Master Systèmes Dynamiques et Signaux

Mémoire de master

Modélisation et outils des systèmes réactifs pour le live-coding collaboratif, développement d'une interface pour le live-coding interactif

---

Auteur :
M^me^ Bérangère Daviaud

Jury :
Pr. S. Lahaye
Dr. M. Lhommeau
Dr. N. Delanoue
Pr. F. Chapeau-Blondeau
Pr. D. Rousseau

Président :
Pr. L. Hardouin

Version du
1^er^ juillet 2021

Remerciements

Je tiens à exprimer toute ma reconnaissance à mes encadrants, M. Sébastien LAHAYE, professeur à Polytech Angers, M. Mehdi LHOMMEAU, maître de conférences à Polytech Angers et M. Mathieu DELALLE, professeur à TALM-Angers. Je les remercie de m'avoir orientée, aidée, conseillée et pour m'avoir fait confiance.

Je désire aussi remercier le responsable Master SDS de Polytech Angers, M. Laurent HARDOUIN, qui m'a permise de me lancer dans cette expérience enrichissante.

Je voudrais exprimer ma reconnaissance envers les amis et camarades de ma promotion qui m'ont apporté leur soutien moral et intellectuel.

Et enfin, un grand merci à ma famille pour leurs encouragements et leur soutien.

Table des matières

Table des figures

Introduction

Au cours de ces vingt dernières années, le live-coding s'est développé et a gagné en popularité dans les sphères universitaires et de la pop culture. Directement issu de la musique informatique, le live-coding travaille une nouvelle perspective, en employant la programmation à la volée. Comme nous l'avons vu dans l'étude bibliographique réalisée au préalable, l'art assisté par ordinateur est apparu dès l'aube de l'informatique, initié au départ par des ingénieurs et des scientifiques pour questionner l'intelligence artificielle ou encore les limites de la machine [1]. D'un point de vue artistique, ces travaux amènent à reconsidérer la place et le rôle de l'auteur. Avec l'évolution technologique rapide de ce siècle et du précédent, les possibilités aujourd'hui paraissent illimitées : l'art dit génératif n'en est qu'à son commencement. Le live-coding est une nouvelle forme récente de la musique assistée par ordinateur, où la machine est utilisée comme un outil de performance live. Naturellement, c'est une pratique basée sur l'improvisation et de plus en plus sur la collaboration. Plusieurs problématiques sont au coeur des travaux actuels, tels que le développement de l'interactivité avec le public [11], l'intuitivité et l'accessibilité des interfaces et des langages de programmation employés [4], ainsi que la synchronisation dans le cas de performances en collaboration[13].

L'objectif du stage est de proposer un outil pour le live-coding interactif en concevant une version étendue de l'interface Troop, qui est un outil de live-coding collaboratif. La preuve de concept s'articulera autour de l'utilisation d'une Raspberry Pi et la création d'orchestrations virtuelles, modélisées comme des système à événements discrets. L'enjeu est de fournir à l'utilisateur le moyen de composer et d'exprimer son intention tout en la paramétrant avec les interactions possibles.

De ce travail est né la preuve de concept TroopRasp que nous exposerons dans ce document. Dans un premier temps, nous présenterons les outils utilisés, tels que l'environnement Troop, le langage pour live-coding FoxDot ainsi que la représentation des orchestrations virtuelles sous forme de systèmes à événements discrets. Dans un deuxième temps, nous évoquerons les aspects techniques rencontrés lors de la conception des fonctionnalités de TroopRasp. Et enfin, dans un dernier temps, nous présenterons l'interface TroopRasp et ses possibilités de création pour live-coding, puis les perspectives du projet.

Chapitre 1

Contexte

Depuis son émergence [10, 14, 15], de multiples plateformes et langages de programmation associés ont été développés pour le live-coding. En particulier, le logiciel Troop [7], [8] permet à plusieurs livecoders de collaborer au sein d'un même document. C'est à partir de cet outil, combiné au langage FoxDot, que nous avons travaillé pour concevoir TroopRasp. Dans cette partie, nous présenterons ces deux outils, puis nous expliquerons et motiverons notre approche pour une solution de live-coding interactif.

1.1 Troop et FoxDot : les outils pour live-coding.

Le logiciel Troop est un éditeur, ou environnement, pour le live-coding. Il permet de collaborer simultanément sur le même document à partir de machines distantes. Pour différencier ses propres contributions de celles des autres, chaque artiste connecté reçoit une étiquette de couleur différente qui contient son nom et qui est associée à l'emplacement de son curseur de texte, comme le montre la figure 1.1. Troop n'est pas un langage de live-coding, mais un outil permettant de connecter plusieurs live-coders sur un réseau. Il faut donc installer le langage de son choix avant de l'utiliser. Par défaut, Troop fonctionne avec FoxDot [5, 6], un langage basé sur Python, mais il est possible de travailler avec TidalCycles, Sonic Pi ou SuperCollider.

FoxDot a été créé en 2015 pour essayer d'ouvrir les voies du live-coding aux utilisateurs qui sont novices en programmation et qui veulent l'utiliser pour créer de la musique rapidement et facilement. Il s'agit d'une bibliothèque Python facile à utiliser qui crée un environnement de programmation interactif et dialogue avec le moteur de synthèse sonore SuperCollider pour générer de la musique. FoxDot est un langage convivial et facile à appréhender qui rend le live-coding facile et amusant, tant pour les débutants en programmation que pour les vétérans. En accord avec la philosophie de la communauté live-coding, Troop et FoxDot sont entièrement gratuits et open source. Leurs pages GitHub contiennent

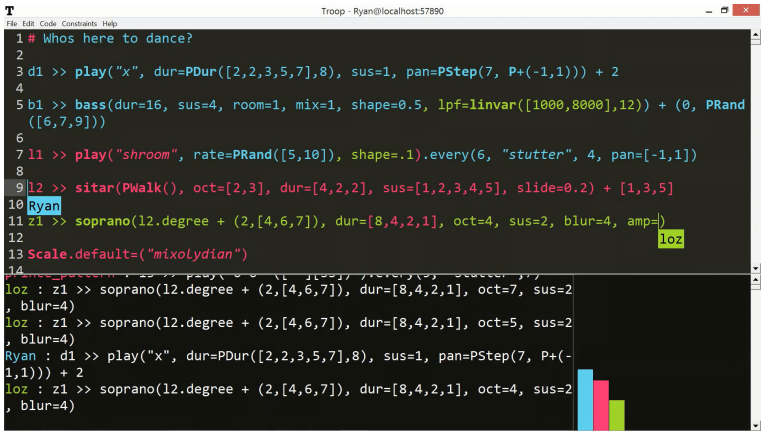

FIGURE 1.1 – Interface de Troop avec plusieurs utilisateurs connectés.

toutes les informations utiles à leur installation et leur configuration. Pour ces différentes raisons, nous avons choisi de travailler avec ces outils déjà diffusées et appropriés par la communauté du live-coding. Pour une meilleure compréhension du sujet et du travail réalisé pour TroopRasp, nous allons parcourir brièvement les notions importantes de ces deux ressources.

1.1.1 Initiation au langage FoxDot.

De manière générale et synthétique, la programmation orientée objet permet de concevoir une application sous la forme d'un ensemble de briques logicielles appelées objets. Un objet représente une entité du monde réel, ou de monde virtuel dans le cas d'objets immatériels, qui se caractérise par une identité, des états significatifs et par un comportement. Chaque objet joue un rôle précis et peut communiquer avec les autres. Les interactions entre eux vont permettre à l'application de réaliser les fonctionnalités attendues.

Bien que FoxDot ne soit pas une application, la structure du langage est celle de la programmation orientée objet. En effet, les "players" sont des objets qui génère du son en fonction des instructions qui leur sont données. C'est en les créant et en les manipulant qu'une session de live-coding se construit et évolue. La première instruction donnée à un player concerne l'instrument joué ou synthé. La double flèche `>>` permet d'assigner un synthé à un player spécifique. Par exemple, le code `p1 >> pluck()` assigne l'instrument "pluck" au player nommé "p1". Comme tout objet, le player a des attributs - dont le synthé - et des méthodes. Parmi celles-ci, nous pouvons trouver une méthode pour mettre à jour les attributs ou encore arrêter le player, et ainsi qu'il ne génère plus de musique.

Pour manipuler les séquences jouées, il existe de nombreuses options, qui se divisent en deux groupes : les attributs et les effets. Les attributs affectent quelle note est jouée et à quel moment, et les effets changent la façon dont le son sonne. Les valeurs de l'attribut ou

de l'effet d'un player sont définis soit en les spécifiant comme argument à l'intérieur d'un appel SynthDef - abréviation pour Synth Definition - comme suit :

```
p1 >> pluck([0, 1, 2, 3], dur=1/2, sus=2)
```

soit en définissant la valeur directement dans les attributs de l'objet player :

```
p1 >> pluck()
p1.degree = [0, 1, 2, 3]
p1.dur = 1/2
p1.sus = 2
```

Les principaux attributs et effets sont présentés et illustrés dans la documentation FoxDot, disponible en ligne. Étant un langage basé sur Python, il est possible d'intégrer les players dans une structure conditionnelle ou de contrôle - avec la syntaxe de Python. Ainsi les possibilités de composition musicale sont vastes et faciles à prendre en main.

1.1.2 La structure de Troop.

Une fois que Troop est installé sur une machine, deux fichiers Python sont exécutables via un terminal de commande : `run-server.py` et `run-client.py`. Comme leur intitulé l'indique, ils permettent d'instancier, respectivement, un serveur Troop et un client Troop. L'interface graphique utilisateur est le client Troop. C'est donc sur cette partie que nous allons nous pencher, puisque TroopRasp est une extension de celle-ci.

Troop est une application développée en python au moyen de la programmation orientée objet. Elle se décompose en plusieurs classes et objets qui interagissent entre eux et permettent ainsi le live-coding collaboratif. Lorsque l'utilisateur exécute le fichier `run-client.py`, un objet de la classe `Client` est instancié. Dans un premier temps, une fenêtre s'ouvre. Cette interface est un objet `ConnectionInput` et sert à obtenir les informations de connexion de l'utilisateur, telles que l'adresse du serveur, le port de connexion, le nom du client, le mot de passe de la session (peut être vide) et le langage utilisé pour la session. En validant les entrées par l'interface, l'objet `ConnectionInput` va les communiquer à l'objet *Client* en tant qu'attribut de classe.

Le client va ensuite instancier un objet de la classe `Sender` et un objet de la classe `Receiver`. L'objet `Sender` se charge d'envoyer les messages au serveur Troop en s'y connectant par une socket et en démarrant une instance d'écoute sur la machine. L'objet `Receiver` écoute les messages du server Troop via la connexion établie par le `Sender`, et met à jour l'interface selon les messages reçus. Au client est aussi associé un interpréteur dépendant du langage choisi lors de la connexion de l'utilisateur, qui exécutera les blocs de code en gérant un sous-processus. Dans notre cas, nous resterons sur le langage FoxDot, c'est donc

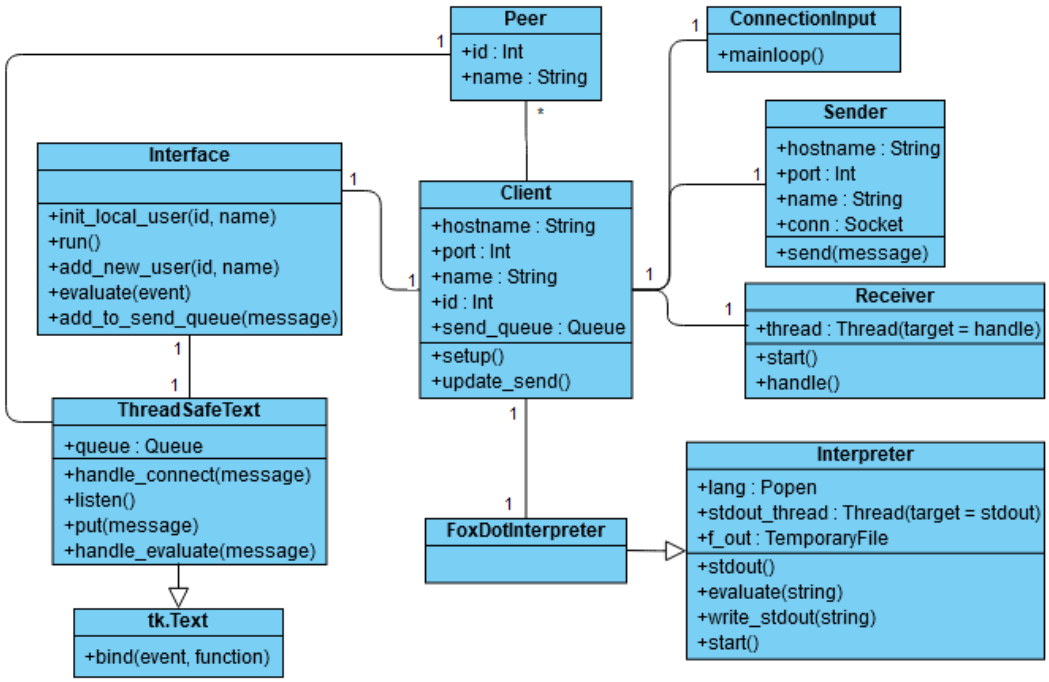

FIGURE 1.2 – Diagramme de Classe non-exhaustif du client Troop.

un objet de la classe `FoxDotInterpreter` qui sera instancié. Toutefois, les interpréteurs spécifiques, dont celui pour FoxDot, sont tous dérivés de la même classe mère `Interpreter`.

L'objet `Client` possède un attribut qui est une liste d'objets `Peer`. La classe `Peer` représente les performeurs connectés à la session et ceux-ci possèdent un identifiant unique et un nom. Enfin, l'interface graphique est créée. Elle est composée de plusieurs éléments graphiques qui sont des objets, tel que la console Troop, la barre de menu ou encore le graphique. Le seul que nous allons présenter est la classe `ThreadSafeText` qui représente la zone de texte où les utilisateurs peuvent coder. C'est via cet objet que nous avons accès à l'objet `Peer` correspondant au client local.

La communication client-serveur se fait via des objets `Message`. La classe `Message` se définit par un contenu et un identifiant qui est celui du `Peer` à l'origine du message. Les messages se déclinent en différents types, chacun représenté par une classe dérivée de la classe `Message`. Suivant le type de message (message de connection, d'opération, d'évaluation de code, message console, etc.), celui-ci peut avoir des attributs spécifiques en plus des deux de bases. C'est grâce à la distinction des messages, que le client Troop va pouvoir gérer la communication et les traiter de manière adaptée.

1.1.3 Communication Client.s-Serveur.

La gestion des messages client s'articule autour de deux instances de la classe `Queue` du module python du même nom. Ce module implémente des files multi-productrices et multi-consommatrices, particulièrement utiles lorsque les informations doivent être échangées sans risques entre plusieurs threads. L'objet `Queue` est une file FIFO, *First In First Out.* Les deux files correspondent à l'attribut `send_queue` de l'objet Client et à l'attribut `queue` de l'objet `ThreadSafeText`. La première stocke les messages qui vont être envoyés au serveur Troop et la deuxième les messages reçus par le serveur.

Voyons plus en détails le chemin d'un message contenant le code FoxDot qu'un client veut évaluer. Depuis l'interface graphique Troop Client, l'utilisateur exécute une ou plusieurs lignes de code en positionnant son curseur à la ligne ou au bloc voulu, puis presse les touches Ctrl+Entr. Grâce à une méthode de la classe `Text` - provenant du module python Tkinter - cette entrée clavier est associée à la méthode `evaluate` de l'objet Interface. Cette méthode trouve le bloc de code à évaluer via la localisation du curseur du peer local et transmet le message à la méthode `add_to_send_queue`. Celle-ci ajoute le message à la file `send_queue`.

Cette file d'attente est constamment scrutée par la méthode `update_send` de l'objet `Client`, une méthode récursive lancée dans le constructeur du client par la méthode `run` de l'interface. La méthode récupère un à un les message de la file et les transmet à l'objet `Sender` qui se charge de les envoyer au serveur Troop. Le serveur va alors diffuser chaque message à tous les clients connectés, également au client qui est à l'origine du message.

Le retour serveur est récupéré par le thread `handle` de l'objet `Receiver` qui stocke chaque message dans la file `queue` de l'objet `ThreadSafeText`. Dès la création de cet objet, sa méthode `listen` scrute la file et passe chaque message dans le bon gestionnaire, en fonction de son type. Concernant le type `MSG_EVALUATE_BLOCK`, le message est pris par le gestionnaire `handle_evaluate` qui transmet le message à l'objet `Interpreter` via sa méthode `evaluate`. Le message, soit les lignes FoxDot évaluées et le nom du client, est affiché dans la console Troop de l'interface graphique et envoyé au sous-processus FoxDot.

La sortie standard du sous-processus est capturée dans un fichier temporaire. Ainsi, dans le cas où le code est une requête, un "print" par exemple, la réponse sera stockée dans ce fichier. Le thread `stdout` le lit continuellement et affiche les réponses dans la console Troop du client.

1.2 Approche pour le live-coding intéractif

Un axe de ce travail vise à étendre la pratique du live-coding à celle des musiques interactives, c'est-à-dire aux réalisations musicales contrôlées tout ou partie par l'audience, et

non seulement par les interprètes. On veut rendre possible l'utilisation d'interfaces durant une performance de live-coding, ou encore d'exploiter la flexibilité des outils de live-coding pour la sonorisation d'installations artistiques interactives [1]. L'enjeu est alors de fournir à l'artiste le moyen de composer et d'exprimer son intention tout en la paramétrant avec les interactions possibles. On peut ici faire référence au concept de "partition virtuelle" (issu de la programmation interactive en musique assistée par ordinateur) qui définit une organisation musicale dans laquelle on spécifie des paramètres dépendant de l'environnement de la performance.

Cette section esquisse et justifie la formalisation retenue pour enrichir Troop afin qu'il puisse être utilisé dans des performances interactives de live-coding.

1.2.1 Formalisation de l'interaction

Une interaction est ici formalisée au travers du concept d'*évènement*. Un tel événement est déclenché ou tiré à partir de mesures sur l'interface. Une mesure provient typiquement d'un capteur connecté à une entrée d'un des calculateurs prenant part à la session Troop, et un événement peut correspondre à :

— l'activation d'un capteur binaire (parfois appelé Tout-Ou-Rien) ;
— au franchissement d'un seuil pour une valeur mesurée à l'aide d'un capteur analogique ;
— la validation d'une expression logique incluant des mesures de capteurs binaires et/ou analogiques.

Pour se conformer au cadre d'étude des systèmes à événements discrets (abordé lors de la section 1.3), l'occurrence des événements est considérée comme instantanée et asynchrone. Le caractère asynchrone transcrit l'hypothèse que l'instant d'occurrence d'un événement ne dépend ni des autres événements, ni du « temps musical » de la performance. Cette dernière hypothèse se justifie par le modèle d'exécution d'une performance avec FoxDot/Troop : le temps y est géré à l'aide d'un unique 'métronome' appelé `Clock` qui cadence l'ensemble des *players* de la performance. Similaire à l'abstration du `TempoClock` de supercollider, ce véritable séquenceur de l'exécution des *players* prend la forme d'une file d'attente. Quand un *player* est instancié, ses notes sont ajoutées à cette file d'attente avec des valeurs temporelles associées qui indiquent quand elles doivent être jouées. A chaque modification, la file d'attente est ré-ordonnée selon l'ordre croissant de toutes les valeurs temporelles. Pendant l'exécution, l'objet `Clock` incrémente continuellement un chronomètre interne. Lorsque ce chronomètre atteint l'heure programmée du premier élément de sa file d'attente, cet élé-

1. Schématiquement, les installations artistiques interactives utilisent des capteurs mesurant des grandeurs au sein du public (voire l'environnement ou la nature), et l'auteur programme les réponses ou réactions particulières aux mesures : le public et la machine travaillent ou jouent ensemble dans un dialogue qui produit en temps réel une œuvre d'art unique.

ment est retiré de la file d'attente et est ensuite appelé (envoi à SuperCollider d'un message OSC pour jouer la note). Notons que les notes peuvent être de n'importe quelle durée et des polyrythmes complexes peuvent être créés facilement en programmant plusieurs *players* avec des durées de notes inégales.

1.2.2 Composition paramétrée par l'interaction

L'objectif est ici de définir le cadre choisi pour la composition d'une performance de live-coding incluant des interactions. Autrement dit, on précise les fonctionnalités que l'on souhaite offrir aux compositeurs pour élaborer une composition musicale dont l'interprétation est paramétrée par l'occurrence d'événements. En programmation interactive pour la musique assistée par ordinateur, on parle de *partition virtuelle*. Chaque *player* pouvant être vu comme un instrument musical indépendant au sein de la 'performance orchestrale' que constitue une session Troop, on préfère parler d'*orchestration virtuelle* dans la suite de ce document.

Concrètement, dans notre travail basé sur Troop, une orchestration virtuelle doit permettre d'instancier des *players* et/ou modifier des attributs des *players* existants lors de l'occurrence d'événements. À la manière d'une partition musicale où une ligne de portée correspondrait à un *player*, un extrait d'interprétation d'une orchestration virtuelle (telle qu'imaginée dans ce travail) est représenté sur la figure 1.3.

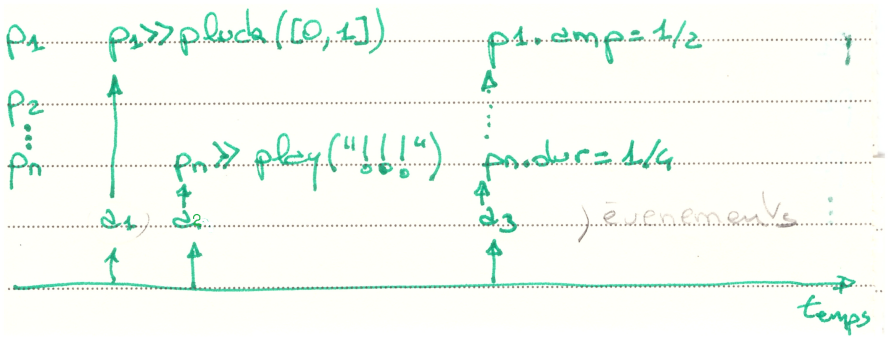

FIGURE 1.3 – Exemple d'interprétation dune orchestration virtuelle.

Sur le schéma 1.3, les événements a_1, a_2 et a_3 surviennent successivement, et au moment de leurs occurrences les *players* p_1 et p_n sont instanciés ou modifiés. Soulignons qu'à la différence d'une partition musicale où la hampe d'une note sert à préciser sa durée et le tempo précise l'intervalle de temps entre deux notes successives, l'axe des temps d'une orchestration virtuelle ne peut pas être discrétisé (puisque les événements surviennent de façon asynchrone) et l'intervalle de temps entre deux événements successifs ne peut pas être précisé (puisqu'il dépend de l'interaction que l'interprète ne maîtrise pas nécessairement).

Pour formaliser l'orchestration virtuelle tout en offrant de la latitude au compositeur, on vise à ce que le modèle retenu permette de définir une orchestration :

— qui tient compte de l'ordre dans lequel les événements sont déclenchés ;
— qui dépend de l'état d'avancement de l'interprétation ;
— qui permet de synchroniser plusieurs instanciations/modifications sur l'occurrence d'un même événement.

Pour satisfaire à ces objectifs, on se propose par la suite de formaliser l'orchestration virtuelle sous la forme d'un système à événements discrets, et plus précisément de la modéliser à l'aide d'automates finis.

1.3 Automates finis comme modèle et outil d'analyse du comportement d'un système à événements discrets

Un système à événements discrets (SED) est un système dont l'espace d'état est discret, et dont l'évolution de l'état dépend entièrement de l'occurrence d'événements instantanés et asynchrones [2]. Le formalisme des SED constitue une abstraction adaptée pour l'étude d'un bon nombre de systèmes, et les développements de cette théorie fournissent des résultats utiles à leur modélisation, leur analyse, leur contrôle ou encore leur validation.

Parmi les modèles pour les SED, les automates finis constituent la classe la plus basique. Ils sont intuitifs, faciles à utiliser, se prêtent aux opérations de composition et à l'analyse. Ils nous apparaissent comme le formalisme le plus simple à même de capturer les phénomènes que l'on souhaite pouvoir mettre en œuvre dans une orchestration virtuelle (voir section 1.2.2).

Dans la suite, nous allons introduire les notions élémentaires de la théorie des langages formels, à savoir les concepts d'*alphabet*, de *mot* et de *langage*, puis nous introduirons brièvement le formalisme des automates finis (pour plus de détails, le lecteur est invité à consulter [2]). Nous présenterons ensuite des éléments d'analyse du comportement d'un système à partir de son modèle sous forme d'automates finis. Cette partie permettra d'esquisser comment ces résultats pourront être utiles à la conception et la validation d'orchestrations virtuelles.

1.3.1 Alphabet et mots

Un *alphabet*, noté Σ, dans le contexte des langages formels, est un ensemble non vide de symboles. Les éléments de Σ sont appelés des *lettres*. Un *mot* sur un alphabet correspond à une séquence finie de lettres. L'ensemble des mots construits sur l'alphabet Σ est noté $\Sigma^\star$.

Exemple 1.1
Prenons l'alphabet $\Sigma = \{0, 1\}$*, voici quelques éléments de* $\Sigma^\star$ *:*

$$0000000, 001000, 11100000, \ldots$$

Ici, $\Sigma^\star$ *n'est rien d'autre que l'ensemble de toutes les suites finies de bits que l'on peut écrire y compris la suite vide.*

1.3.2 Langage

Un *Langage* L défini sur un alphabet Σ est un ensemble de mots sur cet alphabet.

Exemple 1.2
Soit l'alphabet $\sigma = \{a, b\}$*, on peut considérer les langages suivants :*

$$L_1 = \{aab, aa, bbba\}, L_2 = \{a, b\} = \Sigma, \ldots$$

1.3.3 Automates déterministes à états finis

Un automate fini est une machine "abstraite" qui prend en entrée une suite de symboles et qui effectue une reconnaissance de cette suite. Si la reconnaissance se termine, on dit que l'automate accepte ou reconnaît la suite. Le *langage* reconnu par un automate est l'ensemble des suites qu'il accepte. Plus formellement, un *automate fini déterministe* G est un quintuplet :

$$G = (X, \Sigma, \delta, x_0, X_m)$$

où

- X est un ensemble fini d'état ;
- Σ est un alphabet ;
- $\delta : X \times \Sigma \to X$ est une *fonction de transition* ;
- $x_0 \in X$ est un *état initial* ;
- $X_m \subset X$ est un ensemble d'*états finaux* (ou *états marqués*).

L'alphabet représente l'ensemble des évènements tandis que la fonction de transition spécifie la dynamique de l'automate : si $x' = \delta(x, e)$ alors l'occurrence de l'événement e lorsque l'état actuel de l'automate est x, conduit à l'état x'.

Exemple 1.3
Prenons $X = \{x_0, x_1, x_2\}$ *un ensemble d'état,* $\Sigma = \{a, b, c, d\}$ *un alphabet,* x_0 *un état initial,* $X_m = \{x_0\}$ *un ensemble d'états finaux et* δ *définie par le tableau suivant :*

δ	a	b	c	d
x_0	x_1			
x_1		x_2		x_0
x_2			x_2	x_0

Dans ce tableau, la valeur de x_1 à l'intersection entre la ligne x_0 et la colonne a indique que $\delta(x_0, a) = x_1$. Une case vide, comme celle à l'intersection entre la ligne x_0 et la colonne b, indique que la transition correspondante n'est pas définie.

Un automate peut être décrit par un graphe dans lequel chaque état correspond à un nœud et est représenté par un cercle. En particulier, l'état initial est représenté par un cercle avec une flèche d'entrée et un état final est représenté par un double cercle. Si $x_2 = \delta(x_1, e)$ il y aura une flèche dirigée à partir du nœud x_1 au nœud x_2 marquée par le symbole e pour représenter la transition de x_1 à x_2.

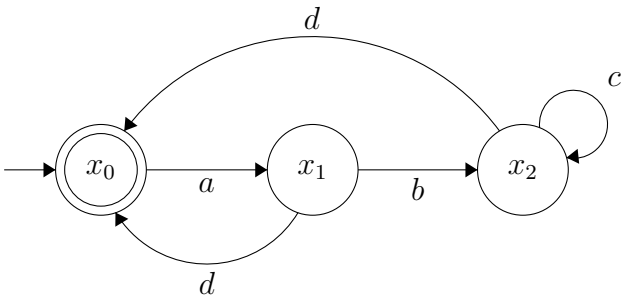

FIGURE 1.4 – Graphe représentant l'automate fini dont les transitions d'état sont définies par le tableau précédent.

1.3.4 Analyse du comportement d'un système.

L'une des motivations à utiliser les automates à états finis est leur propension à répondre à diverses questions d'analyse sur le comportement du système modélisé.

Par exemple, étant donné un automate G, nous pouvons nous intéresser aux questions qui suivent :

— Un état jugé "indésirable" peut-il être atteint depuis certains états dans G ?

— L'état initial de l'automate G est-il atteignable à partir de chaque état ? En d'autres termes, le système peut-il être réinitialisé ?

Parmi les questions d'analyse, on met en avant dans cette section comment vérifier formellement des propriétés, en illustrant l'intérêt pour la composition d'orchestrations virtuelles.

Méthodes formelles de vérification

La modélisation d'une orchestration virtuelle sous la forme d'automates finis partageant des événements va permettre de les vérifier. Tout d'abord, il faut bien distinguer les actions de **Tester** (Fig. 1.5) et de **Vérifier** (Fig. 1.6).

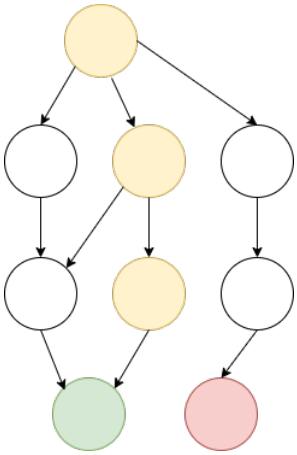

FIGURE 1.5 – Test : à partir d'une donnée, un seul chemin est testé.

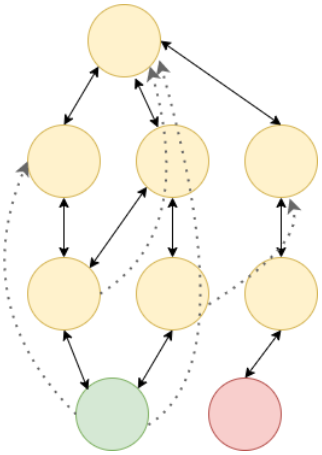

FIGURE 1.6 – Vérification : tous les chemins sont testés.

L'objectif à terme est d'utiliser les méthodes de Model-Checking afin de vérifier des propriétés sur les orchestrations virtuelles. Le model-checking [3] est une approche automatisée permettant de vérifier qu'un modèle de système est conforme à des spécifications. Le comportement du système est formellement modélisé, via des automates, réseaux de Petri, algèbres de processus, ... et les spécifications, exprimant les propriétés attendues du système, sont formellement exprimées par exemple via des formules de logiques temporelles.

La figure 1.7, tirée de [12], donne un aperçu simplifié du cycle d'utilisation du model-checking. Le cycle peut se diviser en trois phases :

1. Modélisation formelle du comportement du système
2. Expression formelle des propriétés attendues
3. Si une propriété n'est pas satisfaite, un contre-exemple est produit qui décrit un scénario possible d'erreur (i.e de violation de la propriété). L'analyse de celui-ci aide à apporter les corrections nécessaires que ce soit sur la modélisation du comportement du système ou sur l'expression formelle des propriétés attendues.

Ce cycle est répété jusqu'à ce que toutes les formules, c'est-à-dire toutes les spécifications, soient vérifiées.

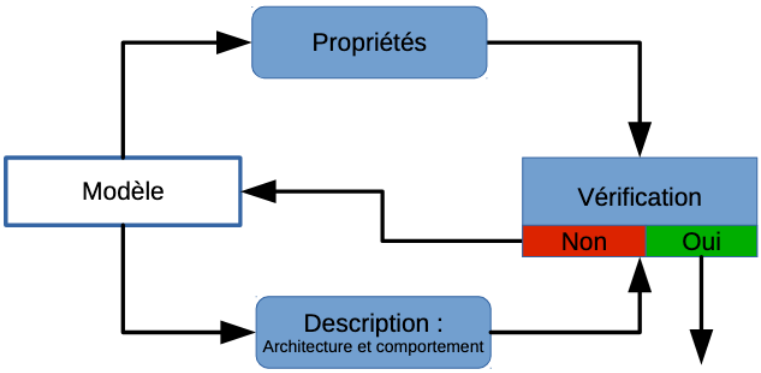

FIGURE 1.7 – Cycle d'utilisation du model-checking.

Dans le cadre du live-coding et des propriétés d'interactivité que nous souhaitons ajouter au logiciel Troop. Plusieurs propriétés dites temporelles peuvent être intéressantes à vérifier :

- **Accessibilité.** *Une certaine situation peut être atteinte.*
  - Une combinaison d'états dans les différents automates finis représentant l'orchestration peuvent-ils atteints simultanément ?
  - L'ensemble des états finaux dans les automates finis modélisant une orchestration peuvent-ils être atteints ?
- **Invariance** *Tous les états du système satisfont une bonne propriété.*
  - Tous les états conduisent à ce qu'au moins un *player* soit actif.
- **Sûreté**. *Quelque chose de mauvais n'arrive jamais.*
  - Chaque fois que je modifie un *player*, celui-ci est-il bien instancié ?
- **Vivacité** . *Quelque chose de bon finira par arriver.*
  - Quand l'amplitude d'un *player* est définie à une valeur importante, elle finira par être baissée.
  - Quand une orchestration est lancée, elle finir par s'achever.
- **Absence de blocage**. *Le système ne se bloque pas.*
  - (non blocage total) Il existe au moins une exécution infinie de la machine.
  - (non blocage partiel) Il n'y a pas d'état bloquant.

Les logiques temporelles - LTL, CTL et CTL*

L'objectif de cette section n'est pas de rentrer dans le détail des logiques temporelles, mais uniquement de donner une intuition de leur utilité dans le cadre de la vérification des orchestrations. Les logiques temporelles permettent de décrire le séquencement d'évènements observés et peuvent exprimer par exemple la causalité : "*chaque fois que j'observe q, j'ai observé p avant*". Ces logiques ne manipulent jamais explicitement le temps comme

durée absolue : entre deux évènements observés (observations), il peut a priori se passer une seconde ou une journée. On ne pourra pas exprimer par exemple que "*quand je vois p, je vois ensuite q exactement 3 secondes après*". On pourra juste dire : "*quand je vois p, je vois ensuite q exactement 3 observations après*". Une formule en logique temporelle peut prendre la forme qui suit :

$$\phi U \psi.$$

Ce qui signifie que la formule ϕ est satisfaite jusqu'à ce que la formule ψ le soit. Le terme U signifie : jusqu'à (U vient de l'anglais Until).

En vérification, trois logiques temporelles sont principalement utilisées. Tout d'abord, la logique LTL permet de modéliser l'avenir d'un chemin infini dans un système de transitions, par exemple une condition va finir par être vraie, une condition sera vraie jusqu'à ce qu'une autre devienne vraie, etc .... Cette logique temporelle est la plus "simple" à comprendre. La logique temporelle arborescente CTL (Computational Tree Logic) permet d'exprimer des propriétés que l'on ne peut exprimer en LTL, et vice-versa. CTL présente l'avantage que l'on peut entièrement vérifier algorithmiquement (par "Model Checking") si une formule est vraie ou fausse sur un graphe d'états donné. En CTL, pour construire une formule logique, il faut considérer deux "dimensions" :

- Sur la première dimension à partir d'un état initial donné, on peut demander qu'une propriété soit vraie pour au moins un choix de branche ("there **E**xists a path such that ...") ou bien qu'elle soit vraie pour tous les choix, c'est-à-dire tous les futurs possibles ("for **A**ll possible paths ...").
- Sur la seconde dimension, partant d'un état initial le long d'un chemin donné, on peut demander qu'une propriété soit vraie : à l'instant juste après l'état initial ("ne**X**t state"), au moins à un instant donné sur le chemin ("some **F**uture state"), tout le temps ("**G**lobally") ou jusqu'à ce qu'une autre propriété devienne vraie ("... **U**ntil ...").

On obtient alors des formules logiques de la forme $AX(\phi)$ qui signifie que ϕ est vraie pour tous les états qui suivent immédiatement l'état initial, $EG(\phi)$ signifie qu'il existe au moins un chemin partant de l'état initial le long duquel ϕ reste tout le temps vraie. Enfin, la logique CTL* est une généralisation de la logique CTL et de la logique temporelle linéaire. Pour plus de détails sur les logiques temporelles, le lecteur pourra se reporter à ouvrage [3].

Model Checking (Vérification de modèle)

Enfin, la finalité est de confronter les modèles aux formules logiques à l'aide du Model Checking. Un outil de model checking [9] est capable de fournir l'ensemble des états du graphe d'états qui satisfont la formule, avec des performances algorithmiques remarquables.

Intuitivement, si on considère la formule

$$(x = 0) \quad \Rightarrow \quad AG(\neg(x = 2))$$

les algorithmes de Model Checking traitent ces formules en partant de leurs atomes ($x = 0$ ou $x = 2$ dans l'exemple précédent) et étiquètent les états du graphe qui les satisfont. Ensuite, il faut remonter les connecteurs et modalités qui apparaissent dans la formule.

La Figure 1.8 résume la future utilisation d'un solveur de Model Checking dans la vérification de propriété dans les orchestrations.

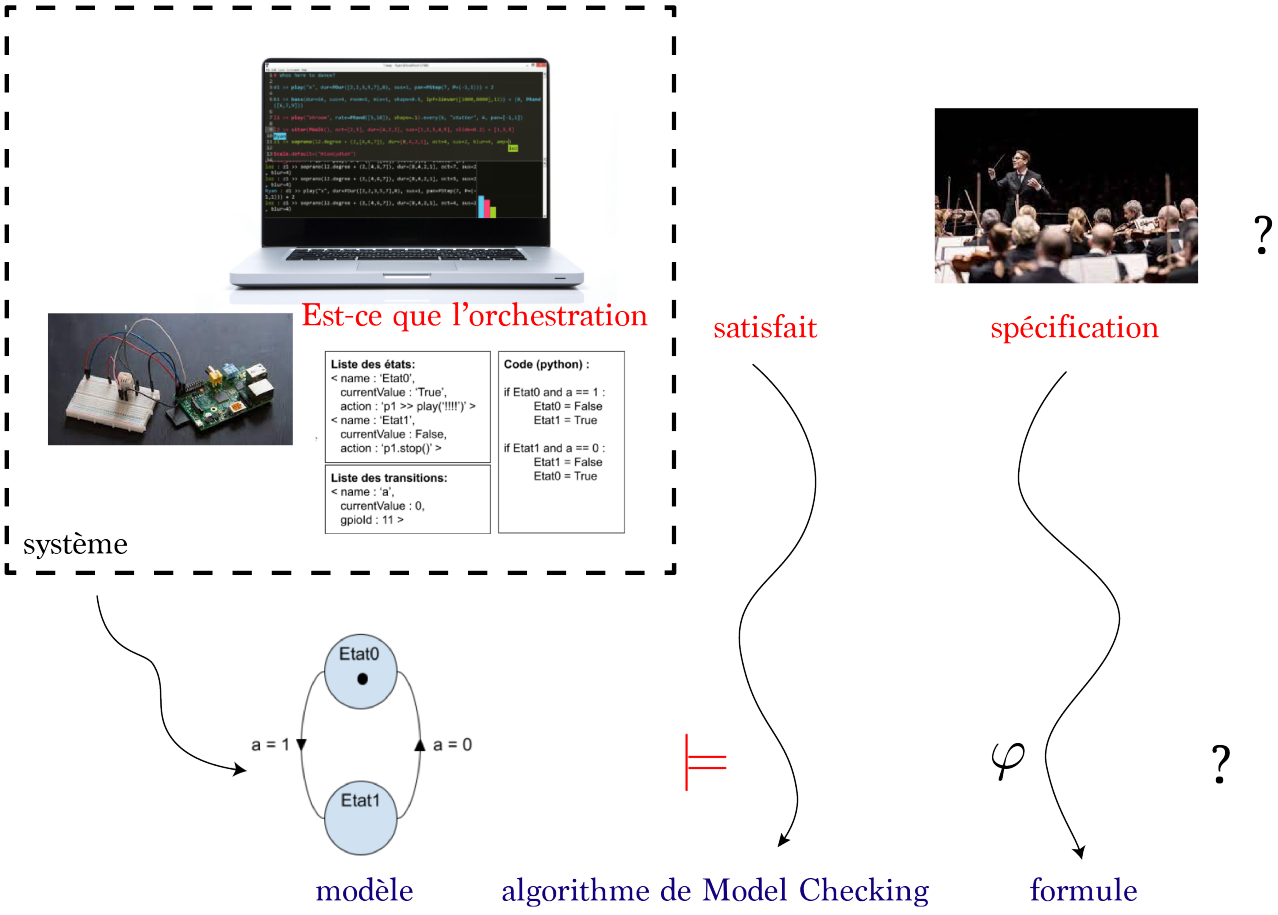

FIGURE 1.8 – Vérification de l'orchestration

Chapitre 2

Travail réalisé

Le travail réalisé durant ce stage s'est déroulé sur deux axes. Le premier, Sensor Interaction, consistait en une preuve de concept d'une interface permettant l'interactivité entre un capteur et un *player* FoxDot et le second, Orchestration, l'implémentation d'orchestrations virtuelles. Le premier a nécessité un travail autour de la communication entre le serveur et le client Troop. Cette dernière déjà existante - voir 1.1.3 - l'objectif fut d'intercepter cette communication pour récupérer la réponse du serveur à la requête du client, sans interrompre le thread *stdout*. Puis, une fois réalisé, il a fallu implémenter et optimiser l'interaction avec les ports GPIO - General Purpose Input Output - soit les ports d'entrée-sortie de la Raspberry Pi. Pour la mise en œuvre d'orchestrations virtuelles, nous avons automatisé la construction d'automates finis en python et intégré les fonctionnalités précédentes. Dans cette partie, nous allons détailler ces points clefs du stage.

2.1 Communication avec l'interpréteur FoxDot.

Pour permettre une interface plus accessible mais surtout pour des soucis de fonctionnement, il est indispensable de pouvoir envoyer des requêtes à l'interpréteur et de récupérer les réponses. Par exemple, avec la fonctionnalité Sensor Interaction, nous associons la valeur d'une entrée de la raspberry pi à un paramètre d'un player. Ce player doit être au préalable instancié dans l'interface Troop. Pour assurer une saisie correcte de la part de l'utilisateur quant au choix du player, nous soumettons à l'utilisateur la liste des players existants et en train de générer de la musique. La commande `print(Clock.playing)` permet d'avoir accès à cette liste. La mise à jour du paramètre se stoppe définitivement lorsque le player est lui-même arrêté. La commande FoxDot utile est `print(nom_du_player.isplaying)`, qui affiche une valeur booléenne : True si le player joue, False sinon. D'autre part, pour l'orchestration, nous avons besoin de récupérer la valeur des variables indiquant si les états des automates finis sont actifs ou non. Dans ces différentes situations, nous devons

communiquer avec l'interpréteur FoxDot et avoir accès aux messages retour.

Or, tel quel, Troop ne permet pas de récupérer les réponses de l'interpréteur. Le flux des messages est lié à un fichier temporaire qui est lu constamment par un thread et affiche les messages dans la console client Troop - voir partie 1.1.3. Les fonctions implémentées dans Troop n'offrent pas d'accès aux messages FoxDot pour les manipuler dans un deuxième temps. Il était donc nécessaire de modifier le processus existant afin de l'adapter à nos besoins. L'objectif n'était pas de déconstruire totalement le fonctionnement de base. Au contraire, il était essentiel de le préserver et uniquement d'y rajouter une deuxième voie donnant accès aux messages que l'on souhaite.

Les messages qui nous intéressent sont identifiables par un préfixe pour qu'ils puissent être distingués des autres. En effet, uniquement les messages réponses aux requêtes que nous soumettons doivent être retournés. Pour ce faire, nous insérons un préfixe - une courte chaîne de caractère - au message dans la requête qui se conservera dans la réponse. Dans le thread *stdout*, qui gère le retour du processus FoxDot, l'emploi d'une expression régulière permet de vérifier la présence, ou non, du préfixe. S'il est identifié, alors le message est retourné. L'identification des messages nous permet également de les trier pour l'affichage console. Comme expliqué préalablement, la méthode *stdout* affiche tous les messages dans la console du client Troop. Pour éviter cette pollution visuelle, nous soustrayons les messages réponses de nos requêtes de cette étape.

Cependant, la fonction *stdout* étant un thread, l'emploi d'un `return` le stopperait. Pour éviter ce comportement, une variable partagée est utilisée pour stocker les messages. C'est dans un autre processus que nous lisons cette variable. Comme elle est sollicitée dans deux processus s'exécutant simultanément, il existe un potentiel conflit de lecture-écriture. Pour y remédier, nous avons implémenté un système de sémaphore pour la protéger. Ceci consiste en un verrou qui va être fermé dès que la variable est sollicitée, puis libérée quand la manipulation est terminée. Lorsque la variable est verrouillée et qu'un processus veut faire une manipulation sur elle, il attend jusqu'à ce qu'elle soit de nouveau libre. Ainsi, aucun conflit n'est possible : à aucun moment un processus tentera de lire la variable alors qu'un autre écrit dedans au même instant.

Avec la modification de la méthode *stdout* de la classe Interpreter et la création d'une nouvelle méthode, nous pouvons envoyer des requêtes au serveur et récupérer la réponse pour connaître l'état de la session de live-coding à tout instant. La seule contrainte est d'utiliser le langage FoxDot pour la session. Toutefois, ce langage est le plus employé et Troop est par défaut configuré pour celui-ci.

2.2 Manipulation du GPIO.

Il existe de nombreuses bibliothèques dédiées au Raspberry Pi permettant de manipuler le port GPIO. Pour la plupart, elles sont accessibles sur Pypi. Celle que nous utilisons pour le projet est la bibliothèque historique : RPi.GPIO. Ce module fournit les ressources nécessaires pour initialiser les entrées/sorties et donne des accès en lecture/écriture. Concernant la configuration générale des pins, la Raspberry Pi autorise deux numérotations (fig. 2.2) : celle de la sérigraphie du connecteur de la carte - GPIO.BOARD - ou la numérotation électronique de la puce - GPIO.BCM. Dans le cas de ce travail, le choix s'est porté sur la première, de manière arbitraire.

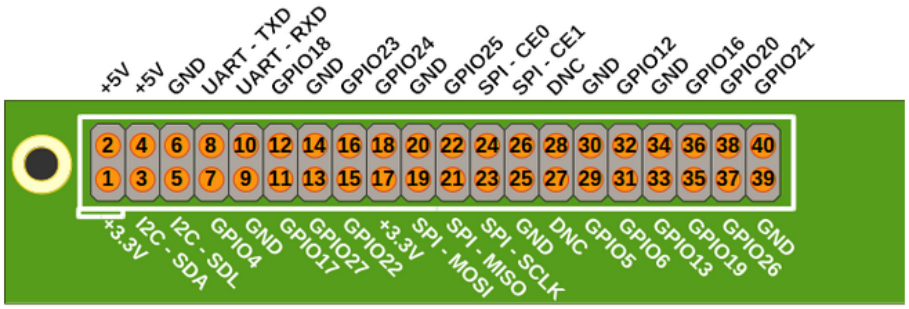

FIGURE 2.1 – Fonctionnalités étendues du port GPIO.

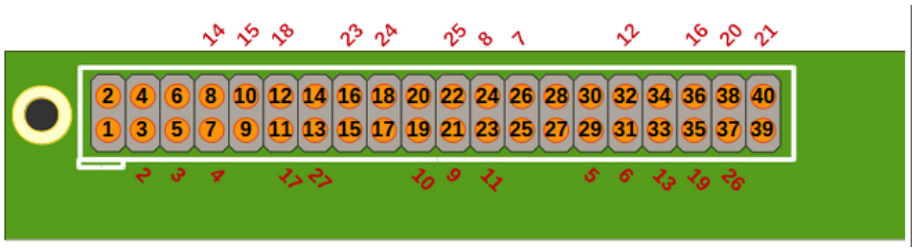

FIGURE 2.2 – En noir la numérotation GPIO.BOARD, en rouge GPIO.BCM.

Afin d'initialiser une entrée/sortie, il suffit de préciser son numéro de pin, en fonction de la configuration précédemment choisie, son paramétrage (entrée ou sortie) et éventuellement son état initial si c'est une sortie. Dans cette étude, nous avons considéré uniquement des entrées connectées à des capteurs de type TOR - tout ou rien - comme des interrupteurs ou des boutons poussoirs. Ensuite, la lecture se fait simplement en précisant le numéro de la pin. Afin d'éviter de laisser flottante toute entrée, il est possible de connecter des résistances de *pull-up* ou *pull-down* en interne. Ceci évite de laisser une entrée - ou une sortie - dans un état incertain, en forçant une connexion à la masse ou à un potentiel donné. Nous avons choisi cette fonctionnalité pour simplifier le montage, étant amené à connecter un ensemble de capteurs TOR pour les transitions d'une orchestration. Actuellement, TroopRasp oblige le système pull-up mais il est envisageable de laisser le choix à l'utilisateur

via l'interface graphique. Ceci laisserait plus de liberté, cependant, cela demanderait une connaissance sur le sujet et impacterait l'accessibilité de l'interface suivant le public visé.

Que ce soit pour la fonctionnalité Sensor Interaction ou Orchestration de TroopRasp, c'est le changement de valeur des entrées GPIO qui déclenche la mise à jour : respectivement, l'évaluation de la nouvelle valeur du paramètre du player FoxDot ou l'évaluation de des états des automates finis. Pour éviter une surcharge en scrutant constamment les entrées, nous instancions un gestionnaire d'évènements basé sur la détection de front. La méthode *add_ event_ detect* du module RPi.GPIO crée un gestionnaire d'évènement - ou *event listener* en anglais - sur un canal précis, soit une entrée GPIO spécifique. La méthode requière également en paramètre d'entrée le type de front qui entraînera un événement : front montant, front descendant ou les deux. En cas d'événement, un processus parallèle est lancé, aussi appelé une fonction de *callback*. C'est au sein de cette dernière que nous codons les opérations de mise à jour.

2.3 Implémentation d'orchestrations virtuelles.

Comme discuté dans la section 1.2.2, le concept d'orchestration virtuelle définit une organisation musicale dans laquelle on spécifie des paramètres dépendant de l'environnement de la performance. On propose de modéliser une orchestration virtuelle sous la forme d'un ou plusieurs automates finis. Nous allons ici aborder la structure d'une orchestration virtuelle- c'est-à-dire, comment nous avons décomposé les automates finis au moyen des ressources python - puis son évolution dynamique et enfin son automatisation.

2.3.1 Structure d'une orchestration.

Une orchestration virtuelle, vue comme un système à évènements discrets, plus précisément modélisée à l'aide d'automates finis, se décompose en trois parties (voir figure 2.3 :

— la liste des états ;
— la liste des transitions ;
— la relation entre les valeurs d'états et les valeurs de transitions (transitions d'état).

Chaque état et transition sont caractérisés par un nom, permettant de les dissocier, et d'une valeur courante. La valeur d'une transition suit la valeur d'une entrée Raspberry Pi dont l'identifiant - le numéro de la pin - est spécifiée à sa création. Quant à la valeur d'un état, elle dépend de la valeur des transitions selon une structure conditionnelle propre à l'automate. Un état est également caractérisé par l'action qu'il engendre quand il est actif. TroopRasp étant un outils pour le live-coding, les actions des états sont des commandes FoxDot générant ou modifiant de la musique.

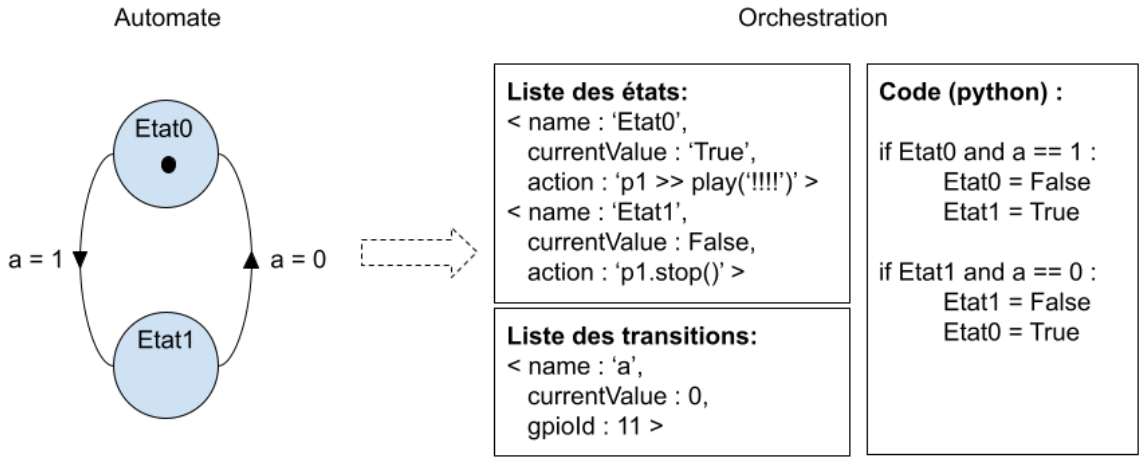

FIGURE 2.3 – Exemple d'automate et d'orchestration correspondant, à l'état initial t=0.

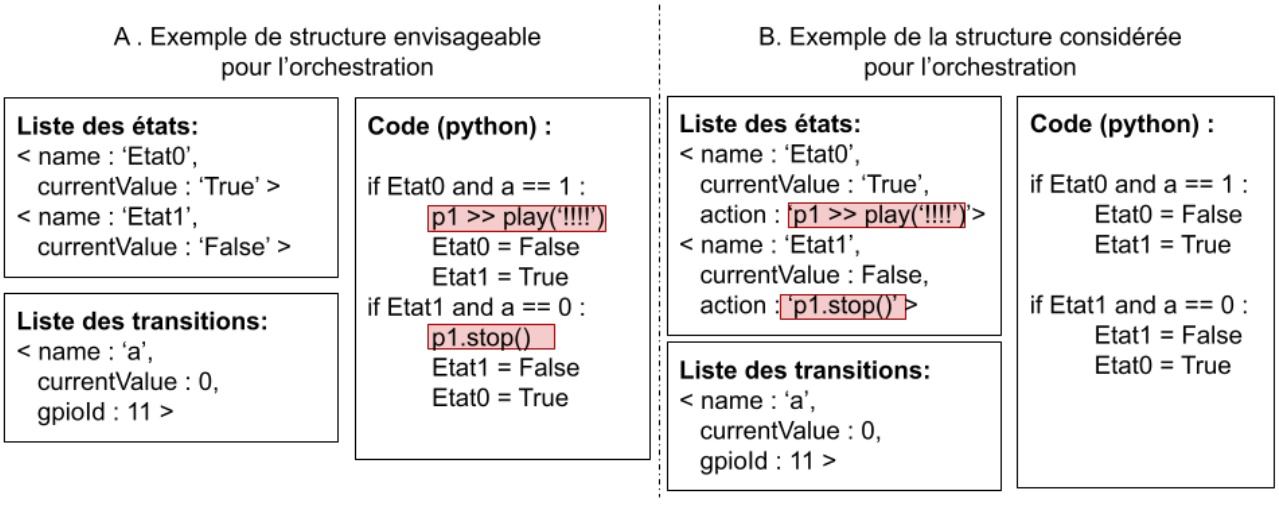

FIGURE 2.4 – Exemples illustrant les structures pour une même orchestration.

Comme l'illustre la figure 2.4, il est envisageable d'intégrer les actions des états dans la structure qui conditionne leur valeur. Cependant, les méthodes d'évaluation des commandes FoxDot en interne, rend cette solution moins adaptée que celle retenue. En effet, nous pouvons soit évaluer une commande en passant directement par l'interpréteur FoxDot, soit en l'ajoutant à la queue des messages pour le serveur Troop (cf. 1.1.3). Par la première méthode, le message sera évalué localement, contrairement à la seconde qui l'évaluera sur toutes les machines clientes et affichera le message dans la console Troop de chaque client. Or, les actions des états doivent être évaluées sur chacune des machines pour que la musique générée soit audible par tous et que les players instanciés soient manipulables par les autres clients, forçant l'emploi de la deuxième méthode. Cependant, si nous évaluons le script python de l'automate ainsi, celui-ci s'afficherait à chaque mise à jour dans les consoles Troop. Ceci entraînerait une pollution visuelle contraignante pour une session de live-coding. En séparant les actions des états du code qui détermine leurs valeurs, nous pouvons utiliser la méthode d'évaluation la plus adaptée pour chacun.

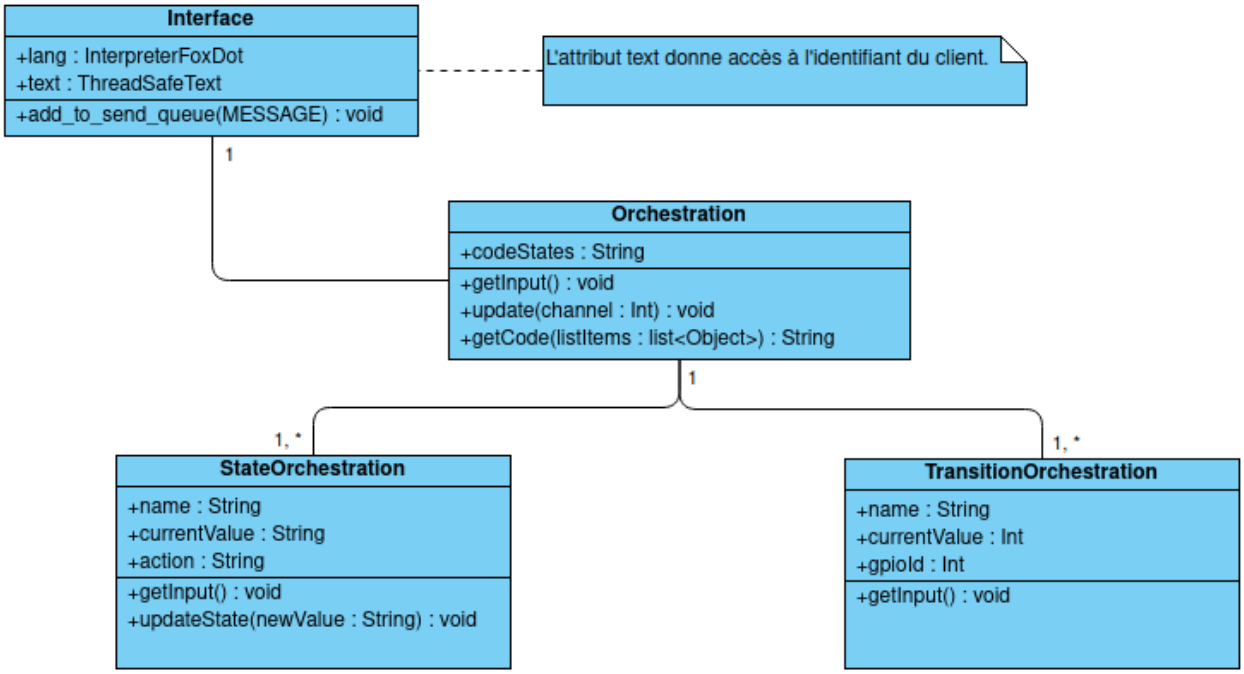

FIGURE 2.5 – Diagramme de classe pour l'orchestration.

Le code source de Troop produit par Ryan Kirkbride est basé sur de la programmation orientée objet (POO) en langage python. Dans cette continuité, nous avons réfléchi l'orchestration comme un objet (voire fig. 2.5) en concevant une classe du même nom, pourvue de quatre attributs :

— un objet de la classe *Interface*, soit l'interface du client qui sollicite la création de l'orchestration ;

— une liste d'objet *StateOrchestration*, soit la liste des états de l'automate ;
— une liste d'objet *TransitionOrchestration*, soit la liste des transitions de l'automate ;
— une chaîne de caractères *codeStates*, soit le code python qui conditionne les valeurs des états en fonction des valeurs de transitions.

La classe Interface donne un accès à l'interpréteur FoxDot et à la méthode *add_ to_ send_ queue* qui sont les deux méthodes pour évaluer les messages FoxDot. L'interpréteur nous permet également de récupérer la réponse du sous-processus FoxDot. Ainsi, nous pouvons connaître la liste des players actifs ou encore la valeur des états de l'orchestration, en tout temps.

2.3.2 Mise à jour de l'orchestration.

Nous allons aborder, étape par étape, le processus de mise à jour de l'orchestration pour mieux saisir son évolution. Tout d'abord, la mise à jour se déclenche par un gestionnaire d'événement qui scrute les entrées GPIO liées aux transitions de l'orchestration. Dès qu'il y a un changement de valeur - que ce soit front montant ou front descendant - la fonction de callback *update* de la classe `Orchestration` est appelée, avec pour paramètre d'entrée l'identifiant de la pin concernée. Cette fonction va enregistrer la nouvelle valeur de la transition liée à cette pin et va la transmettre à l'interpréteur FoxDot. Puis, l'attribut *codeStates* va être évalué pour que la valeur des états soit remise à jour. Via l'interpréteur, nous allons récupérer la valeur de chaque état - voir partie 2.1. Avec la méthode *updateState* de la classe `StateOrchestration`, la nouvelle valeur de l'état va être comparée à l'ancienne. Si elles sont différentes, alors la nouvelle sera enregistrée et si celle-ci vaut True - signifiant que l'état vient d'être actif - l'action associée va être évaluée en étant ajoutée à la queue des messages FoxDot.

Chapitre 3

Résultats

TroopRasp est le résultat du développement d'une preuve de concept pour l'intégration d'interfaces dans les outils de live-coding. Nous avons abordé dans la partie précédente les différents aspects techniques élaborés pour sa réalisation. Dans la suite, nous allons présenter les nouvelles interfaces et les moyens pour l'utilisateur de composer sa performance musicale en la paramétrant avec les interactions possibles. Puis, nous envisagerons les perspectives d'évolution du projet.

3.1 TroopRasp : interactivité et orchestration virtuelle.

L'interface TroopRasp offre deux nouvelles fonctionnalités par rapport à l'interface d'origine Troop. La première, appelée Sensor Interaction, permet de lier une entrée GPIO de la Raspberry Pi à un paramètre d'un player FoxDot. La seconde, appelée Orchestration, offre la possibilité de créer et jouer une orchestration virtuelle, basée sur le modèle d'un système à événements discrets. Chacune est accessible dans le menu de la fenêtre principale via l'onglet "Interactives Features" (voir figure 3.1).

TroopRasp offre la possibilité à un utilisateur de créer une interaction entre un player - par extension, la musique générée par lui - et un capteur. La création de cette interaction est simple et ne requiert que trois éléments. En effet, lorsque l'utilisateur choisit l'option "Sensor", une fenêtre secondaire s'ouvre dans le but de configurer un nouveau Sensor Interaction - fig. 3.2. Il doit alors choisir un player parmi une liste donnée, correspondant aux players déjà instanciés durant la session live-coding, que ce soit par l'utilisateur ou un autre client de la session. S'il n'existe aucun player à cet instant, la liste se composera d'une seule entrée valant 'None' et il ne sera pas possible de créer une interaction. L'utilisateur doit également choisir le paramètre du player auquel il veut lier l'entrée gpio parmi la liste des paramètres existants. Et enfin, le dernier choix se porte sur le numéro du gpio de la Raspberry Pi sur lequel est connecté le capteur.

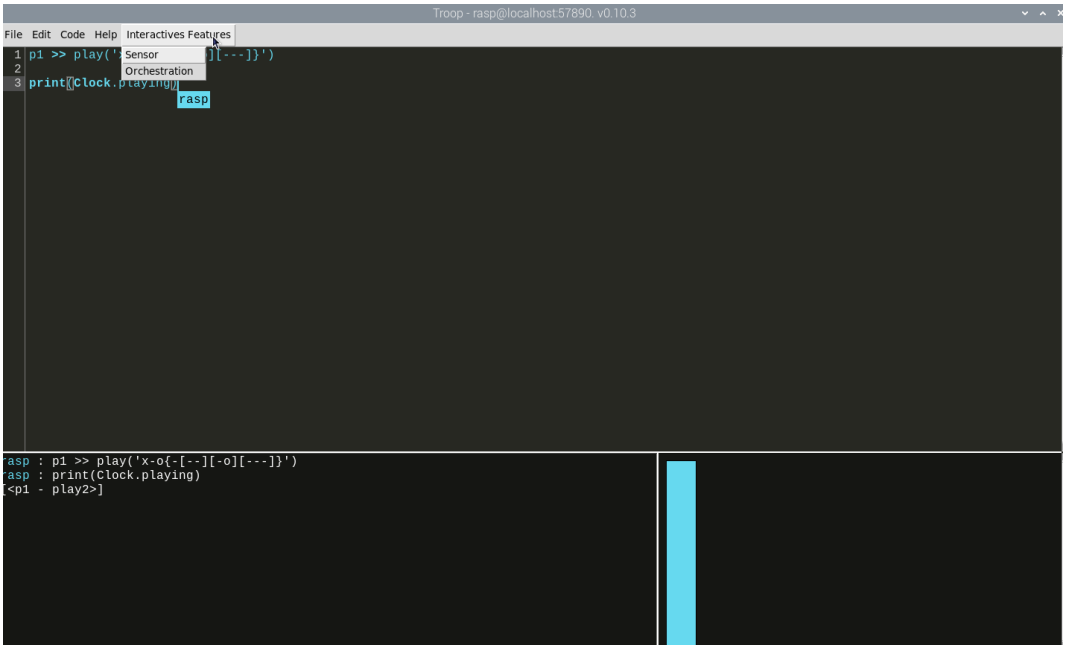

FIGURE 3.1 – TroopRasp : Interface principale.

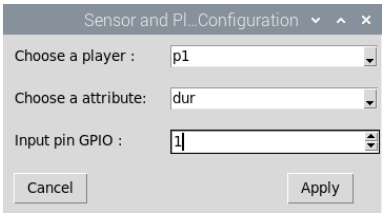

FIGURE 3.2 – TroopRasp : Interface de la fonctionnalité "Sensor Interaction".

Si cette fonctionnalité permet déjà des interactions intéressantes, elle reste encore relativement restrictive. La plupart des paramètres de player FoxDot accepte comme valeur des ensembles générant des mélodies plus complexes et permettant un champ créatif plus vaste. Or, ici, nous imposons une valeur unique et non manipulable. C'est-à-dire que le paramètre vaut la valeur brute, alors qu'on pourrait imaginer l'application d'une formule ou d'un protocole.

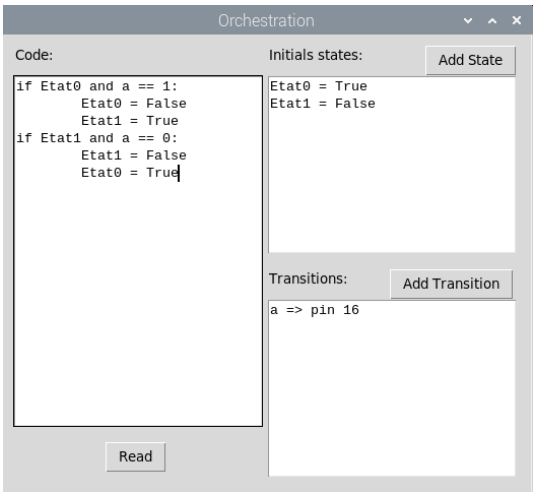

FIGURE 3.3 – TroopRasp : Interface de la fonctionnalité "Orchestration".

La seconde fonctionnalité de TroopRasp est l'implémentation d'orchestration virtuelle. Ceci se traduit par la création d'automates finis, dont les états sont atteints au fil des activations des capteurs. Pour construire son orchestration virtuelle, il faut donc définir les états et le code FoxDot à évaluer quand ils sont actifs, et définir les transitions d'état en fonction des entrées Raspberry Pi. La fenêtre secondaire de l'option "Orchestration" (fig. 3.3) permet de configurer l'ensemble. L'utilisateur peut entrer le code python qui définit les transitions d'état possibles. Un bouton ouvre une fenêtre secondaire pour ajouter un nouvel état (fig. 3.5). Pour chaque état, l'utilisateur doit spécifier un nom, la valeur initiale et le code FoxDot associé. La valeur d'un état est soit vrai (True) pour un état actif, soit faux (False) pour un état inactif. Le code FoxDot peut comprendre plusieurs lignes, et il peut faire référence à des players externes à l'orchestration - instanciés dans la fenêtre principale par l'utilisateur ou un autre client. Chaque nouvel état créé s'affiche dans la fenêtre "Orchestration" avec son nom et sa valeur initiale. Un autre bouton ouvre une fenêtre secondaire pour ajouter une transition (fig. 3.4). La transition est, quant à elle, définie par un nom et le numéro de l'entrée GPIO correspondant. Les noms des états et des transitions sont arbitraires et doivent seulement correspondre aux noms de variables inscrits dans le code. À noter qu'actuellement l'orchestration n'est pas protégée contre les potentielles erreurs de code, que ce soit le code python ou les actions FoxDot.

L'utilisateur peut orchestrer une partition musicale et interagir avec elle en session de

FIGURE 3.4 – TroopRasp : Interface pour ajouter une transition à l'orchestration.

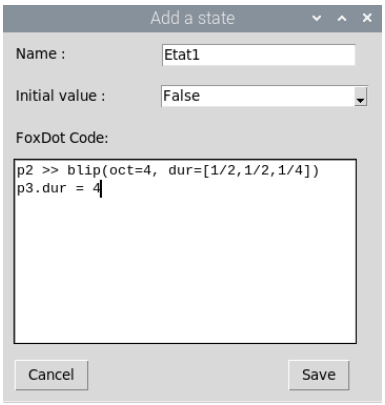

FIGURE 3.5 – TroopRasp : Interface pour ajouter un état à l'orchestration.

live-coding. La taille et la complexité de l'orchestration ne sont pas limitées, offrant un large spectre de composition. Les seules contraintes sont celles de FoxDot. L'emploi de cette fonctionnalité de TroopRasp permet une base musicale à une session de live-coding, capable d'évoluer dans le temps grâce à l'interactivité. Toutefois, l'interface a encore quelque verrou d'accessibilité. En effet, si la logique d'un système à événements discrets n'est pas appréhendée, la construction d'une orchestration peut être difficile. Cependant, cette structure en SED est intéressante pour pouvoir étudier le comportement d'un automate fini dans un nouveau domaine et observer si de nouvelles problématiques en émergent.

3.2 Perspectives.

Dans les perspectives d'évolution de TroopRasp, plusieurs aspects restent à travailler, que ce soit l'accessibilité de l'interface et la flexibilité des fonctionnalités en accord avec les objectifs du stage, ou encore l'application de modèles mathématiques et d'analyse du comportement sur les automates finis de l'orchestration.

Pour offrir plus de possibilités à l'utilisateur et faire de TroopRasp un outil plus polyvalent, il serait intéressant d'élargir le type de capteur compatible, notamment en intégrant les capteurs I2C. L'enjeu est que chaque capteur à son protocole de communication, rendant difficile la généralisation. À première vue, TroopRasp restera limitée - à moins de faire des manipulations dans le code pour l'adapter à une situation précise - mais cet aspect est à étudier.

Comme mentionné dans la partie 3.1, l'interface pour l'orchestration manque d'accessibilité pour les néophytes des systèmes à événements discrets. Néanmoins, une perspective d'évolution est déjà envisagée en proposant une construction de l'orchestration plus haut niveau. Typiquement, l'idée serait de faire une visualisation graphique à l'image des représentations d'automate fini. L'utilisateur construirait son graphe comme il construirait sa partition et pourrait le manipuler à son gré. L'approche graphique rendrait l'interface et le système d'orchestration plus conviviale et facile à appréhender.

Le dernier point à approfondir est l'étude scientifique des systèmes à événements discrets appliqués à la programmation réactive synchrone. Actuellement, la preuve de concept ne comporte pas les fonctionnalités d'analyse, de vérification et de commande. Cet aspect sera implémenté ultérieurement afin d'observer si de nouvelles problématiques en émergent.

Conclusion

TroopRasp est une preuve de concept pour l'intégration d'interfaces dans les outils de live-coding afin d'exploiter leur flexibilité pour la sonorisation d'installations artistiques interactives. Concrètement, cela se résume à l'utilisation de capteurs mesurant des grandeurs au sein de son environnement, à partir desquels l'artiste peut programmer des réponses musicales spécifiques. Le dialogue entre le public, partie intégrante de l'environnement des capteurs et la machine produit en temps réel une performance. L'interface TroopRasp donne le moyen à l'artiste-auteur de composer la structure de ce dialogue via l'implémentation de partition virtuelle. Celle-ci permet de définir une organisation musicale en fonction des paramètres de l'environnement grâce à une orchestration décrivant les interprétations musicales en fonction des séquences d'événements perçus dans le temps. On propose de modéliser les orchestrations à l'aide d'automates à états finis. Ceci permet d'étudier l'application des modèles d'analyse du comportement du système à la programmation réactive synchrone. Notamment, plusieurs propriétés temporelles sont intéressantes à vérifier, telles que l'accessibilité, l'invariance, la sûreté, la vivacité ou encore l'absence de blocage. TroopRasp est une extension de l'environnement Troop, proposant deux fonctionnalités supplémentaires basées sur l'interactivité. Troop est un environnement pour live-coding qui permet à plusieurs live-coders de se connecter et de modifier un même document. Il doit être associé à un langage de live-coding, tel que FoxDot. Pour assurer l'interaction, il a fallu établir un dialogue entre TroopRasp et l'interpréteur FoxDot. Bien que le dialogue soit déjà existant, il n'était pas adapté à notre besoin. En appliquant l'identification des messages requêtes et réponses, l'utilisation d'une variable partagée et sa protection pour éviter les conflits de lecture-écriture, nous avons pu étendre ce dialogue de manière à pouvoir récupérer les informations sur la session de live-coding en temps réel. Un autre aspect du travail s'est porté sur la manipulation du GPIO de la Raspberry. Pour optimiser le processus de mise à jour et pour éviter une surcharge, nous avons implémenté un gestionnaire d'événements basé sur le changement de valeur d'une entrée. L'implémentation des orchestrations nécessite l'application de ces différents aspects. Une structure objet a été également construite pour ordonner la manipulation de l'orchestration. Actuellement, le travail rendu répond aux attentes du stage. Bien que l'interface peut être améliorée pour une meilleure ergonomie et une meilleure accessibilité, elle suffit pour envisager la mise en place d'outils d'analyse. Ceci consistera en un travail ultérieur au cours d'une thèse où les

verrous scientifiques sous l'angle des systèmes concurrents et synchrones seront abordés via le formalisme des automates temporisés.

Bibliographie

[1] P. Bootz. Un historique de la génération numérique de textes. *Laboratoire Paragraphe, Université Paris 8*, 2013.

[2] C. Cassandras and S. Lafortune. Languages and automata. In *Introduction to Discrete Events*, 2008.

[3] Edmund M Clarke, Orna Grumberg, and Doron A. Peled. *Model checking*. MIT Press, London, Cambridge, 1999.

[4] S. Dasari and J. Freema. Directed evolution in live coding music performance. *Georgia Institute of Technology*, 2020.

[5] R. Kirkbride. Foxdot live coding with python and supercollider. In *Proceedings of the International Conference on Live Interfaces*, 2016.

[6] R. Kirkbride. Programming in time : New implications for temporality. In *Proceedings of the International Conference on Live Interfaces*, 2016.

[7] R. Kirkbride. Cooperative live coding of electronic music with troop. In *Proceedings of 15th European Conference on Computer-Supported Cooperative Work - Panels*, 2017.

[8] R. Kirkbride. Troop : A collaborative tool for live coding. In *Proceedings of the 14th Sound and Music Computing Conference*, 2017.

[9] Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. Uppaal in a nutshell. *Int. J. Softw. Tools Technol. Transf.*, 1(1-2) :134–152, 1997.

[10] A. Maclean and J. Rohrhuber. Live coding in laptop performance. *Organised Sound*, 2003.

[11] B. Petit and M. Serrano. Composing and performing interactive music using the hiphop.js language. *NIME 2019 - New Interfaces for Musical Expression*, 2019.

[12] Rodrigo Tacla Saad. *Parallel model checking for multiprocessor architecture*. Theses, INSA de Toulouse, November 2011.

[13] W Sang Lee and G. Essl. Models and opportunities for networked live coding. In *Proceedings of The Live Coding and Collaboration symposium*, 2014.

[14] G. Wang and P. Cook. On-the-fly programming : Using code as an expressive musical instrument. In *Proceedings of the 2004 International Conference on New Interfaces for Musical Expression (NIME)*, 2004.

[15] I. Zolnig and G. Eckel. Live coding : an overview. *International Computer Music Conference Proceedings*, 2007.

Abstract — This manuscript is a master's thesis about modelling and tools of reactive systems for collaborative live coding. We present the development of a new interface TroopRasp, based on Troop environment and FoxDot language, two open source tools for collaborative live coding. In this project, we discuss the representation of virtual score by discrete event systems, such as finite-state automata.

Keywords : Live coding ; Interactive programming ; Automata.

Polytech Angers
62, avenue Notre Dame du Lac
49000 Angers