Chapitre 4

Mise en œuvre

Dans ce chapitre, nous traiterons de la mise en œuvre de notre algorithme dans deux contextes applicatifs différents : un outil de composition et un jeu vidéo. Nous aborderons la question de la conception d'interfaces graphiques dans la première partie du chapitre puisqu'il s'agit d'un prérequis à la conception d'un outil de composition que nous avons nommé *IMTool*[1]. Puis, nous prendrons le point de vue d'un compositeur dont la tâche sera de paramétrer l'algorithme afin d'aborder l'outil de composition lui-même. Enfin, dans la dernière partie du chapitre, nous nous pencherons sur les spécificités reliées à la mise en œuvre de notre algorithme à l'intérieur d'un jeu vidéo. Dans cette dernière partie, nous tenterons aussi d'établir des comparatifs entre un jeu intégrant notre algorithme et un jeu de référence utilisant de la musique linéaire.

Définition 4.1 (*Cohérence*)

Selon *Van Dam et Foley* [17], la *cohérence* implique que tous les objets de l'interface utilisateur se comportent de manière similaire partout à travers l'interface et que les objets aux fonctionnalités similaires se ressemblent. De plus, l'interface doit être uniforme dans l'ensemble du système.

4.1 Les interfaces graphiques

Dans cette section, nous ferons une courte introduction à la conception d'interfaces graphiques. Plus particulièrement, nous nous pencherons sur différents critères à considérer lorsqu'on veut concevoir une interface permettant à l'utilisateur de performer. Par exemple,

[1] Le nom signifie « Interactive Music Tool ».

Foley et al. préconisent un critère de cohérence [17]. Toutefois, ces mêmes auteurs font aussi remarquer qu'il ne faut pas appliquer ce critère aveuglément au détriment de critères plus importants et ainsi risquer de surprendre l'utilisateur. Ceci concorde avec les remarques de *Kellogg* [33] et de *Grudin* [25] qui stipulent qu'une interface incohérente est parfois préférable car la cohérence est un concept inhéremment relatif et n'a pas de sens par elle-même en tant qu'objectif de conception d'interface [25,33]. On lui préfèrera généralement d'autres critères comme la simplicité d'utilisation, l'ergonomie, la rectification des erreurs, la clarté visuelle et l'esthétisme [17, 48, 65]. Certains auteurs proposent l'utilisation d'ensembles de critères totalement différents tels que la manipulation directe, la facilité de discrimination entre les différents éléments de l'interface et bien d'autres encore [14, 55, 56]. Par exemple, *Oren et Nayar* [48] suggèrent de prendre en compte vingt et un critères dont : l'apprentissage minimal, la mémorisation minimale, la simplicité, la familiarité, la séparation des préoccupations, la fonctionnalité, une relation restreinte avec les usagers, l'informativité, la perceptivité, l'habilité d'explication, l'expressivité, l'aspect esthétique et culturel, la fiabilité, la prédictibilité, la cohérence, la sécurité, le support assurance qualité intégré à l'interface, l'adaptabilité, la personnalisabilité, la maintenabilité et la portabilité. Après évaluation, nous avons retenu un ensemble de critères grâce auxquels nous avons conçu un outil de composition, *IMTool*, proposant une interface conviviale à l'utilisateur. Nous allons maintenant décrire plus en détails chacun des critères retenus.

Le premier critère que nous avons considéré est la *simplicité d'utilisation*, car une interface trop complexe à utiliser risque de décourager les utilisateurs potentiels par sa courbe d'apprentissage trop élevée. À l'opposé, une interface simple à utiliser ne nécessite que peu ou pas d'apprentissage de telle sorte qu'un utilisateur débutant peut immédiatement commencer à utiliser celle-ci et un utilisateur compétent peut rapidement se rappeler comment l'utiliser. Ce critère a guidé nos travaux de recherche, y compris l'algorithme de composition choisi.

Comme deuxième critère, nous avons retenu *l'ergonomie de l'interface*. Ce critère est important car une interface ergonomique accélère le travail de l'utilisateur en minimisant le nombre de mouvements répétitifs et de clics inutiles que doit effectuer l'utilisateur au cours de

sa session de travail. Ainsi donc, une interface faisant preuve d'une bonne ergonomie améliore aussi la simplicité d'utilisation d'un logiciel. Afin d'atteindre cet objectif dans *IMTool*, nous avons éliminé la majorité des boîtes de dialogues et rendu les outils de conception d'automates persistants (voir section 4.2).

Définition 4.2 (*Persistant*)
En conception d'interfaces graphiques, on dit qu'un outil est *persistant* s'il reste sélectionné après utilisation. Par constraste, un outil qui n'est pas *persistant* deviendra désélectionné après utilisation au profit d'un outil par défaut, quel qu'il soit.

Nous venons de voir qu'une interface simple et ergonomique permet une utilisation rapide et efficace d'un outil logiciel. Toutefois, l'utilisateur n'est pas infaillible et peut parfois commettre des erreurs dont certaines peuvent avoir des conséquences fâcheuses. Par exemple, dans *IMTool*, la suppression d'un état de l'automate entraîne la suppression de toutes les transitions entrantes et/ou sortantes de cet état. Il est donc important de permettre à l'utilisateur de *rectifier ses erreurs* en revenant en arrière sur ses dernières actions. C'est pourquoi nous avons implémenté un mécanisme robuste d'annulation des opérations.

Le critère suivant qui a orienté la conception de l'interface est la *clarté visuelle* car une interface visuellement claire permet à l'utilisateur de rapidement distinguer les différentes composantes de celle-ci. Il est possible de concevoir une interface claire en éliminant les éléments inutiles de l'interface, en groupant de façon logique les éléments restants et en s'assurant qu'il y ait un espacement suffisant entre les différents groupes [17].

Le critère suivant que nous avons pris en compte est la *rétroaction visuelle* car une bonne interface ne doit jamais forcer l'utilisateur à deviner l'état du programme. Dans *IMTool*, la rétroaction visuelle passe par des changements de couleurs lorsque l'utilisateur sélectionne différents éléments de l'automate ou lorsque le logiciel joue une séquence MIDI associée à un état.

Les deux critères précédents nous amènent à considérer *l'esthétisme* de l'interface. En effet, une étude par Tractinsky [65] a démontré que les interfaces esthétiquement plaisantes produisent une meilleure impression d'utilisabilité apparente au premier coup d'œil, ce qui

affecte aussi l'impression à long terme des utilisateurs sur l'utilisabilité réelle du système. Il est donc important de faire en sorte que l'interface soit la plus plaisante possible du point de vue esthétique. Dans *IMTool*, l'atteinte de cet objectif passe principalement par le choix des couleurs mais aussi par l'organisation de la fenêtre principale, ce qui fait aussi partie du critère de clarté visuelle.

Enfin, le dernier critère que nous avons considéré est la *manipulation directe*. Ceci signifie que l'utilisateur doit pouvoir manipuler directement l'objet d'intérêt [55, 56]. Toutefois, dans notre outil, ce critère n'est que partiellement respecté car la représentation par automates n'est qu'une représentation intermédiaire de la musique interactive. En effet, bien que nous y référions parfois en tant qu'outil de composition, rappelons-nous que notre outil demeure principalement un outil de paramétrage d'un algorithme de composition. Pour pallier à ce problème, nous avons implémenté dans *IMTool* des fonctions de prévisualisation et de déboguage du résultat telles que « Play », « Stop », « Pause », « Step » et « Play State ».

Nous venons de voir différents critères pour la conception d'interfaces. Nous nous sommes bien sûr limités aux critères que nous avons utilisés pour concevoir l'interface de notre outil. Dans les sections qui vont suivre, nous décrirons celui-ci en plus de détails et expliquerons le processus de conception d'une musique interactive à l'aide de cet outil.

4.2 IMTool

Au dernier chapitre, nous avons présenté notre implémentation d'un moteur de musique interactive basé sur les automates étendus probabilistes. Il s'agissait en fait de la première composante d'un système de musique interactive qui ne pourrait être complet sans un outil de composition. En effet, sans outil de composition, les musiciens auraient beaucoup de difficulté à créer du contenu pour un tel engin sans solliciter l'aide de programmeurs qui, vu l'envergure des jeux vidéo modernes, sont souvent déjà débordés. C'est pourquoi nous avons implémenté *IMTool* qui utilise le moteur décrit au précédent chapitre et qui est aussi utilisé, sous une autre forme, dans un jeu vidéo que nous avons implémenté. Nous reviendrons sur l'interface entre le jeu et le moteur à la section 4.3.3.

Le logiciel *IMTool* représente des musiques interactives sous la forme d'automates finis étendus probabilistes. Il permet de manipuler directement ces automates à l'aide d'outils de création d'automates et d'un panneau affichant les propriétés des états et des transitions sélectionnées. À l'aide d'un autre panneau, il permet aussi d'importer des séquences MIDI qui serviront d'alphabet de sortie aux automates ainsi que des Soundfonts qui serviront de banques d'instruments pendant l'étape de rendu des séquences MIDI. À l'exception de la fonctionnalité de rendu MIDI pour laquelle il utilise des bibliothèques provenant de tierces parties, le logiciel a été développé entièrement à partir de zéro.Pour effectuer le rendu des séquences MIDI, nous avons utilisé le synthétiseur *Fluidsynth*[2] [26] et la bibliothèque d'ordonnancement en temps réel *MidiShare*[3] [22], deux bibliothèques « Open Source » sous license « GNU Lesser General Public License v2 » [18].

La figure 4.1 montre la fenêtre principale de l'application. On y retrouve les principaux éléments de l'interface : une barre de menus, une barre d'outils, une barre d'état ainsi que les trois panneaux mentionnés précédemment. Avant de passer à une description plus détaillée de chacun des éléments de l'interface, remarquons les deux caractéristiques suivantes de celle-ci :

1. Chaque panneau est identifié par sa fonction.
2. Les boutons de la barre d'outils sont larges, colorés, espacés et également bien identifiés.

Grâce à ces particularités de l'interface, l'utilisateur peut rapidement se rappeler la fonction de chaque élément de l'interface puisqu'elle est constamment affichée à l'écran. De surcroît, il peut facilement repérer les différents boutons sur la barre d'outils grâce à leurs couleurs vives. Ces deux caractéristiques de l'interface augmentent non seulement la simplicité d'utilisation du logiciel mais aussi la clarté visuelle de l'interface.

Par convention, le menu principal de l'application contient les fonctions d'accès aux fichiers et les fonctions d'aide du logiciel. Il contient aussi les fonctions moins souvent effectuées à l'aide de la souris telles que les fonctions d'édition. Toutefois, les fonctions de création et de déboguage d'automates sont absentes des menus puisqu'elles sont constamment utilisées

[2] `http://www.nongnu.org/fluid/`
[3] `http://midishare.sourceforge.net/`

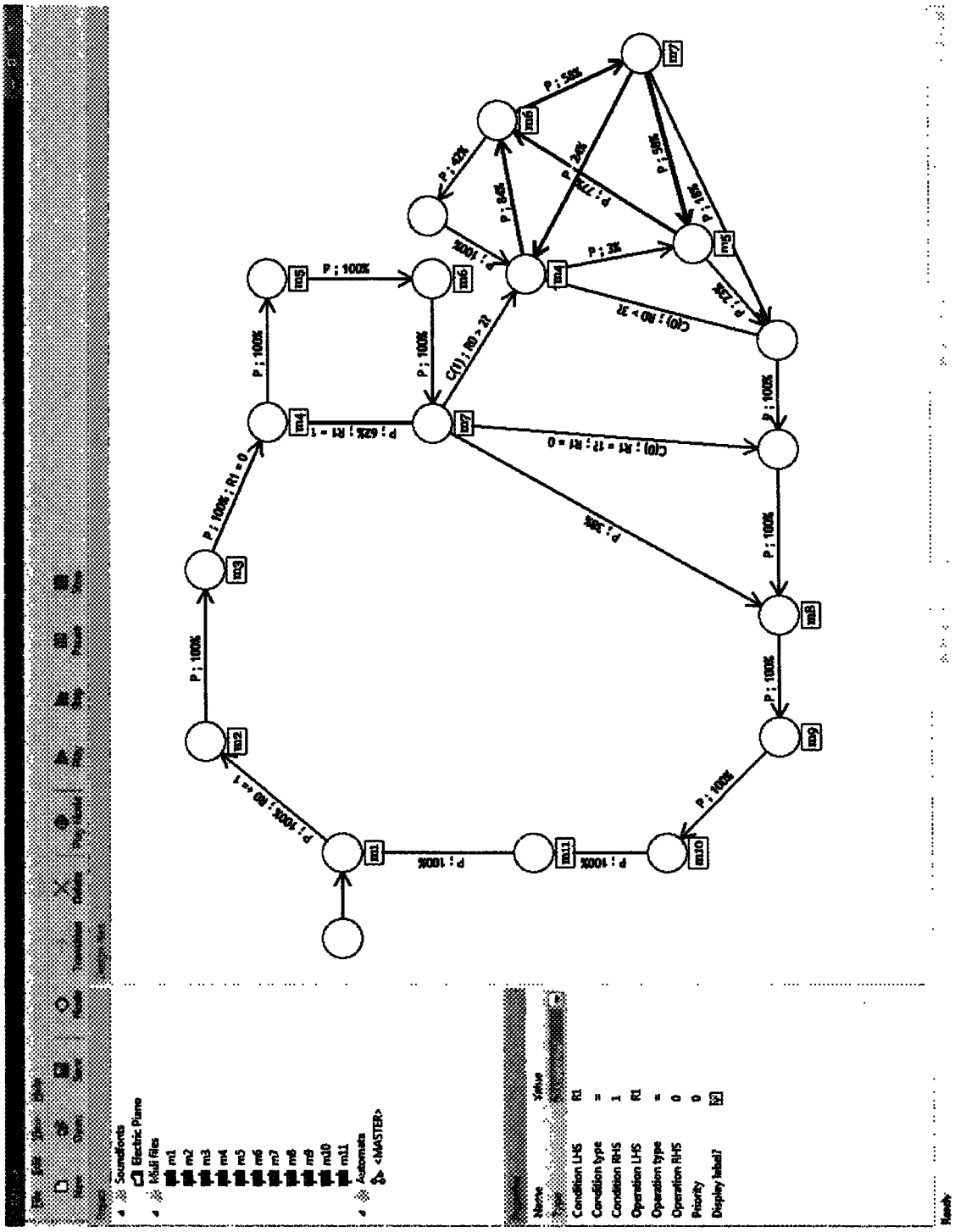


FIG. 4.1 – L'interface utilisateur

et déjà présentes dans la barre d'outils. De ce fait, ces fonctions ne feraient qu'encombrer les menus, ce qui diminuerait la clarté visuelle de l'application.

La barre d'outils est divisée en trois groupes d'outils. Le premier contient les opérations d'accès aux fichiers les plus fréquemment utilisées, soit *nouveau*, *ouvrir* et *enregistrer*. Le deuxième groupe se constitue des outils de création d'automates. Ceux-ci permettent de créer et de supprimer des états et des transitions dans la zone de conception. Ces outils sont persistants, c'est-à-dire qu'ils provoquent un changement de mode du logiciel. Nous verrons plus loin, lorsque nous traiterons de la zone de conception, ce que cela implique. Enfin, le dernier groupe contient les outils de prévisualisation et de déboguage qui permettent à l'utilisateur de corriger les problèmes dans sa musique avant qu'ils ne surviennent dans le jeu vidéo. Malheureusement, cette prévisualisation n'est fiable à 100% que si l'on suppose que le code du jeu vidéo ne contient pas de défauts majeurs au niveau du code chargé de mettre les registres du moteur de musique à jour. Par exemple, il peut arriver que le jeu ne fasse jamais la mise à jour de certains registres ou qu'il écrive des valeurs erronées dans les registres. C'est le programmeur du jeu, soit l'utilisateur de l'API d'*IMTool*, qui doit s'assurer de la cohérence à ce niveau.

La barre d'état permet à l'utilisateur de voir plus rapidement la condition d'activation d'une transition lorsqu'il n'y a pas assez d'espace dans la zone de conception pour afficher celle-ci au dessus de la flèche représentant la transition. Cela évite à l'utilisateur de devoir reconstituer mentalement la condition d'activation courante à partir de la fenêtre de propriétés. Cela lui évite aussi de devoir mémoriser celle-ci s'il prévoit la modifier.

Le panneau en bas à gauche sur la figure 4.1 représente la fenêtre de propriétés que nous venons de mentionner. Il s'agit d'une liste clé-valeur qui se met à jour automatiquement dès que l'utilisateur sélectionne un élément (par un clic du bouton gauche) dans l'explorateur de projet ou dans la zone de conception. Comme cette liste est toujours visible, l'utilisateur n'a pas à effectuer d'actions supplémentaires pour accéder aux propriétés d'un objet. Cela diminue le nombre de clics inutiles et répétitifs, ce qui fait partie du critère d'ergonomie de notre application. Grâce à cette liste, l'utilisateur peut modifier les propriétés de l'élément sélectionné à sa guise. Cet élément est mis à jour lorsqu'un autre objet est sélectionné ou lorsque le projet

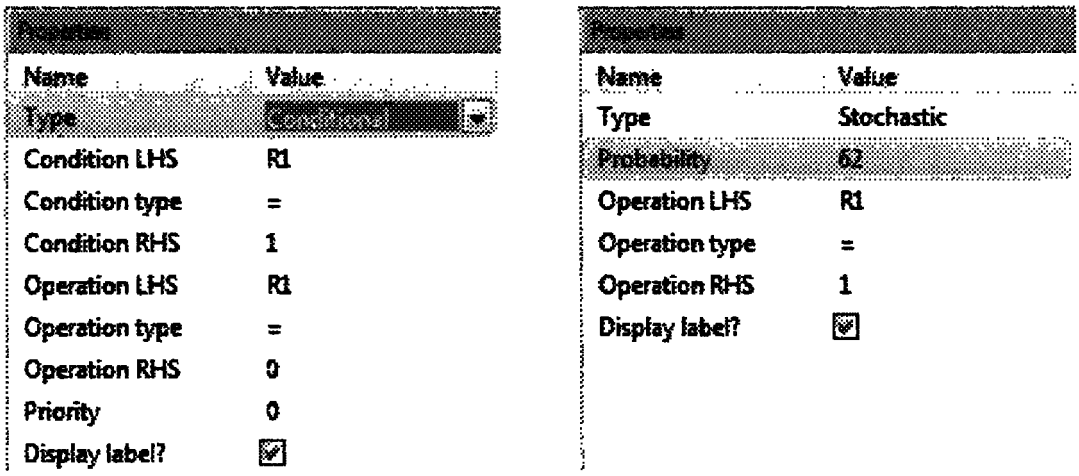
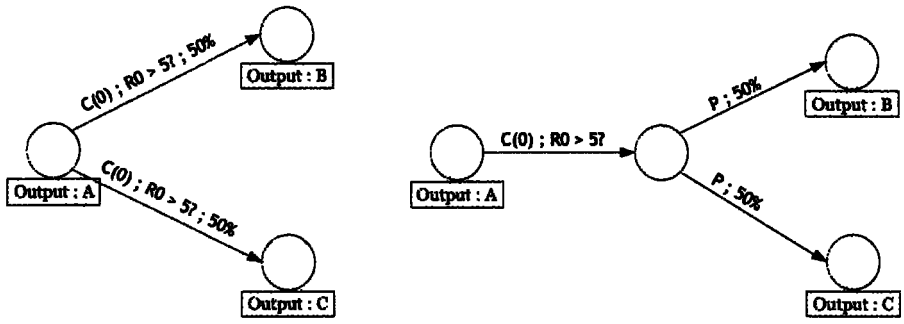


FIG. 4.2 – Fenêtre de propriétés

est sauvegardé. La figure 4.2 montre le panneau affichant les propriétés de deux transitions différentes. On remarque que la plupart des propriétés correspondent à celles que nous avons définies au chapitre précédent, bien que les noms utilisés aient été changés dans le but de rendre l'interface utilisateur plus conviviale. Par contre, la propriété *type* est spécifique à l'éditeur et a été introduite dans un but de prévention d'erreurs. En effet, le modèle d'automates finis étendus probabilistes est complexe et peut sembler difficile à aborder pour un utilisateur néophyte. Nous avons donc divisé les transitions en deux types : les transitions conditionnelles et les transitions stochastiques. Dans le cas des premières, nous fixons la probabilité de transition à 100% mais nous laissons l'utilisateur modifier librement la condition de transition ainsi que la priorité de transition. Pour les deuxièmes, nous faisons le contraire : nous fixons la condition de



(a) Sans « types »

(b) Avec « types »

FIG. 4.3 – La propriété « type » ne limite pas l'expressivité du modèle

transition à la tautologie booléenne VRAI et la priorité de transition à $-\infty$ mais nous laissons l'utilisateur modifier librement la probabilité de transition. On peut voir, en comparant les figures 4.3(a) et 4.3(b), que cette contrainte ne restreint pas l'expressivité du modèle.

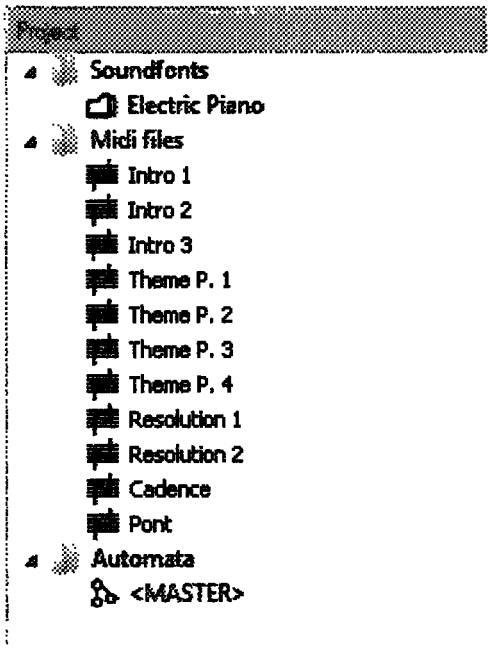


FIG. 4.4 – Explorateur de projet

La figure 4.4 illustre en gros plan l'explorateur de projet. On peut voir qu'il s'agit d'une liste des objets (fichiers) faisant partie du projet et que ces objets sont classés dans trois dossiers. Ces trois dossiers représentent les trois types d'objets qu'un projet peut contenir : des Soundfonts, des séquences MIDI et des automates[4]. L'ajout de fichiers dans le projet peut se faire par le menu principal de l'application ou directement dans ce panneau par un clic du bouton droit de la souris. Dans ce cas, un menu contextuel apparaît et propose non seulement l'ajout de fichiers mais aussi un ensemble de tâches relatives au point de chute du clic de souris. Il est par exemple possible d'écouter une séquence MIDI individuelle à partir de l'explorateur de projet. Quant au bouton gauche de la souris, il permet de sélectionner les objets déjà présents dans le projet et ainsi faire afficher leurs propriétés. Comme propriétés,

[4]Rappelons toutefois que les projets créés avec la version présente du logiciel ne peuvent pas contenir d'autres automates que l'automate maître(<MASTER> sur la figure).

les Soundfonts et les séquences MIDI possèdent tous deux un nom d'affichage et le chemin sur disque du fichier. Toutefois, les Soundfonts possèdent une propriété supplémentaire : un décalage de banque. Cette propriété permet d'utiliser plusieurs Soundfonts dans un même projet car la plupart des Soundfonts ont été conçues pour se charger dans la banque d'instruments 0, ce qui provoque un conflit lorsqu'on tente d'en utiliser plusieurs à la fois. Le synthétiseur que nous utilisons, *Fluidsynth*, propose donc de rémédier à ce problème en spécifiant un décalage de banque (« bank offset » en anglais) pour chaque Soundfont. Ce décalage est ajouté au numéro de banque de chaque instrument dans la Soundfont. Par exemple, si la Soundfont spécifie qu'un instrument devrait se charger dans la banque 4 et qu'on spécifie un décalage d'une position, *Fluidsynth* charge cet instrument dans la banque 5. Ceci amène toutefois le problème qu'il faut informer les fichiers MIDI que les Soundfonts ont été décalées. Ceci se fait grâce au message de changement de banque (voir annexe A).

La zone de conception (panneau de droite sur la figure 4.1) permet à l'utilisateur de manipuler directement un automate. À l'aide des outils précédemment mentionnés, il peut créer et supprimer des états et des transitions en cliquant dans ce panneau. Comme nous l'avons aussi déjà mentionné, ces outils sont persistants, c'est-à-dire que lorsque l'utilisateur clique sur le bouton correspondant dans la barre d'outils, le logiciel passe du mode *sélection* à l'un des modes suivant : *création états*, *création transitions*, *suppression éléments*. Il restera dans ce mode tant que l'utilisateur ne cliquera pas sur le bouton droit de la souris pour revenir au mode *sélection*. Cette mesure d'ergonomie permet à l'utilisateur de travailler plus rapidement par rapport à une première version du logiciel qui n'implémentait pas cette fonctionnalité. Ce panneau implémente aussi la majorité de la rétroaction visuelle de notre logiciel. En effet, lorsque l'utilisateur clique sur un élément de l'automate celui-ci change de couleur en même temps que les propriétés de celui-ci s'affichent dans la fenêtre de propriétés. De plus, lorsque le logiciel est en mode prévisualisation et que l'automate traverse un état auquel est associé une séquence MIDI, il colore cet état en rouge dans la zone de conception pendant que la séquence MIDI joue.

Enfin, avant d'expliquer le processus de composition d'une musique interactive, nous allons d'abord parler du mode de prévisualisation et de déboguage du logiciel, illustré à la

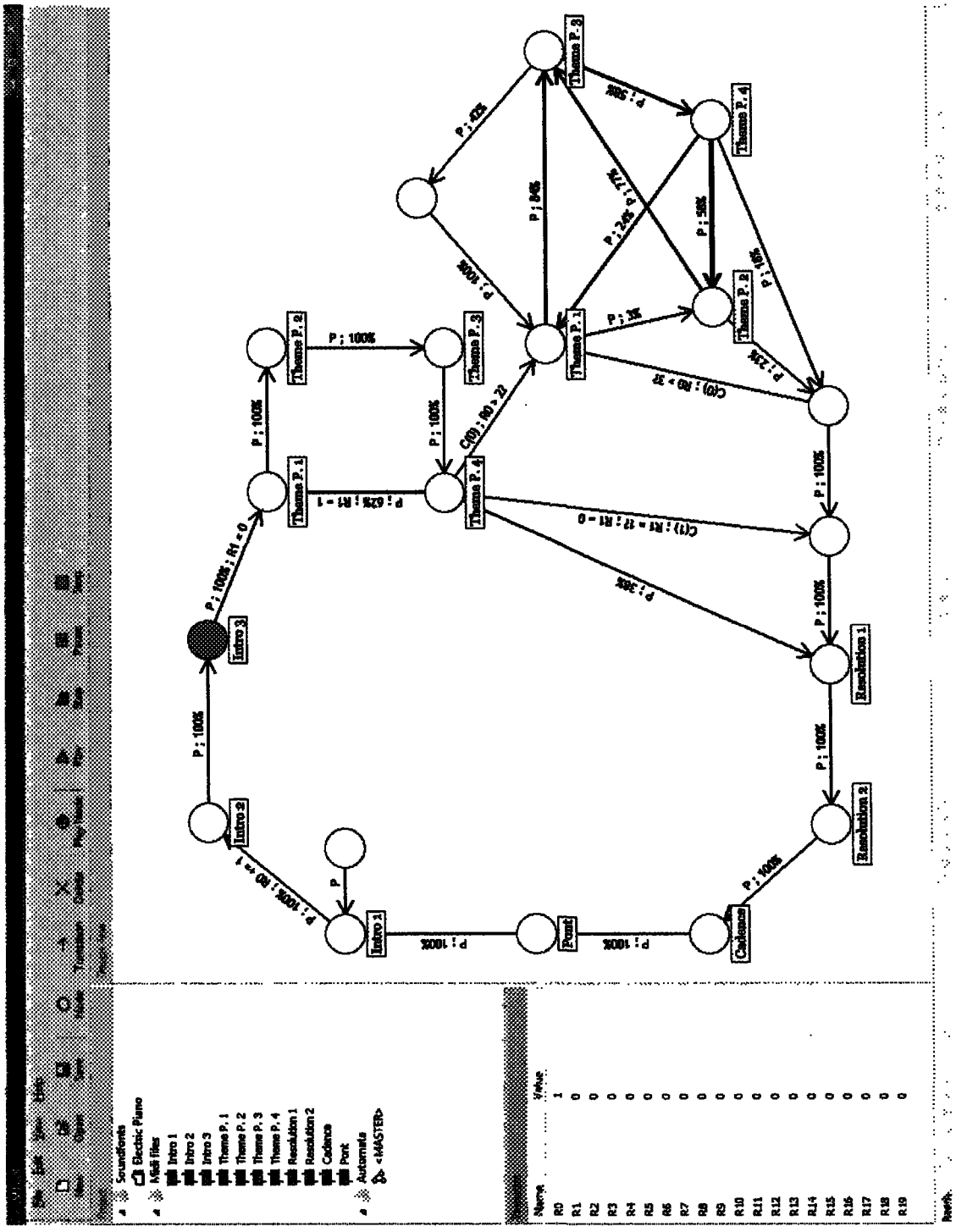


FIG. 4.5 – Le logiciel en mode prévisualisation/déboguage

figure 4.5. On remarque sur cette figure qu'une fenêtre affichant le contenu des registres a pris la place de la fenêtre de propriétés. Il s'agit en fait d'un débogueur qui permet de modifier le contenu des registres lorsque l'utilisateur suspend la simulation. Ces modifications seront prises en compte lors de la reprise de la simulation, au moment où le programme réinitialisera les registres avec les valeurs apparaissant dans la fenêtre de déboguage.

4.3 Mise en œuvre dans un jeu vidéo

Afin de tester notre outil dans un scénario d'utilisation typique, nous avons conçu un prototype de jeu vidéo très simple et composé la trame sonore pour celui-ci à l'aide d'*IMTool*. Cette section portera sur les différentes étapes de cet exercice et sera donc divisée en trois parties : la conception du prototype de jeu, la composition de la trame sonore et l'intégration de la trame sonore dans celui-ci.

4.3.1 Conception d'un prototype de jeu

La première étape de la conception d'un jeu est la production d'un *document de concept*. Dépendamment de la complexité du jeu, la taille de ce document peut varier entre un paragraphe et plusieurs dizaines de pages. Nous nous sommes limités à un seul paragraphe puisqu'il ne s'agissait que d'un prototype de jeu très simple destiné principalement à tester notre algorithme de musique interactive et à valider l'API du moteur.

> L'objectif du jeu est de protéger une cible attaquée par des vagues d'ennemis successives. Le joueur peut éliminer ces ennemis en *cliquant* dessus avec le stylet de la Nintendo DS. Lorsque tous les ennemis d'une même vague ont été éliminés, une nouvelle vague d'ennemis plus agressifs est générée.

Puisque dans ce concept de jeu, le stylet n'est utilisé que de façon superficielle et aurait parfaitement pû être remplacé par la souris d'un ordinateur conventionnel, il est nécessaire ici d'expliquer le choix d'utiliser la Nintendo DS comme plateforme d'implémentation. En effet, rappelons-nous qu'au chapitre 3, nous avons affirmé que notre algorithme était très efficace et ainsi mieux adapté que plusieurs autres pour l'utilisation en temps réel dans un jeu vidéo. Le fait d'utiliser la Nintendo DS nous permettra de justifier cette affirmation puisqu'il s'agit

d'une console ne disposant que de peu de puissance et qu'elle doit partager celle-ci entre tous les sous-systèmes d'un même jeu (graphiques, intelligence artificielle, musique, effets sonores, etc) tout en maintenant une vitesse respectable pour celui-ci. Idéalement, un jeu sur Nintendo DS devrait fonctionner à une vitesse de 60 images (« frames » en anglais) par seconde, soit la vitesse de rafraîchissement de l'écran de la console. En pratique, on peut être un peu plus tolérant.

4.3.2 Composition de la trame sonore

Revenons au document de concept du jeu. Sur celui-ci, aucune direction musicale n'est spécifiée pour le jeu. Vu la nature itérative de ces documents, cette situation est parfaitement normale. Toutefois, elle nous empêche de procéder à la composition de la trame sonore. Il nous faut donc y remédier :

> Initialement, pendant la première vague d'ennemis, la musique sera constituée d'une boucle musicale très simple. La musique change lorsqu'une vague d'ennemis est complètement détruite. Dès lors, on peut soit ajouter une couche supplémentaire à la musique en cours, soit complexifier celle-ci en y ajoutant des notes, des transitions aléatoires, ou en changeant carrément sa structure. On peut aussi combiner les méthodes précédentes pour produire un résultat complètement chaotique.

Ainsi donc, on peut passer à la première étape de la composition de la trame sonore qui consiste en la composition d'une ou plusieurs séquences MIDI initiales. Ces séquences initiales pourront faire partie ou non des séquences qui seront utilisée dans l'œuvre finale[5]. Ces séquences pourront ensuite être importées dans *IMTool* et nous permettront d'immédiatement commencer à utiliser *IMTool* pour spécifier la *méta-structure*[6] de notre composition ainsi que ses interactions avec notre prototype de jeu.

Après avoir importé les séquences MIDI dans *IMTool*, il nous faut maintenant choisir des Soundfonts pour effectuer le rendu de celles-ci. Nous avons choisi d'utiliser une Soundfont fournissant l'ensemble complet des instruments spécifiés par le standard General

[5]Souvent, les séquences utilisées dans l'œuvre finale sont produites en parallèle avec la spécification de l'automate et dérivent des séquences initiales.

[6]On retrouve souvent plusieurs niveaux hiérarchiques de structure dans les pièces musicales. Puisque chaque séquence MIDI possède sa propre structure interne et que l'automate décrit par *IMTool* est une structure d'ordre supérieur qui s'y superpose, on qualifie celui-ci de méta-structure.

MIDI : « Airfont 380 ». Cette Soundfont est disponible gratuitement sur Internet pour usage non commercial.

Ensuite, nous avons défini une correspondance entre les registres de l'automate et les interactions souhaitées entre la musique et l'état du jeu. Selon le document de concept, la seule interaction nécessaire consiste à modifier la musique lorsqu'une vague d'ennemis est éliminée. Nous avons donc choisi de faire correspondre le registre R_0 au nombre d'ennemis restants. De cette façon, la condition $R_0 = 0$ déclenche le changement de structure musicale souhaité à chaque nouvelle vague d'ennemis.

Enfin, nous avons créé les états et les transitions, toujours en suivant le document de concept que nous avons défini. La figure 4.6 illustre le résultat final. Notez qu'à des fins de clarté nous avons supprimé quelques transitions.

4.3.3 Intégration de la trame sonore

La trame sonore en main, il ne reste qu'à l'intégrer au prototype de jeu vidéo précédemment développé. À cet effet, nous avons développé une version optimisée du moteur d'*IMTool* éliminant les calculs inutiles, tel que le rendu à partir du format MIDI vers le format WAV, et les informations redondantes, telles que celles énumérées ci-dessous :

1. Le numéro d'identification des états
2. Le numéro d'identification de l'état initial
3. Les indicateurs d'états finaux
4. L'état de départ sur les transitions
5. La valeur de priorité sur les transitions

Nous considérons que ces informations sont redondantes puisqu'il est possible d'obtenir le même résultat en prétraitant l'automate dans un exporteur de la façon suivante :

1. Réassignation des numéros d'identification des états ; si l'automate possède N états, on leur assigne les numéros $0 \ldots N-1$.
2. Assignation du numéro d'identification 0 à l'état initial de l'automate.

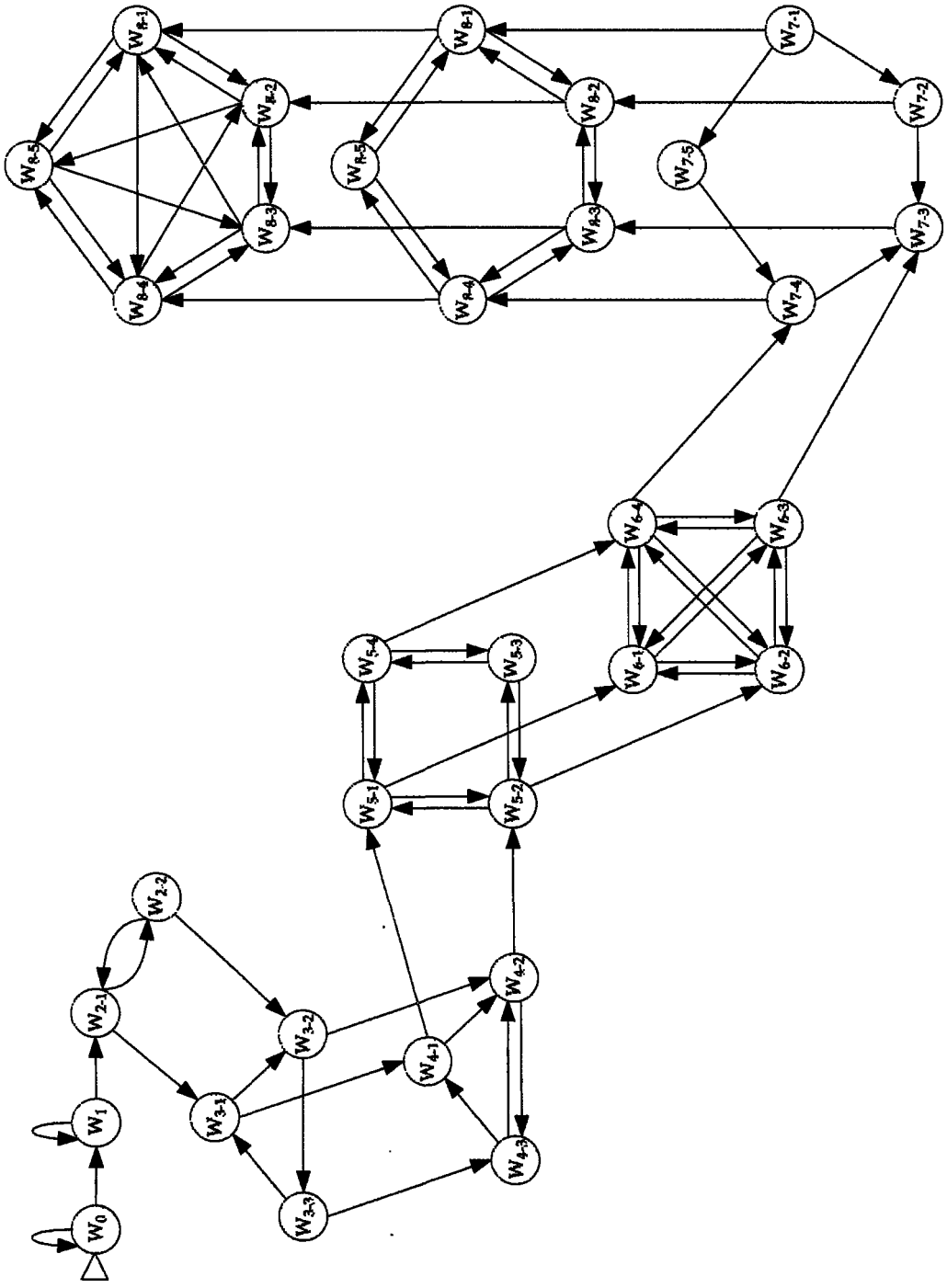


FIG. 4.6 – Automate résultant

3. Suppression des transitions inactivables (telles que définies à la section 3.2.1) incluant les transitions sortant des états finaux.
4. Fusion des tableaux d'états et de transitions pour produire un intercalage. La structure de donnée finale peut être visualisée à la figure 4.7.
5. Tri préalable des transitions par priorité.

En résumé, les différences majeures entre le moteur utilisé dans *IMTool* et celui utilisé dans notre prototype de jeu sont que ce dernier utilise directement le format WAV plutôt que le format MIDI, ce qui permet de l'utiliser sur des ordinateurs disposant de beaucoup moins de puissance de calcul qu'un PC moderne, par exemple la Nintendo DS. C'est grâce à *Fluidsynth* que cela a été rendu possible car sans celui-ci nous n'aurions pas pu convertir les fichiers MIDI en fichiers WAV. De plus, le moteur utilisé dans notre jeu utilise des structures de données optimisées occupant moins d'espace sur le disque et, de ce fait, permettant un chargement plus rapide des fichiers de musique interactive. Des essais ont démontré que ce nouveau format occupe jusqu'à 90% moins d'espace sur disque que le format original, basé sur XML, se charge en mémoire jusqu'à 32 fois plus rapidement sur la Nintendo DS[7].

Nombre d'états
1er pointeur vers les états
...
Nombre de conditions dans le 1er état
1er pointeur vers les conditions dans le 1er état
...
Nb. de destinations de la 1^{ere} condition du 1er état
1^{ere} destination[8] de la 1^{ere} condition du 1er état
...
...

FIG. 4.7 – Structure de données finale

[7]Les résultats exacts sont 76 secondes pour le chargement de vingt mille fichiers dans le format original et 2.35 secondes pour le chargement de vingt mille fichiers dans le format optimisé. Ces résultats sont reproductibles.

[8]La description d'une destination comprend sa probabilité, son état d'arrivée et sa fonction de mise-à-jour.

Un autre avantage de ce format est qu'il représente beaucoup mieux notre modèle tel que nous l'avons décrit au chapitre 3. Ainsi donc, dans ce format, chaque condition représente une fonction d'activation $c(X) \in C$ et la liste de destinations sert à réduire l'espace utilisé par les vecteurs stochastiques (puisque la majorité des probabilités de transitions sont égales à 0).

Avant de conclure, il convient de décrire l'interface entre le moteur de musique et la boucle principale du jeu. Celle-ci est composée de six fonctions : `Reset`, `LoadMusic`, `SetRegister`, `GetRegister`, `StartMusic`, et `StopMusic`. Leur rôle est le suivant :

`Reset` (ré)initialise le moteur de musique en remettant tous les registres à zéro.

`LoadMusic` charge un fichier contenant un automate étendu probabiliste ainsi que tous les segments musicaux nécessaires.

`SetRegister` modifie le contenu d'un registre.

`GetRegister` retourne le contenu d'un registre.

`StartMusic` démarre le moteur de musique.

`StopMusic` arrête le moteur de musique.

4.4 Discussion

Après avoir testé le jeu résultant, il nous apparaît clair que notre algorithme est assez efficace pour être utilisé en temps réel dans un jeu vidéo, même lorsque celui-ci est programmé sur une console disposant d'une puissance limitée. De plus, nous avons trouvé que l'interface d'*IMTool* est possède pour permettre, avec une quantité d'effort raisonnable, la création d'automates étendus probabilistes complexes (comme celui sur la figure 4.6). Ces résultats corroborent nos affirmations du chapitre 3.

Toutefois, il nous apparaît aussi que l'interface d'*IMTool* présente une limitation importante que nous illustrerons à l'aide d'un exemple : notre composition présentée à la figure 4.6. L'automate sur cette figure est formé d'un très grand nombre d'états et de transitions. Nous avons donc dû supprimer certaines transitions afin améliorer la clarté visuelle. Nous proposons ici deux solutions à cette limitation :

- L'introduction d'une représentation hiérarchique pour les automates étendus probabilistes où chaque état de l'automate aurait la possibilité de correspondre à l'état de départ d'un sous-automate.
- L'utilisation de flèches bidirectionnelles ($\leftrightarrow$) lorsque la réciproque $r \rightarrow q$ d'une transition $q \rightarrow r$ existe.

Remarquons au passage qu'il n'est toutefois pas nécessaire d'apporter de changements au modèle tel que nous l'avons présenté au chapitre 3. En effet, il est possible d'aplatir un automate représenté de façon hiérarchique en remplaçant de façon récursive les états qui correspondent à des sous-automates (*méta-états*) par la description de ces derniers. Ne reste qu'à déterminer s'il faut recopier les transitions sortant des méta-états sur tous les états de leurs sous-automates ou s'il ne faut évaluer ces transitions que lorsqu'un sous-automate atteint un état final. Nous proposons de combiner les deux solutions en appelant le premier type de transitions : « transitions prioritaires » et le deuxième type de transitions : « transitions normales ».

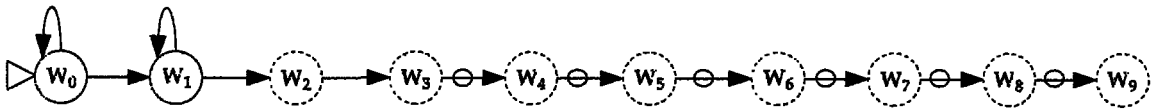


FIG. 4.8 – Structure globale de la pièce utilisée pour notre prototype

En appliquant la première solution, la figure 4.6 pourrait être rescindée en structure hiérarchique à deux niveaux. Le premier représenterait la structure globale de la pièce et pourrait être représenté comme sur la figure 4.8. Sur cette figure, les méta-états sont représentés par des traits pointillés et les transitions hautement prioritaires sont marquées d'un cercle. Chaque méta-état serait associé à un sous-automate décrivant sa composante respective. Ainsi le méta-état W_9 serait associé à un automate contenant les anciens états W_{9-1}, W_{9-2}, W_{9-3}, W_{9-4} et W_{9-5}. En appliquant la seconde solution, ce dernier pourrait être représenté comme sur la figure 4.9.

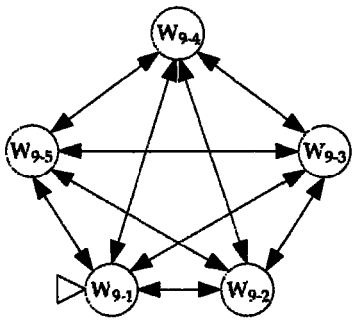


FIG. 4.9 – Utilisation de flèches bidirectionnelles pour améliorer la clarté visuelle