec leurs domaines, des étiquettes, ainsi que des horloges. Soit P un processus, tel que P = (S, H, V, *Act*) où

- S représente un ensemble de scénarios,
- H représente un ensemble d'horloges,
- V représente un ensemble de variables discrètes,
- *Act* représente un ensemble d'étiquettes pour l'interaction avec d'autres processus de l'environnement.

Dans la suite, nous allons présenter les notions de contrainte sur variables et d'affectation de variables discrètes.

4.2.1 Contrainte sur variables et affectation de variables

Soit $Dom(v)$ un ensemble fini et discret qui représente le domaine d'une variable $v \in V$. Nous nous limitons aux variables discrètes à domaines finis. Les contraintes sur les variables sont définies par la syntaxe suivante : $\psi := v \# c \mid \psi \wedge \psi \mid \neg\psi \mid$ Vrai, où v est une variable et c est une constante dans $Dom(v)$. Si v est une variable à valeurs entières nous pouvons considérer que $\# \in \{\leq, <, =, >, \geq\}$, si non $\# \in \{=, \neq\}$. $\Psi(V)$ représente l'ensemble des contraintes sur les variables de V respectant cette syntaxe. Nous définissons une interprétation de variables ω par une application qui associe à chaque variable $v \in V$ une valeur unique $\omega(v) \in Dom(v)$. $\Omega(V)$ représente l'ensemble des interprétations de variables de V. Soient $\psi \in \Psi(V)$ et $\omega \in \Omega(V)$, $\psi(\omega)$ représente la valeur de vérité de ψ en ω.

Une affectation de variables δ permet de modifier les valeurs des variables. Une variable ne peut être modifiée plus qu'une fois par une affectation. Par exemple, considérant l'affectation $\delta=\{v_1 := v_2+1, v_2:= 2\}$ et ω une interprétation de variables, $\omega[\delta]$ est une interprétation de variables définie par $\omega[\delta](v_1) = \omega(v_2) + 1$, $\omega[\delta](v_2) = 2$

et $\omega[\delta](v) = \omega(v)$ pour toute variable v autre que v_1 et v_2. De même, $\omega[\delta]$ est la contrainte satisfaite par toutes les interprétations $\omega[\delta]$ tel que ω satisfait ψ. Nous désignons par $\Delta(V)$ l'ensemble des affectations de variables de V. La figure 4.1 et 4.2 illustrent la description de l'ensemble d'attributs des variables de V, l'ensemble de variables V, l'ensemble d'horloges H et l'ensemble d'étiquettes *Act* pour un commutateur téléphonique à deux horloges.

```
Attributs_Commutateur_téléphonique
begin
DIALING({B})
RING({A})
ECHO_RING({B})
TALKING({A,B})
end
```

Figure 4.1 Exemple de description de l'ensemble d'attributs des variables de **V**

```
Horloges_Labels_Variables_Commutateur_téléphonique
begin
clocks = {h1, h2}
labels = {pickup_A, send_tone_A, dialing_B, busy_tone_A, ring_A_B, pickup_B}
domain (A_status)={BUSY, IDLE}
domain (A_signal)={NONE, TONE, BUSY_TONE, DIALING(B), ECHO_RING(B), ALKING(B)}
domain (B_status)={BUSY, IDLE}
domain (B_signal)={NONE, TONE, BUSY_TONE, RING(A), TALKING(A)}
end
```

Figure 4.2 Exemple de description de l'ensemble d'horloges **H**, d'étiquettes ***Act*** et de variables **V**

4.2.2 Représentation formelle d'un scénario

Soit P un processus et soit S l'ensemble de scénarios décrivant les comportements possibles de P. Nous devons spécifier le changement de l'état du processus après chaque action d'un scénario ou bien après l'écoulement du temps. Nous représentons l'état du processus par le couple $e = (\omega, \theta) \in \Omega(V) \times \Theta(H)$

formé des interprétations de variables et d'horloges. Dans la suite nous adoptons les notations suivantes: $\psi(e) = \psi(\omega)$, $\varphi(e) = \varphi(\omega)$, $e[\delta] = \omega[\delta]$ et $e[\lambda] = \theta[\lambda]$

Définition :

Tout scénario *sc* est un arbre (figure 1.1) : $sc = (N, \rightarrow, A)$ où :

- $N = N_p \cup N_s$ tel que N_p et N_s sont respectivement les ensembles des nœuds principaux et secondaires de *sc*. Nous supposons que $N_p = [N_1, N_2, ..., N_n, N_{n+1}]$ tel que $|N_p| = n+1$. Les nœuds secondaires sont des feuilles de l'arbre.
- A est l'ensemble des actions étiquetant les arcs de l'arbre de *sc*. Chaque action $a \in A$ est un tuple $a = <\varphi_a, \psi_a, \phi_a, lab_a, \delta_a, \lambda_a>$ où:
  - $\varphi_a \in \Phi(H)$ et $\psi_a \in \Psi(H)$ sont des contraintes sur l'état du processus,
  - $lab_a \in Act$,
  - $\phi_a \in \Phi(H)$ est une garde temporelle qui permet l'exécution de l'action *a,*
  - $\delta_a \in \Delta(V)$ est une affectation décrivant la mise à jour des valeurs des variables après l'exécution de l'action *a,*
  - λ_a est une affectation décrivant la mise à jour des valeurs des horloges après l'exécution de l'action *a.*
- $\rightarrow \in N \times A \times N$ est l'ensemble des arcs de l'arbre de *sc.*
- Chaque nœud principal N_i pour $1 \leq i \leq n$: $N_i=[a_{i1},\ldots, a_{iki}]$ tel que :

$(N_i, a_{i1}, N_{i+1}) \in \rightarrow$, a_{i1} est dite l'action principale du nœud N_i.

Nous avons vu dans le premier chapitre, que dans un nœud N_i les actions a_{ij} tel que $1 < j \leq k_i$ sont des alternatives de l'action principale a_{i1}. Ces alternatives peuvent représenter des expirations de temporisateurs ou toute autre interruption. Afin de formaliser la sémantique des actions d'un scénario, nous allons associer à chaque nœud principal N_i, deux contraintes $\psi_{Ni} \in \Psi(V)$ et $\varphi_{Ni} \in \Phi(H)$. Ce couple de contraintes définit le contexte de son nœud. Le contexte de N_i est défini de la manière récurrente suivante :

(1) $\psi_{N1} = \psi_{a11}$ et $\varphi_{N1} = \wedge_{(1 \leq k \leq k1)} \varphi_{a1k}$

(2) $\psi_{Ni} = (\psi_{Ni-1}[\delta_{a(i-1)1}] \wedge \psi_{ai1})$ et $\varphi_{Ni} = \wedge_{(1 \leq k \leq ki)} \varphi_{aik}$ (pour $1 < i \leq n$)

(3) $\psi_{Nn+1} = \psi_{Nn}[\delta_{an1}]$ et $\varphi_{Nn+1} = (\varphi_{Nn} \wedge \phi_{an1})[\lambda_{an1}]$

Après le calcul du contexte des différents nœuds de l'arbre, il ne reste que le calcul des actions-règles qui doivent être indépendantes du contexte des nœuds précédents.

4.2.3 Construction des actions-règles

Pour former une action-règle indépendante, il faut enrichir l'action par son contexte. Chaque action-règle doit comporter toute l'information sur l'état du processus et sa condition d'activité avant et après son exécution. Soit r une action-règle issue de l'action a_{ij} du nœud principal N_i, $r = (\psi_r, \varphi_r, lab_r, \phi_r, \delta_r , \lambda_r, \varphi_r')$ tels que :

- $\psi_r = \psi_{Ni} \wedge \psi_{aij}$
- $\phi_r = \phi_{aij}$
- $\varphi_r = \varphi_{Ni}$
- $\delta_r = \delta_{aij}$
- $lab_r = lab_{aij}$
- $\lambda_r = \lambda_{aij}$
- Pour φ_r', nous distinguons les deux cas suivants :
  - $\varphi_r' = \varphi_{Ni+1}$ pour $j = 1$ et $1 \leq i \leq n$
  - $\varphi_r' = (\varphi_{Ni} \wedge \phi_{aij})[\lambda_{aij}]$ pour $1 \leq j \leq k_i$ et $1 \leq i \leq n$

4.2.4 Construction de l'automate temporisé

Nous désignons par $R(sc)$ l'ensemble des actions-règles issues des actions du scénario *sc*. A partir des actions-règles, nous allons générer les places et les transitions de l'automate. Pour ce faire, nous allons présenter les définitions suivantes :

Définition :

Σ_{sc} représente le système de transitions étiquetées du scénario *sc*. Pour chaque action-règle $r \in R(sc)$, Σ_{sc} comporte deux types de transitions :

- des transitions instantanées de la forme (e, lab_r, e') telles que $e = (\omega,\theta)$ et $e'(\omega[\delta_r] = \theta[\lambda_r])$ avec $\psi_r(\omega) \wedge \varphi(\theta) \wedge \varphi'(\theta[\lambda_r])$.
- et des transitions d'écoulement du temps de la forme (e, d, e') telles que $e = (\omega,\theta)$ et $e' = (\omega,\theta+d)$ avec $\psi_r(\omega) \wedge \varphi(\theta) \wedge (\forall\ d' \leq d.\varphi\ (\theta + d'))$.

Théorème :

Soit *sc* un scénario de S, il existe un TA dont le système de transitions étiquetées est équivalent à Σ_{sc}. Ce TA est dit TA de scénario *sc* et il est noté A_{sc}.

Donc nous pouvons déduire directement les places et les transitions de l'automate temporisé A_{sc} à partir des actions-règles de R(*sc*). Nous commençons par partitionner les places issues des actions-règles, puis, nous ajoutons les transitions entre les places. Les transitions sont alors de deux types: des transitions issues directement des actions-règles et des transitions assurant la progression du temps. Le partitionnement des places issues des actions-règles utilise un algorithme qui assure l'aspect de minimalité n'effectuant le partitionnement d'une place que s'il est nécessaire.

4.3 Intégration explicite de scénarios

Dans les sections précédentes, nous avons présenté une méthode implicite d'intégration de scénarios qui n'utilise aucune directive sur l'ordre d'exécution des scénarios. Cependant, l'intégration explicite de scénarios utilise des directives qui vont contraindre le système par des contraintes d'ordre d'exécution, par exemple, exécuter un scénario avant un autre. L'intégration explicite des systèmes exprimé par des directives est très utile dans la spécification des systèmes existants ou pour exprimer le besoin des concepteurs à un certain ordre d'exécutions des scénarios. Dans la suite, nous allons définir un formalisme sémantique pour les directives explicites.

4.3.1 Les directives explicites d'intégration de scénarios

La méthode d'intégration explicite vise à synthétiser des nouveaux scénarios qui satisfont les directives explicites. Soit *Dir* l'ensemble de directives explicites d'intégration de scénarios d'un système. Le processus de spécification du système P sera noté comme suit : P = (S, *Dir*, V, H, *Act*) où S, V, H et *Act* sont définis dans les sections précédentes et chaque élément de *Dir* est une directive de la forme sc_1 # sc_2 tel que sc_1 et $sc_2 \in$ S et # $\in \{|, ->\}$. "$sc_1 \mid sc_2$" implique que l'exécution de sc_1 et sc_2 est simultanément possible comme deux alternatives mais le système va exécuter

seulement une seule puisqu'on suppose que le système est séquentiel. La directive "sc_1 -> sc_2" implique que sc_1 est exécuté avant sc_2.

Afin de formaliser la sémantique des opérateurs alternatifs et séquentiels, nous définissons la fonction *scen*(ψ, φ, *sc*) où ψ est une contrainte sur variables, φ est une contrainte temporelle et *sc* est un scénario. L'expression *sc'* = *scen*(ψ, φ, *sc*) signifie que le scénario *sc'* a les mêmes actions que *sc* excepté que la première action principale de *sc'* est sur-contraint par ψ et φ comme suit : $\psi_{sc'.a11} = \psi_{sc.a11} \wedge \psi$ et $\varphi_{sc'.a11} = \varphi_{sc.a11} \wedge \varphi$. L'effet de l'action a_{11} du scénario *sc'* sera propagé quand les actions-règles de *sc'* seront calculées.

Une directive sc_1 # sc_2 est considérée comme une équation et l'ensemble *Dir* comme un système d'équations. La méthode d'intégration explicite consiste à résoudre le système d'équations s'il est solvable en satisfaisant toutes les équations. Pour ce faire, nous allons associer à *Dir* un graphe orienté appelé graphe de directives d'intégration (GDI) qui fournit une vue globale du chemin de propagation des contraintes entre les directives. Le GDI de *Dir* est un graphe orienté (V_{Dir}, E_{dir}) où V_{dir} est l'ensemble de sommets et E_{dir} est l'ensemble d'arcs. La construction de GDI associe à chaque sommet $x \in V_{Dir}$ une contrainte sur variables ψ_x et une contrainte temporelle φ_x. Chaque arc (x, *sc*, y) $\in E_{dir}$ permet de propager les contraintes associées au sommet x à travers le scénario *sc* au sommet y. Supposant (x, *sc*, y) $\in E_{dir}$, les scénarios sur-contraints sous la forme *sc'* = *scen*(ψ_x, φ_x, *sc*) sont la solution (si elle existe) au système d'équations de *Dir*.

Avant de construire le GDI, nous construisons un nouvel ensemble de directives noté *Dir'* à partir de l'ensemble *Dir*. L'objectif de cette étape est d'éliminer les confusions qui peuvent se présenter. Par exemple, les deux ensembles de directives suivants {sc_1 -> sc_2, sc_1 -> sc_3} et {sc_1 -> sc_2, sc_2 | sc_3} ont le même GDI mais peuvent avoir une sémantique différente. Par l'application de cette étape nous éliminons cette ambiguïté, par exemple, soit l'ensemble de directives *Dir* = {sc_1 -> sc_2, sc_1 -> sc_3} alors l'ensemble de directives *Dir'* est le suivant : *Dir'* = {sc_1 -> sc_2, sc_1 | $sc_1^{(1)}$, $sc_1^{(1)}$ -> sc_3} avec $sc_1^{(1)}$ est un nouveau scénario qui est une copie de sc_1. Le graphe GDI est construit à partir de l'ensemble de directives

Dir'. La définition de la construction du GDI et de l'ensemble *Dir'* ainsi que les algorithmes de propagation sont présentés dans [Sal 01].

4.4 Conclusion

Dans ce chapitre, nous avons présenté le fondement théorique de notre approche précisément l'étape de génération des actions-règles. Nous avons présenté aussi l'intégration explicite de scénarios en décrivant le formalisme des directives et les étapes de construction du graphe de directives d'intégration. Dans les chapitres suivants, nous allons présenter l'architecture générale du système en terme de fonctionnalités et de modules. Ensuite nous détaillons les modules, les structures de données et les algorithmes nécessaires pour la génération des actions-règles et la génération explicite. Enfin, nous décrivons l'interface de l'application et les grammaires des langages d'acquisition des différentes entrées de l'environnement SCENA et nous fournissons un exemple d'application complet.

Architecture du système

5.1 Introduction

Ce chapitre décrit l'architecture générale du système et les techniques utilisées pour l'implémentation. Dans la section qui suit, nous décrivons les techniques et les outils (C++, UML, Flex, Bison, etc.) choisis afin d'implémenter l'architecture proposée ainsi que les motivations de ces choix. Dans la troisième section, nous décrivons l'architecture du système en terme de fonctionnalités et de modules.

5.2 Environnement de travail

L'utilisation d'une conception objet exprimée sous forme de diagrammes UML (Unified Modeling Language) pour décrire la partie statique de notre système et l'utilisation d'un langage C++ offrant les possibilités de réutilisation, de modification et d'amélioration future sont les points forts de notre architecture.

Pour développer notre système, nous avons opté pour le langage C++ comme langage de programmation. En outre, ce choix est justifié par les facilités qu'offre ce langage dans le développement des applications orientées objets. Ce langage présente les avantages d'être portable, réutilisable, orienté objet et génère un code optimisé.

La conception du système est réalisée à l'aide du langage de modélisation UML. Ce dernier est un langage puissant. Sa notation graphique permet d'exprimer visuellement une solution objet. Sa notation limite aussi les ambiguïtés et les incompréhensions. Son aspect visuel facilite la comparaison et l'évaluation de solutions. Son indépendance par rapport aux langages d'implémentation, aux domaines d'application et aux processus en font un langage universel. Les

diagrammes de classes conçus par UML seront traduits sous forme de classes écrites en C++ qui décrivent le contenu du système en terme de données et de comportement.

Flex et Bison sont deux outils de quatrième génération générant du code C++ à partir d'un langage de spécification plus simple. L'analyseur lexical Flex permet de générer un compilateur lexical en spécifiant les différents lexèmes. L'analyseur syntaxique Bison permet de générer un compilateur syntaxique en fournissant la grammaire du langage. Le choix de Flex et Bison est raisonnable dans la mesure où il y a de bonnes raisons d'utiliser un langage à objets pour la réalisation d'un compilateur.

5.3 Architecture générale du système

L'approche UML décrit l'architecture d'un système de deux manières indépendantes mais complémentaires à savoir l'architecture en terme de fonctionnalités et l'architecture en terme de modules. Dans les sections suivantes, nous décrivons ces deux architectures.

5.3.1 Description de l'architecture en terme de fonctionnalités

Le travail que nous avons élaboré s'inscrit dans le cadre d'un projet global comprenant deux missions principales : la génération implicite des automates temporisés à partir de scénarios et la génération explicite. Nous avons proposé une architecture générale décrivant les principales fonctionnalités du système et les relations entre elles. Cette architecture est décrite dans le diagramme suivant (figure 5.1):

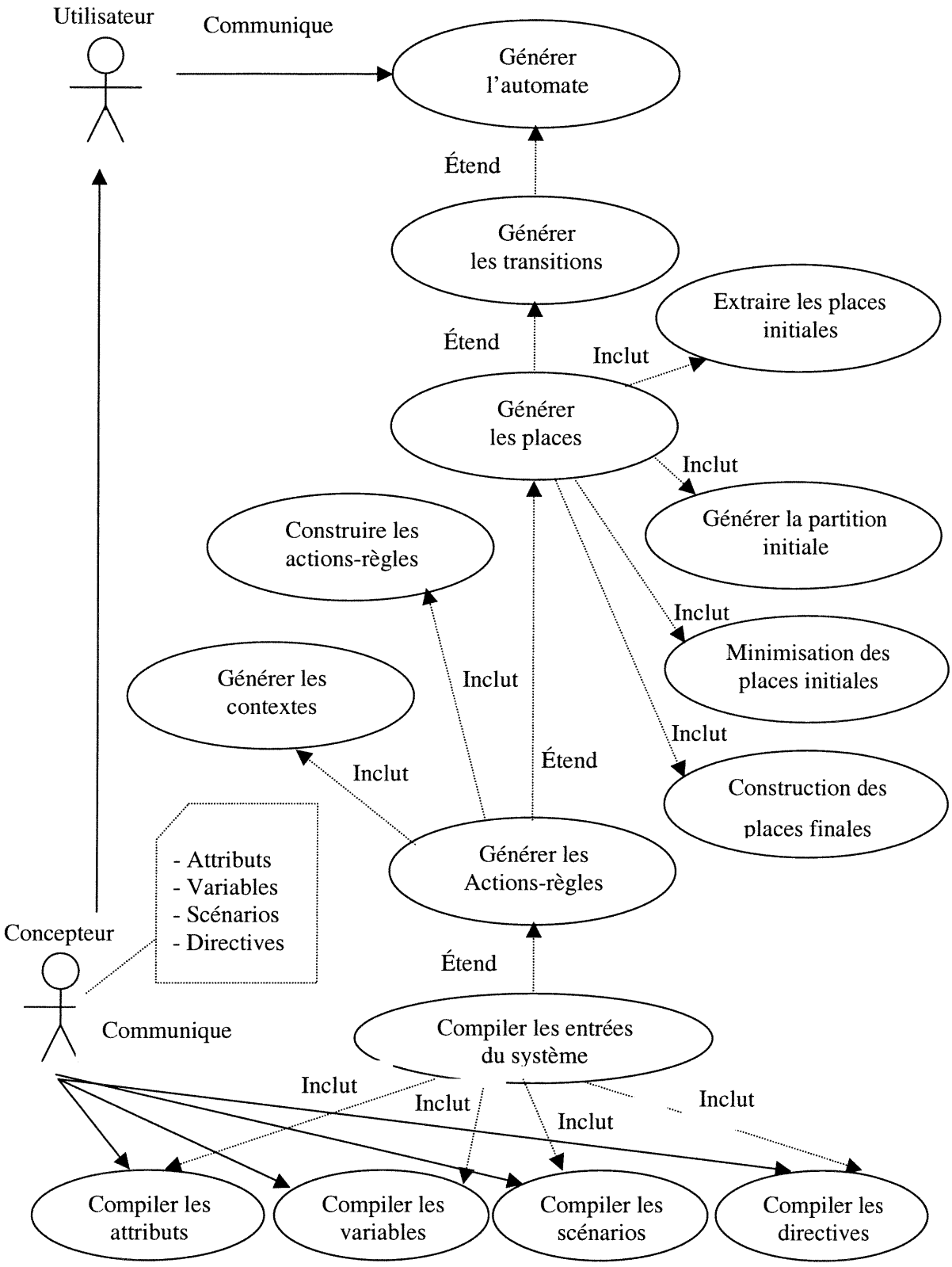


Figure 5.1 Description des fonctionnalités du système SCENA

5.3.2 Description de l'architecture en terme de modules

Notre système est composé d'un ensemble de modules qui répondent à un ensemble de fonctionnalités décrites dans la figure 5.2. Nous rassemblons dans un même module les classes d'objets qui forment une entité indépendante. Ces modules sont définis comme suit :

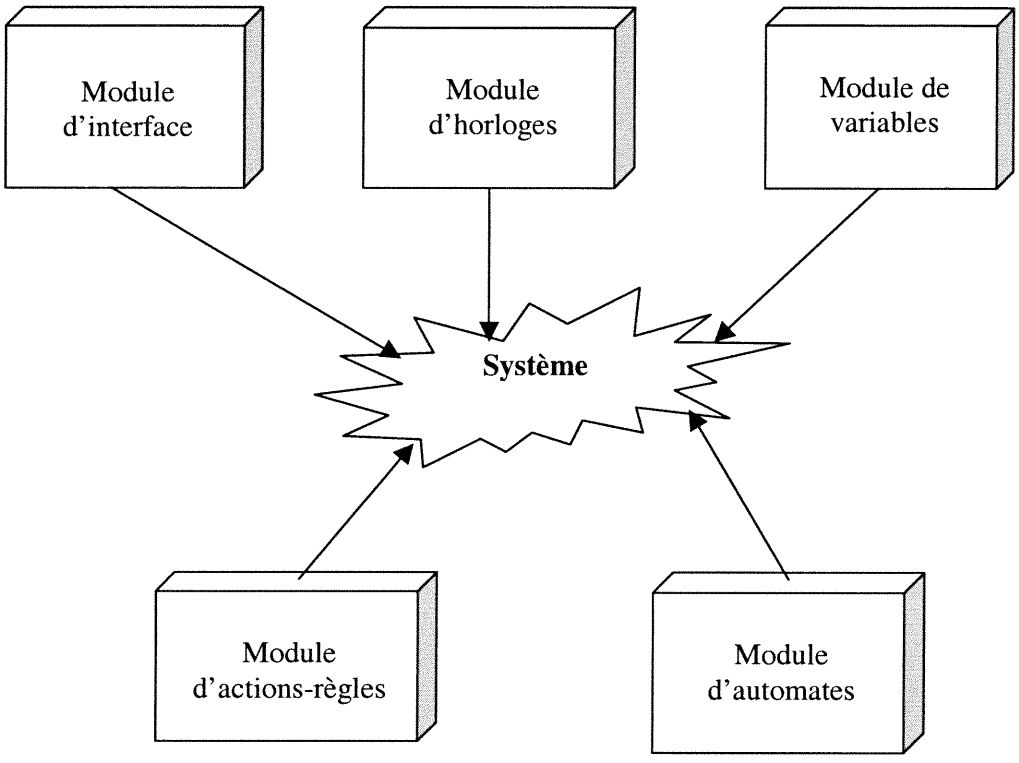


Figure 5.2 Architecture générale du système en modules

- Module d'interface : englobe tous les modules principaux, les compilateurs d'acquisitions d'entrées du système et les interfaces permettant à l'utilisateur d'interagir avec l'application.

- Module d'horloges : englobe les classes qui représentent les horloges et les contraintes temporelles dans le système. Ce module contient toutes les classes de gestion des contraintes temporelles telles que la conjonction, la disjonction et la négation.

- Module de variables : regroupe les classes nécessaires pour représenter les variables, les attributs, les paramètres, les domaines, les contraintes sur variables et les affectations de variables. Ce module intègre toutes les opérations nécessaires pour manipuler les variables et les contraintes sur variables telle que l'affectation d'une variable par une autre et la conjonction de deux contraintes sur variables.

- Module d'actions-règles : ce module est composé de trois sous-modules :
  - Le sous-module de scénarios qui englobe les classes représentant les scénarios, les nœuds des scénarios et les actions des nœuds.
  - Le sous-module de calcul des actions-règles qui intègre les méthodes relatives au calcul du contexte des nœuds des scénarios et la génération des actions-règles.
  - Le sous-module d'intégration explicite qui modélise les directives et le graphe de directives d'intégration (GDI). Ce sous-module contient les algorithmes nécessaires pour la génération du GDI. Le GDI permet la propagation des contraintes exprimées par les directives à travers les scénarios.

- Module d'automates : contient les classes d'extraction des places initiales de l'automate à partir des actions-règles, les classes de génération de la partition initiale, les classes de minimisation des places initiales et les classes de construction des places finales et des transitions de l'automate.

5.4 Conclusion

Dans ce chapitre, nous avons présenté les différents outils techniques nécessaires pour le développement du système. Ensuite, nous avons développé l'architecture générale en la décrivant en terme de fonctionnalités et en terme de modules. Dans les chapitres qui suivent, nous allons détailler les différents modules présentés dans ce chapitre en décrivant pour chacun les principales classes d'objets développées ainsi que les interactions entre les différents objets.

Modules d'horloges et de variables

6.1 Introduction

Dans ce chapitre, nous décrivons le module d'horloges ainsi que le module de variables. Nous développons le fondement théorique du module d'horloges puis nous détaillons les principales classes d'objets développées ainsi que les interactions entre les différents objets. Pour le module de variables, nous présentons les différentes classes d'objets pour chaque type de données et les différentes interactions entre les objets du module.

6.2 Module d'horloges

Pour manipuler les contraintes temporelles, il faut définir les opérateurs élémentaires qui peuvent être utilisés dans les différents modules du système. Nous voulons pouvoir effectuer des opérations d'intersection et d'union sur les contraintes tout en conservant la convexité des contraintes résultantes. Nous allons présenter l'algèbre permettant de représenter les contraintes temporelles et les opérateurs qui permettent de les manipuler.

6.2.1 La matrice de bornes

Nous représentons une contrainte temporelle par la matrice de différence de bornes vue dans le troisième chapitre. Considérons, par exemple, la contrainte temporelle suivante:

$$\varphi : 1 \leq x < 3 \ \wedge \ y \leq 5 \ \wedge \ x\text{-}y \leq 2.$$

Elle peut être représentée au moyen de la matrice suivante:

	0	x	y
0	$(0, \leq)$	$(-1, \leq)$	$(0, \leq)$
x	$(3, <)$	$(0, \leq)$	$(2, \leq)$
y	$(5, \leq)$	$(\infty, <)$	$(0, \leq)$

Chaque entrée de la matrice représente la borne supérieure de la différence entre deux horloges. Une horloge fictive nommée **0** dont la valeur est toujours zéro, a été ajoutée, permettant de représenter la borne inférieure et supérieure des autres horloges. Par exemple, la borne inférieure de x est $\mathbf{0} - x \leq -1$ et la borne supérieure est $x - \mathbf{0} \leq 3$. Nous utilisons la borne $(\infty,<)$ dans le cas d'absence de bornes. Dans le cas d'absence de bornes minimales, nous utilisons la valeur $(0, \leq)$, puisque les valeurs des horloges sont toujours positives. Dans la suite, nous allons définir les bornes qui sont les éléments de la matrice tel que développé par Yovine dans [Yov93]. Ensuite, nous présentons les opérateurs nécessaires pour la manipulation des bornes.

6.2.2 Les bornes

Soit *B* l'ensemble des bornes défini par:

$$\boldsymbol{B} = Z \times \{<,\leq\} \cup \{(-\infty,<),(\infty,<)\}$$

Notons que les opérateurs $<$ et $\leq$ sont totalement ordonnés : $<$ est supposé strictement plus petit que $\leq$. Nous définissons un ordre total d'inclusion sur l'ensemble $\boldsymbol{B}$ des bornes comme suit [Yov93]:

Pour tout $(c_1,\angle_1)$ et $(c_2,\angle_2) \in \boldsymbol{B}$ avec $\angle \in \{<,\leq\}$, nous définissons :

$(c_1,\angle_1) \subseteq (c_2,\angle_2)$ si $c_1 < c_2$ ou si $c_1 = c_2$ et $\angle_1$ est inférieure ou égale à $\angle_2$.

Nous définissons deux autres opérateurs sur l'ensemble de bornes $\boldsymbol{B}$:

- Pour tout $(c_1,\angle_1)$ et $(c_2,\angle_2) \in \boldsymbol{B}$, nous définissons :

  $$(c_1,\angle_1) \otimes (c_2,\angle_2) = (c_1 + c_2, \min(\angle_1,\angle_2))$$

  L'élément neutre pour l'opération $\otimes$ est $\mathbf{e} = (0,\leq)$.

- Pour tout $(c_1,\angle_1)$ et $(c_2,\angle_2) \in \boldsymbol{B}$, nous définissons :

  $$(c_1,\angle_1) \oplus (c_2,\angle_2) = \min((c_1,\angle_1), (c_2,\angle_2))$$

  L'élément neutre pour l'opération $\oplus$ est $\mathbf{n} = (\infty,<)$.

La structure $\langle \boldsymbol{B},\oplus,\otimes,\mathbf{n},\mathbf{e}\rangle$ est un demi-anneau fermé [Yov93]. Nous pouvons déduire que $\oplus$ est commutative et idempotente ($b \oplus b = b$) et que $\otimes$ est distributive par rapport à $\otimes$.

6.2.3 Les opérations sur les matrices de bornes

L'ordre total $\subseteq$ défini sur les bornes peut être étendu de manière naturelle aux matrices de bornes. Soit $H^{\#} = H \cup \{0\}$ (H étant l'ensemble d'horloges du système), pour toute paire de matrices M et M':

$$M \subseteq M' \text{ si pour tout } x,y \in H^{\#}, M_{xy} \subseteq M'_{xy}$$

Les opérations $\oplus$ et $\otimes$ sont étendues sur les matrices de bornes de la façon suivantes :

- Soient M et M' deux matrices de bornes. $M'' = M \otimes M'$ est une matrice de bornes tel que pour tout $x,y \in H^{\#}$,

$$M''_{xy} = \bigoplus_{(z \in H^{\#})} M_{xz} \otimes M'_{zy}$$

- Soient M et M' deux matrices de bornes. $M'' = M \oplus M'$ est une matrice de bornes tel que pour tout $x,y \in H^{\#}$,

$$M''_{xy} = M_{xy} \oplus M'_{xy}$$

- Soit **N** l'élément neutre pour $\oplus$ et **E** l'élément neutre pour $\otimes$ définies de la façon suivante:

$$E_{xy} = \mathbf{e} \text{ si } x = y \text{ ou } \mathbf{n} \text{ sinon}$$

$$N_{xy} = \mathbf{n}$$

Enfin soit $\boldsymbol{M}$ l'ensemble des matrices de bornes définit par:

$$\boldsymbol{M} = \{M \mid \forall x \in H^{\#}, M_{xx} \subseteq (0,\leq)\} \cup \{\mathbf{N}\}$$

La structure $\langle \boldsymbol{M},\oplus,\otimes,\mathbf{N},\mathbf{E}\rangle$ est un demi-anneau fermé permettant les mêmes opérations que sur les bornes. Ces propriétés vont nous permettre la commutativité et la distributivité sur les contraintes temporelles.

Exemple:

Soient les deux contraintes φ_1 et φ_2 et les deux matrices M_1 et M_2 correspondantes:

$\varphi_1 = 2 < x \wedge x - y \leq 3$ et $\varphi_2 = x \leq 3 \wedge y \leq 5$

	0	X	y
0	(0,≤)	(-2,<)	(0,≤)
x	(∞,<)	(0,≤)	(3,≤)
y	(∞,<)	(∞,<)	(0,≤)

$\oplus$

	0	x	y
0	(0,≤)	(0,≤)	(0,≤)
x	(3,≤)	(0,≤)	(∞,<)
y	(5,≤)	(∞,<)	(0,≤)

=

	0	x	y
0	(0,≤)	(-2,<)	(0, ≤)
x	(3,≤)	(0,≤)	(3,≤)
y	(5,≤)	(∞,<)	(0,≤)

D'après cet exemple, la matrice $M_1 \oplus M_2$ représente la contrainte $\varphi_1 \wedge \varphi_2$. En effet, l'opérateur $\oplus$ sur les matrices de bornes coïncide avec l'intersection des contraintes temporelles. Donc, la matrice $M_1 \oplus M_2$ représente la contrainte $\varphi_1 \wedge \varphi_2$ [Yov93].

6.2.4 Calcul de la différence de deux contraintes

Soient deux contraintes φ_1 et φ_2 représentant deux zones d'horloges Z_1 et Z_2. Nous voulons résoudre $\varphi_1 \setminus \varphi_2$, c'est à dire déduire l'ensemble des zones convexes composant $Z_1 \setminus Z_2$. Une contrainte temporelle φ est composée par la disjonction d'une ou de plusieurs contraintes élémentaires de la forme : $\varphi = \Phi_1 \wedge \Phi_2 \wedge ... \wedge \Phi_k$. Une contrainte élémentaire Φ est composée soit d'une seule horloge de la forme $\Phi := x \# c$ soit d'un couple d'horloges formé de la différence de deux horloges de la forme $\Phi := x\text{-}y \# c$. Une zone élémentaire est une zone d'horloges convexe composée d'une ou de plusieurs contraintes élémentaires. Une contrainte φ_1 est incluse dans une contrainte φ_2 si et seulement si la zone Z_1 est incluse dans la zone Z_2. Une contrainte φ est vide s'il existe au moins deux contraintes élémentaires Φ_i et Φ_j sur la même horloge ou couple d'horloges tel que $\Phi_i \cap \Phi_j$ est vide.

Pour calculer $\varphi_1 \setminus \varphi_2$, nous devons dégager toutes les zones élémentaires et identifier celles incluses dans φ_1 et non incluses dans φ_2. Pour extraire les zones élémentaires de $\varphi_1 \setminus \varphi_2$, nous devons découper tout l'espace en des zones élémentaires selon les contraintes élémentaires. Une approche générale consiste à découper l'espace sur les contraintes élémentaires de φ_1 et φ_2. Soit n le nombre d'horloges, k_1 le nombre de contraintes élémentaires de type $x \# c$ dans φ_1 et φ_2 et soit k_2 le nombre de contraintes élémentaires de type $x - y \# c$ dans φ_1 et φ_2. Le

découpage de l'espace en zones élémentaires est déterminé par les combinaisons suivantes:

$$\{\Phi_1,\neg\Phi_1\}\times\{\Phi_2,\neg\Phi_2\}\times...\times\{\Phi_{k1},\neg\Phi_{k1}\}\times\{\Phi'_1,\neg\Phi'_1\}\times\{\Phi'_2,\neg\Phi'_2\}\times...\times\{\Phi'_{k2},\neg\Phi'_{k2}\}$$

Le nombre de combinaison total est de 2^{k1+k2}. Cette approche génère un grand nombre des contraintes vides. Pour éviter la génération de toutes les zones élémentaires générées par les contraintes élémentaires de φ_1 et de φ_2, nous proposons de partitionner l'espace seulement sur les contraintes élémentaires de φ_2.

6.2.4.1 Principe de calcul des zones élémentaires de $\varphi_1 \setminus \varphi_2$

La contrainte $\varphi_1 \setminus \varphi_2$ est égale à $\varphi_1 \cap \neg\varphi_2$. La contrainte $\neg\varphi_2$ est constituée d'un ensemble de zones élémentaires, tel que $\neg\varphi_2 = \{z_1 \cup z_2 \cup ... \cup z_n\}$. Chaque zone élémentaire z_i de $\neg\varphi_2$ est définie par une contrainte φ_i. Pour déterminer les zones élémentaires de $\varphi_1 \setminus \varphi_2$, nous vérifions pour chaque zone élémentaire de $\neg\varphi_2$ si $\varphi_1 \cap \varphi_i$ est vide ou pas. Les zones élémentaires résultats non vides constituent les zones élémentaires de $\varphi_1 \setminus \varphi_2$. Pour faire le calcul de $\varphi_1 \cap \varphi_i$, il suffit d'utiliser l'opérateur $\oplus$ définit auparavant entre les matrices de bornes des contraintes φ_1 et φ_i.

6.2.4.2 Principe de découpage de l'espace sur les contraintes élémentaires de φ_2

Afin d'éviter de générer toutes les combinaisons possibles sur les différentes contraintes élémentaires de φ_2 comme dans l'approche générale, nous proposons de classer les contraintes élémentaires de φ_2. Nous commençons par les contraintes élémentaires de type x # c. Ensuite, nous traitons les contraintes élémentaires de type x – y # c. Nous définissons une bande comme une partie de l'espace délimitée par une ou deux contraintes élémentaires sur une même horloge ou couple d'horloges. Nous considérons que deux bandes se succèdent s'ils ont chacune une contrainte élémentaire dont les valeurs des constantes sont égales sur une horloge ou couple d'horloges. Le découpage en zones élémentaires se fait par la combinaison entre les différentes bandes de toutes les horloges et les couples d'horloges. Pour chaque horloge ou couple d'horloges, nous avons au plus trois bandes. Nous obtenons les zones élémentaires par la combinaison des bandes successives comme suit:

$\{\neg\Phi^1_1, \neg\Phi^1_1\Phi^1_2, \neg\Phi^1_2\} \times$

$\{\neg\Phi^2_1, \neg\Phi^2_1\Phi^2_2, \neg\Phi^2_2\} \times$

$\vdots$

$\{\neg\Phi^n_1, \neg\Phi^h_1\Phi^n_2, \neg\Phi^n_2\} \times$

$\{\neg\Phi'^1_1, \neg\Phi'^1_1\Phi'^1_2, \neg\Phi'^1_2\} \times$

$\{\neg\Phi'^2_1, \neg\Phi'^2_1\Phi'^2_2, \neg\Phi'^2_2\} \times$

$\vdots$

$\{\neg\Phi'^m_1, \neg\Phi'^m_1\Phi'^m_2, \neg\Phi'^m_2\}$

Où n représente le nombre d'horloges et m représente le nombre maximal de couples d'horloges, soit $m = C^n_2 = n(n-1)/2$. Le nombre total des combinaisons des bandes successives est de $3^{n+m} = 3^{1/2*n(n+1)} \approx (1.7)^{n(n+1)}$. Pour le même nombre de contraintes, la complexité de l'approche générale est de $2^{2*1/2*n(n+1)} = 2^{n(n+1)}$. Nous constatons même qu'avec un nombre maximum de contraintes, la complexité est toujours meilleure que celle de l'approche générale. L'algorithme devient très performant avec moins de contraintes élémentaires de type x-y # c, qui est le cas dans la pratique. L'algorithme de calcul de la contrainte $\varphi_1 \setminus \varphi_2$ est présenté dans la figure 6.1.

Soient φ_1 et φ_2 deux contraintes et φ une contrainte vraie
Soit C un ensemble de contraintes initialement vide
Soit R un ensemble de contraintes qui contiendra les contraintes résultats
$C = C \cup \{\varphi\}$
Pour toutes les horloges et couples d'horloges
    Extraire Φinf et Φmax de la contrainte φ_2
    $C = \{\neg\Phi\text{inf}\} \times C \cup \{\Phi\text{inf}, \Phi\text{max}\} \times C \cup \{\neg\Phi\text{max}\}$
Fin Pour
Éliminer les contraintes vides dans C
Éliminer la contrainte φ_i dans C tel que $\varphi_i = \varphi_2$
Pour chaque contrainte dans C
    Si $\varphi_i \cap \varphi_1$ est non vide **alors**
        $R = R \cup \{\varphi_i\}$
    FinSi
Fin Pour

Figure 6.1 Algorithme de calcul de la contrainte $\varphi_1 \setminus \varphi_2$

6.3.2 Description du module d'horloges

Ce module présente l'ensemble de classes qui assurent toutes les opérations sur les horloges et les contraintes temporelles (Figure 6.2). La classe d'objet *Horloge* représente une horloge dans le système qui est caractérisé par le nom de l'horloge. Toutes les horloges du système sont regroupées dans la classe d'objet *HorlogeTable*. La classe d'objet *IndexTable* modélise une affectation de certaines horloges par l'intermédiaire d'une liste qui contient les horloges à mettre à zéro.

Une contrainte temporelle élémentaire est représentée par la classe d'objet *ContrainteElementaire*. La classe d'objet *ContrainteTemp* utilise la classe *ContrainteElementaire* pour modéliser une contrainte temporelle. Elle englobe les méthodes qui manipulent une contrainte temporelle telles que la disjonction par l'intermédiaire de la méthode *Intersection*(*ContrainteTemp*), l'inclusion par l'intermédiaire de la méthode *InclusDans*(*ContrainteTemp*) et la projection par l'intermédiaire de la méthode *Appliquer*(*IndexTable*). Toutes les contraintes temporelles sont regroupées dans la classe d'objet *ContrainteTempTable* qui implémente l'ensemble des méthodes qui manipulent le vecteur de contraintes temporelles.

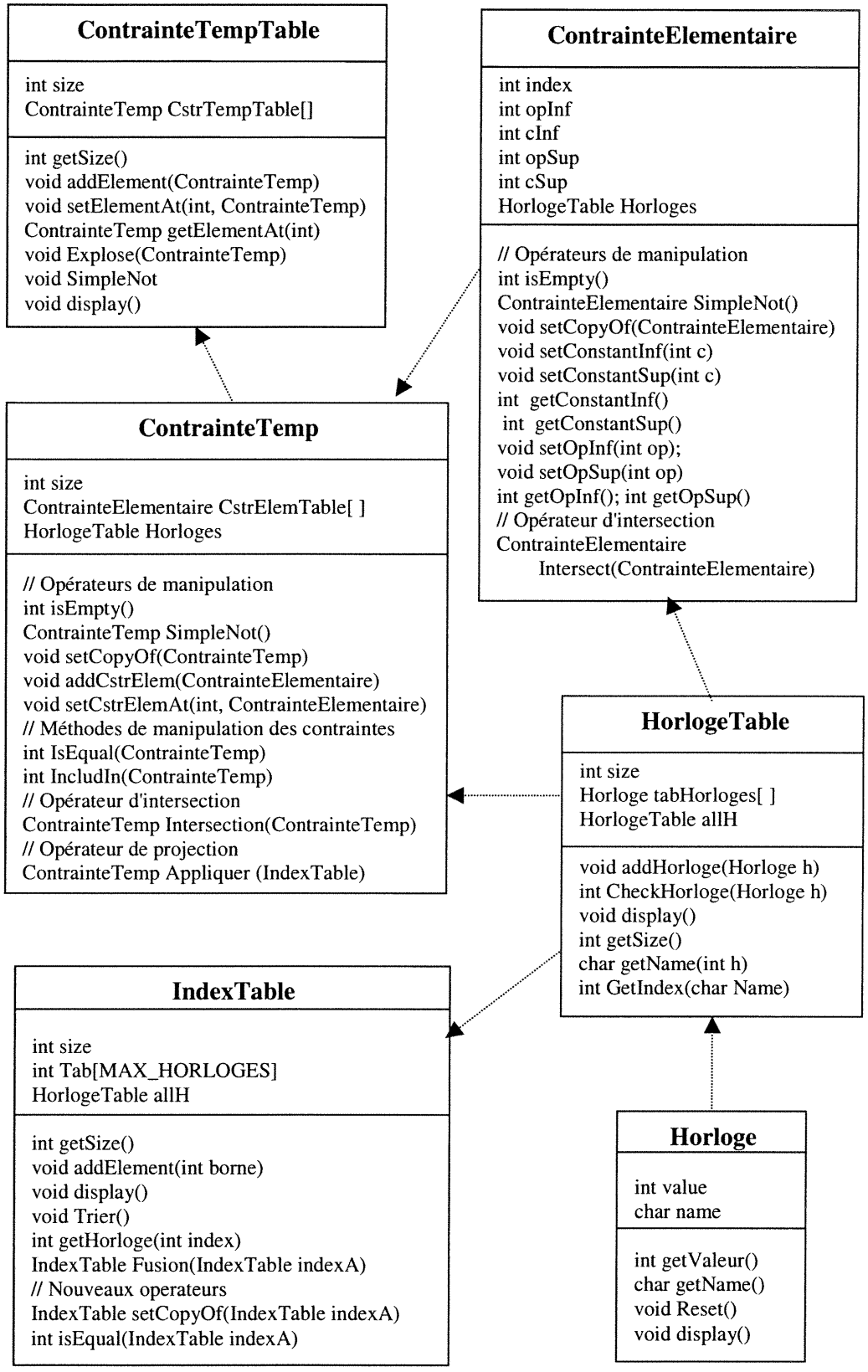


Figure 6.2 Diagramme de classe du module d'horloges

6.3 Module de variables

Le module de variables modélise les attributs avec leurs paramètres, les variables avec leurs domaines, les contraintes de variables et les affectations de variables. Il est composé de cinq sous-modules qui sont les suivants: le sous-module de paramètres, le sous-module d'attributs, le sous-module de variables, le sous-module d'affectation de variables et le sous-module de contraintes sur variables. Nous détaillons les différents sous-modules dans les sections suivantes.

6.3.1 Sous-module de paramètres

Dans ce sous-module, nous modélisons les types de base et le domaine de paramètres. Nous avons opté pour une structure de données qui fait abstraction du type lors de la manipulation des objets. La référence au type d'un objet n'est faite que lors des opérations d'affectation ou de calcul. Nous englobons les deux types, entier et chaîne de caractères dans un nouveau type *TypeValeurParametre* (Figure 6.3) qui représente le nouveau type élémentaire de base. Nous appliquons les différentes opérations sur cet objet sans tenir compte de son type de base. La classe d'objet *TypeValeurDomaineParametre* ajoute un autre type autre qu'un entier et chaîne de caractères qui est l'intervalle. Ce nouveau type doit posséder les caractéristiques d'un entier et d'une chaîne de caractères, qui sont dans le type *TypeValeurParametre*, et les caractéristiques d'un intervalle. La classe d'objet *DomaineParametre* construit une liste d'objets de *TypeValeurDomaineParametre*. Elle modélise un domaine de données de différents types tels que des entiers, des chaînes de caractères et des intervalles. Par exemple, le domaine de paramètres d'un paramètre *P* est Domaine(*P*) = {1, 2, 5, [20:30], 35, [50:60], 99, send, receive}.

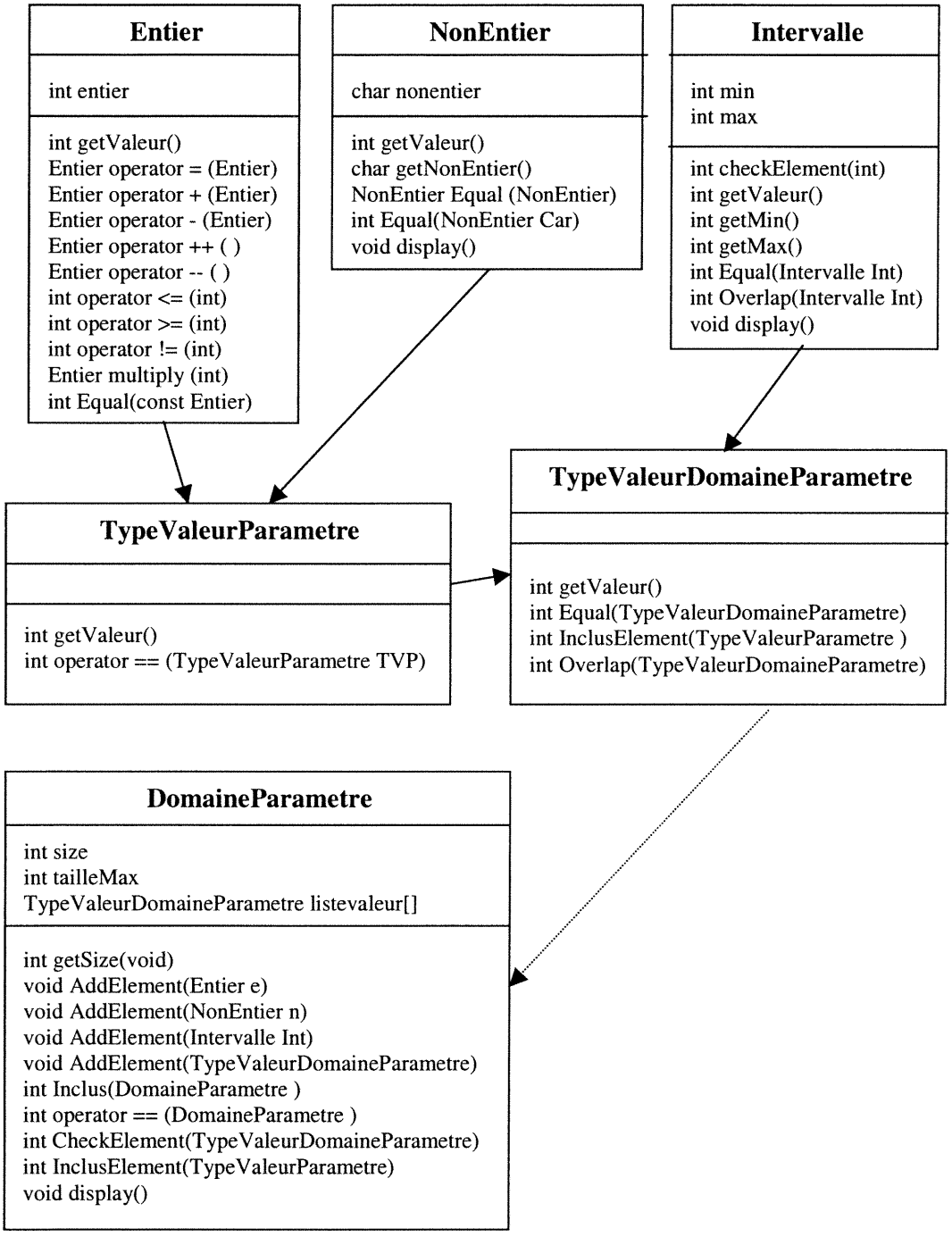


Figure 6.3 Diagramme de classes du sous-module de paramètres

6.3.2 Sous-module d'attributs

Un attribut est caractérisé par son nom, par le nombre de ses paramètres et par les domaines de ses paramètres. Il est modélisé par la classe d'objet *AttributDefinition* (Figure 6.4). L'ensemble de tous les attributs est regroupé dans la classe d'objet *AttributTable*. La classe d'objet *Attribut* modélise une instance d'un attribut dans le système. Elle modélise aussi des instances de ses paramètres qui sont de type *TypeValeurParametre*. Par exemple, l'attribut *Tone_of*(P_1,P_2,P_3) a trois paramètres P_1, P_2 et P_3. Les domaines de chaque paramètre sont les suivants : P_1 = {21, [11:20], 23, [-10:10], 22}, P_2 = {[3:5]} et P_3 = {1, -3, *send*, -6}. Une instance de cet attribut est *Tone_of*(23, 4, *send*).

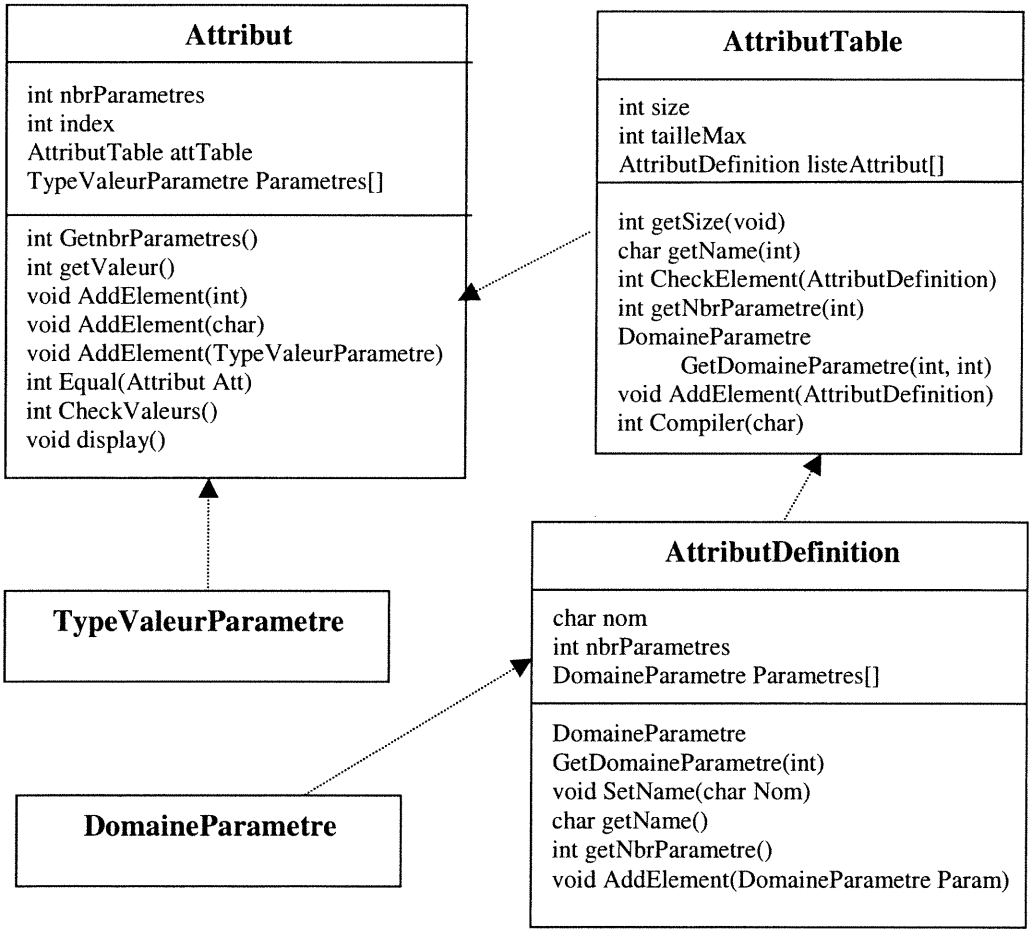


Figure 6.4 Diagramme de classes du sous-module d'attributs

6.3.3 Sous-module de variables

Une variable est caractérisée par son nom et son domaine de variables. Une variable peut prendre des valeurs de différents types tels qu'entier, chaîne de caractères ou attribut. Nous avons défini un autre type de donnée qui est *TypeValeurVariable* (Figure 6.5). Il englobe les anciens types tels qu'entier, chaîne de caractères, intervalle et en ajoutant le type attribut. Un domaine de variables modélisé par la classe d'objet *DomaineVariable* est une liste d'éléments de type *TypeValeurVariable*. La classe d'objet *Variable* concrétise une variable avec son domaine, par exemple, la variable *click* a pour domaine : Domaine(*click*)={0, 1, yes, no, Tone_of(23,4,send)}. Toutes les variables sont regroupées dans la classe d'objet *TousVariables* qui contient la méthode *Compiler(filename)* responsable de la compilation du fichier de variables.

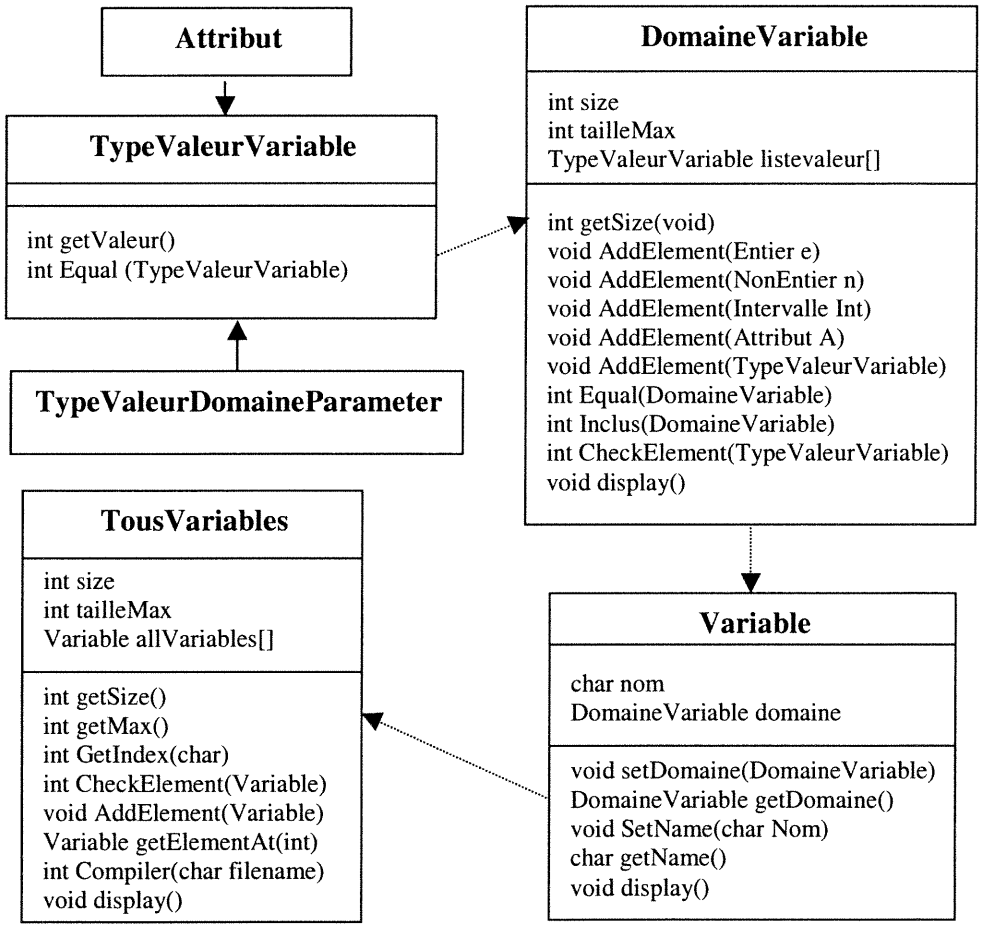


Figure 6.5 Diagramme de classes du sous-module de variables

6.3.4 Sous-module d'affectation de variables

Ce sous-module contient toutes les classes d'objets qui implémentent les affectations de variables. Une affectation de variables est composée de plusieurs affectations élémentaires modélisées par la classe d'objet *AffectationElementaire* (figure 6.6), par exemple, l'affectation suivante : A = {A_status := BUSY **And** B_signal := TONE}. Une affectation élémentaire est de la forme : *variable* := *constante* * *variable* +/- *constante* pour les variables entières et *variable* := *variable* pour les autres. La classe d'objet *Affectation* modélise donc une affectation composée de plusieurs affectations élémentaires.

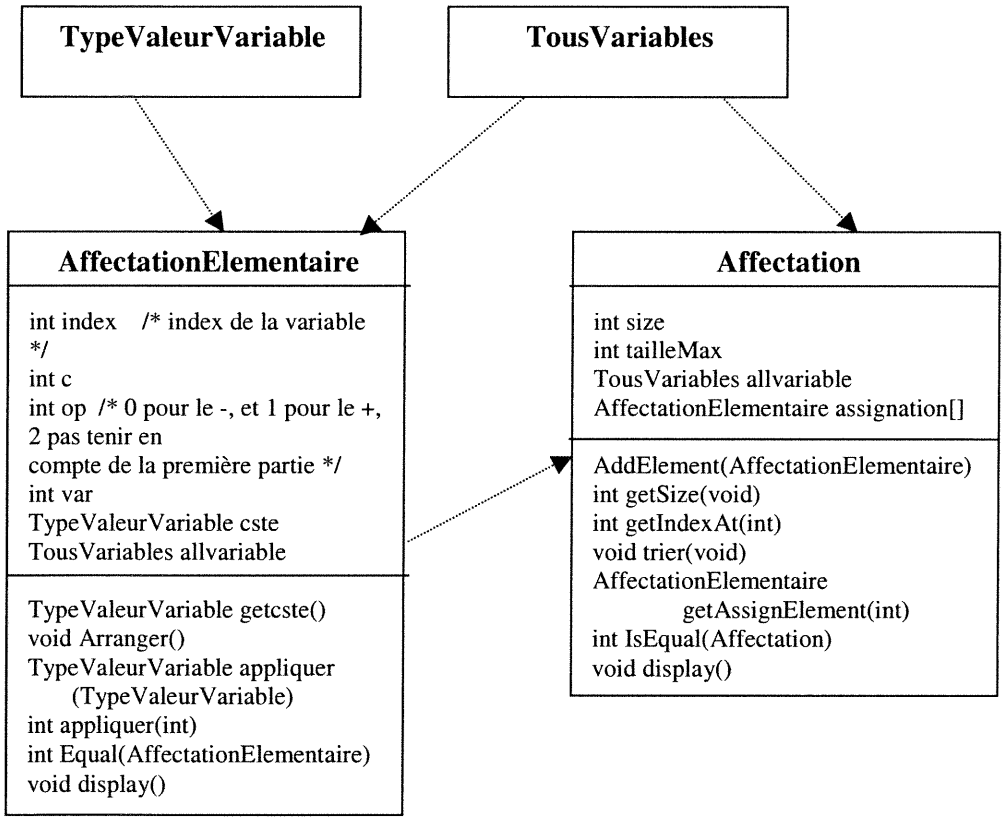


Figure 6.6 Diagramme de classes du sous-module d'affectation de variables

6.3.5 Sous-module de contraintes sur variables

Ce sous-module décrit une contrainte sur variables par la classe d'objet *ContrainteVar* (Figure 6.7). Une contrainte sur variables est modélisée par une matrice à N-dimensions constituée de bits. N étant le nombre de variables. Le nombre de bits de la matrice est le produit des dimensions des différents domaines des variables. Une case de la matrice prend la valeur vraie ou fausse selon l'intersection des différents paramètres de variables sont vrais ou faux. Par exemple, soient les variables suivantes avec leurs domaines : *Direction*{*send*, *receive*}, *State*{*on*, *off*, *idle*} et *Position*{*up*, *down*}. La matrice a trois dimensions et 12 bits de taille. La matrice de la contrainte sur variables suivante : {*Direction* = *send* **and** *State* = *idle* **and** *Position* = *down*} a le bit d'intersection des trois paramètres de variables *send*, *idle* et *down* est à vrai et les autres sont à faux.

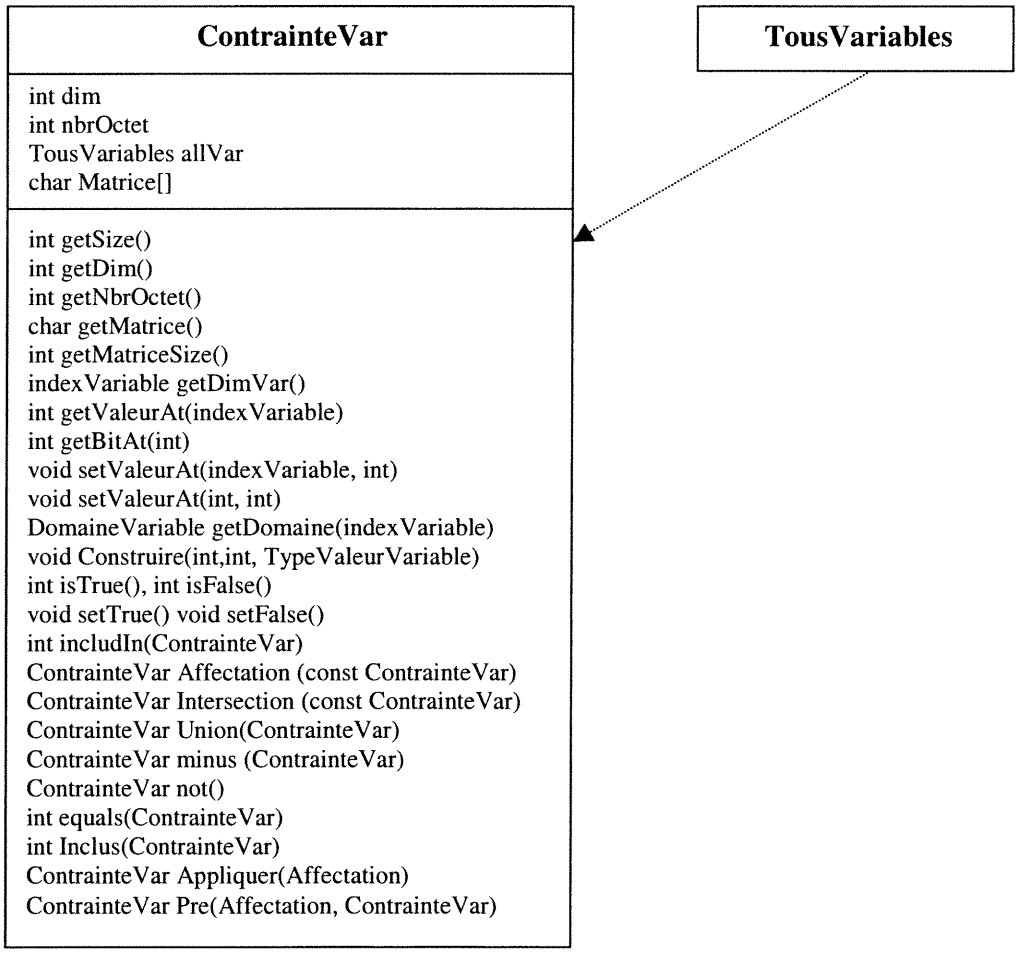


Figure 6.7 Diagramme de classes du sous-module de contraintes

6.4 Conclusion

Nous avons présenté dans ce chapitre, le module d'horloges en développant ses fondements théoriques et la description du module en terme de classes d'objets. Nous avons présenté aussi le module de variables en détaillant ces différentes sous-modules qui sont le sous-module de paramètres, le sous-module d'attributs, le sous-module de variables, le sous-module d'affectation de variables, et le sous-module de contraintes sur variables. Dans le chapitre suivant, nous allons utiliser les structures et les méthodes développées dans les modules d'horloges et de variables pour générer les actions-règles. Ensuite, nous allons détailler le sous-module d'intégration explicite en développant les structures de données du graphe GDI et en présentant les algorithmes de construction de ce graphe.

Génération des actions-règles

7.1 Introduction

Ce chapitre décrit le module de génération des actions-règles. Nous détaillons les différents sous-modules qui le compose, qui sont le sous-module de scénarios, le sous-module d'actions-règles et le sous-module d'intégration explicite. Enfin, nous présentons les principaux algorithmes de génération des actions-règles et d'intégration explicite.

7.2 Sous-module de scénarios

Un scénario est composé d'un ensemble de nœuds. Chaque nœud représente un ensemble d'actions. Une action comporte les informations suivantes: un invariant exprimé par une contrainte temporelle, une contrainte sur les variables, une garde exprimée par une contrainte temporelle, une étiquette de l'action, une affectation de variables et une affectation d'horloges. Une action est modélisée par la classe d'objet *Action*. (Figure 7.1).

La classe d'objet *Nœud* représente un nœud. L'ensemble d'actions attachées au nœud est modélisé par une liste à dimension variable. Différentes méthodes sont implémentées pour manipuler les actions d'un nœud tel que l'ajout d'une nouvelle action à la liste.

Enfin, un scénario est modélisé par la classe d'objet *Scenario*. L'ensemble de nœuds attachés au scénario est modélisé par une liste à dimension variable. Plusieurs méthodes ont été développées pour manipuler les nœuds ou les actions directement.

Pour manipuler tous les scénarios dans le système, nous les avons regroupés dans la classe d'objet *VecteurScenario*. Nous avons implémenté plusieurs méthodes

qui assurent toutes les opérations sur les scénarios tel que l'ajout d'un nouveau scénario ou la duplication d'un scénario. La méthode *compiler(nom_fichier)* de la classe d'objet *VecteurScenario* est responsable de la compilation des scénarios en faisant appel au module de compilation.

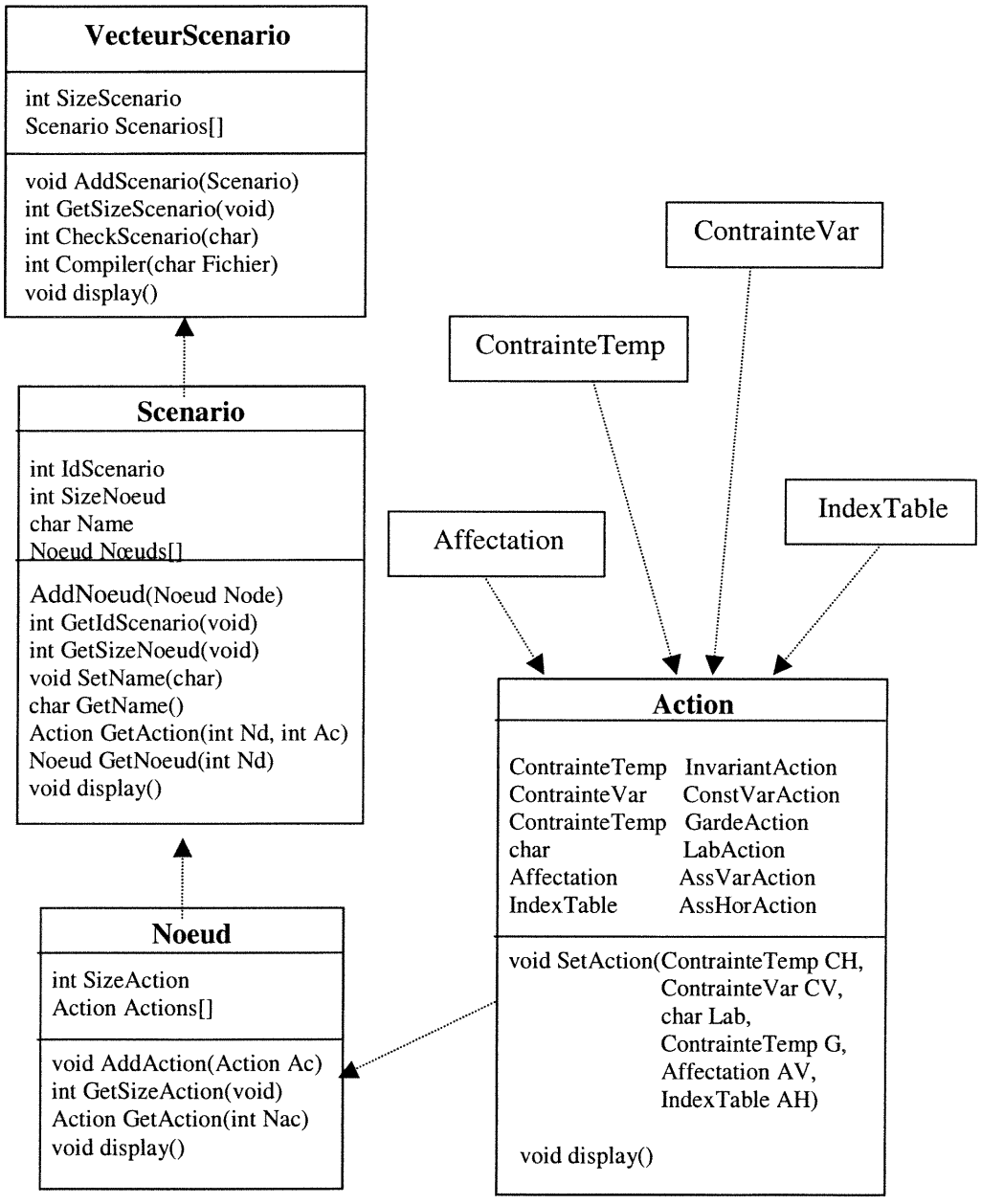


Figure 7.1 Diagramme de classes du sous-module de scénarios

7.3 Sous-module d'actions-règles

Ce sous-module permet la génération des actions-règles à partir d'un ensemble de scénarios. Cette opération consiste à calculer le contexte des nœuds principaux des scénarios et à calculer les actions-règles en se basant sur ce contexte. Les formules de calcul de contexte et de génération des actions-règles sont détaillées dans le quatrième chapitre.

7.3.1 Description du sous-module d'actions-règles

Ce sous-module modélise l'ensemble de classes qui implémentent les actions-règles. La classe d'objet *ActionRegle* (Figure 7.2) modélise une action-règle élémentaire.

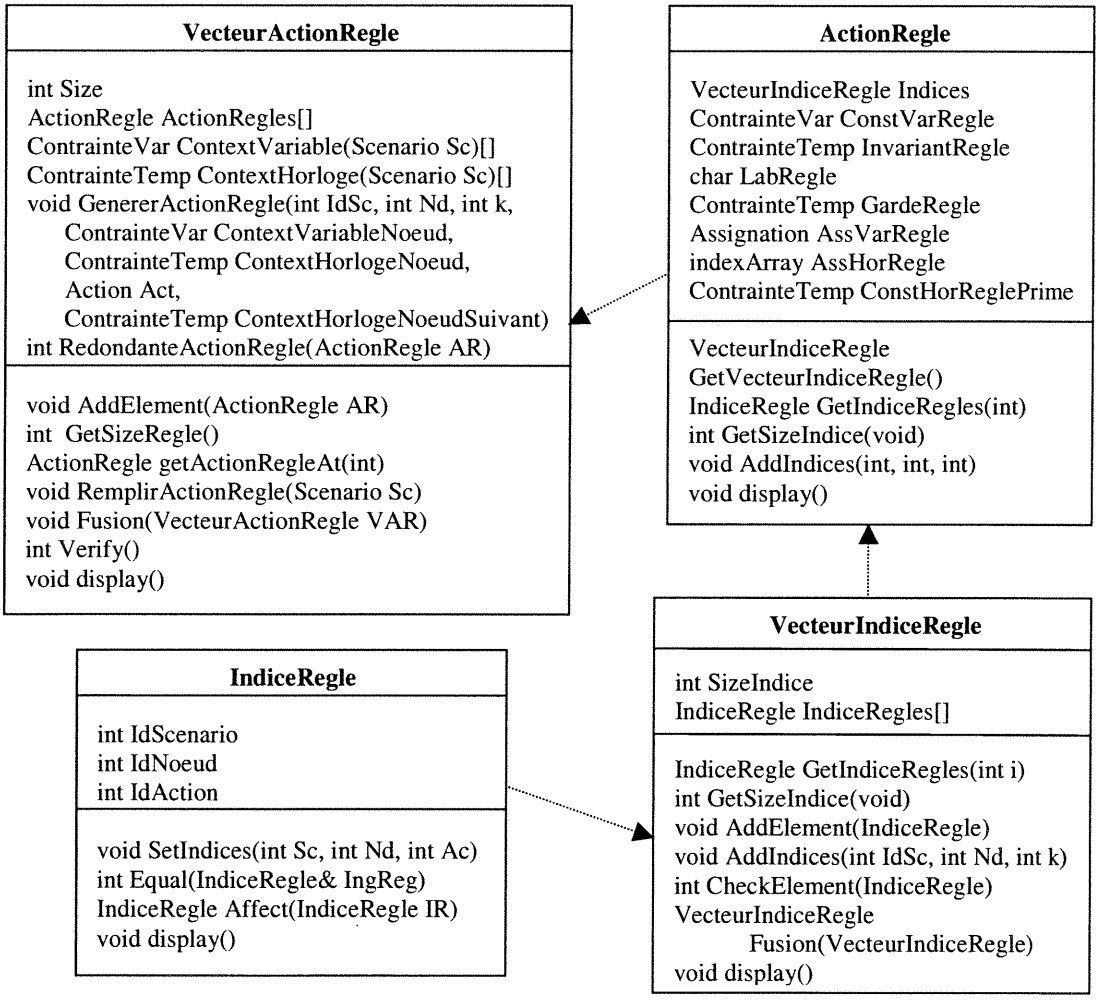


Figure 7.2 Diagramme de classes du sous-module d'actions-règles

Toutes les actions-règles sont regroupées dans la classe d'objet *VecteurActionRegle*. Cette dernière classe contient la méthode principale *RemplirActionRegle(Scenario)* responsable de la génération des actions-règles à partir d'un scénario. La méthode principale *RemplirActionRegle* fait appel à deux méthodes de calcul de contexte de variables et d'horloges. Puis, elle parcourt toutes les actions du scénario pour générer les actions-règles correspondantes. L'algorithme de la méthode est présenté dans la figure 7.3. Les classes d'objet *IndiceRegle* et *VecteurIndiceRegle* sont nécessaires pour garder l'information sur la provenance d'une action-règle et spécialement dans le cas de redondance.

```
Méthode RemplirActionRegle(Sc : Scenario)
Début
Soit ContextVariableArray : ContrainteVar   /* Déclaration du tableau de contexte de variables */
Soit ContextHorlogeArray : ContrainteTemp /* Déclaration du tableau de contexte d'horloges */
ContextVariable(Sc, ContextVariableArray)  /* Appel de la méthode de calcul du contexte de
variables */
ContextHorloge(Sc, ContextHorlogeArray) /*Appel de la méthode de calcul du contexte d'horloges*/
Pour chaque Nœud Nd ∈ Sc
        Pour chaque Action Ac ∈ Nd
                GenererActionRegle(Sc, Nd, Ac, ContextVariableArray[Nd],
                                        ContextHorlogeArray[Nd],
                                        ContextHorlogeArray[Nd+1])
        FinPour
FinPour
Fin
```

Figure 7.3 Méthode principale qui génère des actions-règles à partir d'un scénario

7.3.2 Calcul du contexte des nœuds

Le contexte de chaque nœud principal est composé d'une contrainte temporelle et d'une contrainte sur variables. Nous avons présenté la formule récurrente de calcul de contexte dans le quatrième chapitre. Nous avons séparé le calcul de contexte en deux méthodes : l'une pour le calcul de contexte de variables *ContexteVariable(Scenario)* et l'autre pour le calcul de contexte d'horloges *ContexteHorloge(Scenario)*. Les deux méthodes prennent en entrée un scénario et un

tableau vierge de contraintes correspondantes qui sera rempli en sortie par les contraintes de contexte en question. Les algorithmes des deux méthodes sont présentés dans les figures 7.4 et 7.5. Ces méthodes découlent directement des formules de calcul des contextes présentées dans [Sal 01].

```
Méthode ContexteVariable(Sc : Scenario, ContextVariableArray : ContrainteVar)
Début
/* Calcul du contexte-variable du premier nœud */
  ContextVariableArray[1] := Sc.Action(1,1).ConstVarAction
/* Calcul du contexte-variable de nœuds intermédiaire de 2 à N */
Pour chaque Nœud Nd ∈ Sc de 2 à MaxNœud
ContextVariableArray[Nd] :=
ContextVariableArray[Nd-1].Appliquer(Sc.Action(Nd-1,1).AssVarAction &
                                     Sc.Action(Nd,1).ConstVarAction
FinPour
/* Calcul du contexte-variable du nœud N+1 */
ContextVariableArray[Nd+1] :=
ContextVariableArray[Nd].Appliquer(Sc.Action(Nd,1).AssVarAction)
Fin
```

Figure 7.4 Méthode de calcul du contexte variable des nœuds

```
Méthode ContexteHorloge(Sc : Scenario, ContextHorlogeArray : ContrainteTemp)
Début
/* Calcul du contexte-horloge de nœuds intermédiaire de 1 à N */
Pour chaque Nœud Nd ∈ Sc de 1 à MaxNœud
        ContextHorlogeArray[Nd] := Sc.Action(Nd,1).InvariantAction
        Pour chaque Action Ac ∈ Nd de 2 à  MaxAction
                ContextHorlogeArray[Nd] = ContextHorlogeArray[Nd] &
                Sc.Action(Nd,Ac).InvariantAction
        FinPour
FinPour
/* Calcul de contexte-horloge du nœud N+1 */
ContextHorlogeArray[Nd+1] := (ContextHorlogeArray[Nd] &
Sc.Action(Nd,1).GardeAction).Appliquer(Sc.Action(Nd,1).AssHorAction)
Fin
```

Figure 7.5 Méthode de calcul du contexte horloge des nœuds

7.3.3 Calcul des actions-règles

Après le calcul du contexte des nœuds principaux, nous allons calculer les actions-règles. La méthode *GenererActionRegle* est responsable de générer une action-règle à partir d'une action. Les entrées de la méthode sont une action, le contexte variable du nœud courant, le contexte horloge du nœud courant et le contexte horloge du nœud suivant. Une action est identifiée par le numéro de scénario, le numéro de nœud dans le scénario et le numéro de l'action dans le nœud. En sortie de la méthode, nous obtenons une action-règle qui est ajoutée à la liste des actions-règles dans la classe d'objet *VecteurActionRegle*. L'algorithme de la méthode est présenté dans la figure 7.6. Cette méthode découle directement de la formule de la génération des actions-règles présentée dans [Sal 01].

```
Méthode GenererActionRegle(Sc : Scenario, Nd : Noeud, Ac : Action,
                           ContextVariableNoeud : ContrainteVar,
                           ContextHorlogeNoeud : ContrainteTemp,
                           ContextHorlogeNoeudSuivant : ContrainteTemp)
Début
Soit Regle : ActionRegle
/* Remplir les éléments de l'action-règle */
Regle.ConstVarRegle := ContextVariableNoeud & Ac.ConstVarAction
Regle.InvariantRegle := ContextHorlogeNoeud
Regle.LabRegle := Ac.LabAction
Regle.GardeRegle := Ac.GardeAction
Regle.AssVarRegle := Ac.AssVarAction
Regle.AssHorRegle := Ac.AssHorAction
Si Ac est l'action principale
    Regle.ConstHorReglePrime := ContextHorlogeNoeudSuivant
Sinon
    Regle.ConstHorReglePrime := (ContextHorlogeNoeud &
                                 Ac.GardeAction).Appliquer(Ac.AssHorAction)
FinSi
Regle.AjoutIndices(Sc, Nd, Ac)  /* Ajout des indices dans l'action-règle pour conserver l'action
d'origine */
AddVecteurActionRegle(Reg)  /* Ajout de l'action-règle dans le vecteur d'actions-règles */
Fin
```

Figure 7.6 Méthode de génération des actions-règles à partir des contextes et de l'action

A chaque ajout d'une action-règle dans le vecteur d'actions-règles, nous vérifions la redondance de celle-ci. Une action-règle *AR1* est redondante, s'il existe une action-règle *AR2* dans le vecteur d'actions-règles tel que les deux actions-règles vérifient toutes les conditions suivantes dans l'ordre de leurs apparitions :

1. Les étiquettes de deux actions-règles sont identiques,
2. Les affectations de variables de deux actions-règles sont identiques,
3. Les affectations d'horloges de deux actions-règles sont identiques,
4. La contrainte sur variables de *AR1* est incluse dans la contrainte sur variables de *AR2*,
5. L'invariant de *AR1* est inclus dans l'invariant de *AR2*,
6. La garde de *AR1* est incluse dans la garde de *AR2*,
7. La deuxième contrainte d'horloges de *AR1* est incluse dans la deuxième contrainte d'horloges de *AR2*.

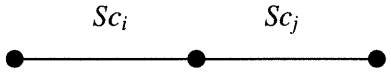
C'est la méthode *RedondanteActionRegle(ActionRegle)* qui vérifie la redondance d'une action-règle en vérifiant toutes les conditions précédentes avec toutes les actions-règles. La méthode est invoquée lors de l'ajout des indices de provenance de l'action-règle par l'intermédiaire de la méthode *AjoutIndices(Scenario, Nœud, Action)* appartenant à la classe d'objet *ActionRegle*. Dans le cas de redondance, nous éliminons l'action-règle en question en se limitant à ajouter les identificateurs du scénario, du nœud et de l'action dans *VecteurIndiceRegle* de l'ancienne action-règle. Nous gardons ainsi la trace de l'origine de l'action-règle en conservant l'information sur les actions d'origines.

7.4 Sous-module d'intégration explicite

Le sous-module de génération explicite décrit la phase de génération explicite de scénarios. Cette phase nécessite des structures de classes d'objets intermédiaires pour représenter le GDI. L'approche consiste à générer le vecteur de directives *Dir'* (Voir chapitre 4) puis générer le GDI sous forme d'une matrice. Une fois le GDI construit, nous appliquons l'algorithme de propagation des contraintes à travers les arcs du graphe. L'algorithme de propagation est présenté dans l'article [Sal 01].

7.4.1 Cohérence temporelle

Soit la directive élémentaire séquentielle Sc_i -> Sc_j qui est représentée par le GDI suivant :



Nous considérons que la directive est cohérente temporellement si le contexte du premier nœud du scénario Sc_j est inclus dans le contexte du dernier nœud de Sc_i :

$Sc_j.\varphi_r^{(1)} \subset Sc_i.\varphi_N$ avec $\varphi_N = (\varphi_r^{(N)} \wedge \phi_{a11})[\lambda_{a11}]$ est le contexte du dernier nœud,

et $\varphi_r^{(i)} = \wedge \varphi_{aik}$ $(1 \leq k \leq k_i)$ est le contexte du nœud i.

7.4.2 Génération de l'ensemble de directives *Dir'*

Nous modélisons une directive par la classe d'objet *Directive* (Figure 7.7). La classe d'objet *VecteurDirective* représente un vecteur de directives. Cette dernière classe contient la méthode *compiler(File)* qui permet de lire le fichier de directives et de le compiler en créant des instances de l'objet *Directive* et une instance de l'objet *VecteurDirective*. La méthode *DirPrime()* dans la classe d'objet *VecteurDirective* permet de générer un autre vecteur de directives *Dir'*. La méthode prend en entrée un vecteur de directives *Dir* et fournit en sortie un nouveau vecteur de directives *Dir'* après remplacement des directives qui présentent des ambiguïtés d'ordonnancement. L'algorithme de la méthode est décrit dans la figure 7.8. Après génération du vecteur de directives *Dir'*, nous vérifions la cohérence temporelle de chaque directive appartenant à *Dir'* pour s'assurer de la propagation des contraintes temporelles à travers le GDI. Dans la section suivante, nous allons présenter la construction du GDI.

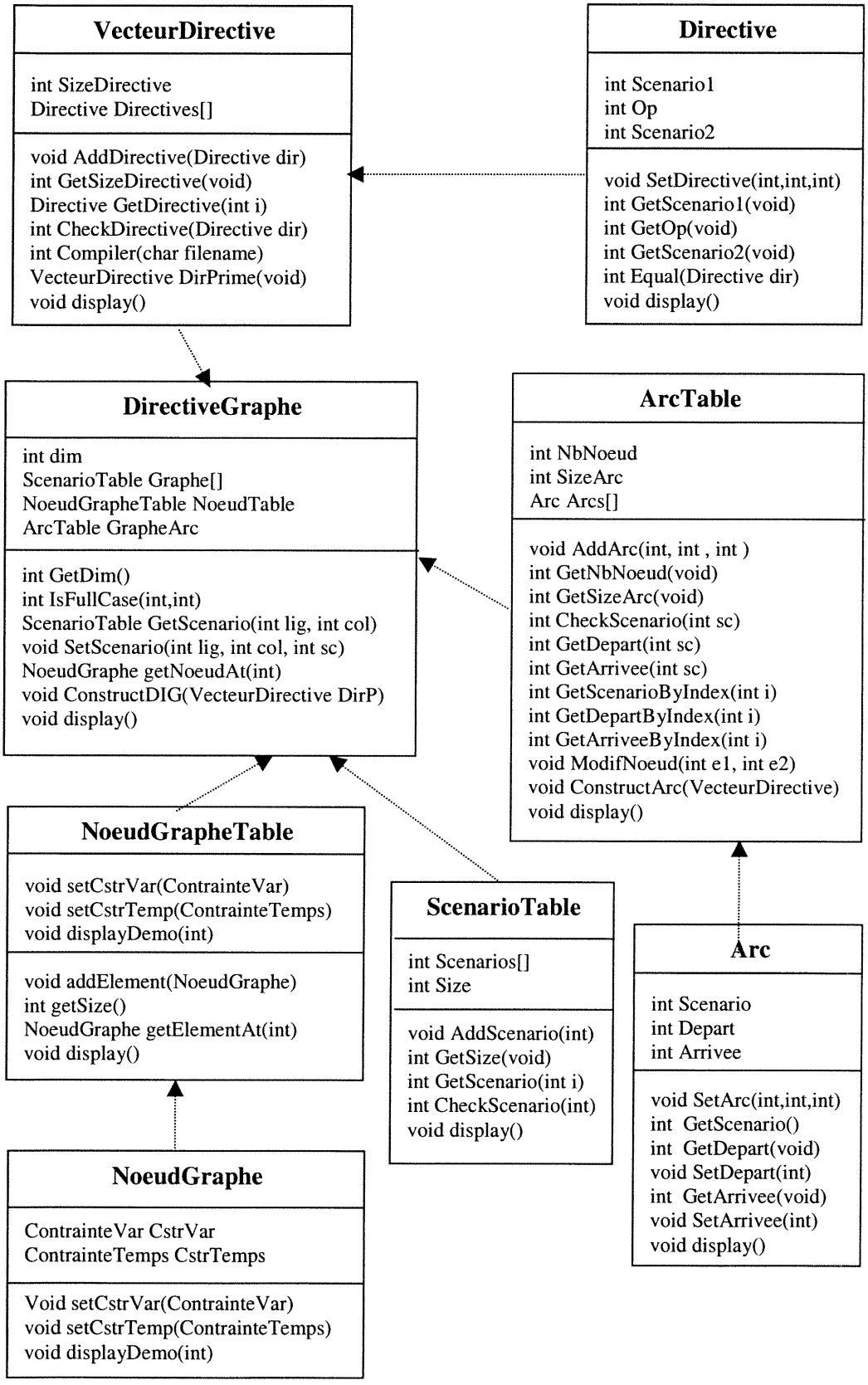


Figure 7.7 Diagramme de classes du module d'intégration explicite

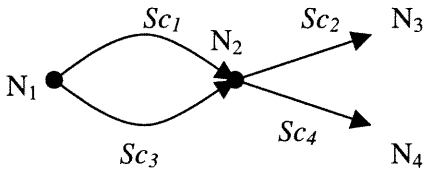
```
Méthode Génération_Dir_Prime(Dir : VecteurDirective)
Début
Soit Dir' : VecteurDirectivePrime   /* Un nouveau Vecteur de directives */
Pour chaque Directive d ∈ Dir      /* d est composé de Scenario1, Op et Scenario2 */
        Occurrence := Faux
        Pour chaque Directive d' ∈ Dir qui précède d
                Si d.Scenario1 = d'.Scenario2 ET d.Op = "->" ET d'.Op = "->" alors
                        Occurrence := Vrai
                FinSi
        FinPour
        Si occurrence = Vrai alors
                Sc' = Dupliquer(sc)
                AjouterDirective(Dir', sc', "->", d.Scenario2)
                AjouterDirective(Dir', sc', " | " , d.Scenario1)
        Sinon
                AjouterDirective(Dir', d.Scenario1, d.Op, d.Scenario2)
        FinSi
FinPour
Fin

La fonction Dupliquer() fait une copie d'un scénario.
La procédure AjouterDirective() ajoute une directive dans un VecteurDirective.
```

Figure 7.8 Méthode de génération de l'ensemble de directives *Dir'*

7.4.3 Représentation du GDI

Nous représentons le GDI par une matrice carrée. La dimension de la matrice est le nombre de nœud du graphe. Les lignes de la matrice représentent les nœuds de départs des arcs du graphe. Les colonnes représentent les nœuds d'arrivées. Les cases de la matrice représentent les arcs du graphe. Nous pouvons avoir plusieurs scénarios entre un nœud de départ et un nœud d'arrivée. Soit, par exemple, le vecteur de directives *Dir'* = { Sc_1 | Sc_3, Sc_1 -> Sc_2, Sc_3 -> Sc_4, Sc_2 | Sc_4 }. Le GDI de *Dir'* est le suivant :



La matrice du GDI est la suivante :

	N_1	N_2	N_3	N_4
N_1	_	Sc_1, Sc_3	_	_
N_2	_	_	Sc_2	Sc_4
N_3	_	_	_	_
N_4	_	_	_	_

Pour calculer le nombre de nœuds nécessaire pour la génération de la matrice, nous allons construire le graphe dans une structure intermédiaire, puis, créer et remplir la matrice du GDI à partir de cette structure. La classe d'objet *DirectiveGraphe* (Figure 7.7) est la classe principale du module de génération explicite. Elle contient la matrice carrée qui implémente le GDI. Les éléments de la matrice sont de type *ScenarioTable*. C'est cette dernière classe d'objet qui permet d'avoir plusieurs scénarios entre un nœud de départ et un nœud d'arrivée. Chaque nœud est caractérisé par une contrainte sur variables et une contrainte temporelle. Un nœud est modélisé par la classe d'objet *NoeudGraphe*. Tous les nœuds du graphe sont regroupés dans un vecteur dans la classe d'objet *NoeudGrapheTable*. La structure intermédiaire est composée de la classe d'objet *Arc* qui modélise les arcs du graphe et de la classe d'objet *ArcTable*. Cette dernière classe regroupe tous les arcs du graphe. Elle contient la méthode *construireArc(VecteurDirective)* responsable de la construction du graphe à partir de *Dir'* en créant des instances de l'objet *Arc*. Dans la figure 7.9, nous présentons l'algorithme de la méthode

principale *ConstruireGDI(VecteurDirective)* de la classe d'objet *DirectiveGraphe*. Cette méthode prend en entrée le vecteur de directives *Dir'* et fait appel à la méthode *ConstruireArc*. Cette dernière construit les arcs et calcule le nombre total de nœuds. Connaissant ce nombre, la méthode principale poursuit son exécution en créant la matrice carrée et la remplissant à partir du vecteur d'arcs.

```
Méthode ConstruireGDI(Dir' : VecteurDirective)
Début
Soit GrapheArc : ArcTable                    /* Un nouveau  tableau d'arcs */
GrapheArc.ConstruireArc(Dir')                /* Appel de la méthode ConstruireArc */
Soit dim = GrapheArc.GetNbNoeud()            /* Méthode qui retourne le nombre de nœuds */
Soit GrapheMatrice : ScenarioTable[dim * dim]      /* Construction de la matrice de dimension
dim*dim. Les éléments sont de type ScenarioTable */
/* Initialisation de la matrice */
Pour chaque i entre 1 et dim*dim
   GrapheMatrice[i] := 0
FinPour
/* Remplir la matrice à partir du tableau d'arcs */
Pour Chaque i entre 1 et  GrapheArc.GetSizeArc()  /* Renvoi le nombre des arcs totaux */
 SetScenario(GrapheArc.GetDepart(i), GrapheArc.GetArrivee(i), GrapheArc.GetScenario(i))
FinPour
/* SetScenario(Nd : entier, Na : entier, Sc : entier) permet de mettre à jour la matrice au niveau de la
ligne Nd, la colonne Na par le scénario Sc.
GetDepart(entier), GetArrivee(entier) et GetScenario(entier) permettent de récupérer le nœud de
départ, d'arrivée et le scénario correspondant */
Fin
```

Figure 7.9 Construction du GDI

Dans la figure 7.10, nous représentons l'algorithme de construction du graphe sous forme d'un vecteur d'arcs à partir du vecteur de directives *Dir'*. Nous utilisons la structure intermédiaire du graphe basé sur les arcs. Nous représentons un arc par un nœud de départ, un nœud d'arrivée et un identificateur de scénario. Cette structure d'arcs est meilleure que la structure basée sur les nœuds qui présente plus de complexité dans l'élimination des nœuds en surplus au cours d'exécution.

```
Méthode ConstruireArc(Dir' : VecteurDirective)
Début
Soit e1, e2, e3, e4 : entier                    /* Les noeuds
Soit NbNoeud = 0                                /* Le nombre total de nœuds */
   Pour chaque Directive d ∈ Dir'               /* d est composé de Scenario1, Op, Scenario2 */
        Si d.Scenario1 ∉ ArcTable
                NbNoeud  := NbNoeud + 2
                e1 := NbNoeud - 1
                e2 := NbNoeud
                AddArcTable(d.Scenario1, e1, e2);
        Sinon
                e1 = GetDepart(d.Scenario1)
                e2 = GetArrivee(d.Scenario1)
        FinSi
        Si d.Scenario2 ∉ ArcTable
                Si d.Op = "-> "
                        NbNoeud  := NbNoeud + 1
                        AddArcTable(d.Scenario1, e2, NbNoeud)
                Sinon
                        NbNoeud  := NbNoeud + 1
                        AddArcTable(d.Scenario2, e1, NbNoeud)
                FinSi
        Sinon
                e3 = GetDepart(d.Scenario2)
                e4 = GetArrivee(d.Scenario2)

                Si d.Op = "-> "
                        Si e2 < e3 ou e2 > e3           /* Détection d'un nœud dupliqué */
                           ModifNoeud(e2, e3)
                Sinon
                        Si e1 < e3 ou e1 > e3           /* Détection d'un nœud dupliqué */
                           ModifNoeud(e1, e3)
                FinSi
        FinSi
   FinPour
Fin
```

La méthode ModifNoeud(e1 : **entier**, e2 : **entier**) permet d'éliminer un nœud en surplus et de faire la propagation de la modification dans tout le tableau ArcTable qui contient tous les arcs du graphe.

Figure 7.10 Construction des arcs du GDI

7.5 Conclusion

Nous avons présenté, dans ce chapitre, le module de génération des actions-règles en détaillant ces différents sous-modules qui sont le sous-module de scénarios, le sous-module d'actions-règles et le sous-module d'intégration explicite. Chaque action-règle résultat est indépendante du contexte et comporte toute l'information sur l'état du système et sa condition d'activité avant et après l'exécution. Une fois toutes les actions-règles obtenues, nous pouvons générer les places et les transitions de l'automate en appliquant les algorithmes correspondants. Dans le chapitre suivant, nous allons présenter l'interface de l'application et les grammaires des langages d'acquisition des entrées du système. Enfin, nous présentons un exemple d'application complet.

Interface et compilation

8.1 Introduction

Ce chapitre décrit l'interface de l'application ainsi que la description des langages des grammaires d'acquisition des entrées du système. Dans la deuxième section, nous décrivons l'interface de l'outil SCENA. Dans la troisième section, nous présentons en détail les outils de génération automatique de code Bison et Flex et nous décrivons les grammaires des langages d'acquisition des attributs, des variables, des scénarios et des directives. Enfin, nous présentons un exemple d'application complet et nous proposons dans la conclusion les points forts de l'outil et les améliorations futures proposées.

8.2 Description de l'outil SCENA

L'outil que nous avons développé vise l'automatisation de l'activité de spécification d'un système réactif temps-réel. Le menu principal (Figure 8.1) donne accès à toutes les fonctionnalités de l'outil. Cinq principaux sous-menus sont accessibles à partir du menu principal. L'ordre d'apparition des sous-menus dans le menu principal est important. Il permet de suivre étape par étape le déroulement des calculs depuis l'acquisition des domaines et des scénarios en passant par la génération des actions-règles jusqu'à la génération des places et des transitions de l'automate. Voici les cinq sous-menus:

- Gestion des domaines: ce menu propose l'acquisition et l'affichage des attributs et des variables.
- Gestion des scénarios: ce menu permet l'acquisition et l'affichage des scénarios et des directives.
- Gestion des actions-règles : ce menu permet la génération des actions-règles et leurs consultations.

- Gestion des places : ce menu propose la génération des places ainsi que leur consultation et leur enregistrement dans un fichier.
- Gestion des transitions : ce menu propose la génération des transitions ainsi que la consultation et l'enregistrement de ces transitions dans un fichier.

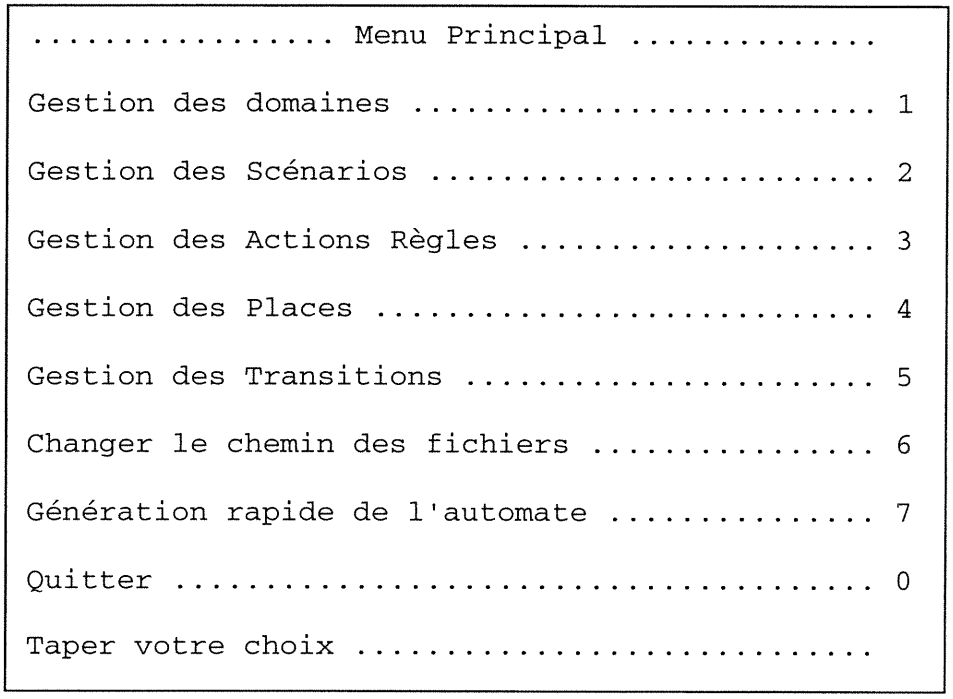


Figure 8.1 Menu principal de l'environnement SCENA

Le menu principal offre aussi une génération rapide de l'automate en passant par toutes les fonctionnalités automatiquement. Dans les sections suivantes, nous allons détailler les fonctionnalités des différents sous-menus.

8.2.1 Menu de gestion des domaines

Le menu de gestion des domaines (Figure 8.2) propose l'acquisition des attributs et des variables avec leurs domaines. Pour ce faire, nous saisissons les noms des fichiers textes décrivant les attributs et les variables avec leurs domaines dans un langage bien défini. La grammaire de ce langage sera détaillée ultérieurement. Une fois les noms saisis, une étape de compilation des fichiers va se déclencher. Si la

phase d'analyse lexicale et syntaxique ne génère aucune erreur, les structures internes correspondantes seront remplis et nous pouvons visualiser les attributs et les variables avec leurs domaines.

```
.............. Menu des Domaines .............
Acquisition des Domaines ..................... 1
Affichage des Domaines ....................... 2
Initialiser le Domaine ....................... 3
Retour au Menu Principal ..................... 0
Taper votre choix ............................
```

Figure 8.2 Menu de gestion des domaines

8.2.2 Menu de gestion des Scénarios

Le menu de gestion des scénarios (Figure 8.3) propose l'acquisition des scénarios et des directives en saisissant les noms des fichiers textes décrivant les scénarios et les directives dans un langage bien défini que nous allons détailler ultérieurement. Une fois les noms saisis, une étape de compilation des fichiers va se déclencher. Si la phase d'analyse lexicale et syntaxique ne génère aucune erreur, les structures internes correspondantes seront remplies et nous pouvons visualiser les scénarios et les directives. Si l'intégration est implicite, nous n'avons pas à saisir les directives. Dans le cas d'intégration explicite le fichier des directives doit être non vide.

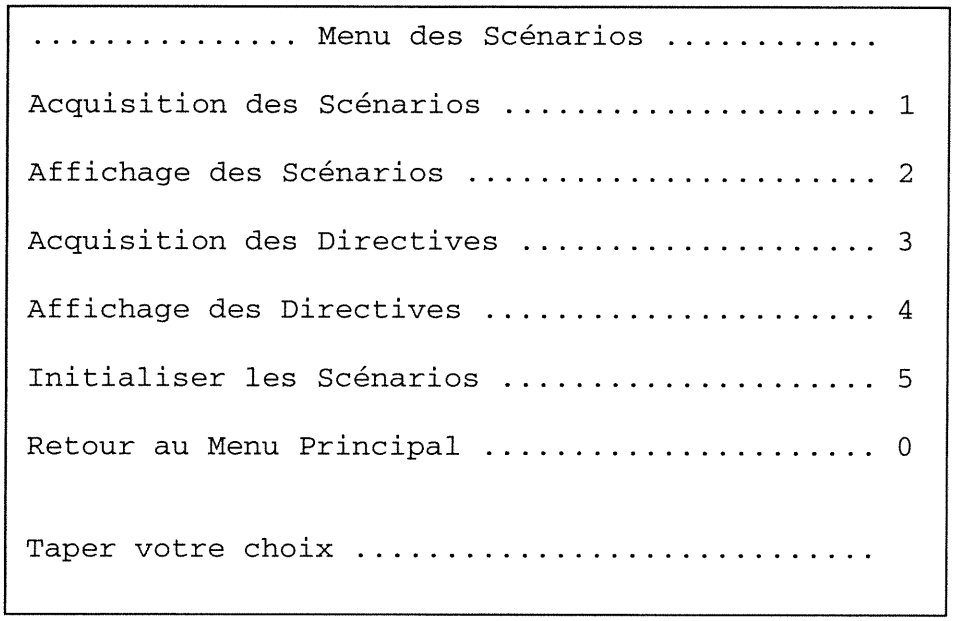


Figure 8.3 Menu de gestion des scénarios

8.2.3 Menu de gestion des Actions-règles

Le menu de gestion des actions-règles (Figure 8.4) propose la génération des actions-règles en sélectionnant quelques scénarios ou tous. Le menu permet encore de visualiser les actions-règles résultantes du traitement.

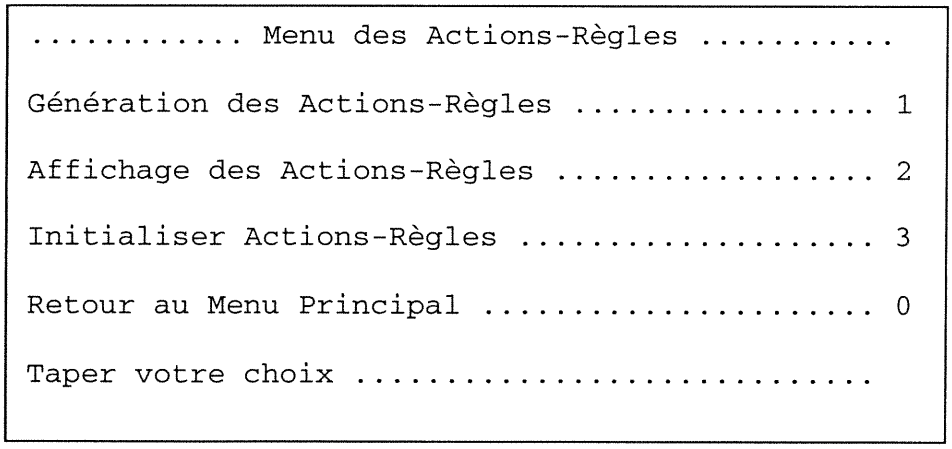


Figure 8.4 Menu de gestion des actions-règles

8.2.4 Menu de gestion des places

Le menu de gestion des places (Figure 8.5) est dédié à la génération des places de l'automate. Nous pouvons procéder à la génération des places de deux façons. La première consiste à générer directement les places de l'automate sans rentrer dans les détails de calcul sur les classes (places initiales). Toutefois, des messages sont émis pour avertir l'utilisateur du succès de chaque étape. La deuxième façon consiste à suivre pas à pas la génération des places en lançant les instructions une à une dans l'ordre d'apparition dans le menu. Cette dernière façon est très utile pour retracer l'action-règle d'origine d'une place ou d'une transition. Dans la section suivante, nous allons présenter le sous-menu de génération des places pas à pas.

```
.............. Menu des Places ................

Génération Rapide des Places  ................. 1

Génération pas à pas des Places  .............. 2

Affichage des Places ........................... 3

Enregistrer les Places dans un ficher ......... 4

Retour au Menu Principal ....................... 0


Taper votre choix ..............................
```

Figure 8.5 Menu de gestion des places

8.2.4.1 Menu de génération pas à pas

Le menu de génération pas à pas (Figure 8.6) est invoqué à partir de la deuxième option du menu de gestion des places intitulé *Génération pas à pas des Places*. La construction des places de l'automate passe par plusieurs étapes. La première étape est celle de l'extraction des classes à partir des actions-règles. L'étape suivante est celle du partitionnement pour générer la partition initiale.

Ensuite, nous procédons à la minimisation des classes. Enfin, la construction des places de l'automate.

```
..... Menu de génération pas à pas .....
Extraction des Classes ..................... 1
Affichage des Classes ...................... 2
Génération de la Partition Initiale ........ 3
Affichage de la Partition Initiale ......... 4
Minimisation des Classes ................... 5
Affichage des Classes Minimisées............ 6
Construction des Places de l'automate....... 7
Affichage des Places de l'automate.......... 8
Enregistrer les Classes dans un fichier ..... 9
Retour au Menu des places .................. 0
Taper votre choix ..........................
```

Figure 8.6 Menu de génération pas à pas

8.2.5 Menu de gestion des transitions

Le menu de gestion des transitions (Figure 8.7) est dédié à la génération des transitions de l'automate. Le menu offre la possibilité de visualiser les transitions et de les enregistrer dans un fichier.

Nous avons présenté toutes les fonctionnalités de l'outil SCENA avec la description de tous les menus. Dans les sections suivantes, nous allons décrire les phases de compilation ainsi que les grammaires utilisées pour compiler les fichiers d'entrées du système.

```
..... Menu des Transitions .....

Transition de l'automate ..... 1

Affichage des Transitions ..... 2

Enregistrer les Transitions dans un ficher .... 3

Retour au Menu Principal ..... 0

Taper votre choix .....
```

Figure 8.7 Menu de gestion des transitions

8.3 Module de compilation

La syntaxe d'un langage est décrite par une grammaire qui représente un ensemble de règles définissant toutes les constructions valides pouvant être acceptées dans le langage. Une grammaire est définie par un ensemble de terminaux qui sont des chaînes de caractères appartenant au langage, un ensemble de non terminaux qui ne sont pas inclus dans le langage lui-même, mais ils sont utilisés pour représenter des définitions intermédiaires et un ensemble de productions qui définissent les non terminaux dans la grammaire. Elles sont de la forme E -> F ou E est un non terminal et F est une séquence de terminaux et / ou non terminaux. Un élément de l'ensemble de non terminaux est spécifié comme l'élément de démarrage. Deux règles sont à respecter dans une grammaire.

- Chaque non terminal doit apparaître à gauche du signe '->' dans au moins une production
- L'élément de démarrage ne doit pas apparaître dans le membre de droite d'une production.

8.3.1 Activité de conception d'un langage

La conception d'un langage est une activité difficile. Elle nécessite la prise en considération de plusieurs critères qui peuvent guider la conception d'un langage. Parmi ces critères :

- L'uniformité qui se traduit par le fait que les caractéristiques similaires doivent agir de la même manière et que les caractéristiques non similaires ne doivent pas agir de la même manière. Par exemple, dans le langage Pascal, l'instruction de contrôle *while* utilise des *begin* et *end* alors que *repeat* ne les utilise pas.
- La clarté : le langage doit utiliser une notation qui permet aux concepteurs de comprendre facilement et précisément la nature des traitements.
- La simplicité : tout langage facile à apprendre est qualifié de simple. Ce critère chevauche celui d'uniformité qui tend à éliminer les cas particuliers.

- La sûreté : elle se traduit par la mise en oeuvre de mécanismes qui empêcheraient le programme d'agir de manière désastreuse. Pour ce faire, la conception doit prendre en considération l'origine des erreurs usuelles d'écriture, dans le but de les combattre, et doit également fournir un moyen efficace de détection d'erreurs.

Nous distinguons plusieurs types de grammaire selon la forme de leurs règles de production. Nous allons présenter les deux grammaires les plus utilisées qui sont les grammaires régulières et les grammaires non textuelles. Les grammaires régulières ont des règles de production de la forme suivante.

- E -> T avec E est un non terminal et T est un ensemble de terminaux.
- E ->T F avec E et F sont deux non terminaux et T est un terminal.

Cette grammaire génère des expressions régulières simples qui limite l'étendue des langages réguliers. En effet il est impossible de générer la chaîne a^nb^n avec n entier à l'aide d'une grammaire régulière.

Les grammaires non contextuelles permettent de générer entre autres les chaînes a^nb^n. La forme de leurs règles est la suivante :

- E ->F J avec E, F et J sont des non terminaux.
- E -> T avec E est un terminal et T une chaîne de terminaux.

Les grammaires non contextuelles sont celles qui sont les plus utilisées pour décrire la structure des langages de programmation les plus connus. Les grammaires contextuelles se caractérisent par le fait que leurs parties gauches de leurs productions peuvent contenir plusieurs non terminaux au contraire aux langages non contextuels qui ont seulement un seul non terminal dans la partie gauche de leurs productions. En réalité, les grammaires des langages de programmation sont contextuelles, mais les concepteurs des langages essayent de se ramener à une grammaire non contextuelle renforcée par les actions sémantiques pour lever les ambiguïtés qui peuvent régissent lors du choix de la production à consommer.

8.3.2 Description des outils de développement

La construction d'analyseurs lexicaux et syntaxiques d'un compilateur peut être largement automatisée au moyen de Flex, pour l'analyse lexicale, et Bison, pour l'analyse syntaxique. Ces outils acceptent, sous un format relativement naturel, la description formelle des aspects lexicaux ou syntaxiques d'un langage source. L'utilisateur peut compléter ces analyseurs au moyen d'actions sémantiques. En choisissant, Flex et Bison, nous pouvons rédiger les actions sémantiques requises en C++.

8.3.2.1 Description de Flex

FLex produit des analyseurs lexicaux reconnaissant des expressions régulières décrites dans un fichier. Le but de l'analyse lexicale est de transformer une suite de symboles en terminaux (un terminal peut être, par exemple, un nombre, un identificateur, etc.). Une fois cette transformation effectuée, la main est repassée à l'analyseur syntaxique (Figure 8.1). Le but de l'analyseur lexical est donc de consommer des symboles et de les fournir à l'analyseur syntaxique.

8.3.2.2 Description de Bison

Bison permet de produire des analyseurs syntaxiques selon une grammaire non contextuelle décrite dans un fichier. En fait, il est même possible, à l'aide de Bison, de produire des interpréteurs ou des compilateurs complets, c'est-à-dire comportant en plus l'analyse sémantique et l'interprétation et/ou la génération de code. Bison

s'utilise la plupart du temps conjointement à Flex. Flex fournit les unités lexicales à Bison qui se charge d'analyser leur organisation syntaxique et qui, au fur et à mesure de la reconnaissance, est capable d'exécuter certaines actions sémantiques.

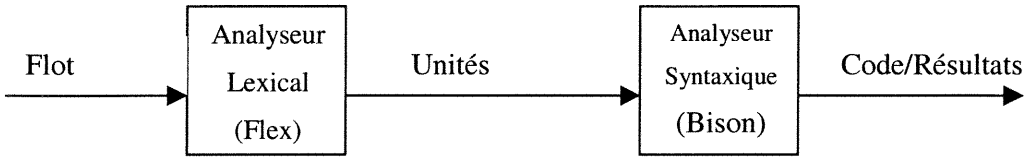


Figure 8.8 Principe de fonctionnement de Flex et Bison

8.3.3 Écriture des grammaires et implémentation

Nous avons quatre types d'entrées dans le système, les attributs, les variables, les scénarios et les directives. Nous avons défini quatre grammaires, une pour chaque type dont les fichiers d'acquisition portent des extensions différentes. Les extensions *.att*, *.var*, *.sce et .dir* sont utilisées respectivement pour les fichiers d'acquisition des attributs, des variables des scénarios et des directives. Les grammaires développées sont présentées dans l'annexe A et des exemples de leurs fichiers seront aussi présentés dans l'exemple d'application plus loin dans ce chapitre.

8.3.3.1 Description d'une expression logique

Une contrainte sur variables est exprimée par une expression logique dont la grammaire est la suivante :

expression -> *expression* **or** *and_expression*
expression -> *and_expression*
and_expression -> *and_expression* **and** *feuille*
and_expression -> *feuille*
feuille -> '(' *expression* ')'
feuille -> **identificateur '=' entier** /* L'identificateur représente une variable */
feuille -> **identificateur '<' entier**
feuille -> **identificateur '>' entier**
feuille -> **identificateur '<=' entier**

feuille -> **identificateur '>=' entier**

feuille -> **identificateur '!=' entier**

feuille -> **identificateur '=' identificateur**

feuille -> **identificateur '='** *attribut*

Pour évaluer l'expression logique d'une contrainte sur variables, nous avons besoin de construire un arbre binaire dont les nœuds internes sont les opérateurs logiques et les feuilles sont les variables, les entiers, les chaînes de caractères ou les attributs. Nous avons regroupé les deux types de nœuds dans un nouveau type *TypeNoeud* (Figure 8.9). La classe d'objet *TypeSimple* représente les feuilles de l'arbre et la classe d'objet *NoeudContrainte* représente les nœuds internes. L'évaluation de l'arbre donne lieu à la matrice de bits de la contrainte sur variables.

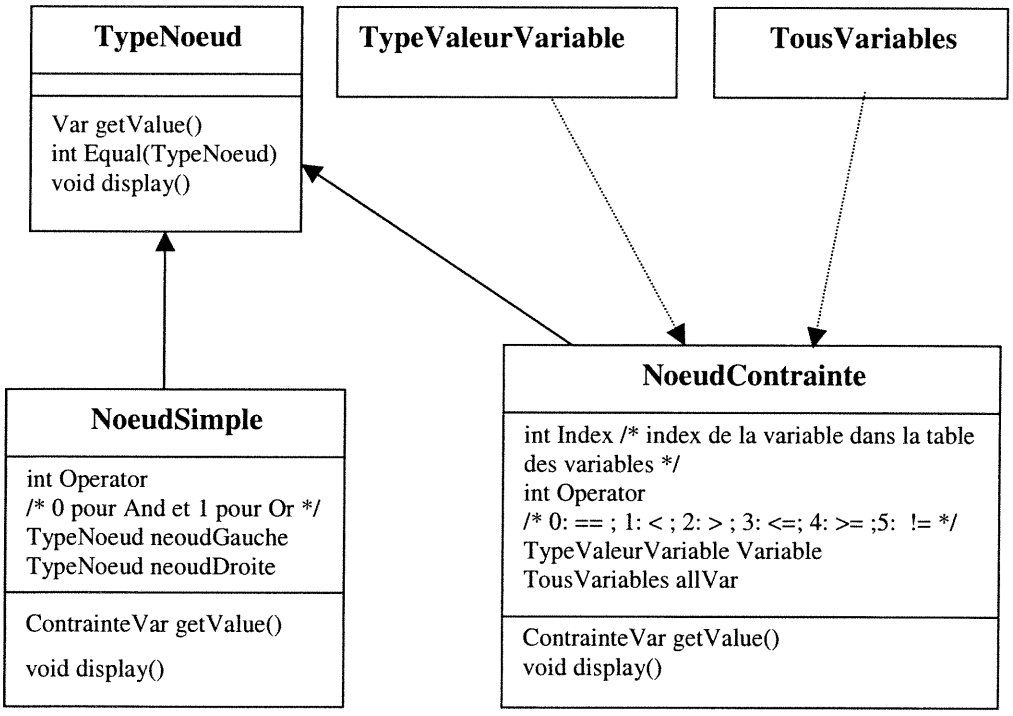


Figure 8.9 Structure de l'arbre binaire

8.3.3.2 Gestion des erreurs dans les compilateurs

Lorsqu'une erreur est rencontrée, l'analyse s'arrête et affiche un message d'erreur expliquant la nature de l'anomalie et le numéro de la ligne dans le fichier texte. Flex et Bison gèrent automatiquement les erreurs lexicales et syntaxiques en appelant une fonction spécifique en cas d'erreur. Pour pousser la détection d'erreurs, nous ajoutons des règles de production simulant les erreurs les plus fréquentes produites par les concepteurs. Dans le cas de succès de la phase de compilation, le compilateur donne la main à la méthode appelante après mise à jour des structures internes du système.

8.4 Exemple d'application

Nous avons choisi l'exemple du commutateur téléphonique à deux horloges. La description du fonctionnement du commutateur est la suivante :

Lorsque le poste **A** est non occupé, si le commutateur reçoit le message PICKUP(A) alors il envoie la tonalité au poste **A**. Si le poste **A** appelle **B** avant 30 unités de temps alors le commutateur déclenche la sonnerie lorsque le poste **B** est libre. Mais si le poste **A** ne fait rien pendant 30 unités de temps alors le commutateur lui envoie BUSY_TONE(A). Lorsque le terminal du poste **B** est entrain de sonner, si celui-ci décroche avant 60 unités de temps, le commutateur établit la communication sinon il l'annule et envoie BUSY_TONE(A).

Les attributs, les variables, les étiquettes et les horloges du commutateur sont ceux présentés dans la figure 4.1 et la figure 4.2. Le scénario du commutateur écrit sous le format du langage d'acquisition des scénarios est décrit dans la figure 8.10. Dans la figure 8.11, nous présentons les actions-règles générées par l'environnement SCENA.

```
Commutateur_telephonique
begin_scenario Sc_phone
   begin_node n0
      begin_action a01
         invariant: true
         label: pickup_A
         variable_constraint:A_status=IDLE and A_signal=NONE
         temporal_guard: true
         variables_assignment: A_status := BUSY
         clocks_reset: {h2}
      end_action
   end_node
   begin_node n1
      begin_action a10
         invariant: h2 = 0
         label: send_tone_A
         temporal_guard: h2 = 0
         variables_assignment: A_signal := TONE
         clocks_reset: {h1}
      end_action
   end_node
   begin_node n2
      begin_action a20
         invariant: h1 <= 30
         label: dialing_B
         variable_constraint:
         temporal_guard: h1 < 30
         variables_assignment: A_signal := DIALING(B)
         clocks_reset: {h2}
      end_action
      begin_action a21
         invariant: h1 <= 30
         label: busy_tone_A
         variable_constraint:
         temporal_guard: h1 = 30
         variables_assignment: A_signal := BUSY_TONE
         clocks_reset: {}
      end_action
   end_node
   begin_node
      begin_action
         invariant: h2 = 0
         label: ring_A_B
         variable_constraint:B_status=IDLE and B_signal=NONE
         temporal_guard: h2 = 0
         variables_assignment:A_signal:=ECHO_RING(B),B_signal:=RING(A),B_status:=BUSY
         clocks_reset: {h2}
      end_action
   end_node
   begin_node
      begin_action
         invariant: h2 <= 60
         label: pickup_B
         variable_constraint: true
         temporal_guard: h2 < 60
         variables_assignment:A_signal:=TALKING(B),B_signal := TALKING(A)
         clocks_reset: {}
      end_action
      begin_action
         invariant: h2 <= 60
         label: busy_tone_A
         variable_constraint: true
         temporal_guard: h2 = 60
        variables_assignment: A_signal:=BUSY_TONE,B_signal:=NONE,B_status:=IDLE
         clocks_reset: {}
      end_action
   end_node
end_scenario
```

Figure 8.10 Le scénario du commutateur téléphonique

```
******** Debut Action-Regle ********
Scenario: 0 Noeud: 0 Action: 0
Contrainte Variable:
{IDLE,NONE,BUSY,NONE}
{IDLE,NONE,BUSY,TONE}
{IDLE,NONE,BUSY,BUSY_TONE}
{IDLE,NONE,BUSY,RING(A)}
{IDLE,NONE,BUSY,TALKING(A)}
{IDLE,NONE,IDLE,NONE}
{IDLE,NONE,IDLE,TONE}
{IDLE,NONE,IDLE,BUSY_TONE}
{IDLE,NONE,IDLE,RING(A)}
{IDLE,NONE,IDLE,TALKING(A)}
Invariant: True
Lab = pickup_A
Garde: True
Affectation Variable: Vecteur
d'assignation
[ assignation : A_status <- BUSY ]
Assignation Horloge: 1
Contrainte Horloge: ( h2 == 0 )
********* Fin Action-Regle **********
******** Debut Action-Regle *********
Scenario: 0 Noeud: 1 Action: 0
Contrainte Variable:
{BUSY,NONE,BUSY,NONE}
{BUSY,NONE,BUSY,TONE}
{BUSY,NONE,BUSY,BUSY_TONE}
{BUSY,NONE,BUSY,RING(A)}
{BUSY,NONE,IDLE,BUSY_TONE}
Invariant: ( h2 == 0 )
Lab = send_tone_A
Garde: ( h2 == 0 )
Affectation Variable: Vecteur
d'assignation
[ assignation : A_signal <- TONE ]
Assignation Horloge: 0
Contrainte Horloge: ( h1 <= 30 )
************* Fin Action-Regle ******
************* Debut Action-Regle ****
Scenario: 0 Noeud: 2 Action: 0
Contrainte Variable: 00000000
00111111
{BUSY,TONE,BUSY,NONE}
{BUSY,TONE,BUSY,TONE}
{BUSY,TONE,BUSY,BUSY_TONE}
{BUSY,TONE,IDLE,BUSY_TONE}
{BUSY,TONE,IDLE,RING(A)}
{BUSY,TONE,IDLE,TALKING(A)}
Invariant: ( h1 <= 30 )
Lab = dialing_B
Garde: ( h1 < 30 )
Affectation Variable: Vecteur
d'assignation
[assignation:A_signal <- DIALING(B)]
Assignation Horloge: 1
Contrainte Horloge: ( h2 == 0 )
********* Fin Action-Regle **********
```

```
************* Debut Action-Regle ****
Scenario: 0 Noeud: 2 Action: 1
Contrainte Variable:
[assignation : A_signal<- BUSY_TONE]
Assignation Horloge:
Contrainte Horloge: ( h1 == 30 )
************* Fin Action-Regle ******
************* Debut Action-Regle ****
Scenario: 0 Noeud: 3 Action: 0
Contrainte Variable:
{BUSY,DIALING(B),IDLE,NONE}
Invariant: ( h2 == 0 )
Lab = ring_A_B
Garde: ( h2 == 0 )
Affectation Variable: Vecteur
d'assignation
[ assignation : A_signal <-
ECHO_RING(B) ]
[ assignation : B_status <- BUSY ]
[ assignation : B_signal <- RING(A) ]
Assignation Horloge: 1
Contrainte Horloge: ( h2 <= 60 )
************* Fin Action-Regle ******
************* Debut Action-Regle ****
Scenario: 0 Noeud: 4 Action: 0
Contrainte Variable:
{BUSY,ECHO_RING(B),BUSY,RING(A)}
Invariant: ( h2 <= 60 )
Lab = pickup_B
Garde: ( h2 < 60 )
Affectation Variable: Vecteur
d'assignation
[ assignation : A_signal <-
TALKING(B) ]
[ assignation : B_signal <-
TALKING(A) ]
Assignation Horloge:
Contrainte Horloge: ( h2 < 60 )
************* Fin Action-Regle ******
************* Debut Action-Regle ****
Scenario: 0 Noeud: 4 Action: 1
Contrainte Variable:
{BUSY,ECHO_RING(B),BUSY,RING(A)}
Invariant: ( h2 <= 60 )
Lab = busy_tone_A
Garde: ( h2 == 60 )
Affectation Variable: Vecteur
d'assignation
[ assignation : A_signal <- BUSY_TONE
]
[ assignation : B_status <- IDLE ]
[ assignation : B_signal <- NONE ]
Assignation Horloge:
Contrainte Horloge: ( h2 == 60 )
********* Fin Action-Regle **********
```

Figure 8.11 Les actions-règles du commutateur téléphonique

L'automate résultant de la spécification formelle générée par SCENA est sous un format textuel. La figure 8.12 ci dessous représente l'automate résultat sous un format graphique.

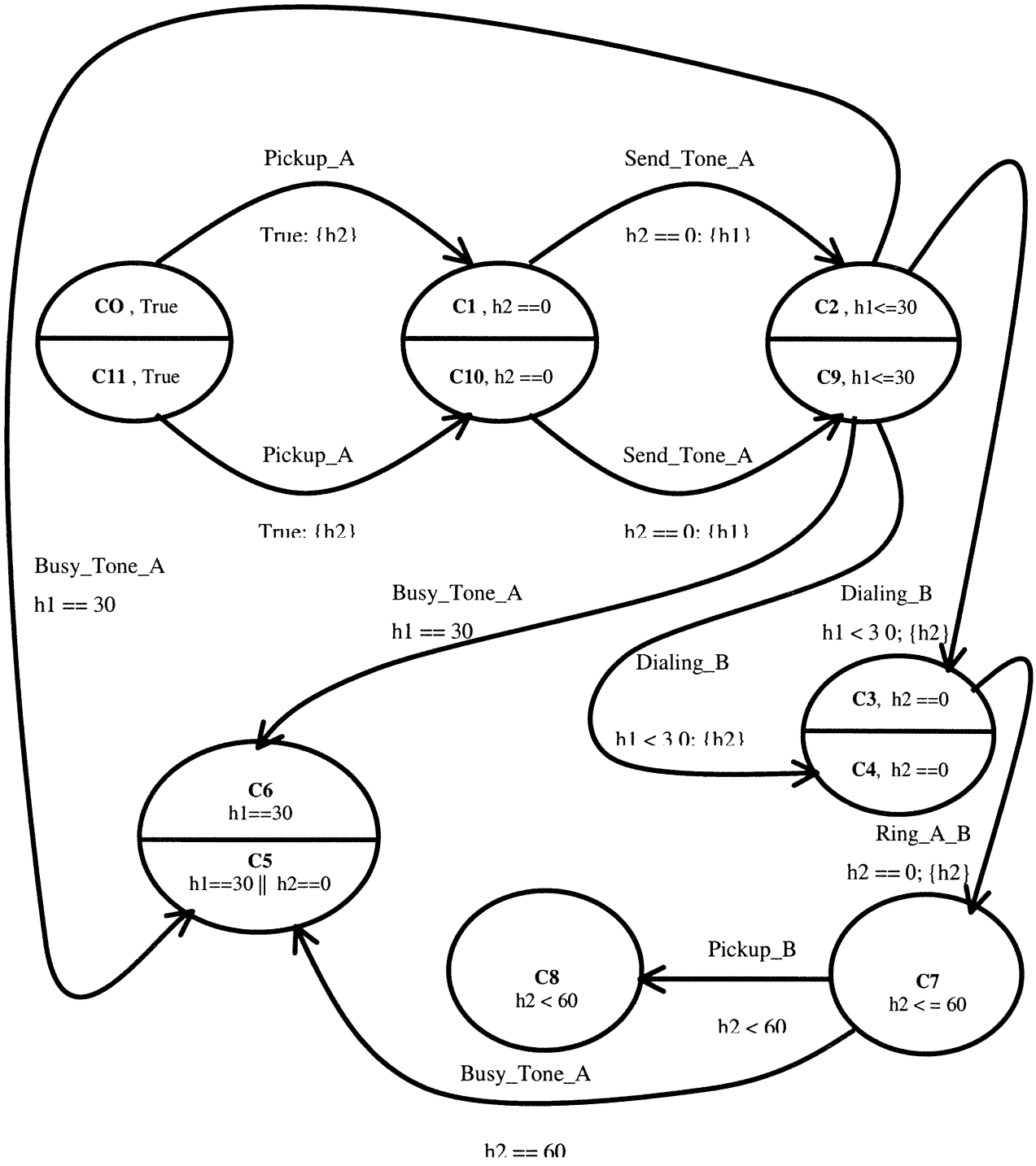


Figure 8.12 L'automate du commutateur téléphonique

8.5 Conclusion et améliorations futures

Dans ce chapitre, nous avons décris l'interface de l'outil SCENA. Ce dernier permet de générer des automates temporisés étendus à partir d'un ensemble de scénarios et de directives le cas échéant. Cependant quelques améliorations futures seront proposées afin de rendre l'outil plus mature et plus convivial. Trois types d'améliorations sont envisagés dans cette perspective :

1. Ajout d'autres fonctionnalités à l'outil tels que l'affichage des contraintes sur variables sous forme d'expressions logiques et le développement d'un module de simulation intégré au système.
2. Développement d'une interface ergonomique permettant l'acquisition interactive des scénarios et la représentation graphique des automates.
3. Amélioration de la flexibilité, de la souplesse d'utilisation et de la performance (temps d'exécution, optimisation).

Pour des fins de flexibilité et de souplesse nous prévoyons intégrer de l'aide et de guidance dans l'exécution des fonctionnalités de l'outil. Nous prévoyons également de faire tester l'outil par des experts du domaine de validation des protocoles. Ces derniers vont détecter les anomalies et les conflits à corriger et vont proposer leurs suggestions d'amélioration.

Conclusion

Lors du développement des systèmes réactifs temps-réel, les concepteurs passent directement de la spécification informelle à l'implémentation. Ce passage est dû à l'absence d'outils de spécification formelle, réduisant ainsi la détection d'erreurs et augmentant les coûts. La génération automatique d'un prototype du système peut être très utile dans le reste des phases de cycle de vie de développement tels que la vérification, la simulation ou encore le test.

L'approche scénario développée dans [Sal 01] permet de synthétiser un prototype du système, sous forme d'un automate temporisé [Alu 94], à partir d'un ensemble de scénarios. Un scénario est une description partielle du système modélisant une séquence d'interactions entre celui-ci et l'environnement. L'extraction des scénarios se fait à partir des exigences informelles.

Dans le cadre du projet SCENA, nous avons mis en œuvre un environnement pour automatiser la génération de spécification formelle à partir des scénarios sous forme d'un automate temporisé. Nous avons mis en oeuvre les algorithmes de compilation de scénarios en automates temporisés. L'approche consiste à transformer les scénarios en actions-règles, puis, construire les places et les transitions de l'automate résultat.

Nous avons développé les structures de données nécessaires pour manipuler les horloges, les variables et les différentes contraintes et affectations. Nous avons choisi des structures qui facilitent les différentes opérations telles que les disjonctions et les conjonctions des contraintes temporelles et des contraintes sur variables, tout en tenant compte de l'aspect performance.

Nous avons développé les algorithmes nécessaires de calcul de différentes opérations sur les éléments de données et nous avons implémenté les algorithmes de génération des actions-règles. Deux types d'intégration sont implémentés: intégration implicite et intégration explicite. L'intégration explicite de scénarios, au

contraire de l'implicite, utilise des directives qui vont contraindre le système par des contraintes d'ordre d'exécution. L'outil développé permet aux concepteurs de construire un prototype de services qu'ils veulent étudier. L'intégration explicite et très utile pour les systèmes existants ou pour exprimer le besoin des concepteurs à un certain ordre d'exécution des scénarios. L'entrée de l'outil est la description du domaine d'application, les scénarios et les directives écrits dans des langages bien définis.

Nous avons développé aussi, les grammaires des langages d'acquisition des entrées du système qui sont les attributs, les variables, les scénarios et les directives et nous avons implémenté les compilateurs correspondants. Enfin, nous avons présenté un exemple d'application d'un commutateur téléphonique en détaillant les entrées et les sorties du système.

Vu les difficultés de transformer une spécification informelle en une formelle, nous proposons comme prolongement envisageable de notre travail le développement d'un langage évolué de quatrième génération générant des scénarios à partir d'une spécification plus simple. Ce langage doit faciliter l'extraction des exigences informelles d'un système réactif temps-réel.

Bibliographie

[Aho 90] Aho A., Sethi R., Ullman J., Compilateurs: Principes, Techniques et Outils, *Inter Editions*, 1990.

[Alu 93] Alur R., Courcoubetis C., Dill D., Model-Checking in Dense Real-time, *Information and Computation*, Vol. 104, No 1, p. 2-34, 1993.

[Alu 94] Alur R., Dill D., A Theory of Timed Automata, *Theoretical Computer Science*, Vol. 126, p. 183-235, 1994.

[Dan 97] Dano B., Une démarche d'ingénierie des besoins orientée objet guidée par les cas d'utilisation, *Ph.D. thesis, Université de Nantes*, novembre 1997.

[Des 98] Desharnais J., Frappier M., Khediri R., and Mili A.: Integration of Sequential Scenarios. *Transactions on Software Engineering*, Vol. 24, No. 9, p. 695-704, Octobre 1998.

[Elk 00] Elkoutbi M. and Keller R. K., User Interface Prototyping based on UML Scenarios and High-level Petri Nets, *In Application and Theory of Petri Nets 2000*, Springer-Verlag LNCS 1825, p.166-186, June 2000.

[Hol 82] Holbrook H. III., A scenario-based methodology for conducting requirements elicitation. *ACM SIGSOFT Software engineering Notes*, Vol. 15, No 1, p. 95-104, Janvier 1982.

[His 94] Hsia P., Samuel J., Gao J., Kung D., Toyoshima Y. and Chen C., Formal Approach to Scenario Analysis, *IEEE Software*, Vol 11, p. 33-41, Mars 1994.

[Lus 97] Lustman F., A formal approach to scenario integration, *Annals of Software Engineering*, Vol 3, p. 255-271, 1997.

[Mas95] Mason T., Brown D., Lex & Yacc, *O'Reilly & Associates Inc.*, 2 ème Edition, 1995.

[Sal 00] Salah A., Dssouli R., Lapalme G., Vincent D., Vers un environnement de création de services fondé sur les scénarios enrichis, *Proceedings of CFIP*, 2000.

[Sal 01] Salah A., Dssouli R., Lapalme G., Compiling real-time scenarios into a Timed Automaton, *Proceedings of FORTE/PSTV*, 2001.

[Sal 02] Salah A., Dssouli R., Lapalme G., Intégration de scénarios temps-réel en automates temporisés, *Proceedings of CFIP,* 2002.

[Som 96] Somé S., Dssouli R. et Vaucher J., Toward an automation of requirements engineering using scenarios, *Journal of Computing and Information*, Vol. 2, No 1, p.1110-1132, 1996.

[Yov 93] Yovine S., Méthodes et outils pour la verification symbolique de systèmes temporisés, *Ph. D. thesis, Institut National Polytechnique de Grenoble*, France, mai 1993.

Annexe

I. Grammaire des attributs

Extention du fichier : Le fichier doit avoir l'extension **.att**.

Mot clés : begin et end. Tous les mots clés sont en minuscules.

Un **identificateur** est une suite des lettres minuscules ou majuscules, des chiffres et du caractère '_'. Un identificateur doit commencer par une lettre.

Un **entier** est une suite des chiffres qui peuvent être préfixes par le signe '+' ou '-'.

Syntaxe :

debut -> < *titre_attribut* > *corps_attribut* /* *titre_attribut est optionnel* */

corps_attribut -> **begin** *liste_attribut* **end**

liste_attribut -> *liste_attribut un_attribut*

un_attribut -> **identificateur** '(' *liste_domaine_parametre* ')'
un_attribut -> **identificateur** '(' '{' '}' ')'
un_attribut -> **identificateur** '(' ')'

liste_domaine_parametre -> *liste_domaine_parametre* ';' '{' *liste_parametre* '}'

liste_parametre -> *liste_parametre* ',' *parametre*

parameter -> **entier**
parameter -> **identificateur**
parameter -> '[' **entier** ' :' **entier** ']'

Exemple :

```
Exemple_click_sourie
begin
Tone_of({21,[11:20],23,[-10:10],22};{[3:5]};{1,-3,send,-6})
Dial({})
Send({0,1,ON,OFF})
Tone()
end
```

II. Grammaire des variables

Extention du fichier : Le fichier doit avoir l'extension **.var**.

Mot clés : begin, clocks, labels, domain et end. Tous les mots clés sont en minuscules.

Un **identificateur** est une suite des lettres minuscules ou majuscules, des chiffres et du caractère '_'. Un identificateur doit commencer par une lettre.

Un **entier** est une suite des chiffres qui peuvent être préfixes par le signe '+' ou '-'.

Syntaxe :

debut -> *<titre_variable>* **begin** *corps_variable* **end** -- *titre_variable* est optionnel

corps_variable -> *liste_clock liste_label liste_variable*

liste_clock -> **clocks = '{'** *clocks* **'}'**

clocks -> **identificateur**
clocks -> *clocks* **',' identificateur**

liste_label -> **labels = '{'** *labels* **'}'**

labels -> **identificateur**
labels -> *labels* **',' identificateur**

variables -> *une_variable*
variables -> *variables une_variable*

une_variable -> **domain '(' identificateur ')' '=' '{'** *liste_parametre* **'}'**

liste_parametre -> *un_parametre*
liste_parametre -> *liste_parametre* **','** *un_parametre*

un_parametre -> **entier**
un_parametre -> **identificateur**
un_parametre -> **'[' entier ':' entier ']'**
un_parametre -> *attribut*

attribut -> **identificateur '('** *attribut_parametre* **')'**
attribut -> **identificateur '(' ')'** /* L'attribut n'a pas de parametres */

attribut_parametre -> *attribut_parametres* **','** *un_attribut_parametre*
attribut_parametre -> *un_attribut_parametre*

un_attribut_parametre -> **entier**
un_attribut_parametre -> **identificateur**

Exemple :
Exemple_click_sourie
begin
clocks = {h1, h2}
labels = {D, U, SC, DC, SS, ES}
domain (pos)={UP, DOWN, Tone_of(0,3,-3), Tone_of(22,5,-6)}
domain (click)={Send(0), YES, NO, [1:10], Send(1), [-11:0]}
domain (sel)={ON, OFF, Dial()}
end

III. Grammaire des scénarios

Extention du fichier : Le fichier doit avoir l'extension **.sce**.

Mot clés : begin_scenario, end_scenario, begin_node, end_node, begin_action, end_action,

invariant, label, variable_constraint, temporal_guard, variables_assignment, clocks_reset, true, or, and. Tous les mots clés sont en minuscules.

Un **identificateur** est une suite des lettres minuscules ou majuscules, des chiffres et du caractère '_'. Un identificateur doit commencer par une lettre.

Un **entier** est une suite des chiffres qui peuvent être préfixes par le signe '+' ou '-'.

Syntaxe :

debut -> *<titre_scenario>* liste_scenario /* *titre_scenario* est optionnel */

liste_scenario -> **begin_scenario identificateur** *liste_node* **end_scenario**
liste_scenario -> *liste_scenario* **begin_scenario identificateur** *liste_node* **end_scenario**
/* L'identificateur représente le nom du scénario */

liste_node -> **begin_node [identificateur]** *liste_action* **end_node**
liste_node -> *liste_node* **begin_node [identificateur]** *liste_action* **end_node**
/* L'identificateur est facutatif et représente le nom d'un noeud */

liste_action -> **begin_action [identificateur]** *action* **end_action**
liste_action -> *liste_action* **begin_action [identificateur]** *action* **end_action**
/* L'identificateur est facultatif et représente le nom d'une action */

action -> invariant label variable_constraint temporal_guard variables_assignment clocks_reset

invariant -> **invariant ':'** *liste_clock_constraint*

invariant -> **invariant ':' true** /* La contrainte est vrai */
invariant -> **invariant ':'** /* La contrainte est vrai */
invariant -> **' '** /* Toute la ligne est vide, la contrainte est vrai */
label -> **Label ':' identificateur** /* L'identificateur représente le label de l'action */

variable_constraint -> **variable_constraint ':'** *expression*
variable_constraint -> **variable_constraint ':' true** /* la contrainte est vrai */
variable_constraint -> **variable_constraint ':'** /* la contrainte est vrai */
variable_constraint -> **' '** /* Pas de ligne, la contrainte est vrai */

temporal_guard -> **temporal_guard ':'** *liste_clock_constraint*
temporal_guard -> **temporal_guard ':' true** /* la contrainte est vrai */
temporal_guard -> **temporal_guard ':'** /* la contrainte est vrai */
temporal_guard -> **' '** /* Pas de ligne, la contrainte est vrai */

variables_assignment -> **variables_assignment ':'** *liste_assignation*
variables_assignment -> **variables_assignment ':'** /* Pas d'assignation */
variables_assignment -> **' '** /* Pas de ligne, pas d'assignation */

clocks_reset -> **clocks_reset ':''{'** *liste_clock* **'}'**
clocks_reset -> **clocks_reset ':''{' '}'**
clocks_reset -> **clocks_reset ':'**
clocks_reset -> **' '**

liste_clock -> **identificateur** /* L'identificateur représente une horloge */
liste_clock -> *liste_clock* **',' identificateur**

liste_clock_constraint -> *clock_constraint*
liste_clock_constraint -> *liste_clock_constraint* **','** *clock_constraint*

clock_constraint -> **identificateur '=' entier** /*l'identificateur représente une horloge*/
clock_constraint -> **identificateur '<' entier**
clock_constraint -> **identificateur '<=' entier**
clock_constraint -> **identificateur '>' entier**
clock_constraint -> **identificateur '>=' entier**
clock_constraint -> **identificateur '-' identificateur '=' entier**
clock_constraint -> **identificateur '-' identificateur '<' entier**
clock_constraint -> **identificateur '-' identificateur '<=' entier**
clock_constraint -> **identificateur '-' identificateur '>' entier**
clock_constraint -> **identificateur '-' identificateur '>=' entier**

expression -> *expression* **or** *and_expression*
expression -> *and_expression*

and_expression -> *and_expression* **and** *feuille*
and_expression -> *feuille*

feuille -> **'('** *expression* **')'**
feuille -> **identificateur '=' entier** /* L'identificateur représente une variable */

feuille -> **identificateur '<' entier**
feuille -> **identificateur '>' entier**
feuille -> **identificateur '<=' entier**
feuille -> **identificateur '>=' entier**
feuille -> **identificateur '!=' entier**
feuille -> **identificateur '=' identificateur**
feuille -> **identificateur '='** *attribut*

liste_assignation -> *une_assignation*
liste_assignation -> *liste_assignation* **','** *une_assignation*

une_assignation -> **identificateur ':''=' identificateur**
une_assignation -> **identificateur ':''=' entier**
une_assignation -> **identificateur ':''=' entier '*' identificateur '+' entier**
une_assignation -> **identificateur ':''=' entier '*' identificateur '-' entier**
une_assignation -> **identificateur ':''=' identificateur '+' entier**
une_assignation -> **identificateur ':''=' identificateur '-' entier**
une_assignation -> **identificateur ':''=' entier '*' identificateur**
une_assignation -> **identificateur ':''='** *attribut*

attribut -> **identificateur '('** *attribut_parametres* **')'**

attribut_parametres -> *attribut_parametres* **','** *un_ attribut_parametre*
attribut_parametres -> *un_ attribut_parametre*
attribut_paramatres -> ' ' /* L'attribut n'a pas de paramètres */

un_attribut_parametre -> **entier**
un_attribut_parametre -> **identificateur**

Exemple :

```
Exemple_click_sourie
begin_scenario Sc1
 begin_node
  begin_action
    invariant: h1-h2>=2
    label: D
    variable_constraint: (pos=UP and click=NO) or sel=Dial( )
    temporal_guard: h1<3, h2>=2
    variables_assignment: click:=NO, pos:=Tone_of(22,5,-6)
    clocks_reset: {h1}
  end_action
  begin_action
    label: U
    variable_constraint: pos = UP and click = YES and sel = OFF
  end_action
 end_node
end_scenario
```

```
begin_scenario Sc2
 begin_node
  begin_action
    invariant: h1 <= 5
    label: SC
    variable_constraint: pos=DOWN or sel = OFF
    temporal_guard: true
    variables_assignment: click:= 2*click + 3
    clocks_reset: {h1, h2}
  end_action
 end_node
end_scenario
```

IV. Grammaire des directives

Extention du fichier: Le fichier doit avoir l'extension **.dir**.

Mot clés : begin et end. Tous les mots clés sont en minuscules.

Un **identificateur** est une suite des lettres minuscules ou majuscules, des chiffres et du caractère '_'. Un identificateur doit commencer par une lettre.

Un **entier** est une suite des chiffres qui peuvent être préfixes par le signe '+' ou '-'.

Syntaxe :

debut -> < *titre_directive*> *corps_directive* /* *titre_directive est optionnel* */

corps_directive -> **begin** *liste_directive* **end**

liste_directive -> *liste_directive un_directive*

un_directive -> **identificateur '|' identificateur**
un_directive -> **identificateur '->' identificateur**

Exemple :

```
Directives_MouseClick
begin
Sc1 | Sc3
Sc1 -> Sc2
Sc3 -> Sc4
Sc2 | Sc4
end
```