Programmation web réactive

Colin Vidal

UNIVERSITÉ CÔTE D'AZUR

ÉCOLE DOCTORALE
SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DE LA COMMUNICATION

THÈSE DE DOCTORAT

Programmation web réactive

Colin Vidal

Inria

Présentée en vue de l'obtention du grade de docteur en Informatique
d'Université Côte d'Azur **et du** Collège de France**.**

Dirigée par : Manuel Serrano
Co-encadrée par : Gérard Berry

Soutenue le : 06 juillet 2018

Devant le jury, composé de :

Gérard Berry	Professeur	Collège de France
Emmanuel Chailloux	Professeur	Sorbonne Université
Roland Ducournau	Professeur émérite	Université de Montpellier
Alain Girault	Directeur de recherche	Inria
Manuel Serrano	Directeur de recherche	Inria

Reactive Web Programming

Composition du jury :

Rapporteurs :		
Emmanuel Chailloux	Professeur	Sorbonne Université
Alain Girault	Directeur de recherche	Inria
Examinateur :		
Roland Ducournau	Professeur émérite	Université de Montpellier
Co-directeur de thèse :		
Gérard Berry	Professeur	Collège de France
Directeur de thèse :		
Manuel Serrano	Directeur de recherche	Inria

Remerciements

Un grand merci à Manuel Serrano d'avoir encadré cette thèse en me laissant une grande liberté tout en étant disponible quand j'en avais besoin. Aussi, merci pour ces nombreuses discussions de « geeks » autour de GNU/Linux et de la face cachée de l'informatique en général ! Un grand merci également à Gérard Berry qui a co-encadré cette thèse en y apportant un regard critique d'une grande exigence ainsi que de m'avoir dévoilé, avec humour, les secrets d'Esterel.

Merci à Emmanuel Chailloux et Alain Girault d'avoir accepté d'être les rapporteurs de cette thèse. Merci à Roland Durcournau de m'avoir montré la joie des compilateurs, de m'avoir mis en contact avec Manuel il y a trois ans et de participer au jury.

Merci aussi à Francis Dolière Somé, avec qui j'ai partagé mon bureau et qui m'a prêté son parking de voiture pendant trois ans. Merci à toutes les autres personnes de l'Inria et de l'équipe INDES avec qui j'ai partagé de nombreux repas (délicieux soit dit en passant, merci aussi à la cantine de l'Inria pour ça !) et discussions.

Je tiens aussi à remercier mes parents Laurence Destrade et Patrick Vidal de m'avoir soutenu et encouragé pendant cette période ainsi que d'avoir relu ce manuscrit (si vous voyez des fautes d'orthographe, il ne faut pas leur en vouloir : il y en avait vraiment beaucoup au début !)

Enfin, le dernier remerciement – mais pas le moindre – te revient, Manel Zarrouk - *Boucachiche*, ma moitié. Je te remercie tendrement pour tes nombreux et chaleureux encouragements, ton grand soutien et ta bonne humeur infaillible qui m'ont permis d'aller de l'avant et conclure cette aventure qu'est la thèse.

Résumé

Le web est une plate-forme universelle pour développer des applications riches en interactions avec les utilisateurs et des services distants. Ces interactions sont implémentées sous forme d'évènements asynchrones pouvant survenir à n'importe quel instant de l'exécution de l'application. JavaScript, le langage du web, gère les évènements asynchrones de façon peu abstraite, ce qui rend l'écriture, la vérification et la maintenance d'applications interactives difficile.

La contribution de cette thèse est l'élaboration et l'implémentation du langage Hiphop.js qui dote JavaScript d'abstractions de plus haut niveau pour gérer les évènements asynchrones. Hiphop.js est une implémentation JavaScript de constructions temporelles du langage réactif synchrone Esterel. Grâce à ces constructions, le flot de contrôle d'une application Hiphop.js est explicite. Il est donc possible de savoir précisément quand et sous quelles conditions un évènement est traité par simple lecture du code source de l'application. Ceci facilite la vérification et la maintenance de l'application. L'intégration profonde du langage Hiphop.js avec l'environnement dynamique du web est une part importante des travaux entrepris dans cette thèse. Les programmes sont construits et compilés pendant l'exécution de l'application JavaScript ce qui permet d'adapter automatiquement le traitement des évènements asynchrones en fonction des changements de l'environnement au cours de l'exécution (par exemple, la connexion ou déconnexion de participants pendant une visioconférence).

Mots-clés : web, programmation asynchrone, JavaScript, programmation réactive, langage synchrone

Abstract

The web is a universal platform used to develop applications interacting with users and remote services. These interactions are implemented as asynchronous events that can be fired anytime. JavaScript, the mainstream language of the web, handles asynchronous events using low-level abstractions that makes it difficult to write, verify, and maintain interactive applications.

We have addressed this problem by designing and implementing a new domain specific language called Hiphop.js. It offers an alternative to JavaScript event handling mechanism by reusing temporal constructions coming from the synchronous programming language Esterel. These constructions make the control flow of the program explicit and deterministic. Hiphop.js is embedded in JavaScript and suits the traditional dynamic programming style of the Web. It is tighly coupled to JavaScript with which it can exchange values and access any data structures. It can also support dynamic modifications of existing programs needed to support on-demand download on the Web. It can run on both end of Web applications, namely on servers and on clients.

In this thesis, we present Hiphop.js, its design and implementation. We overview its programming environment and we present the prototypical web applications we have implemented to validate the approach.

Keywords : web, asynchronous programming, JavaScript, reactive programming, synchronous language

Table des matières

Chapitre 1

Introduction générale

Sommaire

1.1 Motivations et objectifs

1.1.1 Le web : une plate-forme universelle

Le web est historiquement un moyen d'échange et de lecture de documents structurés, statiques et non interactifs. Des technologies se sont ensuite greffées au web et ont permis la création de pages web interactives ayant un contenu dynamique, puis *d'applications web* : gestionnaires de mails, éditeurs de documents, lecteur de musiques et vidéos, plate-forme de communications et réseaux sociaux, etc. Le « Web 2.0 » était né [O'Reilly, 2003]. Le web continue à évoluer aujourd'hui grâce à des technologies qui permettent l'adaptation automatique de l'apparence de l'application et de son interaction avec l'utilisateur en fonction du périphérique (ordinateur, tablette tactile, téléphone portable) et l'utilisation de l'application en mode hors-ligne.

Les technologies web englobent JavaScript, HTML, CSS et HTTP pour les échanges réseaux. De fait, le terme application web désigne toute application dépendant fortement de ces technologies et généralement riche en interaction avec l'utilisateur. Elles sont de plus en plus utilisées en dehors du cadre initial d'un navigateur web. Par exemple, Electron [GitHub, 2013] et React Native [Facebook, 2015] sont des outils qui reposent sur les technologies web et qui permettent la création d'applications autonomes visant tous types de périphériques, des ordinateurs classiques aux téléphones mobiles. L'environnement de bureau libre Gnome préconise l'utilisation de JavaScript et de CSS pour implémenter des applications intégrées à son écosystème. Dans le cadre de l'avènement de l'*Internet des objets*, des environnements d'exécution adaptés à des ressources matérielles minimales permettent l'utilisation de JavaScript pour implémenter les applications exécutées par des *objets connectés*.

De part sa normalisation, ses possibilités techniques et son usage massif, le web est devenu aujourd'hui une plate-forme universelle pour écrire des applications interactives et échanger des données. Cette plate-forme est dynamique par nature : le comportement d'une application varie en fonction des actions des utilisateurs mais aussi en fonction du comportement des pairs avec lesquels elle communique. L'objet de cette thèse est l'étude de la gestion de ces interactions dans le contexte dynamique du web par le programmeur d'applications.

1.1.2 Orchestration d'évènements

Les applications web modernes agissent en interaction constante avec l'utilisateur et des serveurs distants. Ces interactions sont variées : choix et lecture de musiques et vidéos en ligne, visioconférences, réservation de trains et d'hôtels, interrogations de bases de

données et accès au système de fichier, navigation et zoom dans des cartes interactives, etc. Les applications peuvent elles-mêmes communiquer avec plusieurs services web distants pour répondre aux requêtes de l'utilisateur.

Ces interactions sont asynchrones par nature puisqu'il est impossible de prédire à quel moment de l'exécution du programme elles se produisent. Du point de vue de l'implémentation, une grande complexité est induite par la gestion d'évènements asynchrones, par exemple pour traiter les entrées de l'utilisateur, les terminaisons de calculs ou de temporisations locales, les réponses de requêtes à des services web distants et diverses erreurs. De plus, chaque réception d'évènement peut déclencher en réaction des calculs et l'émission d'autres évènements. L'*orchestration d'évènements* est le terme dénommant cette gestion réactive de calculs et d'interactions en tout genre.

La boucle d'évènements

Dans les applications web [W3C, 2017], des *gestionnaires d'évènements* permettent le traitement des évènements asynchrones en appelant des fonctions associées à chaque type d'évènement. Chaque fois qu'un évènement est reçu, le gestionnaire idoine ajoute les fonctions associées à ce type d'évènement à la fin d'une file d'attente. Un environnement d'exécution JavaScript dispose d'une *boucle d'évènements* qui, tant que la file d'attente n'est pas vide, retire la première fonction et l'exécute [1].

Lorsque le nombre d'évènements à orchestrer est grand, comme dans les applications web modernes, ce modèle rend difficile la compréhension de la dynamique d'exécution du programme [Mikkonen et Taivalsaari, 2008]. En effet, la syntaxe et l'ordre dans lequel les gestionnaires d'évènements sont définis ne reflètent pas les différentes possibilités d'ordonnancement à l'exécution. Il faut dérouler mentalement les différents ordonnancements potentiels pour comprendre les différents états et comportements possibles du programme. De plus, la modification, la composition et la réutilisation de code sont plus difficiles avec ce modèle d'orchestration, notamment lorsqu'une modification du code entraîne de nouveaux états : il faut alors à nouveau dérouler mentalement les différents ordonnancements potentiels en considérant une combinatoire plus grande d'évènements et d'états.

1.2 Contribution

Le langage Hiphop.js est le fruit de cette thèse. Il apporte une solution d'orchestration alternative en introduisant la programmation réactive synchrone dans l'environnement

1. Bien que la gestion asynchrone des évènements ne soit pas spécifiée par JavaScript, la très grande majorité des environnements JavaScript l'implémentent.

intrinsèquement dynamique du web. Pour cela, Hiphop.js étend le langage JavaScript en y incorporant la sémantique et les constructions du langage réactif synchrone Esterel v5 [Berry et Gonthier, 1992, Berry, 2000b]. Ces constructions sont aussi adaptées au contexte dynamique du web : les programmes Hiphop.js sont dynamiquement construits puis connectés aux évènements JavaScript, c'est-à-dire pendant l'exécution du programme JavaScript environnant. Des constructions permettent également d'étendre un programme Hiphop.js déjà construit. Par ailleurs, Hiphop.js dispose d'abstractions facilitant la mise à jour de l'interface graphique d'une application web.

Ainsi, Hiphop.js introduit dans le web les constructions et abstractions permettant une vision hiérarchique des évènements et de leur traitement. Ceci rend la dynamique du programme explicite dès la syntaxe du code, facilitant l'écriture et la maintenance d'applications web interactives.

1.3 Plan de la thèse

Le chapitre 2 présente les travaux connexes ayants des connexions avec Hiphop.js. Le chapitre 3 met en évidence par des exemples pratiques les problèmes liés au modèle de gestion des évènements asynchrones par JavaScript. Ces mêmes exemples sont ensuite implémentés en Hiphop.js, ce qui permet de comparer le modèle de JavaScript au modèle réactif de Hiphop.js tout en introduisant ce nouveau langage. Le chapitre 4 est un manuel exhaustif de Hiphop.js, décrivant l'ensemble des concepts et des instructions du langage. Le chapitre 5 explique comment Hiphop.js est intégré et se connecte au modèle de JavaScript. Ensuite, le chapitre 6 détaille le fonctionnement interne de Hiphop.js et son implémentation. Le chapitre 7 présente le débogueur symbolique et distribué conçu pour Hiphop.js et le chapitre 8 présente et explique des applications réelles écrites en Hiphop.js. Enfin, le chapitre 9 dresse des perspectives d'évolution de Hiphop.js et le chapitre 10 conclut l'ensemble de ces travaux.

Les chapitres 3, 4, 5, 7 et 8 s'adressent principalement au lecteur intéressé par l'utilisation de Hiphop.js pour écrire des programmes, tandis que le chapitre 6 s'adresse au lecteur intéressé par la modification du langage.

Chapitre 2

État de l'art

Sommaire

2.1 Les langages réactifs synchrones

On classe les systèmes informatiques en deux grandes familles [Harel et Pnueli, 1985, Pnueli, 1986]. La première correspond aux systèmes dits *transformationnels*. Un système transformationnel est caractérisé par un début et une fin. Dans l'intervalle, un état initial correspondant aux entrées du système est transformé jusqu'à l'obtention d'un état final correspondant aux sorties du système. Ces systèmes correspondent par exemple aux compilateurs, à des traitements par lots ou aux logiciels de calcul scientifique. La seconde famille correspond aux systèmes *interactifs* et *réactifs*. Il s'agit de systèmes dont le comportement est défini selon une série d'états et d'évènements au cours du temps et peut théoriquement ne jamais terminer. Les systèmes interactifs réagissent aux évènements de l'environnement sans contrainte de temps, alors que les systèmes réactifs réagissent dans un temps imparti par l'environnement. Par exemple, le noyau d'un système d'exploitation (OS) généraliste est interactif (son temps de réponse aux évènements dépend de sa charge et des capacités matérielles), alors que le pilote automatique d'un avion est un système réactif (son temps de réponse aux évènements est spécifié et doit être respecté).

Plus précisément, un système réactif est caractérisé par le fait que le résultat de la réaction de plusieurs composants concurrents aux évènements est déterministe. L'hypothèse synchrone est un outil intellectuel qui assure ce déterminisme. Un langage réactif synchrone s'exécute sous forme d'*instants* ou de *réactions* qui sont atomiques, c'est-à-dire exécutés dans un temps conceptuellement nul. Les différents composants communiquent via des *signaux* dont l'émission dans une réaction est instantanée et diffusée à l'ensemble des composants dans un instant. Cette propriété assure que le comportement d'un composant ne sera pas influencé par l'ajout ou le retrait d'un autre composant du système et aide ainsi à rendre le programme facilement composable.

Les langages Lustre [Caspi *et al.*, 1987] et Signal [LeGuernic *et al.*, 1991] ont été conçus pour raisonner en terme de flot de données, alors que les langages StateCharts [Harel, 1987] et Esterel [Berry et Gonthier, 1992, Berry, 2000b] ont été conçus pour raisonner en terme de contrôle de flot. Ces deux visions ne sont pas exclusives pour autant, par exemple les lois de vol d'un avion (position de gouvernes en fonction de l'incidence et de la vitesse) sont spécifiées en terme de flot de données alors que les différentes phases de vol (décollage, atterrissage) sont spécifiées en terme de contrôle de flot. Ainsi, des langages et techniques hybrides [Berry, 2007, Colaço *et al.*, 2005] permettent d'écrire des programmes selon ces deux visions. Ces langages disposent d'ailleurs de compilateurs permettant de générer un code dans un formalisme commun (format OC) [Paris *et al.*, 1993]. Il s'agit d'un langage commun permettant d'exprimer la sémantique d'un système réactif synchrone ainsi que son interface avec un environnement d'exécution asynchrone.

La caractéristique d'instantanéité de l'hypothèse synchrone implique que le statut d'émission d'un signal dans une réaction est unique : le signal est soit présent, soit absent. Cette restriction peut amener à l'écriture de programmes invalides, bien que syntaxiquement corrects. Par exemple :

```
present S else
  emit S
end
```

Ce code Esterel est invalide à cause d'un *cycle de causalité* [Berry, 2002]. Si le signal `S` est absent, alors la branche `else` est exécutée et `S` est émis. C'est contradictoire avec l'hypothèse synchrone : le signal ne peut pas être absent puis présent « plus tard » dans le même instant. Une variante de *non réaction immédiate à l'absence* du modèle réactif synchrone introduite dans ReactiveC puis dans ReactiveML [Boussinot, 1991, Mandel et Pouzet, 2008] permet d'éliminer les cycles de causalité en ajoutant implicitement un retard d'un instant après le test d'absence d'un signal. Avec cette variante, la branche `else` du code vu précédemment ne serait exécutée qu'à la réaction suivante de celle du test de présence du signal `S`. Une autre variante de l'hypothèse synchrone appelée *Sequentially Constructive* (SC) [Hanxleden *et al.*, 2014] considère des fils d'exécutions concurrents partageant en lecture et écriture des variables partagées. L'ordonnancement de ces fils d'exécution pendant l'exécution garantit que l'ensemble des écritures d'une variable sont exécutées avant ses lectures. Le langage Sequentially Constructive Esterel (SCEst) implémente la variante SC de l'hypothèse synchrone en reprenant les constructions d'Esterel [Rathlev *et al.*, 2015].

2.2 La programmation fonctionnelle réactive

La programmation fonctionnelle réactive (FRP, pour *Functional Reactive Programming*) est un paradigme permettant l'écriture d'applications en raisonnant en terme de flot de données. Avec un langage de type FRP, une expression est automatiquement réévaluée lorsqu'une valeur qu'elle référence est modifiée. Cette expression s'évaluant elle-même en une autre valeur pouvant être référencée dans une autre expression, cela permet la construction de dépendances entre expressions automatiquement réévaluées lorsque qu'une valeur est modifiée. Contrairement au modèle réactif synchrone, la notion d'instant atomique n'existe pas.

Historiquement, Fran [Elliott et Hudak, 1997] est le premier langage de type FRP. Ce langage définit deux type de signaux. Le premier appelé *évènement* est une valeur qui change dans le temps de façon discrète. Le second appelé *comportement* est une valeur continue qui varie dans le temps. La valeur d'un comportement est modifiée par un évènement ou par un autre comportement dont il dépend. Lorsqu'une expression du langage

référence un comportement, elle devient elle même un comportement. Le mécanisme de transformation d'une expression en comportement est appelé *lifting*. Il est à l'origine de la création de dépendance permettant la propagation des valeurs et de la réévaluation automatique des expressions. Avec Fran, le lifting d'une expression est explicite. Avec d'autres langages de type FRP, FrTime [Cooper et Krishnamurthi, 2004] par exemple, ce mécanisme est implicite.

2.3 Technologies web réactives

Il est communément admis que le langage de programmation du web est JavaScript [W3Tech, 2018]. Il s'agit d'un langage très souple par plusieurs aspects. D'abord par son système de type qui est dynamique et « permissif ». Par exemple, des incompatibilités de type vont être automatiquement résolues par le langage en appliquant des conversions implicites. Ainsi, le test `"" == 0` est `vrai` [1]. Un autre exemple est le modèle objet à base de prototype permettant l'ajout ou la suppression d'attributs à tout instant de l'exécution, le type d'un objet ne donnant ainsi aucune garantie sur les propriétés qui le caractérisent. Par ailleurs, une forme de méta-programmation est possible par la modification de mécanismes objets (accès à un attribut ou appel de fonction) via des *proxy* [Van Cutsem et Miller, 2010]. Enfin, JavaScript dispose d'un noyau fonctionnel proche de Scheme ou ML permettant la construction de fermetures lexicales ou la manipulation de fonctions comme des valeurs du langage.

Cette souplesse est une caractéristique importante pour programmer le web : les programmes clients typiques sont construits depuis le serveur puis envoyés au client. Une fois sur le client, le programme est étendu par des bibliothèques qui sont chargées dynamiquement. Des structures de données, comme celles permettant la construction de l'IHM sont constamment créées et mutées, des objets provenant de sources diverses sont sérialisés et manipulés. La souplesse de JavaScript facilite largement l'ensemble de ces opérations. Le modèle de compilation « just-in-time » utilisé par les principaux environnements d'exécution JavaScript [Mozilla, 2011, Google, 2008, Ha *et al.*, 2009, Chudnov et Naumann, 2015] permet d'obtenir de bonnes performances tout en conservant la souplesse de l'environnement. Ces performances permettent aussi l'utilisation de JavaScript pour implémenter le côté serveur des applications web [Dahl, 2009, Tilkov et Vinoski, 2010].

Enfin, des dialectes de JavaScript comme TypeScript [Rastogi *et al.*, 2015] ou Flow [Facebook, 2014a] offrent des possibilités de typage graduel en permettant au programmeur d'ajouter des annotations de type. Une autre approche consiste à écrire

1. Pour cette raison, il est généralement recommandé d'utiliser les opérateurs === et !== qui ne font pas de conversion implicites.

un programme web dans un langage différent puis à le compiler vers JavaScript comme par exemple Emscripten [Zakai, 2011], Scala.js [Doeraene, 2013], Hop [Serrano *et al.*, 2006], JS_of_ocaml [Vouillon et Balat, 2014] ou encore BuckleScript [Bloomberg, 2016]. Cette dernière approche semble confirmée par le standard WebAssembly [Mozilla *et al.*, 2015] supporté par la majorité des navigateurs web modernes. Il s'agit d'un environnement d'exécution d'un code assembleur vers lequel il est possible de compiler depuis n'importe quel langage. La motivation est de produire un code plus rapide que JavaScript.

2.3.1 Les promesses et constructions `async` / `await`

Dans ses déclinaisons récentes, le langage JavaScript dispose de deux nouveaux mécanismes simplifiant l'orchestration des appels asynchrones [ECMA, 2015].

Les promesses

Le premier mécanisme est appelé *Promesse*. Il est initialement introduit en Lisp [Friedman et Wise, 1977] pour permettre la synchronisation de tâches parallèles. Une promesse est un objet représentant un calcul dont le résultat n'est pas encore connu. En JavaScript, une promesse est un objet qui encapsule un appel asynchrone et qui dispose d'une interface permettant d'associer des *fonctions de rappel*. Il s'agit de fonctions automatiquement appelées par JavaScript lors de la terminaison de la tâche de fond afin d'en notifier le programme. Une promesse est *résolue* si la tâche de fond déclenchée par l'appel asynchrone se termine avec succès ou *rejetée* si la tâche échoue. Enfin, une valeur optionnelle retournée par la tâche est passée en paramètre des fonctions de rappel.

Une utilisation typique des promesses est le chaînage de plusieurs opérations asynchrones qui doivent se produire en séquence. Lorsque le premier appel asynchrone se termine, sa promesse est résolue ce qui permet la création d'une nouvelle promesse encapsulant un autre appel asynchrone et ainsi de suite. Le chaînage des promesses évite l'imbrication complexe de fonctions de rappel lors de la construction de séquences d'appels asynchrones [Kambona *et al.*, 2013]. Par exemple [2] :

2. Ici, les promesses ne sont jamais rejetées afin de garder l'exemple simple.

```
const p1 = new Promise(resolve => {
   setTimeout(() => resolve("first promise resolved"), 2000);
});

const p2 = p1.then(firstResult => {
   console.log(firstResult);
   return new Promise(resolve => {
      setTimeout(() => resolve("second promise resolved"), 100);
   });
});

p2.then(console.log);
```

La promesse `p1` est construite, la tâche de fond est déclenchée par la fonction passée en paramètre au constructeur `new Promise` : il s'agit d'un minuteur dont la terminaison entraînera la résolution de la promesse par l'appel à `resolve`. La promesse `p1` est donc résolue après 2 secondes avec la valeur « first promise resolved ». La fonction de rappel passée à `p1.then` automatiquement appelée à la résolution de `p1` affiche la valeur retournée par `p1` puis construit et retourne une nouvelle promesse. Lorsque cette nouvelle promesse est résolue, la fonction de rappel passée à `p2.then` est appelée et affiche « second promise resolved ». Autant de promesses que nécessaire peuvent être chaînées de cette façon afin d'implémenter des séquences d'actions asynchrones.

Les promesses permettent donc aux bibliothèques d'exposer des appels asynchrones via une interface de programmation (API) standard et l'écriture de code plus composable. Par ailleurs, une primitive permet d'inclure des promesses démarrées simultanément dans une promesse englobante qui n'est résolue que lorsque l'ensemble des promesses incluses sont résolues, il s'agit là d'une forme de synchronisation pour les traitements parallèles. Une primitive antagoniste existe aussi, permettant de rejeter la promesse englobante si l'une des promesses incluses est rejetée.

Les constructions `async/await`

Si l'imbrication de fonctions de rappel est limitée par l'utilisation des promesses, il est nécessaire de créer des fonctions de rappel pour implémenter le comportement d'une promesse lorsqu'elle est résolue ou rejetée. Par exemple, pour créer une nouvelle promesse (chaînage) ou autre action quelconque. Le second mécanisme introduit dans ECMAScript 6 permet d'abstraire entièrement les fonctions de rappel en donnant l'illusion d'un code synchrone. Une fonction dont la définition est préfixée par `async` permet l'appel de fonctions retournant une promesse en préfixant l'appel par `await`. Le code après `await` est exécuté une fois que la promesse est résolue. Ainsi, le code est plus concis et sans fonctions de rappel. Du point de vue local à la fonction le code est syn-

chrone : l'instruction `await` ne se termine que lorsque la promesse est résolue. Cependant, du point de vue global au programme, ce code n'est en réalité pas synchrone : l'ensemble du code après un appel préfixé par `await` est implicitement encapsulé dans une fonction de rappel appelée à la résolution de la promesse passée à `await`. Cela a deux implications : d'abord, l'utilisation de `await` dans des structures de contrôle itératives n'est pas possible. Ensuite, l'état du programme peut avoir changé entre le moment où `await` est exécutée et le moment où elle se termine. Par exemple, l'exécution du code après `await` peut n'être valide que si `await` se termine moins de x secondes après son démarrage. Le programmeur doit alors explicitement vérifier l'état du programme. Cette dernière implication est vraie aussi pour la résolution des promesses et les fonctions de rappel, mais la structure d'apparence synchrone du code d'une fonction `async` ne donne pas l'intuition de le faire puisque le programme semble « bloqué » jusqu'à la terminaison de `await`.

Technologies connexes

Le langage Dart [Google, 2011] est conçu pour écrire des applications web client et serveur. Il dispose comme JavaScript d'un unique fil d'exécution et de constructions permettant de gérer des évènements asynchrones. Une *Future* Dart est similaire à une promesse JavaScript : c'est un objet représentant un calcul en cours, sa résolution ou son rejet retournant une valeur éventuelle. Dart dispose d'un mécanisme supplémentaire appelé *Streams*. Un stream est une séquence d'évènements asynchrones qui, contrairement aux futures, permet de recevoir plusieurs valeurs au fil du temps. Des constructions permettent d'itérer sur un stream comme avec n'importe quelle collection, l'itération terminant lorsque il n'y a plus d'évènements dans le stream ou si une erreur est détectée.

Le langage C# [Microsoft, 2012] et le langage F# [Syme *et al.*, 2011] introduisent respectivement les types `Task<T>` et `Async<T>` permettant des mécanismes similaires aux promesses et fonctions `async` de JavaScript.

Enfin, une approche alternative [Philips *et al.*, 2016] consiste à transformer statiquement un programme synchrone en un programme composé de continuations. Ainsi, chaque continuation peut-être appelée de façon asynchrone, ce qui évite de bloquer le programme en attendant la complétion d'une opération en tâche de fond, mais le programmeur dispose d'une représentation synchrone du flot de contrôle. L'avantage de cette approche par rapport à `async/await` est qu'il n'est pas nécessaire d'expliciter le code synchrone qui doit être transformé en continuation : une analyse statique de dépendance de contrôle de flot détermine automatiquement les transformations à appliquer.

2.3.2 Interfaces graphiques web réactives et applications interactives

L'IHM d'une application web est définie de façon déclarative par l'écriture d'un arbre HTML. Dans les applications web modernes, cet arbre est généralement automatiquement généré via différentes techniques décrites ici. Le *DOM* (*Document Object Model*) [W3C, 2004] est une représentation objet arborescente de l'arbre HTML. Il est accessible via une API standard de JavaScript, ce qui permet de modifier dynamiquement l'IHM de l'application au cours de son exécution et de lire des valeurs rentrées par l'utilisateur, par exemple dans des champs texte. Une IHM web réactive se caractérise par la synchronisation du DOM en fonction de l'état de l'application au cours de son exécution. D'un point de vue technique, l'API JavaScript du DOM est peu abstraite. D'abord parce que pour accéder à un nœud HTML depuis le DOM il faut l'identifier de façon unique ou connaître sa position dans l'arbre. Ensuite, parce que le DOM est indépendant de l'état de l'application : le programmeur doit donc explicitement mettre à jour le DOM lorsque l'état de l'application est modifié. Ces caractéristiques rendent l'implémentation d'IHM web réactive complexes et peu compositionnelles. Plusieurs travaux connexes permettent d'identifier plus précisément ces problèmes et y apportent des solutions.

Flapjax [Meyerovich *et al.*, 2009] est l'une des premières tentatives d'adaptation du style de programmation FRP dans le web. Il s'agit d'une implémentation JavaScript de FrTime. Le principe est analogue : des comportements sont créés à partir d'évènements : lorsque qu'un nouvel évènement se produit, la valeur de chaque comportement en dépendant est automatiquement mise à jour. Cela implique la réévaluation automatique des expressions JavaScript utilisant ces comportements. Une expression JavaScript e_1 contenant un comportement c devient elle-même un comportement, ainsi, si la valeur de e_1 est utilisée dans une expression e_2 et que la valeur de b change, e_1 puis e_2 sont automatiquement réévaluées. Le flot de contrôle de l'application ne dépend donc pas de fonctions de rappel, il est implicitement dicté par des flux d'évènements. Cette approche offre donc un style de programmation fonctionnel tout en assurant la réévaluation automatique du code dépendant d'un comportement. Du point de vue du programmeur, cela simplifie la construction et la maintenance de l'IHM d'une application : lorsqu'un comportement est mis à jour, tous les éléments graphiques qui le référencent sont automatiquement mis à jour. Cet exemple est tiré de la documentation officielle de Flapjax [3] :

3. `http://www.flapjax-lang.org/try/index.html?edit=mouse1.fx`

```
<div style={! { position: "absolute",
                left: mouseLeftB(document),
                top: mouseTopB(document) } !}>
  the mouse!
</div>
```

Ici, `mouseLeftB` et `mouseTopB` sont des comportements dépendant d'évènements générés par la page web (récupérés via la variable globale `document`). Lorsque la souris bouge, des évènements sont déclenchés et mettent à jour ces comportements. Ceci entraîne une réévaluation automatique de l'objet littéral passé à l'attribut `style` et donc une modification de la position de « the mouse ! » affiché à l'écran. La vue du programme est donc toujours synchronisée, par construction, avec le modèle. Cela évite l'écriture de fonctions de rappel explicitant la mise à jour de chaque élément graphique et il n'est pas nécessaire nommer chaque élément graphique afin de pouvoir l'identifier lors de la mise à jour. Plus récemment, une implémentation à base de promesses JavaScript propose un modèle analogue [Jeffrey et Van Cutsem, 2015].

Elm [Czaplicki et Chong, 2013] est un langage de programmation de type FRP dédié à la création d'IHM web réactives. De façon analogue aux comportements de Flapjax, des *signaux* correspondent à des valeurs pouvant changer dans le temps suite à des évènements. Lorsque la valeur d'un signal change, toutes les expressions qui le référencent sont réévaluées, formant un arbre de dépendance comme avec Flapjax. La particularité d'Elm est de permettre des opérations asynchrones lors du calcul de l'arbre de dépendances d'un signal : alors que la nouvelle valeur d'un signal est propagée, elle peut en un nœud donné de l'arbre déclencher un appel asynchrone. La propagation ne continue qu'une fois l'appel asynchrone terminé.

Ur/Web [Chlipala, 2015] est un langage de programmation web *multitier*, c'est-à-dire que le même programme implémente la partie cliente et la partie serveur de l'application. Cela permet un modèle de programmation unifiée qui, par exemple, abstrait les appels réseaux du serveur au client. En terme d'IHM réactive, Ur/Web se distingue de Flapjax et Elm. D'abord, les éléments HTML sont des valeurs du programme, ce qui permet la construction littérale du DOM. Des signaux (similaires aux signaux Elm) sont injectés dans le DOM comme n'importe quelle autre valeur. Lorsque la valeur du signal change, les parties du DOM en dépendant sont automatiquement mises à jour. Il est aussi possible d'injecter des fonctions de rappel référençant un signal, et qui sont automatiquement exécutées à chaque changement de valeur du signal. Dans [Reynders *et al.*, 2014], un modèle et une implémentation de langage web multitier dont le style de programmation FRP est semblable à Flapjax et Elm est présenté. Des expressions clientes qui dépendent d'un signal défini sur le serveur seront automatiquement recalculées lorsque la valeur de ce signal change. L'inverse est vrai : des valeurs du serveur peuvent dépendre de signaux définis sur le client. Contrairement à Ur/Web, la

frontière entre le passage de valeurs entre code client et le code serveur est explicite.

Sunny est un autre langage dédié au développement d'applications web interactives [Milicevic *et al.*, 2013]. Comme Ur/Web, ce type de langage unifie la programmation client et serveur à travers un formalisme unique. Au lieu de supprimer entièrement les gestionnaires d'évènements comme avec une approche FRP, une construction du langage permet de lier les évènements à un code en forçant le programmeur à expliciter quelles sont les conditions à remplir pour l'exécuter. Lorsque l'état de l'application change, des dépendances implicites entre le modèle et l'IHM permettent la mise à jour automatique du DOM, à l'image de la programmation FRP.

Les travaux académiques présentés jusqu'ici, bien qu'ils aient des approches différentes, ont tous une chose en commun. Ils forment une couche d'abstraction au DOM et aux gestionnaires d'évènements asynchrones de JavaScript permettant la mise à jour automatique du DOM lors de la modification de l'état de l'application, notamment lors d'un évènement asynchrone. Ceci évite au programmeur de maintenir des noms uniques pour identifier les éléments HTML et de gérer les mises à jour du DOM. Cette abstraction facilite aussi la composition et la maintenance des programmes.

Parallèlement des technologies utilisées dans l'industrie comme jQuery [Resig, 2006] ou Mootools [Proietti, 2006] ont permis de simplifier la programmation du DOM en proposant une API plus simple et plus abstraite. Cela rend des manipulations courantes plus concises, mais ne règle pas les problèmes fondamentaux comme le nommage ou la manipulation des éléments du DOM dans du code asynchrone. La manipulation du DOM est résolue par des *moteurs de template*, par exemple Mustache [Wanstrath, 2009] ou dans des environnements web complets comme Angular [Google, 2016] ou Vue.js [You, 2014]. Il s'agit de fichiers modèles comprenant du code HTML dont le contenu est paramétré par des variables. À l'exécution, un mécanisme de rendu génère du code HTML à partir de ces fichiers modèles en substituant les variables par les valeurs correspondantes à l'état du programme. Ceci évite le nommage des éléments du DOM : lorsque l'état du programme change, il suffit de réappliquer le mécanisme de rendu au lieu d'identifier puis de modifier les éléments HTML un à un. À noter qu'un mécanisme similaire est utilisé pour mettre à jour le DOM dans Sunny.

Une approche alternative proposée par la bibliothèque React [Facebook, 2013] consiste à définir des fragments d'IHM appelés *composants*. Un composant est une fonction [4] JavaScript dont l'appel construit une représentation du DOM correspondante à l'état de l'application. Les composants sont conçus pour être combinés et former une hiérarchie de composants représentant l'ensemble de l'arbre HTML. Ainsi, lorsque l'état de l'application change, le composant « racine » est exécuté et entraîne une cascade d'exécution de ses composants fils, chaque composant mettant à jour la partie du DOM

4. Les composants peuvent aussi être définis en tant que classe JavaScript, leur permettant de disposer d'un état et de plusieurs opérations avancées.

qu'il représente. Ce mécanisme assure que l'ensemble de l'IHM est synchronisée à chaque changement d'état [5]. Afin de rendre l'écriture des composants plus facile, JSX [Facebook, 2014c] permet d'écrire du code HTML directement dans le code JavaScript, comme n'importe quelle valeur littérale. Redux [Abramov, 2015], une implémentation de la spécification Flux [Facebook, 2014b], permet de centraliser l'état de l'application dans un modèle immuable qui est connecté au composant React racine de l'application. Lorsque qu'un nouvel état de Redux est calculé, il est automatiquement propagé à l'ensemble des composants. L'architecture est la suivante :

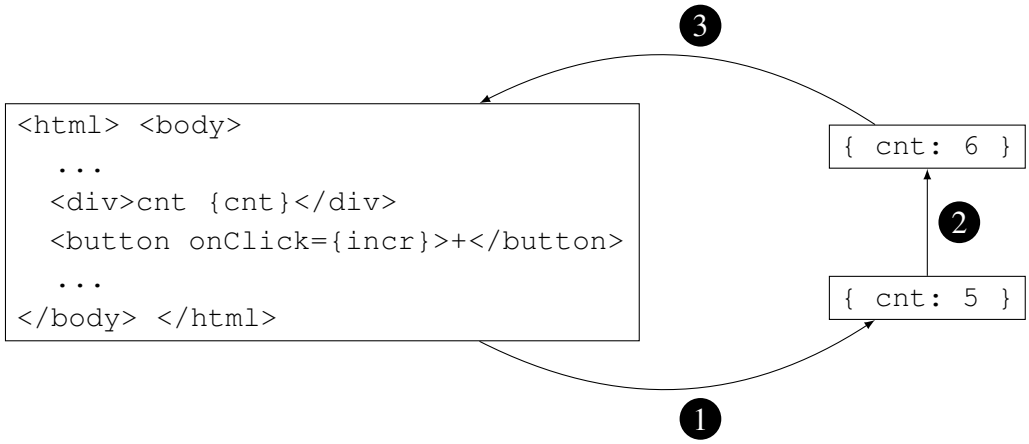

FIGURE 2.1 – Architecture globale de Redux.

La figure 2.1 présente un exemple schématisé de l'utilisation de React et Redux qui met en évidence le flot de données unidirectionnel dans le programme. Initialement, la page affiche « cnt 5 » ainsi qu'un bouton pour incrémenter cette valeur [6]. Lorsque l'utilisateur clique sur le bouton « + », la fonction `incr` indique à Redux d'incrémenter la valeur de `cnt` (1). Ensuite, le nouvel état est calculé et `cnt` vaut 6 (2). Enfin, Redux informe React de mettre à jour l'IHM (3). Le flot de données est unidirectionnel dans l'application : tout changement d'état est explicitement demandé à Redux. À ce moment, Redux calcule le nouvel l'état et le propage ensuite à l'ensemble de l'IHM. Ce modèle se rapproche du style de programmation FRP mais a une granularité moins fine : les dépendances sont entre un état et les composants graphiques, pas entre des comportements et expressions du langage.

RxJS [Podwysocki *et al.*, 2013] et Bacon.js [Paananen, 2012] permettent de construire et composer des flux de données asynchrones. L'émission d'une valeur dans un flux

5. En interne, React est optimisé pour ne mettre à jour que les parties du DOM dont le contenu a effectivement changé.

6. Les accolades sont une extension de JSX. Cette extension permet de calculer et d'injecter une valeur JavaScript dans l'arbre HTML au moment de la construction (ou la mise à jour) du DOM.

est de sources multiples : clic de la souris ou appui d'une touche, terminaison d'une promesse, l'émission de valeur d'un autre flux, etc. Ceci ressemble à la programmation FRP, mais il est aussi possible de combiner des flux en les synchronisant, par exemple, pour attendre des valeurs spécifiques de plusieurs flux, ou pour faire un calcul et émettre le résultat dans un autre flux. Dans le cas où un calcul prend du temps dans un flux, la synchronisation de valeurs en entrée et en sortie de flux est également possible. Par exemple :

```
inputField.map(event => event.target.value)
          .switchMap(value => translator(value)
          .subscribe(console.log);
```

Ici, `inputField` est un champ texte émettant un nouvel évènement à chaque fois qu'il est modifié, `translator` est une fonction retournant une promesse qui interroge un service web pour obtenir une traduction de la valeur du champ texte. L'opérateur `switchMap` permet de remplacer chaque requête en cours (non résolue) par une nouvelle si le champ texte est modifié. Ainsi, seule la réponse correspondante à la dernière valeur rentrée dans le champ texte est affichée à la console : la valeur du champ texte et sa traduction sont toujours synchronisées. Ce type de fonctionnalité est également proposé par Orc [Kitchin *et al.*, 2009].

Un *formlet* est une abstraction permettant, à partir de valeurs du programme, la construction automatique du DOM et en particulier de formulaires HTML [Cooper *et al.*, 2008]. La particularité est que la liaison avec le code HTML est typée et vérifiée statiquement. L'idée est ensuite reprise avec les *flowlets*, où les modifications des formulaires génèrent des flux d'évènements pouvant être combinés [Bjornson *et al.*, 2010].

Pendulum [El Sibaïe et Chailloux, 2016] est un langage réactif synchrone. Il s'agit du travail connexe le plus proche de Hiphop.js car Pendulum est conçu pour orchestrer les applications web et repose également sur les constructeurs et la sémantique d'Esterel. Contrairement à Hiphop.js, Pendulum est exclusivement dédié à l'orchestration de la partie cliente des applications. Ceci permet une intégration fine avec le DOM : les éléments HTML correspondent à un type spécifique de signal et la modification de leur valeur déclenche implicitement une réaction. Un autre aspect traité par Pendulum est de rendre la programmation plus sûre par un typage statique fort. En effet, Pendulum est une extension d'OCaml et permet donc d'avoir une vérification statique des types avant l'exécution du programme et donc d'éliminer une grande famille d'erreur lors de la génération de code JavaScript. Hiphop.js a fait le choix d'une intégration plus poussée avec le langage hôte. Hiphop.js ne dispose pas de vérification de types et les programmes sont construits dynamiquement pendant et en fonction de valeurs connues qu'au moment de l'exécution du programme JavaScript (voir section 5).

2.3.3 Le modèle d'acteur dans le web

Le modèle d'acteur est un formalisme de programmation parallèle et dynamique qui définit des entités de calcul indépendantes les une des autres [Hewitt *et al.*, 1973]. Un acteur dispose de son propre espace d'adressage, communique avec d'autres acteurs exclusivement via des messages asynchrones et peut créer de nouveaux acteurs au cours de son exécution. Bien qu'un acteur ne puisse initialement communiquer qu'avec son parent et vice-versa, la référence d'un acteur peut-être envoyée par message à n'importe quel autre acteur. Ceci permet donc aux acteurs n'ayant pas de lien de parenté direct de communiquer entre eux.

Il est possible d'implémenter ce modèle dans les applications web par l'utilisation de *web worker*, une tâche de fond démarrée depuis une page web [W3C, 2017]. Il s'agit d'un processus exécuté en parallèle et dans un espace d'adressage différent. Il permet de faire des calculs longs sans bloquer le fil d'exécution JavaScript. Enfin, un web worker communique par envoi de messages avec la page web qui l'a démarré.

Cependant, l'utilisation de web workers est bornée par plusieurs contraintes. La première est liée au fait qu'une API spécifique permet de créer et d'échanger des messages avec un web worker. Dans le contexte d'une application web, massivement distribuée, des acteurs peuvent être locaux, mais aussi sur le serveur. Les *generic web workers* proposent une API unifiée pour échanger des messages depuis un web worker local au navigateur ou depuis un serveur distant [Welc *et al.*, 2010]. Akka.js est un mécanisme analogue utilisé dans Scala.js [Doeraene, 2013] qui permet de définir des web workers côté client et serveur et de communiquer entre deux indifféremment. Par ailleurs, les données échangées entre web workers sont de types primitifs (nombres ou chaînes de caractères) et les échanges ne sont possibles qu'entre un web worker et la page web qui l'a créé. Spider.js [Myter *et al.*, 2016] propose un modèle unifié permettant la communication entre web workers (pas seulement d'un web worker avec son parent), supportant le passage d'objets lors de l'échange de messages et qui abstrait le passage du réseau dans le cas d'envoi de message avec un acteur non local.

Enfin, [Van den Vonder *et al.*, 2017] unifie la programmation par acteur avec la programmation réactive via des *réacteurs*, un acteur spécialisé qui communique avec d'autres acteurs par des flots de données au lieu d'envoi de messages classiques.

2.3.4 Vue globale

L'ensemble des travaux présentés jusque ici traitent des problèmes différents. De plus, un même problème peut être abordé de façon très différente par deux technologies. Par exemple, Redux et Flapjax traitent le problème de la gestion et de propagation de l'état dans une application, mais Redux choisit une approche centralisée alors que Flap-

jax est entièrement dissocié dans un graphe de dépendances. La figure 2.2 donne un aperçu global des trois problèmes principaux traités : la catégorie 1 traite le problème de l'abstraction du DOM, la 2 celui de la gestion de l'état et de sa propagation dans le programme et la 3 celui de la programmation distribuée ou multitier. Des technologies comme Pendulum et Hiphop.js sont difficilement classables dans ce type de figure. Pendulum et Hiphop.js traitent le problème de la gestion de l'état de l'application ainsi que sa propagation dans le programme et l'IHM (intégration avec les éléments du DOM pour Pendulum, et proxy réactifs pour Hiphop.js, voir le chapitre 5). Cependant, l'approche employée est radicalement différentes : au lieu de raisonner en terme d'état de variable, il faut raisonner en terme d'état de contrôle.

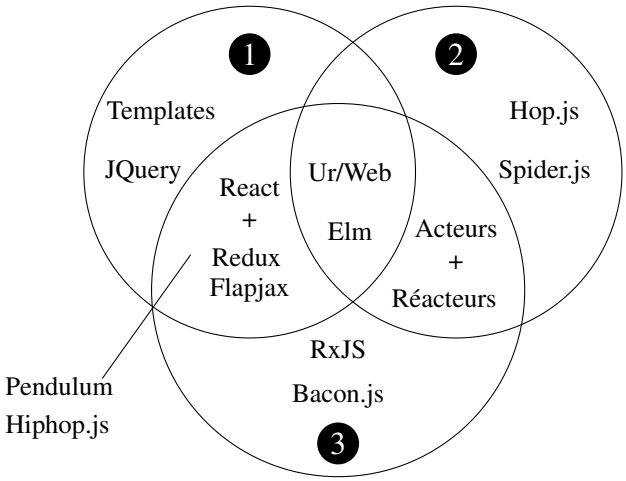

FIGURE 2.2 – Classement de travaux en fonction des problèmes traités

2.4 Positionnement de Hiphop.js

D'un côté, les langages réactifs synchrones semblent prédestinés à un environnement déterministe et statique. Par exemple, le nombre de capteurs d'un avion est connu avant l'exécution du programme calculant les paramètres de vol et ne changera pas pendant le vol. Par ailleurs, la compilation d'un langage réactif synchrone est coûteuse à cause de la complexité liée aux analyses statiques allant jusqu'à l'interprétation abstraite afin de garantir la causalité d'un programme. D'un autre côté, le modèle du web est dynamique par nature : les différentes entités avec lequel le programme va interagir varient au cours de l'exécution. Par exemple, dans une application web de visioconférence des participants peuvent se connecter et se déconnecter de façon imprévisible durant l'appel.

Ces deux mondes semblent contradictoires, pourtant, la sémantique des langages réactif synchrones n'interdit pas de fusionner les deux mondes : l'environnement du pro-

gramme doit être figé dans un instant afin de conserver l'hypothèse synchrone, mais il peut évoluer entre deux instants. Par exemple, ReactiveML permet créer des programmes synchrones et de modifier ses canaux de communications avec l'environnement de façon dynamique [Mandel et Pouzet, 2008].

Le langage Hiphop.js propose donc d'intégrer la programmation réactive synchrone dans le web. À noter que Hiphop.js fait suite à un premier langage implémenté en Scheme [Berry *et al.*, 2011]. Bien que ces deux langages ont vocation à orchestrer le web en se basant sur le langage Esterel, Hiphop.js adopte une approche différente beaucoup plus axée sur l'intégration avec le langage hôte JavaScript.

Chapitre 3

Le web 2.0 et Hiphop.js

Sommaire

3.1 Introduction

Plusieurs concepts caractérisent une application web 2.0 : la navigation hypertexte, l'interopérabilité des données, la composition de programme, la génération (ou la distribution) de code client par le serveur, son caractère dynamique ou encore son interactivité. Ce chapitre précise d'abord le concept d'interactivité dans le contexte des applications web. Ensuite, deux variantes d'une application sont spécifiées et implémentées en JavaScript. Ces implémentations sont discutées et mettent en évidence des difficultés liées à l'implémentation d'une application interactive en JavaScript. Ensuite, cette application est implémentée en Hiphop.js en introduisant de façon informelle le langage. Enfin, les implémentations Hiphop.js sont comparées aux versions JavaScript et montrent le bénéfice qu'elles apportent.

3.2 Interactivité, évènement et asynchronisme

La capacité d'une application web à réagir aux évènements permettent de la rendre interactive. Ces évènements sont d'origines hétérogènes, les plus communs sont les suivants :

- l'utilisateur : un clic ou un déplacement de la souris, l'appui sur une touche du clavier ;
- interne à l'application : un minuteur, le résultat d'un accès à une ressource du système ;
- caractéristique au réseau : la réception d'un message d'une autre machine ou la réponse d'un message envoyé depuis l'application.

La réaction à ces évènements peut entraîner une mise à jour de l'IHM, des changements de l'état interne à l'application, ou encore des actions générant à leur tour de nouveaux évènements. Ces évènements ont tous une chose en commun : ils sont *asynchrones*. En effet, il n'est pas possible de prédire à quel instant de l'exécution du programme un évènement se produit. JavaScript dispose de gestionnaires d'évènements pour traiter ces évènements asynchrones : il s'agit d'associer l'appel d'une fonction à l'arrivée d'un évènement. Ce type de programmation est connu pour être difficile à mettre au point et à maintenir lorsque la combinatoire d'évènements et d'états est grande.

3.3 Application de minuteur en JavaScript

Le fil conducteur de cette section est une application de minuteur s'exécutant dans un navigateur web. Cette application est simple, mais son comportement dépend d'évènements asynchrones. La spécification est tirée du *7 GUIs* challenge [1] qui propose des spécifications d'IHM correspondant à des cas d'usage classiques. Le but est d'identifier les meilleures approches de programmation en terme de compréhension et de maintenance de code pour implémenter des contraintes dépendantes de la combinatoire d'évènements et d'états.

3.3.1 Minuteur basique

La spécification du minuteur est décrite puis son implémentation JavaScript est présentée et brièvement discutée.

Spécification du minuteur

L'IHM du minuteur est présentée à la figure 3.1.

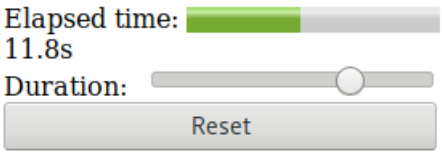

FIGURE 3.1 – Le minuteur.

La durée est indiquée par le curseur horizontal, le temps écoulé est indiqué par la jauge ainsi que par le champ numérique et le bouton Reset réinitialise le minuteur. Lorsque la minuterie est active, le jauge et le champ numérique sont mis à jour chaque dixième de seconde jusqu'au moment où le temps écoulé est égal à la durée du minuteur. À cet instant, le minuteur s'arrête. Le déplacement du curseur met immédiatement à jour la durée du minuteur et le reprend avec le temps écoulé courant. Un clic sur le bouton Reset réinitialise le temps écoulé et redémarre le minuteur.

La figure 3.2 représente un automate d'états finis correspondant à la spécification du contrôle du minuteur. Elle permet de visualiser l'ensemble de la structure de contrôle de l'application et la combinatoire des évènements et états.

1. `https://github.com/eugenkiss/7guis/wiki#time`

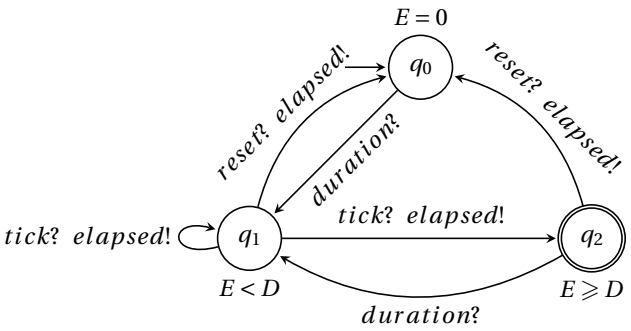

FIGURE 3.2 – Visualisation de la structure de contrôle du minuteur.

D correspond à la durée du minuteur, E au temps écoulé et les états aux différents prédicats entre le temps écoulé et la durée du minuteur. Enfin, les transitions représentent les évènements gérés par l'application. Les évènements reçus par l'application sont `tick`, `duration` et `reset`. Ils correspondent respectivement à chaque dixième de seconde écoulé, au déplacement du curseur et au clic sur le bouton Reset. L'évènement `elapsed` est généré par l'application pour mettre à jour la jauge et le champ numérique.

Implémentation de l'IHM

L'implémentation de l'IHM est écrite en HTML :

```
<div>Elapsed time: <meter id="meter"/></div>
<div> <span id="elapsed"></span>s</div>
<div>
  Duration:
  <input id="range" type="range" max="100"
         onchange="setD(this.value)"/>
</div>
<div>
  <button onclick="resetE()">Reset</button>
</div>
```

Cette IHM réagit à deux évènements que l'on peut qualifier « d'entrée » du point de vue de l'utilisateur :

- Lorsque l'utilisateur change la position du curseur, un gestionnaire d'évènements évalue l'expression `onchange` du curseur appelant la fonction `setD`. Il s'agit de l'évènement `duration`.
- Lorsque l'utilisateur clique sur le bouton Reset, un gestionnaire d'évènements

évalue l'expression `onclick` du bouton appelant la fonction `resetE`. Il s'agit de l'évènement `reset`.

Implémentation de l'orchestration

Une implémentation JavaScript possible de l'application de minuteur est la suivante [2] :

```
var D = 0, E = 0, id = -1;

window.onload = function() {
   $("meter").value = E;
   $("meter").max = D;
   $("elapsed").innerHTML = E;
   $("range").value = D;
}

function tick() {
   if (E < D) {
      E += 0.1;
      $("meter").value = E;
      $("elapsed").innerHTML = E;
   }
   if (E == D) {
      clearTimeout(id);
      id = -1;
   }
}

function startIntervalIfNeeded() {
   if (id == -1 && E < D) {
      id = setInterval(tick, 100);
   }
}

function setD(value) {
   D = value;
   $("meter").max = D;
   startIntervalIfNeeded();
}

function resetE() {
   E = 0;
   $("meter").value = E;
   clearTimeout(id);
   id = -1;
   startIntervalIfNeeded();
}
```

L'état du programme est ici défini par trois variables globales. Les variables globales `D` et `E` correspondent respectivement à la durée et au temps écoulé du minuteur. Une troisième variable globale `id` correspond à l'identifiant du minuteur interne à JavaScript permettant l'avancement du temps ; une valeur négative signifie que la minuterie est inactive.

La fonction `window.onload` est automatiquement appelée par le navigateur lorsque l'ensemble de la structure HTML et le programme de page sont chargés en mémoire. Elle peut s'assimiler au point d'entrée du programme. Ici, cette fonction synchronise les éléments graphiques avec l'état initial de l'application.

Les fonctions `setD` et `resetE` implémentent la mise à jour de l'état interne du minuteur et de l'IHM dans le cas de modification de la position du curseur ou du clic sur Reset. Enfin, les fonctions auxiliaires `tick` et `startIntervalIfNeeded` per-

2. La variable `$` est conventionnellement utilisée comme raccourci syntaxique de l'expression `document.getElementById`, API standard pour accéder aux éléments graphiques d'une page web.

mettent l'avancement du temps chaque dixième de seconde si la minuterie est active. L'appel de la fonction `tick` correspond à l'évènement $tick$ de l'automate.

Remarques

Pour comprendre ce type de programme et déterminer son état à chaque instant de l'exécution, le programmeur doit mentalement dérouler le flot de contrôle du programme et se poser les questions suivantes :

- Quels sont les différents appels asynchrones ?
- Quels effets de bords provoquent-ils ? Notamment sur la modification de variables globales ou la modification de l'IHM ?
- Ces effets de bords sont-ils systématiques, ou appliqués seulement lorsque l'application est dans un état spécifique ?

Ici, cette analyse est simple car la combinatoire d'évènements et d'états est faible. Elle n'est cependant pas directe : il faut faire de nombreux aller-retours dans le code source du programme et répéter cette opération plusieurs fois en supposant différents états. Cette analyse peut-elle devenir un point critique pour la vérification du programme lorsque la combinatoire des évènements et états augmente ?

3.3.2 Minuteur avec suspension

Une nouvelle fonction est ajoutée à la spécification du minuteur. Le but est de savoir si l'analyse du programme devient plus difficile après l'adaptation du code à la nouvelle spécification, notamment à cause du flot de contrôle complexifié par de nouveaux états, de nouvelles fonctions asynchrones, ou encore par la modification du code existant.

Extension de la spécification

La nouvelle spécification étend celle présentée en section 3.3.1 en ajoutant la possibilité de suspendre le minuteur lorsqu'il est actif. Dans ce cas, le temps écoulé n'est plus incrémenté jusqu'à ce que le minuteur soit réactivé. Le curseur et le bouton Reset restent fonctionnels durant la suspension : le déplacement du curseur met toujours à jour la durée du minuteur et un clic sur le bouton Reset réinitialise toujours le minuteur. Un bouton Suspend est ajouté dans l'IHM : il permet de basculer entre le mode normal et le mode suspendu. Lorsque le minuteur est suspendu, le bouton Suspend est coloré en orange.

L'automate de la figure 3.3 est une évolution de celui de la figure 3.2. Il montre que la nouvelle fonctionnalité augmente considérablement la combinatoire des évènements et états.

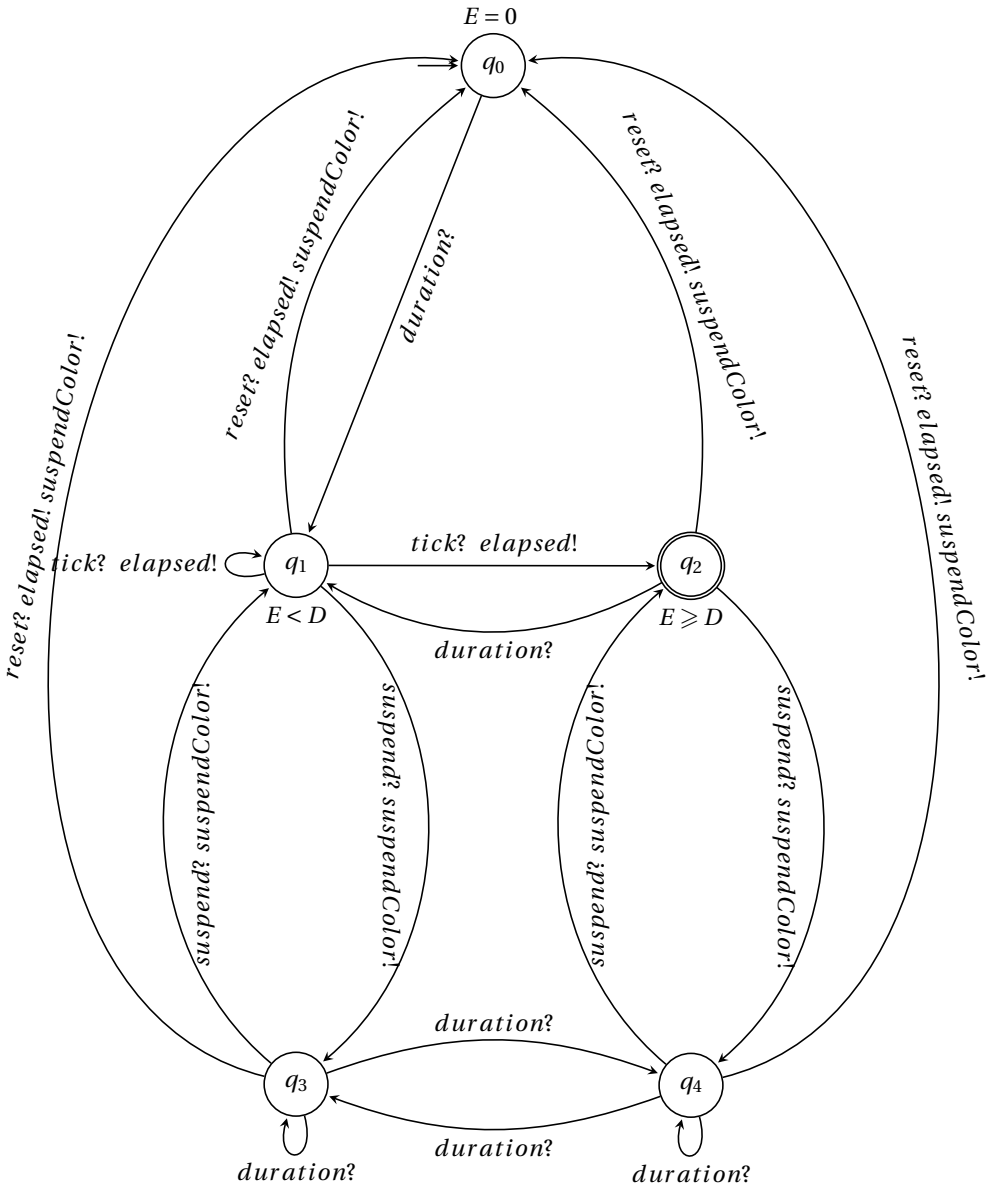

FIGURE 3.3 – Visualisation de la structure de contrôle du minuteur avec suspension.

Modification de l'IHM

La modification de l'IHM consiste à d'ajouter le bouton Suspend :

```
<button id="susp-btn" onclick="susp()">
  Suspend
</button>
```

La fonction `susp` est appelée chaque fois que l'utilisateur clique sur le bouton Suspend.

Modification de l'implémentation de l'orchestration

Dans un premier temps, la fonction `susp` est implémentée. Elle est appelée à chaque clic sur le bouton Suspend et suspend ou réactive la minuterie en fonction de l'état de la variable globale `isSusp` :

```
var isSusp = false;
function susp() {
   if (isSusp) {
      isSusp = false;
      $("susp-btn").style.backgroundColor = "transparent";
      startIntervalIfNeeded();
   } else {
      isSusp = true;
      $("susp-btn").style.backgroundColor = "orange";
      clearTimeout(id);
      id = -1;
   }
}
```

Enfin, les fonctions `setD` et `resetE` doivent être modifiées pour prendre en compte l'état suspendu ou actif de la minuterie :

```
function resetE() {
   if (isSusp) {
      isSusp = false;
      $("susp-btn").style.backgroundColor = "transparent";
   }
   E = 0;
   $("meter").value = E;
   clearTimeout(id);
   id = -1;
   startIntervalIfNeeded();
}

function setD(value) {
   D = value;
   $("meter").max = D;
   if (!isSusp) { startIntervalIfNeeded(); }
}
```

Remarques

L'analyse décrite précédemment pour analyser ce type de programme est à refaire entièrement :

- le nouvel appel à `susp` est asynchrone et il modifie l'état du programme par la mise à jour de la variable `isSusp` et de l'IHM. De plus, `susp` a un comportement différent selon l'état courant de `isSusp`;
- les fonctions `resetE` et `setD` sont modifiées afin de faire dépendre leur comportement de la variable `isSusp`.

Par ailleurs cette analyse est maintenant plus complexe : la taille du code a sensiblement augmenté, la majeure partie du code existant a été modifiée et le comportement de l'application dépend non plus de 6 évènements et 3 états mais de 14 évènements et 5 états. Ici, l'application est simple pour les besoins de l'exemple, donc l'analyse reste relativement simple. Cependant, cela montre que le programme initial n'est pas modulaire, ce qui lève une réelle difficulté dans le cas d'applications plus complexes : chaque modification de spécification implique potentiellement une modification significative du programme dont la combinatoire d'évènements et états à considérer est possiblement décuplée. Lorsque le code orchestrant ces programmes atteint les milliers de ligne, la modification est longue et la vérification consistant à l'analyse mentale précédemment présentée devient difficile.

3.4 Application de minuteur en Hiphop.js

Dans cette section, l'application de minuteur est implémentée avec Hiphop.js. Dans un premier temps, la version simple est présentée et introduit les concepts et constructions de Hiphop.js. Ensuite, la version avec suspension est présentée. Enfin, les différences entre JavaScript et Hiphop.js sont discutées.

3.4.1 Minuteur basique

Le code de l'IHM étant commun aux versions Hiphop.js et JavaScript, seule l'implémentation de l'orchestration est présentée. Dans un premier temps, considérons l'implémentation Hiphop.js d'un minuteur sans bouton Reset :

```
MODULE BasicTimer (IN duration(0), OUT elapsed) {
   EMIT elapsed(0);
   LOOP {
      IF (VAL(elapsed) < VAL(duration)) {
         RUN(TimeoutMod(100));
         EMIT elapsed(PREVAL(elapsed) + 0.1);
      } ELSE {
         PAUSE;
      }
   }
}
```

Avant de rentrer dans l'explication de ce code, étudions d'abord quelques concepts fondamentaux de Hiphop.js qui seront détaillés dans les chapitres 4 et 5. Le module `BasicTimer` est un programme Hiphop.js qui s'exécute en une succession de *réactions* ou d'*instants* (les deux termes sont synonymes et interchangeables). Une réaction est l'exécution d'une fonction déterministe prenant un ensemble de *signaux d'entrée* et qui calcule un ensemble de *signaux de sortie*. Les signaux jouent un rôle similaire aux évènements JavaScript. La réaction est dite *atomique* car du point de vue de Hiphop.js elle s'exécute toujours jusqu'à complétion sans interruption et de façon instantanée, c'est-à-dire en un temps conceptuellement nul. Enfin, comme le montre la figure 3.4, un programme Hiphop.js et le programme JavaScript environnant sont exécutés en alternance de façon coopérative. Les réactions sont déclenchées depuis le programme JavaScript, par exemple lors de la survenue d'un évènement. Lorsque la réaction se termine le contrôle est rendu au programme JavaScript.

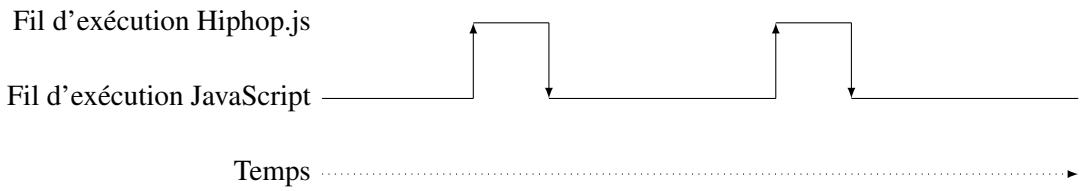

FIGURE 3.4 – Visualisation du modèle d'exécution de Hiphop.js.

Contrairement à l'implémentation JavaScript où le programme est composé d'un ensemble de fonctions exécutées de façon indépendante, le programme Hiphop.js comprend un unique code séquentiel composé d'instructions temporelles définies par une durée de vie dont l'unité est la réaction. Par exemple, l'instruction `EMIT` démarre et se termine dans la même réaction. À l'opposé, l'instruction `PAUSE` démarre dans un instant et se termine dans le suivant. Par exemple, `PAUSE; PAUSE; EMIT foo` pause pendant deux réactions et émet le signal `foo` lors d'une troisième réaction. La durée de vie de chaque instruction Hiphop.js est détaillée au chapitre 4.

Expliquons maintenant comment `BasicTimer` s'exécute. À la première réaction, le signal `elapsed` est émis avec la valeur 0, ce qui permet d'initialiser le temps écoulé. Ensuite, `LOOP` démarre son corps qui commence par évaluer le test `IF`.

- Si la valeur de `elapsed` est strictement inférieure à cette de `duration`, la première branche de `IF` est prise et l'instruction `RUN` démarre. Cette instruction appelle un autre module Hiphop.js `TimeoutMod` utilisant la fonction JavaScript `setTimeout` pour créer une attente d'un dixième de seconde. Ce module est détaillé au chapitre 5. La fonction JavaScript se termine immédiatement ce qui évite de bloquer l'exécution Hiphop.js, mais `RUN` est bloquant pour Hiphop.js : il n'y a donc plus rien à exécuter et la réaction se termine. Lorsque le dixième de seconde est écoulé, une fonction interne à Hiphop.js est appelée ; elle informe le programme Hiphop.js que `RUN` doit se terminer à la prochaine réaction puis elle déclenche une réaction. `RUN` se termine et `EMIT` émet le signal `elapsed` en incrémentant sa valeur. Puisque `EMIT` est instantanée, le corps de la boucle se termine et redémarre immédiatement sa première instruction. Le test est immédiatement réévalué et une attente est à nouveau démarrée si le temps écoulé est toujours strictement inférieur à la durée du minuteur.
- Si la valeur de `elapsed` est supérieure ou égale à celle de `duration`, il n'y a rien à faire. Le programme marque une pause et la réaction se termine : l'instruction `PAUSE` est démarrée et ne terminera qu'à la réaction suivante. Lorsqu'un évènement JavaScript déclenchera une nouvelle réaction, par exemple lors de la modification de la durée du minuteur, `PAUSE` se terminera, `LOOP` se terminera également et redémarrera sa première instruction. Ainsi, si la durée du minuteur est toujours supérieure au temps écoulé, le test prendra alors la première branche du `IF`.

Le bouton Reset

Puisque Hiphop.js permet la composition de modules grâce à l'instruction `RUN`, l'implémentation du minuteur basique en Hiphop.js est un petit module englobant `BasicTimer` dans une instruction temporelle appelée `LOOPEACH`. Le module `BasicTimer` est inchangé :

```
MODULE Timer (IN duration(0), IN reset, OUT elapsed) {
   LOOPEACH(NOW(reset)) {
      RUN(BasicTimer);
   }
}
```

L'instruction LOOPEACH démarre BasicTimer à la première réaction. À chaque clic sur le bouton Reset le signal reset est émis par JavaScript. L'expression NOW(reset) devient vrai et LOOPEACH *préempte* son corps : l'état de contrôle du corps du LOOPEACH est détruit et la première instruction du corps est démarrée comme à la première réaction. Ici, cela a pour effet de réinitialiser elapsed à 0 et redémarrer le test de IF du module BasicTimer.

La connexion entre l'IHM, le programme JavaScript et le programme Hiphop.js permettant la propagation des évènements aux signaux Hiphop.js et vice-versa fait l'objet de la section 5.2.

3.4.2 Minuteur avec suspension

L'implémentation Hiphop.js du programme de minuteur avec suspension est la suivante :

```
MODULE SuspendableTimer (IN duration(0), IN reset, IN suspend,
                         OUT elapsed, OUT suspendColor)  {
   LOOPEACH(NOW(reset)) {
      FORK {
         SUSPEND TOGGLE(NOW(suspend)) {
            RUN(Timer);
         }
      } PAR {
         EMIT suspendColor("transparent");
         LOOP {
            AWAIT(NOW(suspend));
            EMIT suspendColor("orange");
            AWAIT(NOW(suspend));
            EMIT suspendColor("transparent");
         }
      }
   }
}
```

Ce code utilise les constructions Hiphop.js FORK et PAR qui permettent l'exécution parallèle, c'est-à-dire durant la même réaction, de la gestion de l'état du minuteur et du changement de couleur du bouton Suspend. Ce code parallèle est englobé par l'instruction LOOPEACH, ce qui a pour effet de forcer la terminaison et le redémarrage du parallèle si le signal reset est présent lors de la réaction.

L'état du minuteur est géré par le module Timer vu en section 3.4.1 qui est appelé par RUN. La différence est liée à l'instruction SUSPEND TOGGLE englobante : l'exécution et l'état de ce code seront « gelés » à chaque réaction à partir de celle où le signal suspend est présent, ils seront « dégelés » à la réaction où suspend sera à nouveau

présent.

L'alternance de couleur du bouton Suspend est gérée grâce aux instructions `LOOP`, `AWAIT` et `EMIT`. L'instruction `AWAIT` démarre lors d'une réaction et ne se termine qu'à une réaction suivante si le signal `suspend` est présent. Ceci permet l'attente d'un clic sur le bouton Suspend. À ce moment, `AWAIT` se termine et `EMIT` émet le signal `suspendColor` avec la valeur `orange` associée, indiquant à l'environnement de changer la couleur du bouton en orange. Immédiatement après, une nouvelle instruction `AWAIT` démarre. Elle ne se termine que lors d'un prochain clic, où de façon similaire `suspendColor` sera émis avec la valeur `transparent`, indiquant à l'environnement d'enlever la couleur orange du bouton. Immédiatement après, `LOOP` redémarre la première instruction de son corps, et l'alternance peut continuer. L'émission initiale de `suspendColor` assure la non-coloration du bouton Suspend lors de la première réaction et lorsque le minuteur est réinitialisé par `LOOPEACH`.

3.5 Discussion

Plusieurs particularités de l'implémentation Hiphop.js du minuteur sont à noter. D'abord, il n'existe aucune variable globale, l'état du programme étant explicitement connu par le flot de contrôle grâce à la durée de vie des instructions. Ensuite, l'exécution synchrone du code et l'emboîtement syntaxique des instructions rend explicite la préemption ou la suspension. Enfin, les signaux d'entrée et les signaux de sortie permettent de distinguer clairement la source et la destination des évènements.

Ces particularités caractérisent l'alternative à la boucle d'évènements de JavaScript proposée par Hiphop.js. Lorsqu'un évènement survient, si le flot de contrôle est à une instruction dépendant de l'évènement, alors il est pris en compte, sinon il est ignoré. L'implémentation et la vérification d'un programme Hiphop.js ne nécessitent donc pas de dérouler mentalement le code avec la combinatoire des évènements et des états possibles. Le code est modulaire : l'instruction `RUN` permet d'inclure un code existant sans le modifier dans un autre code indépendant.

Dans l'exemple du minuteur, l'ajout de la suspension augmente de façon significative la combinatoire d'évènements et d'états, comme le montrent les figures 3.2 et 3.3. L'implémentation JavaScript du minuteur avec suspension est plus complexe à implémenter et vérifier que la version sans suspension alors que la version avec suspension de Hiphop.js n'est pas plus complexe à implémenter ni à vérifier que la version sans suspension.

3.6 Conclusion

À travers le développement d'une application simple mais significative vis-à-vis de l'utilisation de mécanismes asynchrones, ce chapitre met en évidence deux problèmes majeurs induits par la gestion des évènements asynchrones par JavaScript. D'abord, le déroulement mental par le programmeur du flot de contrôle d'un programme comprenant de nombreuses fonctions de traitement d'évènements asynchrones est difficile, en particulier lorsque la combinatoire d'évènements et d'états possible est grande. Ensuite, l'ajout d'une fonctionnalité, même simple, à un programme existant nécessite une refonte profonde de la logique temporelle du programme. Cela entraîne la modification de l'implémentation existante, l'ajout de nouveau code et une revérification globale du programme.

Hiphop.js répond au problème de la gestion de mécanisme asynchrone en permettant une compréhension visuelle directe du contrôle dans les différents cas de traitement d'évènements grâce à la notion de réaction ainsi que de durée de vie et d'emboîtement syntaxique de ses instructions. Par ailleurs, plusieurs propriétés importantes comme la composition de programme, la réutilisation de code ainsi que la séparation des préoccupations peuvent être facilement appliquées dans un programme Hiphop.js, facilitant l'ajout de nouvelles fonctionnalités sans modification de la logique temporelle du programme originel.

Chapitre 4

Le langage Hiphop.js

Sommaire

4.1 Introduction

Ce chapitre suit le modèle d'un manuel d'utilisation exhaustif du langage Hiphop.js. Les différents constructeurs du langage y sont présentés de façon analogue à la documentation d'Esterel v5 [Berry, 2000a]. Une carte du langage synthétise la grammaire de Hiphop.js à l'annexe A. Ce chapitre s'adresse donc aux programmeurs désireux d'utiliser Hiphop.js. Aucune connaissance préalable de la programmation réactive synchrone n'est requise.

Conventions booléennes

En Hiphop.js comme en JavaScript, les valeurs `false`, `undefined`, `null`, `0` ou la chaîne vide correspondent à la valeur booléenne `faux`. Les autres valeurs correspondent à la valeur booléenne `vrai` [ECMA, 2015].

4.2 Réactions et durée de vie des instructions

Un programme Hiphop.js est exécuté sous la forme de réactions – ou instants – atomiques : conceptuellement, une réaction s'exécute en temps 0, c'est-à-dire instantanément par rapport au flot des évènements de l'environnement. Une instruction *démarre* dans la réaction lorsque le flot de contrôle du programme arrive à elle. De façon analogue, une instruction se *termine* dans la réaction où le flot de contrôle en sort. L'instruction est dite *active* entre son démarrage et sa terminaison. On distingue deux types d'instructions.

- Une instruction *instantanée* démarre et se termine dans la même réaction.
- Une instruction ayant une *épaisseur temporelle* ne se termine jamais à la réaction où elle a démarrée. Elle reste donc active d'une réaction à la suivante jusqu'à sa terminaison.

4.3 « Hello world ! » en Hiphop.js

Cette section présente une variante du classique « Hello world ! » en Hiphop.js. Le but est de présenter un exemple de programme Hiphop.js simple mais complet, exécutable et interagissant avec l'environnement. Le code est écrit dans fichier JavaScript :

```
const hh = require("hiphop");
const hello = MODULE (IN sayHelloAgain, OUT cnt(0)) {
  LOOPEACH(NOW(sayHelloAgain)) {
    EMIT cnt(PREVAL(cnt) + 1);
    ATOM {
      console.log("Hello world from Hiphop.js!", VAL(cnt));
    }
  }
}
const m = new hh.ReactiveMachine(hello);
m.addEventListener("cnt", function(evt) {
  console.log("Hello world from JavaScript!", evt.signalValue);
});
m.react();
m.inputAndReact("sayHelloAgain");
```

La variable `hh` rend accessibles depuis JavaScript l'environnement d'exécution et les instructions de Hiphop.js, `hello` représente le programme Hiphop.js et `m` une instance exécutable. Les appels aux méthodes `react` et `inputAndReact` permettent d'interagir avec le programme Hiphop.js depuis l'environnement JavaScript, et la méthode `addEventListener` permet de déclencher des actions dans l'environnement JavaScript depuis Hiphop.js.

Les programmes écrits en Hiphop.js sont exécutables par l'environnement JavaScript Hop.js [Serrano et Prunet, 2016]. L'intégration entre Hiphop.js et JavaScript est détaillée au chapitre 5, notamment l'écriture de programme Hiphop.js dans des fichiers JavaScript ainsi que la *machine réactive*.

4.4 Syntaxe concrète

La grammaire de Hiphop.js étend celle de JavaScript.

hiphopjs:
    js
    hh-module
    hh-statement

Le symbole non terminal *hiphopjs* représente l'ensemble de la grammaire de Hiphop.js, *js* représente la grammaire de JavaScript, et *identifier* un identifiant JavaScript. Enfin, les symboles non terminaux *hh-module* et *hh-statement* sont définis dans la suite de ce chapitre.

4.4.1 Module

L'unité de programmation d'un programme Hiphop.js est le module [1] :

```
hh-module:
        MODULE identifier_opt (signal-list_opt) {
          hh-statement
        }
```

Un module peut être nommé et dispose d'une *interface* qui définit les signaux globaux du module, voir section 4.4.2. L'interface du module correspond au symbole non terminal optionnel *signal-list*. Le corps du module comprend ses instructions [2].

4.4.2 Signaux

Un signal est une entité nommée faisant partie d'un programme Hiphop.js.

```
signal-list:
        signal
        signal-list, signal

signal:
        direction identifier combine_opt
        direction identifier(expression) combine_opt

direction:
        IN
        OUT
        INOUT

combine:
        COMBINE(functional-value)
```

Dans toute réaction, un signal possède un statut unique *présent* ou *absent* et optionnellement une valeur JavaScript quelconque. Par défaut, un signal est absent à chaque réaction alors que sa valeur persiste d'un instant à l'autre. Un signal défini dans l'interface du module est *global* : il est accessible depuis n'importe quelle instruction du module. Une API JavaScript permet d'émettre ou lire les signaux globaux depuis l'environnement, elle est décrite dans le chapitre 5.

1. La convention grammaticale pour formaliser la syntaxe du langage est celle utilisée dans [Kernighan et Ritchie, 1978].

2. Dans la suite de cette présentation et à l'exception de l'instruction `ATOM`, le corps d'une instruction correspond toujours au symbole non terminal *hh-statement* compris entre les caractères `{` et `}`.

Les signaux globaux sont définis avec une *directionnalité* : les signaux d'*entrée* peuvent être émis depuis l'environnement, les signaux de *sortie* peuvent être lus depuis l'environnement, et enfin les signaux d'*entrée-sortie* peuvent être émis et lus par l'environnement. Les mots-clés `IN`, `OUT` et `INOUT` identifient la directionnalité d'un signal, respectivement un signal d'entrée, de sortie, et d'entrée-sortie.

Une valeur initiale peut être associée à un signal lors de sa définition. Cette valeur correspond à une expression Hiphop.js calculée au premier instant du programme :

```
IN cnt(1), OUT cntSum(VAL(cnt) + 1)
```

Au premier instant, le signal d'entrée `cnt` vaut 1, et le signal de sortie `cntSum` vaut 2 : la somme de la valeur du signal `cnt` et de 1. L'instruction `VAL` permettant de lire la valeur d'un signal est expliquée en section 4.4.3. L'initialisation cyclique de signaux est interdite, par exemple :

```
IN cnt(1 + VAL(cntSum)), OUT cntSum(VAL(cnt) + 1))
```

Ce code est rejeté dès la compilation du programme.

Signaux combinés

Par défaut, une unique valeur peut-être émise pour un même signal lors d'un instant [3]. La définition d'une fonction de *combinaison* permet l'émission multiple de valeurs pour ce même signal lors d'un instant. Le symbole non terminal $functional\text{-}value$ correspond à la fonction de combinaison : il s'agit soit d'une définition littérale de fonction, soit d'une variable référençant une fonction. L'ordre d'émission des valeurs n'étant pas déterministe, la fonction doit donc être associative et commutative [4] :

```
MODULE (OUT a(2) COMBINE((x, y) => x + y)) {
   EMIT a(1);
   EMIT a(3);
}
```

Dans cet exemple, un signal de sortie `a` est défini avec la valeur initiale 2. L'instruction `COMBINE` syntaxiquement positionnée immédiatement après la déclaration et initialisation (optionnelle) de `a` indique que le signal est combiné. `COMBINE` prends un unique paramètre qui est la fonction JavaScript de combinaison d'arité 2, ici, une fonction d'addition.

3. Une émission depuis l'environnement est considérée comme une émission dans l'instant.
4. C'est la responsabilité du programmeur de s'assurer que la fonction de combinaison soit associative et commutative.

À la première réaction de ce programme, `EMIT a(1)` émet le signal `a` avec la valeur 1. La fonction de combinaison n'est jamais appelée à la première émission du signal dans l'instant. La valeur initiale 2 est donc écrasée et `a` vaut 1. Immédiatement après dans le même instant, `EMIT a(3)` émet `a` avec la valeur 3 : la fonction de combinaison de `a` est automatiquement appelée avec les paramètres `(1, 3)` et retourne 4. L'instant est terminé, le signal `a` est présent et vaut 4. À noter que puisque la fonction de combinaison est associative et commutative, le résultat serait le même en inversant les deux émissions.

4.4.3 Expressions

Les expressions Hiphop.js étendent les expressions JavaScript par l'utilisation d'instructions supplémentaires appelées *accesseurs*.

hh-expression:
- *js-expression*
- **NOW**(*identifier*)
- **PRE**(*identifier*)
- **VAL**(*identifier*)
- **PREVAL**(*identifier*)

Le symbole non terminal *js-expression* représente une instruction JavaScript étendue par des constructions supplémentaires définies par *hh-expression*. La signification des symboles terminaux est la suivante, pour `s` un signal donné :

- `NOW(s)` : retourne `vrai` si `s` est présent dans l'instant, `faux` sinon.
- `PRE(s)` : retourne `vrai` si `s` était présent dans l'instant précédent, `faux` sinon.
- `VAL(s)` : retourne la valeur de `s` dans l'instant.
- `PREVAL(s)` : retourne la valeur de `s` dans l'instant précédent.

Si aucune valeur n'est définie pour `s`, `VAL(s)` et `PREVAL(s)` retournent `undefined`. L'exemple suivant présente trois expressions valides :

```
NOW(s) && !PRE(s)
NOW(s) || stateBool
VAL(s) + 5 + PREVAL(s)
```

La première expression vaut `vrai` si le signal `s` est présent dans l'instant et ne l'était pas dans l'instant précédent, `faux` sinon. La seconde expression vaut le *OU* logique entre la présence de `s` dans l'instant et la variable JavaScript `stateBool`. La troisième

expression vaut la somme de la valeur de `s` dans l'instant, de 5, et de `s` à l'instant précédent.

4.4.4 Délais

Certaines instructions Hiphop.js sont paramétrées par un *délai*.

```
delay:
        (hh-expression)
        COUNT(hh-eprexssion, hh-expression)
        IMMEDIATE(hh-expression)
```

Chaque clause correspond à un type de délai :

- Un *délai standard* est une expression Hiphop.js dont l'évaluation à `vrai` entraîne la terminaison du délai. L'expression est évaluée à chaque instant où l'instruction est active, sauf celui où l'instruction est démarrée.
- Un *compteur* est une expression Hiphop.js dont l'évaluation à `vrai` N instants entraîne la terminaison du délai, où $N > 0$. La première expression s'évalue en un entier $N > 0$ à l'instant où l'instruction est démarrée. Ensuite, la seconde expression est évaluée aux instants suivants où l'instruction est active (comme pour les délais standards) : si elle vaut `vrai`, alors N est décrémenté. À l'instant où $N = 0$, le délai se termine. Par exemple :

  ```
  COUNT(3, NOW(s))
  COUNT(VAL(s) + 3, VAL(t) == 5)
  ```

  Plaçons-nous à l'instant où ces instructions sont démarrées. Le premier compteur se termine au troisième instant où `s` est présent. Le second compteur se termine au $N^{ème}$ instant où la valeur du signal `t` est égale à 5, et où N = `VAL(s) + 3`. Par exemple, si la valeur de `s` est égale à 2 à l'instant où le compteur est démarré, alors le compteur se termine au deuxième instant suivant où la valeur de `t` est égale à 5.
- Un délai standard est rendu immédiat en le préfixant par le mot clé `IMMEDIATE`. Dans ce cas, l'expression du délai est évaluée à l'instant de démarrage de l'instruction. Ainsi, l'instant de démarrage de l'instruction est pris en compte dans la terminaison du délai, qui peut donc être instantané.

4.4.5 Instructions Hiphop.js

Cette section présente l'ensemble des instructions du langage ainsi que les différentes façons de les utiliser. Certaines instructions peuvent être construites par la composition d'autres instructions du langage, elles sont détaillées dans [Berry, 2000a, Berry, 2018a]. Ce chapitre ayant vocation à servir de référence pour l'utilisation du langage, ces compositions ne sont pas systématiquement décrites.

```
hh-statement:
        basic-statement
        sequence-statement
        parallel-statement
        emit-statement
        await-statement
        cond-statement
        preemption-statement
        suspension-statement
        atom-statement
        local-statement
        loop-statement
        trap-statement
        exec-statement
        run-statement
```

4.4.6 Instructions basiques de flot de contrôle

Le langage définit trois instructions basiques de flot de contrôle.

```
basic-statement:
        NOTHING
        PAUSE
        HALT
```

L'instruction `NOTHING` est instantanée et n'a aucun effet. L'instruction `PAUSE` se termine à l'instant suivant celui où elle est démarrée. Enfin, l'instruction `HALT` ne se termine jamais.

4.4.7 Séquence

La séquence d'instructions Hiphop.js est soit implicite, soit explicite.

```
sequence-statement:
        hh-statement; hh-statement
        SEQUENCE { hh-statement }
```

Pour des raisons d'uniformité avec JavaScript le caractère « `;` » placé en fin de ligne est optionnel.

Lorsqu'une instruction de séquence démarre, la première instruction du corps est immédiatement démarrée. La terminaison de cette première instruction entraîne le démarrage immédiat de l'instruction suivante, et ce jusqu'à la première instruction qui ne se termine pas instantanément. La séquence se termine à l'instant où la dernière instruction de son corps se termine. Par exemple :

```
SEQUENCE {
  NOTHING;
  PAUSE;
  PAUSE;
}
```

À l'instant où `SEQUENCE` est démarrée, `NOTHING` démarre et se termine instantanément, la première `PAUSE` démarre ensuite mais ne se terminera qu'à l'instant suivant. À l'instant suivant, la première `PAUSE` se termine puis la seconde `PAUSE` démarre. À un troisième instant, la dernière `PAUSE` se termine ainsi que la `SEQUENCE`.

4.4.8 Exécution parallèle

Les constructions `FORK` et `PAR` permettent l'exécution d'instructions en parallèle.

```
parallel-statement:
        FORK identifier_opt { hh-statement } branch_opt

branch:
        PAR { hh-statement } branch_opt
```

Lorsque `FORK` est démarrée, le corps de chaque branche est immédiatement démarré. Le parallèle se termine à l'instant où la dernière branche active se termine.

Pour permettre à Hiphop.js une dynamicité qui n'existe pas en Esterel mais qui est indispensable dans le contexte du web, il est possible d'ajouter ou de retirer des branches d'un parallèle nommé entre deux réactions. Ces manipulations sont possibles depuis JavaScript via une API de la machine réactive décrite au chapitre 5.

Ordonnancement de l'exécution de branches parallèles

Le langage Hiphop.js garantit que l'exécution des branches est ordonnancée dans la réaction de façon à respecter les règles suivantes [Berry, 2002] :

- L'exécution d'une branche dont le flot de contrôle atteint la lecture du statut d'un signal `s` sera bloquée jusqu'à ce que `s` soit émis depuis au moins une autre branche, ou jamais émis dans l'instant.
- L'exécution d'une branche dont le flot de contrôle atteint la lecture de la valeur d'un signal `s` sera bloquée jusqu'à ce que l'ensemble des valeurs de `s` soient émises depuis les autres branches, ou que `s` ne soit jamais émis avec une valeur dans l'instant.

L'exemple suivant illustre les règles d'ordonnancement de branches parallèles :

```
FORK {
  AWAIT(VAL(s) > 5 && NOW(v))
} PAR {
  EMIT s(3);
  EMIT v
}
```

Lorsque l'instruction `FORK` est démarrée, le corps de chaque branche parallèle est immédiatement démarré. L'exécution de `AWAIT` de la première branche est bloquée jusqu'à ce que les instructions `EMIT` de la seconde branche soient exécutées : sachant que `NOW(v)` et `EMIT v` sont exécutées dans le même instant, `EMIT v` est exécutée avant `NOW(v)`. Le raisonnement est analogue pour l'émission avec valeur de `s`. À noter que cet ordonnancement n'est pas perceptible depuis l'environnement JavaScript, pour qui la réaction est atomique et donc pour qui l'exécution des branches est instantanée.

Enfin, un *cycle de dépendance*, c'est-à-dire un cycle entre émission et lecture d'un même signal est interdit :

```
FORK {
   IF(NOW(v)) {
      EMIT s
   }
} PAR {
   IF(NOW(s)) {
      EMIT v
   }
}
```

Le cycle de dépendance est lié à l'émission du signal `s` dépend de l'émission du signal `v` qui dépend lui même de l'émission du signal `s`. Cet exemple déclenche une erreur à la première réaction du programme : puisque les programmes Hiphop.js sont construits dynamiquement, aucune analyse de causalité n'est appliquée à la compilation du programme, à l'exception de l'initialisation et émission cyclique de la valeur d'un signal.

4.4.9 Émission de signal

Les instructions `EMIT` et `SUSTAIN` permettent l'émission de signaux.

```
emit-statement:
        EMIT emit-list
        SUSTAIN emit-list

emit-list:
        emit-signal
        emit-list, emit-signal

emit-signal:
        identifier
        identifier(hh-expression)
```

L'instruction `EMIT` émet un ensemble de signaux et se termine instantanément, alors que `SUSTAIN` émet continuellement à chaque instant un ensemble de signaux et ne se termine donc pas. Chaque signal peut être émis avec une valeur correspondante à une expression évaluée à chaque émission. Plusieurs émissions sont possibles dans le même instant dans le cas des signaux combinés.

4.4.10 Attente

L'instruction `AWAIT` attend la terminaison d'un délai.

```
await-statement:
        AWAIT delay
```

Le démarrage de l'instruction `AWAIT` démarre immédiatement le délai et l'instruction se termine à l'instant où le délai se termine. Par exemple, l'attente de la présence d'un signal `s` s'écrit de la façon suivante :

```
AWAIT(NOW(s))
```

L'évaluation d'un délai étant retardée par définition (voir section 4.4.4), `AWAIT` ne se

termine pas même si s est présent au démarrage. Au contraire, si le délai est immédiat :

```
AWAIT IMMEDIATE(NOW(s))
```

Alors AWAIT se termine immédiatement si s est présent au démarrage. Ainsi, une attention particulière doit être portée à ce type de construction :

```
AWAIT IMMEDIATE(NOW(s));
AWAIT IMMEDIATE(NOW(s));
```

Si s est présent dans l'instant où cette séquence est démarrée, les deux instructions AWAIT seront démarrées et se termineront dans l'instant.

Enfin, l'exemple suivant illustre l'utilisation d'un compteur :

```
AWAIT COUNT(3, NOW(S))
```

Le compteur est initialisé à 3 lorsque l'instruction est démarrée. L'instruction se terminera au troisième instant où le signal s est présent.

4.4.11 Branchement conditionnel

cond-statement:
IF (*hh-expression*) { *hh-statement* } $else_{opt}$

else:
ELSE { *hh-statement* }
ELSE *cond-statement*

Au démarrage de IF, son expression est évaluée. Si elle vaut vrai, le corps de IF est immédiatement exécuté, sinon le contrôle passe au corps de ELSE s'il existe. Dans le cas où plusieurs clauses ELSE IF sont emboîtées, elles sont évaluées en séquence jusqu'à la première expression qui s'évalue à vrai. L'instruction se termine à l'instant où le corps correspondant à la clause sélectionnée se termine ou si aucune clause n'est sélectionnée.

4.4.12 Préemption

Les instructions ABORT et WEAKABORT permettent la préemption d'instructions.

preemption-statement:
ABORT *delay* { *hh-statement* }
WEAKABORT *delay* { *hh-statement* }

Les instructions ABORT et WEAKABORT démarrent immédiatement leur corps et leur délai à l'instant où elles sont démarrées. Elles se terminent à l'instant où leur corps se termine.

Dans le cas où le délai se termine avant le corps, une *préemption* est appliquée sur le corps : les instructions du corps ne sont plus actives et leur état de flot de contrôle est réinitialisé. ABORT et WEAKABORT diffèrent par le moment où elles préemptent leur corps.

- ABORT applique une une préemption *forte*. Le corps de l'instruction ne reçoit pas le contrôle à l'instant où le délai se termine. Considérons l'exemple suivant :

  ```
  ABORT(NOW(a)) {
     SUSTAIN s;
  }
  ```

  Le signal s est émis à chaque instant jusqu'à la terminaison du délai **exclu**. Ainsi, dans l'instant où le signal a est présent, SUSTAIN n'est pas exécutée et s n'est pas émis.

- WEAKABORT applique une préemption *faible*. Le corps de l'instruction reçoit le contrôle à l'instant où le délai se termine. Par exemple :

  ```
  WEAKABORT(NOW(a)) {
     SUSTAIN s;
  }
  ```

  Le signal s est émis à chaque instant jusqu'à celui de la terminaison du délai **inclus**. Ainsi, dans l'instant où le signal a est présent, s est également émis car l'instruction SUSTAIN est exécutée à cet instant.

4.4.13 Suspension

L'instruction SUSPEND permet la suspension de l'exécution de son corps.

suspension-statement:
SUSPEND *delay* { *hh-statement* }
SUSPEND FROM *delay* **TO** *delay* *emit-suspended*$_{opt}$ { *hh-statement* }
SUSPEND TOOGLE *delay* *emit-suspended*$_{opt}$ { *hh-statement* }

emit-suspended:
EMITWHENSUSPENDED *identifier*

L'instruction SUSPEND interrompt l'exécution de son corps pendant un ou plusieurs instants et permet de le redémarrer à un instant ultérieur. L'état du flot de contrôle du

corps est conservé pendant la suspension. Le démarrage de l'instruction exécute immédiatement son corps. Le corps est suspendu à l'instant où le délai se termine, et SUSPEND se termine à l'instant où son corps se termine. Soit l'exemple suivant :

```
SUSPEND(NOW(r)) {
   EMIT a;
   PAUSE;
   EMIT b
}
```

À l'instant où SUSPEND est démarrée, le signal a est émis, PAUSE est démarrée et l'instant se termine. Si le signal r est présent à l'instant suivant, le corps est alors suspendu : l'instruction PAUSE ne se termine pas et b n'est pas émis. Ce comportement se répète tant que r est présent. Au prochain instant où r est absent, PAUSE se termine, b est émis et SUSPEND se termine. Le corps peut être suspendu et redémarré plusieurs fois, le délai étant immédiatement repris à l'instant où il se termine et réévaluera son expression dès l'instant suivant.

La variante SUSPEND FROM TO suspend son corps à la terminaison du premier délai, puis chaque instant jusqu'à la terminaison du second délai. Si les deux délais se terminent au même instant, le corps n'est pas suspendu. La variante SUSPEND TOGGLE suspend son corps à la terminaison d'un délai. Comme pour SUSPEND, le délai est immédiatement repris à l'instant où il se termine. Ceci permet de redémarrer le corps dès l'instant suivant si le délai se termine à nouveau.

Par ailleurs, l'option EMITWHENSUSPENDED commune à ces deux variantes [5] permet d'émettre un signal à chaque instant où le corps de l'instruction est suspendu. Par exemple :

```
SUSPEND TOGGLE(NOW(a)) EMITWHENSUSPENDED b {
   SUSTAIN c
}
```

Le signal b est émis à chaque instant où l'instruction SUSTAIN est suspendue.

4.4.14 Exécution de code JavaScript durant la réaction

L'instruction ATOM permet l'exécution de code JavaScript durant la réaction.

atom−statement:
ATOM { *hiphopjs* }

5. Techniquement, EMITWHENSUSPENDED pourrait être utilisé avec SUSPEND. Cette restriction peut donc être simplement levée si des cas d'usage se présentent.

ATOM est instantanée au sens de Hiphop.js car elle démarre et se termine dans le même instant. Le corps de ATOM correspond au symbole non terminal *hiphopjs* compris entre les caractères { et }. Lorsque ATOM démarre, le code JavaScript composant son corps est exécuté. Lorsque le code JavaScript se termine, ATOM se termine.

Par ailleurs, le corps de ATOM est du code JavaScript étendu par les constructions Hiphop.js. Les valeurs de signaux sont donc accessibles depuis JavaScript au moyen des constructions vues en section 4.4.3. Par exemple :

```
ATOM {
   afficheTraduction(VAL(trad));
}
```

L'exécution de cette instruction ATOM exécute la fonction JavaScript afficheTraduction avec en argument la valeur du signal trad dans l'instant.

4.4.15 Définition de signaux locaux

L'instruction LOCAL permet la définition de signaux *locaux* de la même façon que les signaux globaux mais sans directionnalité. Un signal local est un signal accessible uniquement depuis le corps défini par LOCAL. Il n'est donc pas accessible depuis l'environnement ni par le code Hiphop.js défini en dehors du corps de LOCAL.

local-statement:
        LOCAL *local-signal-list* { *hh-statement* }

local-signal-list:
        local-signal
        local-signal-list, *local-signal*

local-signal:
        identifier $combine_{opt}$
        identifier(*hh-expression*) $combine_{opt}$

À l'instant où l'instruction LOCAL est démarrée, les signaux définis par *signal-list* sont créés et éventuellement initialisés puis le corps est démarré. À l'instant où la dernière instruction du corps se termine, LOCAL se termine et les signaux définis par *signal-list* sont détruits. Par exemple :

```
LOCAL l, m(5 + VAL(s)), n(6) COMBINE((a, b) => a * b)) {
   instructions
}
```

Trois signaux locaux sont créés : `l` sans valeur initiale, `m` avec la valeur initiale correspondant à l'expression `5 + VAL(s)` évaluée au démarrage de l'instruction, et `n` en tant que signal combiné avec la valeur initiale 6. Par ailleurs, la définition d'un signal local `s` dans la portée lexicale courante masque le signal `s` de la portée lexicale englobante :

```
LOCAL s(1) {
   LOCAL s(2) {
      ATOM {
         console.log(VAL(s))
      }
   }
   ATOM {
      console.log(VAL(s))
   }
}
```

L'exécution de ce code affiche donc sur la console « 2 » puis « 1 ».

4.4.16 Boucles

Les instructions `LOOP`, `WHILE`, `FOR`, `LOOPEACH`, et `EVERY` permettent la constructions de différents types de boucles.

```
loop-statement:
        simple-loop
        temporal-loop

simple-loop:
        LOOP { hh-statement }
        WHILE (hh-expression) { hh-statement }
        FOR (signal-list_opt; hh-expression_opt; hh-statement_opt) {
          hh-statement
        }

temporal-loop:
        LOOPEACH delay { hh-statement }
        EVERY delay { hh-statement }
```

Boucles simples

Le démarrage de l'instruction `LOOP` démarre immédiatement son corps. Lorsque son corps se termine, il est instantanément redémarré. Le corps de la boucle ne doit pas être

instantané, c'est-à-dire qu'il ne doit pas pouvoir se terminer à l'instant de son démarrage. Cette condition peut-être vérifiée statiquement [Tardieu, 2004] pour générer une erreur de compilation si elle n'est pas respectée [6]. Considérons l'exemple suivant :

```
LOOP {
    IF(NOW(v)) {
        PAUSE
    }
}
```

Ce programme est invalide car si le signal `v` est absent `PAUSE` n'est pas exécutée et l'instruction `IF` se termine instantanément. La boucle aurait alors à redémarrer immédiatement son corps, ce mécanisme devant se répéter infiniment.

La boucle `WHILE` est similaire à `LOOP`, mais ne démarre son corps que si l'expression passée en paramètre est vraie. Lorsque la dernière instruction de son corps se termine, l'expression du `WHILE` est réévaluée, et le corps est instantanément redémarré si elle est vraie.

Enfin, la boucle `FOR` est composée de trois parties optionnelles. Le démarrage de la boucle `FOR` crée les signaux locaux optionnellement définis dans sa première partie et démarre son corps si l'expression de sa seconde partie est vraie. Chaque instant où son corps se termine, l'instruction de sa troisième partie est exécutée et l'expression de la seconde partie est évaluée. Le corps est instantanément redémarré si elle est vraie. Par exemple :

```
FOR (l(0), m(4); VAL(l) != VAL(m); EMIT(PREVAL(l) + 1)) {
  ATOM {
    console.log("VAL(l) =", VAL(l));
  }
  PAUSE;
}
```

Deux signaux locaux `l` et `m` sont définis et respectivement initialisés à 0 et 4. Comme les valeurs sont différentes, le test d'inégalité initial s'évalue à `vrai`. Le corps est donc exécuté : la valeur de `l` est affichée sur la console et `PAUSE` démarre. À la réaction suivante, `PAUSE` se termine et l'instruction `EMIT` incrémente `l`. Le test est immédiatement réévalué. Cette itération continue jusqu'à que ce que test d'inégalité retourne `faux`, c'est-à-dire après 4 itérations.

6. L'implémentation actuelle Hiphop.js ne détecte pas les boucles instantanées.

Boucles temporelles

Une boucle temporelle est une boucle dont les itérations dépendent de la terminaison d'un délai. Si le délai se termine avant le corps de la boucle, une préemption forte est appliquée sur le corps, réinitialisant son état de contrôle. Si le corps se termine avant le délai, l'instruction attend la terminaison du délai. Dans les deux cas, le délai est redémarré et la boucle débute une nouvelle itération à l'instant où le délai se termine. Deux variantes de boucles temporelles existent.

- La première variante est la boucle `LOOPEACH`. Elle démarre son corps à l'instant où elle est démarrée. Elle ne se termine pas et le corps est redémarré quelque soit sont état à l'instant où le délai du `LOOPEACH` se termine. Par exemple :

  ```
  LOOPEACH(NOW(r)) {
     EMIT a;
     PAUSE;
     EMIT b
  }
  ```

  À l'instant où `LOOPEACH` est démarrée, le signal `a` est émis, `PAUSE` démarre et l'instant se termine. Si le signal `r` est absent à l'instant suivant, alors `PAUSE` se termine et le signal `b` est émis. En revanche, si le signal `r` présent à l'instant suivant, une préemption forte est alors appliquée sur le corps, `b` n'est donc pas émis. Une nouvelle itération débute, `a` est émis, `PAUSE` démarre et l'instant se termine.

- La seconde variante est la boucle `EVERY`. Contrairement à `LOOPEACH`, l'instruction `EVERY` n'exécute pas son corps lorsqu'elle est démarrée, mais attend pour cela la terminaison du délai une première fois. `EVERY` correspond donc à une séquence composée de `AWAIT` puis de `LOOPEACH`.

Réincarnation

L'emboîtement de boucles et de signaux locaux est à l'origine d'un phénomène appelé *réincarnation* [Berry, 2002]. Il s'agit de la création de plusieurs instances d'un même signal local dans la même réaction. Par exemple :

```
LOOP {
  LOCAL l {
    IF(NOW(l)) {
      ATOM {
        console.log("L est présent (1).");
      }
    }
    PAUSE;
    EMIT l;
    IF(NOW(l))  {
      ATOM {
        console.log("L est présent (2).");
      }
    }
  }
}
```

À la première réaction, le signal local `l` est créé, le premier test de présence est `faux` car `l` est absent par défaut et «L est présent (1)» n'est donc pas affiché. La pause démarre ensuite. À la réaction suivante la pause se termine, `l` est émis, le second test de présence de `l` est vrai et «L est présent (2)» est donc affiché. Dans le *même* instant `LOCAL` se termine, `l` est détruit et la boucle démarre une nouvelle itération. Une *nouvelle instance* de `l` est alors créée et le premier test de présence est `faux`. Il s'agit d'un nouveau signal qui, bien que le nom `l` soit le même, est indépendant du signal `l` créé lors de la précédente itération de `LOOP`.

4.4.17 Trappes

Les instructions `TRAP` et `EXIT` permettent de détourner le flot de contrôle d'un programme Hiphop.js.

trap-statement:
TRAP *identifier* { *hh-statement* }
EXIT *identifier*

Le démarrage de `TRAP` démarre immédiatement son corps. La trappe se termine lorsque le corps se termine ou si une instruction `EXIT` identifiant cette trappe et contenue dans le corps est exécutée. Lorsque `EXIT` est exécutée, une préemption faible est immédiatement appliquée à l'ensemble du corps. Par exemple :

```
TRAP GetOut {
  EMIT a;
  IF (NOW(aux)) {
    EXIT GetOut;
  }
  EMIT b;
}
EMIT c;
```

Au démarrage de l'instruction `TRAP`, le signal `a` est émis. Ensuite, si le signal `aux` est présent dans l'instant, `EXIT GetOut` est exécutée et a pour effet d'amener le flot de contrôle du programme directement en sortie du corps de la trappe. Ainsi, le signal `c` est émis, mais le signal `b` n'est pas émis. En revanche, si le signal `aux` est absent, alors les signaux `b` et `c` sont tous deux émis.

Il est possible d'implémenter l'instruction `WEAKABORT` par l'utilisation d'une trappe :

```
TRAP WeakAbort {
  FORK {
    hh-statement
    EXIT WeakAbort;
  } PAR {
    AWAIT delay;
    EXIT WeakAbort;
  }
}
```

Cet exemple montre aussi que lorsque `EXIT` est exécuté, la préemption faible de la trappe est propagée à l'ensemble des branches parallèles contenues dans le corps de la trappe.

Terminaison de trappes emboîtées

Lorsque des trappes sont emboîtées, la trappe extérieure est prioritaire. Par exemple :

```
TRAP GetOutOuter {
  EMIT a;
  TRAP GetOUTInner {
    IF (NOW(aux)) {
      FORK {
        EXIT GetOutOuter;
      } PAR {
        EXIT GetOutInner;
      }
    }
    EMIT b;
  }
}
EMIT c;
```

Dans cet exemple, si le signal `aux` est présent, les deux instructions `EXIT GetOutOuter` et `EXIT GetOutInner` sont démarrées en même temps. La trappe extérieure étant prioritaire, seule `EXIT GetOutOuter` est exécutée. Ainsi, le signal `b` n'est pas émis et le contrôle passe directement à `EMIT c`.

4.4.18 Contrôle d'exécution JavaScript asynchrone

Les instructions `EXEC` et `EXECEMIT` permettent l'exécution et le contrôle de code JavaScript asynchrone.

exec-statement:
EXEC *js-bridge exec-params*$_{opt}$
EXECEMIT *identifier js-bridge exec-params*$_{opt}$
promise-statement

js-bridge:
js-instruction
THIS
DONE
DONEREACT

exec-params:
ONSUSP *js-bridge*
ONRES *js-bridge*
ONKILL *js-bridge*

Le symbole non terminal *js-instruction* représente une instruction JavaScript étendue par *js-bridge*. Au démarrage de `EXEC` une instruction JavaScript déclenchant une opération asynchrone est exécutée. L'instruction ne se termine que lorsque l'opération asynchrone JavaScript est terminée. `EXEC` n'est pas instantanée : même si l'opération

JavaScript se termine instantanément de façon atomique, EXEC ne se terminera qu'à l'instant suivant. Par exemple :

```
EXEC setTimeout(DONE, 1000)
```

Le démarrage de EXEC déclenche un minuteur JavaScript d'une durée de une seconde. Une fois la seconde écoulée, la fonction passée au premier paramètre de setTimeout est appelée afin de signaler la terminaison de minuteur. Ici, le constructeur DONE crée dynamiquement une fonction JavaScript qui, lorsqu'elle est appelée, indique à Hiphop.js que l'opération asynchrone démarrée par EXEC est terminée. Ainsi, EXEC se terminera à la prochaine réaction. Le constructeur DONEREACT est similaire, mais la fonction qu'il construit dynamiquement déclenche en plus automatiquement de réaction lors de son appel : ceci permet de faire réagir immédiatement le programme Hiphop.js à la terminaison de l'opération.

La variante EXECEMIT permet d'émettre automatiquement un signal lors de la terminaison de l'action asynchrone. Par exemple, dans le contexte d'une application web exécutée par un navigateur :

```
function waitForKeypress(doneReactFn) {
   document.addEventListener("keydown", function(evt) {
      doneReactFn(evt.keyCode);
   });
}
```

Cette fonction asynchrone attend qu'une touche du clavier soit pressée. Elle est appelée dans Hiphop.js de la façon suivante :

```
EXECEMIT key waitForKeypress(DONEREACT);
```

Lorsque EXECEMIT est démarrée et que l'utilisateur appuie sur une touche, la fonction retournée par DONEREACT (et identifiée par doneReactFn dans JavaScript) est appelée avec la valeur de la touche passée en paramètre. Cela aura pour effet de terminer EXECEMIT, d'émettre le signal key avec la valeur de la touche passée en paramètre à DONEREACT, et de déclencher une réaction.

Suspension et préemption

Dans cette partie, EXEC identifie communément EXEC et sa variante EXECEMIT. Comme toute instruction Hiphop.js, EXEC peut être suspendue ou préemptée. Suspendre ou préempter une instance de EXEC signifie la suspension ou la préemption de l'action asynchrone que cette instruction déclenche. Les paramètres ONSUSP, ONRES et ONKILL permettent de signaler à l'environnement ces changements.

- L'instruction JavaScript passée à `ONSUSP` est appelée dans le cas où l'instance d'exécution de `EXEC` est suspendue dans l'instant.
- L'instruction JavaScript passée à `ONRES` est appelée dans le cas où l'instance d'exécution de `EXEC` était suspendue à l'instant précédent et ne l'est plus à l'instant courant.
- L'instruction JavaScript passée à `ONKILL` est appelée dans le cas ou l'instance d'exécution de `EXEC` est préemptée dans l'instant (qu'il s'agisse d'une préemption forte ou faible).

La construction `THIS` est utilisable dans l'ensemble des instructions JavaScript passées à `EXEC`. Il s'agit d'un objet JavaScript créé dynamiquement par Hiphop.js et propre à chaque instance d'exécution de `EXEC`. `THIS` définit deux propriétés :

- `THIS.killed` : cette valeur est un booléen valant vrai si l'instance du `EXEC` a été préemptée.
- `THIS.id` : cette valeur est un entier permettant d'identifier de façon unique l'instance du `EXEC`.

Par ailleurs, `THIS` peut être utilisé pour échanger des valeurs entre les différentes instructions passées à `EXEC`. Par exemple :

```
function start(self, doneReact) {
  self.foo = "bar";
  setTimeout(doneReact, 2000);
}
const hh = require("hiphop");
const m = new hh.ReactiveMachine(MODULE (IN stop) {
  ABORT(NOW(stop)) {
    EXEC start(THIS, DONEREACT) ONKILL console.log(THIS.foo);
  }
});
m.react();
m.inputAndReact("stop");
```

La fonction `start` appelée par `EXEC`, modifie `self` en ajoutant un attribut `foo` et la réaction se termine. Avant que l'attente de 2000 millisecondes ne se termine, une nouvelle réaction est déclenchée et, puisque le signal `stop` est présent, `ABORT` préempte `EXEC`. L'instruction `ONKILL` est appelée et « bar » est affiché sur la console. Ainsi, l'attribut `foo` est conservé entre les différents appels des instructions d'une même instance de `EXEC`.

Intégration avec les promesses

Une promesse est un objet JavaScript représentant la complétion ou l'erreur éventuelle d'un appel asynchrone [ECMA, 2015], le concept général est décrit à la section 2.3.1. Lorsque l'appel asynchrone se termine, la valeur ou l'erreur retournée est accessible depuis la promesse. Les promesses sont aujourd'hui massivement utilisées dans les programmes et bibliothèques JavaScript afin d'encapsuler des opérations asynchrones. Il est possible de manipuler les promesses JavaScript directement depuis Hiphop.js grâce à l'instruction `PROMISE`.

promise-statement:
PROMISE $identifier_{then}$, $identifier_{catch}$ *js-bridge* $exec\text{-}params_{opt}$

`PROMISE` abstrait `EXEC` en évitant d'expliciter depuis le code asynchrone JavaScript la connexion avec Hiphop.js, notamment à cause de l'appel aux fonctions `DONE` ou `DONEREACT` lors de la complétion du code asynchrone. L'instruction JavaScript démarrée par `PROMISE` doit retourner une promesse. Si la promesse est résolue, le signal $identifier_{then}$ est émis avec pour valeur la valeur renvoyée par la promesse. Si la promesse est rejetée, le signal $identifier_{catch}$ est émis avec la valeur renvoyée par la promesse. L'instruction se termine ensuite instantanément.

Par exemple, le code JavaScript suivant est une fonction qui interroge le service web MyMemory pour obtenir la traduction anglaise d'un mot français passé en paramètre. La valeur retournée est une promesse dont la résolution retourne la traduction anglaise :

```
function translateXhr(word) {
  const srv = "http://mymemory.translated.net/api/get";
  const xhr = new XMLHttpRequest();
  return new Promise((resolve, reject) => {
    xhr.onreadystatechange = () => {
      if (xhr.readyState == 4 && xhr.status == 200) {
        const resp = JSON.parse(xhr.responseText);
        resolve(resp.responseData.translatedText);
      }
    }
    xhr.onerror = reject;
    xhr.open("GET", `${srv}?langpair=fr|en&q=${word}`, true);
    xhr.send();
  });
}
```

Il s'agit d'une fonction standard utilisable depuis n'importe quel programme JavaScript mais aussi Hiphop.js, par exemple, dans le programme suivant :

```
MODULE (IN word, OUT trans, OUT err) {
  EVERY(NOW(word)) {
    PROMISE trans, err translateXhr(VAL(word));
  }
}
```

Ici, le service web MyMemory est appelé à chaque instant où le signal d'entrée `word` est présent. Comme `EXEC`, l'instruction `PROMISE` est bloquante tant que l'action asynchrone n'est pas terminée, c'est-à-dire tant que la promesse n'est pas résolue ou rejetée. Lorsque la traduction est reçue et que la promesse est résolue, le programme Hiphop.js réagit automatiquement : l'instruction `PROMISE` termine et le signal de sortie `trans` est émis avec pour valeur la traduction de la valeur de `word`. Dans le cas d'un échec, `PROMISE` termine et le signal de sortie `err` est émis avec pour valeur le paramètre passé à la fonction de rejet de la promesse. Ici, il s'agit d'un objet de type `XMMLHttpRequest`, représentant la requête web vers MyMemory.

Comme `EXEC`, `PROMISE` peut être suspendue ou préemptée. Ainsi, Hiphop.js garanti par construction que lorsque `PROMISE` se termine la valeur `trans` est la traduction de la valeur courante `word`. En effet, si `PROMISE` est active et que le programme Hiphop.js réagit avec le signal `word` présent, alors `PROMISE` est préemptée par `EVERY` avant d'être immédiatement redémarrée. Ceci a pour effet d'ignorer la résolution ou le rejet de la promesse correspondante à la traduction d'une ancienne valeur de `word`.

4.4.19 Réutilisation de modules

L'instruction `RUN` permet la réutilisation de modules Hiphop.js en les appelant depuis un autre module.

```
run-statement:
        RUN (hh-expression)
        RUN (hh-expression, signal-assoc-list)

signal-assoc-list:
        identifier = identifier
        signal-assoc-list, identifier = identifier
```

L'appel de sous-module permet la construction hiérarchique d'applications Hiphop.js. Cette construction forme un arbre de modules représentant le programme Hiphop.js global et dont la racine est le point d'entrée. La forme la plus simple est la suivante, où `Bar` est une variable JavaScript référençant un module :

```
RUN(Bar)
```

Lors de la compilation du programme, cette instruction est substituée par une instruction `LOCAL` dont le corps correspond au module `Bar`. Une *instance* de `Bar` est créée dans le module appelant. L'appel récursif d'un même module est interdit [7]. Par défaut, les signaux globaux du module instancié deviennent locaux au bloc et ne sont donc pas accessibles depuis le module appelant, sauf dans les cas suivants :

- Si le nom d'un signal global du module instancié est identique au nom d'un signal du module appelant accessible depuis l'instruction `RUN`, il s'agit alors du même signal. Toute émission de ce signal depuis le module appelant est visible dans le module instancié et vice-versa. Ce mécanisme est appelé *capture* dans la nomenclature d'Esterel v5 [Berry, 2000a].
- Si le nom d'un signal global du module instancié est explicitement associé à un signal du module appelant, alors le signal du module instancié est substitué par celui du module appelant. Les arguments de l'instruction `RUN` permettent d'associer les signaux. Par exemple `RUN (Bar, sBar=sFoo)` substitue le signal `sBar` du module instancié par le signal `sFoo` du module appelant. Autant d'associations que nécessaire peuvent être définies.

Dans tous les cas, les directionnalités `IN`, `OUT` et `INOUT` du module instancié sont ignorées puisque les signaux du module appelant substituent ceux du module instancié. L'exemple suivant illustre différentes situations possibles :

```
MODULE Bar {
   IN i, b1, b3;
   OUT o, b2, b4;
   instructions
}

MODULE Foo {
   IN i, j;
   OUT o, p;
   RUN(Bar, b1=j, b2=p)
}
```

Le signal `b1` de l'instance du module `Bar` est substitué par le signal `j` du module `Foo`. De même, `b2` est substitué par `p`. Le signal `i` est le même dans `Foo` et l'instance de `Bar` puisque le nom est identique dans les deux modules, tout comme pour le signal `o`. Enfin, les signaux `b3` et `b4` sont locaux à l'instance de `Bar`.

7. À noter que contraitement à Esterel v5 ou Hiphop.js, Esterel v7 permet une récursion statique de modules [Berry, 2008].

Chapitre 5

Architecture d'intégration Hiphop.js/JavaScript

Sommaire

L'intégration de Hiphop.js dans les technologies du web est une contribution importante de ces travaux. En effet, Hiphop.js n'a pas pour vocation de remplacer la façon de programmer les applications web, mais de la rendre plus simple et vérifiable en se focalisant sur la gestion d'évènements asynchrones. Plusieurs aspects de l'intégration de Hiphop.js dans JavaScript sont donc abordés dans ce chapitre. D'abord, une première partie traite l'intégration syntaxique de Hiphop.js dans le code source de JavaScript ainsi que de l'exécution des programmes Hiphop.js au sein d'un programme JavaScript, ces mécanismes permettant une communication naturelle entre les deux langages. Une seconde partie présente ensuite la connexion temporelle entre JavaScript et Hiphop.js. La troisième partie aborde la construction dynamique de programmes, une particularité importante permettant l'adaptation de Hiphop.js au web. Enfin, le cas du traitement des actions asynchrones par Hiphop.js est étudié.

Ce chapitre tient pour acquises par le lecteur les notions introduites au chapitre 3, c'est-à-dire les notions de programmation réactive, de programmation asynchrone et le positionnement de Hiphop.js vis-à-vis de JavaScript.

5.1 Du code à l'exécution de programme Hiphop.js

Cette section traite d'abord de l'intégration de Hiphop.js dans JavaScript du point de vue de la syntaxe et du code source. Ensuite, l'intégration logique en terme d'API est présentée. Ces deux mécanismes sont des fondements sur lesquels d'autres mécanismes d'intégration plus évolués sont construits. Ils sont traités par la suite.

5.1.1 Code source commun

Un programme Hiphop.js est écrit dans un fichier JavaScript qui doit impérativement réserver la variable `hh` et l'initialiser de la façon suivante :

```
const hh = require("hiphop");
```

Les mots-clés de Hiphop.js sont en majuscule afin de les distinguer des mots clés JavaScript. Par ailleurs, les constructions et plus généralement les programmes Hiphop.js sont des valeurs de première classe du point de vue de JavaScript. Une instruction ou un module Hiphop.js peut donc être affecté à une variable JavaScript ou passé en paramètre à une fonction.

Un mécanisme de transformation statique de code source, décrit au chapitre 6, permet l'exécution d'un programme JavaScript contenant des programmes Hiphop.js dans n'importe quel navigateur web, ainsi que sur le serveur.

5.1.2 Compilation et exécution : les *machines réactives*

Afin d'avoir une API JavaScript permettant par exemple de signaler à Hiphop.js la survenue d'un évènement JavaScript, un programme Hiphop.js est compilé en une *machine réactive*. Il s'agit d'un objet JavaScript représentant une instance exécutable d'un programme Hiphop.js. Par exemple :

```
const hh = require("hiphop");
var fooMachine = new hh.ReactiveMachine(fooModule);
```

Dans ce code, `fooModule` correspond à un programme Hiphop.js et `ReactiveMachine` est un constructeur JavaScript qui prend un module Hiphop.js en paramètre, qui le compile, et qui retourne une machine réactive ici référencée par la variable `fooMachine`.

La machine réactive implémente une méthode `react` permettant de déclencher une réaction, typiquement lorsqu'un évènement JavaScript se produit. Cette méthode est synchrone, ce qui, grâce à la sémantique de JavaScript, garantit l'atomicité d'une réaction Hiphop.js. D'autres fonctionnalités de l'API de la machine réactive sont décrites dans la suite de ce chapitre.

Enfin, le constructeur `ReactiveMachine` prend un second paramètre optionnel à des fins de débogage. Il s'agit d'un objet JavaScript pouvant comprendre des attributs décrits dans les chapitres 6 et 7.

5.2 Évènements et signaux

Cette section présente la connexion temporelle entre JavaScript et Hiphop.js. Il s'agit de la façon d'informer un programme Hiphop.js depuis JavaScript de la survenue d'un évènement. Cette information permet au programme Hiphop.js de calculer son nouvel état et d'en informer à son tour le programme JavaScript par l'émission de signaux de sortie. Les machines réactives disposent d'une API permettant ces différents échanges. Cette API est maintenant décrite et illustrée par des exemples qui complètent les versions Hiphop.js du programme de minuteur étudié au chapitre 3. L'API est composée des méthodes suivantes :

- `ReactiveMachine.prototype.input`
- `ReactiveMachine.prototype.inputAndReact`
- `ReactiveMachine.prototype.addEventListener`

Informer un programme Hiphop.js d'un évènement

La méthode `input` permet l'émission d'un signal avant une réaction. Elle prend en premier paramètre une chaîne de caractères correspondant au nom du signal d'entrée à émettre et une valeur quelconque en second paramètre si l'émission comprend une valeur. La méthode `inputAndReact` est identique et déclenche une réaction immédiatement après l'émission du signal.

Afin d'illustrer l'utilisation de ces méthodes, reprenons l'implémentation de l'IHM du programme de minuteur. Un clic sur le Reset appelle la fonction `resetE` grâce à un gestionnaire d'évènements JavaScript. Dans les versions Hiphop.js du minuteur, l'implémentation de cette fonction est la suivante, où `suspendableTimerMachine` est une variable référençant une machine réactive correspondant à `SuspendableTimer` :

```
function resetE() {
   suspendableTimerMachine.inputAndReact("reset");
}
```

Les autres fonctions appelées par des gestionnaires d'évènements JavaScript lors du déplacement du curseur de durée du minuteur ou le clic sur le bouton Suspend sont implémentées de façon analogue.

Informer JavaScript de l'émission d'un signal de sortie

La méthode `addEventListener` permet de créer un gestionnaire d'évènements associé à l'émission d'un signal de sortie. C'est une version simplifiée de la fonction éponyme des éléments graphiques HTML :

- Elle prend en premier paramètre une chaîne de caractères correspondante au nom du signal de sortie auquel doit être associé le gestionnaire d'évènements. Elle prend aussi un second paramètre qui est une fonction f appelée à l'issue de chaque réaction où le signal de sortie est émis.
- Plusieurs gestionnaires d'évènements peuvent être associés à un même signal de sortie. Dans ce cas, la fonction f de chaque gestionnaire d'évènements est appelée séquentiellement du point de vue de JavaScript, dans l'ordre dans lequel le gestionnaire d'évènements a été associé au signal (via l'appel à `addEventListener`).
- La fonction f est définie avec un argument : un objet qui est créé dynamiquement par Hiphop.js lors de l'appel à f. Il comprend les champs `signalName` et `signalValue` correspondant respectivement au nom du signal de sortie émis et

à sa valeur (si la valeur n'est pas définie, ce champ vaut `undefined`). Enfin, un troisième champ `stopPropagation` est une fonction dont l'appel inhibe les gestionnaires d'évènements suivants associés au même signal.

Afin d'illustrer l'utilisation de `addEventListener`, reprenons maintenant l'exemple du minuteur. L'émission du signal `elapsed` doit mettre à jour la jauge et le champ numérique de l'IHM. L'implémentation est la suivante :

```
suspendableTimerMachine.addEventListener("elapsed", evt => {
   let e = evt.signalValue;
   document.getElementById("meter").value = e;
   document.getElementById("elapsed").innerHTML = e;
});
```

5.3 Dynamicité des programmes Hiphop.js

Historiquement, les langages réactifs synchrones sont conçus pour être utilisés dans des environnement statiques où tous les acteurs interagissant avec le programme sont connus au moment de l'écriture du programme. À l'opposé, le web est un environnement intrinsèquement dynamique où le nombre d'acteurs interagissant avec le programme peut varier à tout moment. Le mélange de codes JavaScript et Hiphop.js dans un même code source ouvre la voie à la construction et la modification dynamique de programmes.

5.3.1 Construction dynamique de programme

Prenons par exemple le programme de minuteur du chapitre 3. Il est intéressant d'offrir la possibilité, suite à un clic utilisateur, d'ajouter ou enlever dynamiquement des minuteurs. Pour cette raison, il est possible de construire dynamiquement, c'est-à-dire pendant l'exécution du programme JavaScript, les programmes Hiphop.js. Afin d'illustrer ce concept, considérons le code suivant :

```
function makePause(n) {
   if (n <= 0) {
      return NOTHING;
   } else if (n == 1) {
      return PAUSE;
   } else {
      return SEQUENCE {
         PAUSE;
         ${makePause(n - 1)};
      } } }
```

Cette fonction JavaScript retourne un fragment de programme Hiphop.js composé de n pauses. La forme syntaxique `${`*js-expression*`}` déclenche l'évaluation de l'expression JavaScript *js-expression* lors de la construction du programme Hiphop.js. Cette expression doit retourner une valeur Hiphop.js qui est insérée à la place de `${`*js-expression*`}`. Ainsi, le programme Hiphop.js suivant utilise cette forme pour y intégrer le nombre de pauses voulues :

```
var fooModule = MODULE Foo {
   OUT a, b;
   LOOP {
      EMIT a;
      ${makePause(4)};
      EMIT b;
   }
}
```

Ici, le programme Hiphop.js généré par ce code comprendra 4 pauses entre l'émission du signal `a` et celle du signal `b`. Par ailleurs, puisque toute construction Hiphop.js est une valeur JavaScript, il est possible d'affecter un module à une variable. Dans ce code, la variable `fooModule` référence le module `Foo`.

5.3.2 Plasticité du parallèle

Il est possible de créer ou de supprimer des branches d'un parallèle entre deux instants. Pour cela, l'instruction `FORK` est appelée avec un identifiant. Par exemple :

```
FORK parallelFoo {
   HALT
}
```

L'instruction `FORK` identifiée par `parallelFoo` contient une unique branche ne se terminant jamais. Soit `m` la machine réactive d'un programme Hiphop.js comprenant `parallelFoo`, et soit le code JavaScript suivant exécuté depuis l'environnement entre deux instants :

```
var lp = LOOP {
   ATOM {
      console.log("nouvelle branche");
   }
   PAUSE
}
m.getElementById("parallelFoo").appendChild(lp);
```

Ce code ajoute dans une nouvelle branche de `parallelFoo` le code référencé

par `lp`. La méthode `getElementById` permet d'obtenir un objet représentant `parallelFoo`. Cet objet implémente la méthode `appendChild` dont l'appel ajoute au parallèle une instruction Hiphop.js passée en argument. À l'instant suivant, la nouvelle branche est démarrée, et « nouvelle branche » s'affiche. Le parallèle correspond maintenant au code suivant :

```
FORK parallelFoo {
   HALT
} PAR {
   LOOP {
      ATOM {
         console.log("nouvelle branche");
      }
      PAUSE
   }
}
```

La branche ajoutée par `appendChild` est démarrée à l'instant suivant si le parallèle était démarré et ne s'est pas terminé à l'instant précédent. Il est aussi possible de supprimer une branche précédemment ajoutée. Soit `lp` introduite dans l'exemple précédent, et soit le code JavaScript suivant exécuté dans l'environnement entre deux instants :

```
m.getElementById("parallelFoo").removeChild(lp);}
```

À l'instant suivant, la branche contenant la boucle n'existera plus. Il ne restera que la branche contenant l'instruction `HALT`.

Techniquement, dans l'implémentation actuelle de Hiphop.js, l'ajout ou le retrait d'une branche modifie l'AST du programme (gardé en mémoire par la machine réactive) et recompile l'ensemble du programme. À terme et dans le cas de l'ajout d'une branche, il serait intéressant de ne compiler que la branche ajoutée et de connecter le circuit généré au circuit du programme existant, donc sans recompiler l'ensemble du programme.

De façon plus concrète, supposons une application de visioconférence comprenant un nombre variable de participants. Il est possible de gérer les interactions des participants avec Hiphop.js grâce à la dynamicité du parallèle. La connexion d'un participant ajoute une nouvelle branche parallèle qui gère interactions du participant avec le programme. La déconnexion du même participant supprime la branche lui correspondant. Ainsi, il est possible de gérer l'ajout et le retrait des utilisateurs sans reconstruire un nouveau programme et perdre l'état du programme précédent à chaque connexion ou déconnexion d'un participant.

5.4 Actions asynchrones

L'action asynchrone est caractérisée par un début, l'exécution d'une fonction se terminant immédiatement et par une fin, un évènement signalant la terminaison de l'action avec succès ou erreur et valeur éventuelle. Ce mécanisme est omniprésent dans le web : l'attente de la réponse d'une requête réseau, l'attente de la terminaison d'un minuteur ou encore l'attente de la réponse d'un accès au système de fichier sont des exemples d'actions asynchrones.

Démarrage et terminaison

Dans le cas général en JavaScript, la fonction permettant le démarrage d'une action asynchrone prend une fonction de rappel en paramètre. Cette fonction de rappel est associée à un gestionnaire d'évènements qui est déclenché lors de la terminaison de l'action. Puisqu'il s'agit d'un cas d'utilisation courant, Hiphop.js dispose de l'instruction `EXEC` dédiée à la manipulation temporelle d'actions asynchrones qui est définie formellement à la section 4.4.18. Cette instruction démarre une action asynchrone et ne se termine qu'à une réaction suivante, lorsque l'action asynchrone est terminée. Grâce à `EXEC`, une action asynchrone *est* synchrone du point de vue de Hiphop.js dans la mesure où elle bloque le fil d'exécution du programme Hiphop.js jusqu'à sa terminaison comme l'instruction `AWAIT`.

Par exemple, le code suivant est une version simplifiée de `TimeoutMod` utilisé comme délai pour l'exemple de minuteur du chapitre 3 :

```
function TimeoutMod(nms) {
   return MODULE() {
      EXEC setTimeout(DONEREACT, nms);
   }
}
```

`TimeoutMod` est une fonction dont l'appel crée et retourne un module paramétré par le nombre de millisecondes à attendre entre le moment où `EXEC` démarre et celui où il se termine. Lorsque `EXEC` démarre, la fonction prédéfinie `setTimeout` est appelée et se termine immédiatement. Le premier paramètre passé à `setTimeout` est une fonction appelée par JavaScript lorsque les `nms` millisecondes sont écoulées. Ici, `DONEREACT` s'évalue lors du démarrage de `EXEC` en une fonction créée dynamiquement par Hiphop.js. Son appel informe le programme Hiphop.js de la terminaison de l'action asynchrone, et déclenche une réaction.

Préemption et suspension

Les notions de préemption et de suspension ont été brièvement présentées dans l'exemple de minuteur du chapitre 3. Comme toute instruction du langage, `EXEC` peut également être préemptée ou suspendue, ce qui peut impliquer que l'action asynchrone soit interrompue. Si c'est nécessaire, `EXEC` dispose des paramètres optionnels permettant d'en informer JavaScript. Par exemple, le code suivant est une version complétée de `TimeoutMod` :

```
function TimeoutMod(nms) {
   let id;
   return MODULE() {
      EXEC id = setTimeout(DONEREACT, nms)
         ONKILL clearTimeout(id)
         ONSUSP clearTimeout(id)
         ONRES id = setTimeout(DONEREACT, nms);
   }
}
```

Dans le minuteur du chapitre 3, si `TimeoutMod` est préemptée avant d'arriver à complétion, alors la fonction prédéfinie `clearTimeout`, appelée par le paramètre `ONKILL`, désactive le délai interne à JavaScript. C'est ce qui se produit lorsque le signal `reset` est présent : la préemption englobante de `LOOPEACH` s'applique. Le phénomène analogue se produit lorsque l'utilisateur met le minuteur en pause : la suspension englobante de `SUSPEND` s'applique, le paramètre `ONSUSP` appelle `clearTimeout` et le délai interne à JavaScript est désactivé. Quand l'utilisateur réactive le minuteur, le délai interne à JavaScript est réactivé par l'appel à `setTimeout` défini par le paramètre `ONRES`. À noter que JavaScript ne permet pas de suspendre son minuteur interne (permettant de faire réagir le programme Hiphop.js à chaque centième de seconde). C'est pour cette raison que le minuteur interne à JavaScript est simplement désactivé lors de la suspension puis réactivé lorsque la suspension se termine. Ainsi, l'état du minuteur interne de JavaScript est perdu, mais l'état du minuteur Hiphop.js est en revanche conservé puisqu'il est enregistré dans un signal global du programme.

Par ailleurs, l'appel à `DONE` ou `DONEREACT` n'a aucun effet si l'instruction `EXEC` est préemptée ou suspendue. Dans le cas général, `ONKILL` est utile pour libérer les ressources prises par l'action asynchrone.

Enfin, `DONE` et `DONEREACT` acceptent un paramètre optionnel. Il s'agit de la valeur, si elle existe, retournée par la terminaison de l'action asynchrone. Dans ce cas de figure, les variantes `EXECEMIT` et `PROMISE` présentées à la section 4.4.18 permettent l'émission de cette valeur dans un signal.

5.5 Interfaces graphiques réactives

La synchronisation entre l'état d'un programme et son IHM est une part importante du développement d'une application web. Par exemple, dans l'implémentation JavaScript de l'application de minuteur toutes les fonctions du programme à l'exception de `startIntervalIfNeeded` mettent à jour l'IHM. La version Hiphop.js isole le code orchestrant le programme de celui mettant à jour l'IHM en le déportant dans des gestionnaire d'évènements, mais la mise à jour de chaque élément de l'IHM doit tout de même être explicitement implémentée dans chaque gestionnaire d'évènements. Cela pose deux problèmes. D'abord, il faut que chaque élément de l'IHM soit nommé de façon unique afin de pouvoir y accéder depuis le code JavaScript. Ensuite, la modification de l'IHM (ajout ou suppression d'un élément) force aussi la modification du code la mettant à jour. L'émergence de bibliothèques comme Facebook React [Facebook, 2013] illustrent globalement le besoin d'abstraire cette synchronisation en rendant implicites les étapes suivantes :

- Nommer explicitement chaque élément graphique ;
- Mettre explicitement à jour la valeur d'un élément graphique, par exemple dans un gestionnaire d'évènements lorsqu'un évènement modifie l'état du programme.

La suite de cette section détaille comment Hiphop.js abstrait cette synchronisation explicite du statut et de la valeur d'un signal de sortie avec l'IHM.

5.5.1 Proxy réactif Hop.js

Le langage JavaScript définit l'objet *proxy* [Van Cutsem et Miller, 2010, ECMA, 2015]. Il s'agit d'un objet P encapsulant un autre objet C. Les opérations fondamentales comme la lecture, la modification, l'invocation de fonction, etc. sur P sont implicitement appliquées à C et peuvent être personnalisées. Une analogie possible est celle de la programmation par aspects [Kiczales *et al.*, 1997].

L'environnement JavaScript Hop.js introduit le concept de *proxy réactif*. Il s'agit d'un proxy JavaScript spécialisé pour être lu par les éléments d'IHM d'une page web. La modification de la valeur d'un proxy réactif entraîne implicitement la mise à jour de l'ensemble des éléments utilisant le proxy. Par exemple :

```
<script>var meterP = hop.reactProxy({v: 5})</script>
<button onclick=~{meterP.v++}>+</button>
<meter min="0" value=~{meterP.v} max="10"/>
```

Un proxy réactif `meterP` est défini. Il est lu par une jauge initialement remplie à moitié : l'expression `~{meterP.v}` est évaluée au chargement de la page et retourne la valeur initiale de `meterP.v`. Lorsque l'utilisateur clique sur le bouton « + », `meterP.v` est incrémentée et la jauge se met à jour automatiquement : l'expression `~{meterP.v}` est automatiquement réévaluée et retourne la nouvelle valeur de `meterP.v`. Il n'est donc pas nécessaire de nommer la jauge, ni de mettre explicitement à jour la valeur de son attribut `value` lors du clic sur le bouton « + », ces opérations étant abstraites par le proxy réactif. L'utilisation d'un même proxy réactif dans plusieurs éléments est directe :

```
<div>
  <span>La valeur de  meterP.v est :</span>
  <span><react>~{meterP.v}</react></span>
</div>
```

Ici, un clic sur le bouton « + » remplit la jauge et affiche la nouvelle valeur de `meterP.v`. Il n'y a donc aucune modification à faire dans le code JavaScript du programme. Deux extensions syntaxiques de Hop.js sont à noter. D'abord, l'utilisation de la forme `~{`*expr*`}` définit une expression dont la syntaxe est vérifiée à la compilation du programme mais dont l'évaluation est reportée à plus tard, lorsque le programme est reçu et exécuté sur le client. Ici, l'expression sera évaluée après chaque modification de la valeur de `meterP.v` (après un clic sur le bouton « + »). Ensuite, la balise `<react>` permet à Hop.js d'identifier à quelle position du DOM doit être injectée la valeur du proxy réactif.

5.5.2 Signal de sortie utilisé comme *proxy*

Les proxies réactifs de Hop.js sont utilisés par Hiphop.js afin de permettre l'injection de l'état (statut ou valeur) d'un signal de sortie dans l'IHM. À chaque réaction où un signal de sortie *s* est modifié, les éléments de l'IHM lisant l'état de *s* sont automatiquement mis à jour avec le nouvel état. Soit le programme Hiphop.js suivant :

```
const hh = require("hiphop");
const prg = MODULE (INOUT a, INOUT b, INOUT r, OUT o(0)) {
  LOOPEACH(NOW(r)) {
    FORK {
      AWAIT(NOW(a));
    } PAR {
      AWAIT(NOW(b));
    }
    EMIT o(PREVAL(o) + 1);
  }
}
```

```
const m = new hh.ReactiveMachine(prg);
m.react();
```

Il s'agit de l'implémentation Hiphop.js du programme « ABRO » utilisée pour introduire Esterel [Berry, 2000a]. Contrairement à la version originale Esterel, `o` est ici un signal avec valeur afin de compter le nombre de fois qu'il est émis. Soit le code HTML de l'IHM qui interagit avec ce programme :

```
<button onclick=~{m.inputAndReact("a")}>A</button>
<button onclick=~{m.inputAndReact("b")}>B</button>
<button onclick=~{m.inputAndReact("r")}>R</button>
<button onclick=~{m.react()}>-</button>


<div>
  <span>o emissions number:</span>
  <span><react>~{m.value.o}</react></span>
</div>
```

Initialement, « o emissions number : » affiche 0. Chaque fois que l'utilisateur clique sur les boutons « A » puis « B » (ou l'inverse), le signal `o` est émis en incrémentant sa valeur et l'IHM est mise à jour avec la nouvelle valeur de `o`. Techniquement, l'expression `m.value.o` retourne un proxy réactif que Hiphop.js met automatiquement à jour avec la nouvelle valeur de `o` lorsqu'il est émis. La réaction initiale suivant immédiatement la création de machine réactive permet d'initialiser la valeur du proxy lié à `o` avec sa valeur initiale, ici 0. Autrement « o emissions number : » n'afficherait initialement aucune valeur. L'IHM est maintenant complétée avec un message qui indique si le signal `r` a été émis lors de la dernière réaction afin d'informer l'utilisateur de la réinitialisation de l'attente de `a` et `b` :

```
<div>
  <span>Signal r present?</span>
  <span><react>~{m.present.r ? "yes" : "no"}</react></span>
</div>
```

Initialement, « Signal r present ? » affiche « no ». Lorsque l'utilisateur clique sur le bouton « R », la machine réagit, et « no » devient « yes ». Lorsque l'utilisateur clique ensuite sur un autre bouton que « r », « yes » redevient « no ». L'expression `m.present.r` est un proxy réactif dont la valeur est un booléen indiquant si le signal `r` était présent ou non à l'instant précédent.

Grâce aux proxies réactifs et à l'intégration dans Hiphop.js, il n'est donc pas nécessaire de nommer explicitement les éléments de l'IHM dont la valeur dépend de signaux de sortie, et il n'est pas non plus nécessaire d'implémenter des gestionnaires d'évènements pour les mettre à jour. Il s'agit d'une solution alternative à celles présentées au chapitre 2

pour abstraire et simplifier la mise à jour du DOM en réaction à des changements d'états dans le programme suite à des évènements asynchrones.

5.5.3 Diffusion d'une valeur associée à un signal d'entrée

Reprenons maintenant l'IHM de l'application de minuteur. Il est possible de la réécrire en utilisant les proxies réactifs. Par exemple, le code suivant est le code HTML implémentant la jauge de progression avec les proxies réactifs :

```
<div>
  <span>Elapsed time:</span>
  <meter value=~{m.value.elapsed} max=~{m.value.duration}/>
</div>
```

À chaque réaction où le temps du minuteur progresse, le signal de sortie `elapsed` est émis et l'attribut `value` de la jauge est automatiquement mis à jour. La jauge se remplit alors un peu plus (ou se vide, dans le cas de la réinitialisation du minuteur). Par ailleurs, lorsque l'utilisateur change la durée du minuteur, l'attribut `max` de la jauge est mis à jour afin d'ajuster son échelle. Mais il y a ici un problème : la durée du minuteur correspond à la valeur du signal `duration`, qui est un signal d'entrée puisqu'il s'agit d'une action de l'utilisateur et non pas du résultat d'une réaction. L'expression `m.value.duration` n'est donc pas valide car l'environnement JavaScript n'a pas un accès de lecture à un signal d'entrée.

Fondamentalement, le problème est le suivant : l'environnement JavaScript (ici, l'IHM) doit être mis à jour suite à un évènement extérieur au programme Hiphop.js (ici, la modification du temps du minuteur par l'utilisateur) et par une valeur ne dépendant pas du résultat d'une réaction. Dans l'exemple du minuteur, plusieurs solutions sont possibles. La première consiste à modifier manuellement la valeur de l'attribut `max` de la jauge dans le gestionnaire d'évènements du curseur paramétrant la durée du minuteur :

```
function setD(duration) {
  $.getElementById("meter").max = duration;
  m.inputAndReact("duration", duration);
}
```

Cette solution n'est pas idéale : d'abord, la jauge est maintenant modifiée par deux sources, le programme Hiphop.js et le gestionnaire d'évènements du curseur de durée. Cela rend l'analyse de l'état de la jauge complexe et augmente le risque d'une incohérence. Ensuite, cela implique de nommer explicitement la jauge. Une meilleure solution consiste à créer un signal de sortie auxiliaire appelé `durationOut`. Ce signal est émis continuellement à chaque réaction avec comme valeur celle du signal `duration` :

```
MODULE SuspendableTimerDurationOut (
       IN duration(0), IN reset, IN suspend,
       OUT elapsed, OUT suspendColor, OUT durationOut)  {
  FORK {
    SUSTAIN durationOut(VAL(duration));
  } PAR {
    RUN (SuspendableTimer);
  }
}
```

Il suffit ensuite de changer l'attribut `max` de la jauge :

```
<meter value=~{m.value.elapsed} max=~{m.value.durationOut}/>
```

Ainsi, la jauge ajustera automatiquement son échelle à chaque modification de la durée du minuteur.

Cette solution préserve une vision claire de l'origine et de la cible des évènements et évite de nommer la jauge. Cependant, cette solution a le défaut d'alourdir la définition de l'interface du programme Hiphop.js, notamment si de nombreux éléments graphiques posent le même problème que la jauge : cela va multiplier des signaux de sorties auxiliaires dont le seul but est de rediriger l'émission d'un signal d'entrée dans un signal de sortie.

Les signaux d'entrées-sorties

Une solution alternative proposée par Hiphop.js est l'utilisation d'un signal d'entrée-sortie. Il s'agit d'un signal global lisible à la fois en lecture et en écriture par l'environnement. L'interface du module `SuspendableTimer` avec le signal `duration` en entrée-sortie est :

```
MODULE SuspensableTimer(INOUT duration(0),
                        IN reset, IN suspend,
                        OUT elapsed, OUT suspendColor)
```

Ainsi, le gestionnaire d'évènements du curseur de durée ne fait qu'émettre le signal `duration` en déclenchant une réaction, et l'échelle de la jauge est automatiquement mise à jour par une unique source, le programme Hiphop.js, que ce soit lors de l'émission de `elapsed` ou de `duration`.

Un signal d'entrée-sortie est donc utile lorsque qu'un évènement extérieur au programme Hiphop.js entraîne la mise à jour de l'environnement sans dépendre d'un état calculé par Hiphop.js car :

- Il permet de garder compacts le code et l'interface d'un module ;
- Il évite la modification de l'environnement par des gestionnaires d'évènements autres que ceux des signaux de sorties.

Cependant, un signal d'entrée-sortie est à utiliser de façon parcimonieuse. En effet, l'utilisation massive de ce type de signal mélange l'origine et la destination des évènements, ce qui rend plus complexes la compréhension et la vérification d'un programme.

Chapitre 6

Technique de compilation et modèle d'exécution

Sommaire

Le but de ce chapitre est de présenter les différents éléments techniques permettant au lecteur intéressé par l'implémentation de Hiphop.js de faire évoluer le langage. Une première section présente une syntaxe alternative de Hiphop.js appelée *syntaxe abstraite*. Elle est utilisée pendant la compilation de Hiphop.js comme un langage intermédiaire et peut également être utilisée à des fins prospectives pour faire évoluer Hiphop.js. La chaîne de compilation de Hiphop.js, entièrement écrite en JavaScript, est ensuite détaillée. La construction de l'*arbre de syntaxe abstraite* (AST) d'un programme Hiphop.js est abordée, puis le modèle de compilation d'Esterel reposant sur le principe de circuits logiques, adapté et réimplémenté dans le compilateur de Hiphop.js, est présenté. Enfin, les optimisations utilisées dans Hiphop.js afin de générer des circuits plus compacts sont expliquées et le gain qu'elles apportent est mis en évidence à travers plusieurs programmes. L'architecture globale de l'implémentation du compilateur est présentée en annexe C. Il s'agit d'une référence rapide récapitulant le rôle de chaque fichier dans la chaîne de compilation et d'exécution de Hiphop.js.

La figure suivante illustre la chaîne de compilation, et montre à quelle section se réfère chaque étape :

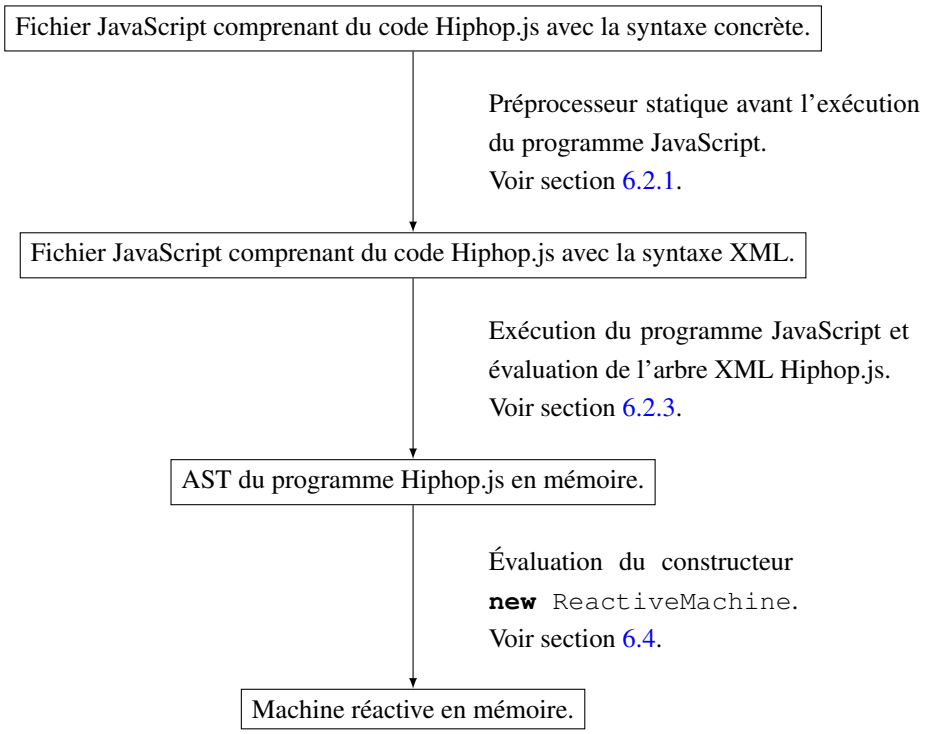

FIGURE 6.1 – Schéma de la compilation d'un programme Hiphop.js.

6.1 La syntaxe abstraite

Comme avec la syntaxe concrète, les programmes utilisant la syntaxe abstraite partagent le même code source que le programme JavaScript. La syntaxe abstraite étant un dialecte de XML, la distinction avec une instruction JavaScript est directe : les instructions Hiphop.js sont toujours bornées par les caractères « < » et « > ». L'analyse lexicale, syntaxique et la transformation de structure XML est une fonctionnalité offerte par Hop.js et décrite en section 6.2.2. Par ailleurs, et comme avec la syntaxe concrète, chaque construction du langage est une valeur JavaScript, elle peut donc être passée en argument de fonction, affectée à une variable, une expression, etc.

Par exemple, le programme introductif « Hello world » équivalent à celui du chapitre 4 écrit avec la syntaxe abstraite est le suivant :

```
const hh = require("hiphop");
const hello = <hh.module
    sayHelloAgain=${{ direction: hh.IN }}
    cnt=${{ direction: hh.OUT, initValue: 0 }}>
  <hh.loopeach sayHelloAgain>
    <hh.emit cnt apply=${function() {
      return this.preValue.cnt + 1;
    }}/>
    <hh.atom apply=${function() {
      console.log("Hello world from Hiphop.js!", this.value.cnt);
    }}/>
  </hh.loopeach>
</hh.module>;
const m = new hh.ReactiveMachine(hello);
m.addEventListener("cnt", function(evt) {
  console.log("Hello world from JavaScript!", evt.signalValue);
});
m.react();
m.inputAndReact("sayHelloAgain");
```

La syntaxe abstraite dispose d'un sous ensemble des instructions de la syntaxe concrète. Cette partie présente seulement des aspects syntaxiques et techniques spécifiques à l'utilisation de la syntaxe abstraite. La définition complète de cette syntaxe est présentée en annexe B.

Échappement XML

Lorsque Hop.js analyse un arbre XML, la valeur passée à chaque attribut d'un nœud XML est automatiquement convertie en chaîne de caractères. Cela pose un problème car la syntaxe abstraite de Hiphop.js nécessite de pouvoir passer des valeurs fonctionnelles

à certains attributs XML, voir section 6.1.2. Ce problème est résolu par la construction `${`*js-expression*`}` de Hop.js. Lorsque le nœud XML est évalué, *js-expression* est évaluée puis passée sans conversion en chaîne de caractères à l'attribut du nœud. Par exemple :

```
<foo attrInteger=${42}
     attrFunc=${function(p) {return p}}
     attrIntToStr=43 />
```

Lors de l'évaluation du nœud XML `foo` par Hop.js, le type de l'attribut `attrInteger` est un nombre et celui de `attrFunc` est une fonction. En revanche, le type de `attrIntToStr` est une chaîne de caractères, la valeur 43 étant implicitement convertie en chaîne de caractères.

6.1.1 Module et signaux

Les modules sont définis de la façon suivante :

hh-module:
`<hh.module` $name_{opt}$ $signal\text{-}list_{opt}$`>` *hh-statement* `</hh.module>`

name:
`__hh_debug_name__="`*identifier*`"`

signal-list:
signal $signal\text{-}list_{opt}$

signal:
identifier
identifier`=${`*js-expression*`}`

L'attribut `__hh_debug_name__` permet de nommer le module [1].

Les signaux globaux sont définis par une suite d'attributs. Par défaut, un signal est défini avec une directionnalité d'entrée-sortie et n'a pas de valeur initiale. La spécification de la directionnalité, de la valeur initiale et de la fonction de combinaison est possible en associant un objet JavaScript à l'attribut définissant le signal. Les propriétés lues sont les suivantes :

- `direction` : directionnalité du signal. La valeur peut être `hh.IN` (signal d'entrée), `hh.OUT` (signal de sortie) ou `hh.INOUT` (signal d'entrée-sortie).

1. Les blanc soulignés rappellent conventionnellement qu'il s'agit d'un attribut réservé ne pouvant donc pas être utilisé pour nommer un signal.

- `initValue` : valeur initiale du signal.
- `initApply` : fonction appelée à la première réaction dont le retour est la valeur initiale du signal.
- `combine` : fonction de combinaison.

La valeur initiale d'un signal est calculée soit statiquement, soit dynamiquement : les propriétés `initValue` et `initApply` sont ainsi exclusives et ne peuvent donc pas être utilisées conjointement. Par exemple :

```
<hh.module A=${{direction: hh.IN, initValue: 15}}
           B C=${{combine: (a, b) => a + b}}>
   statement
</hh.module>
```

Dans cet exemple, `A` est un signal d'entrée initialisé à 15. Le signal `B` est un signal d'entrée-sortie (par défaut) et le signal `C` est un signal combiné.

6.1.2 Expressions

Expression Hiphop.js

Le langage JavaScript est utilisé pour définir une expression Hiphop.js. Contrairement à la syntaxe concrète, où une expression est toujours évaluée à l'exécution du programme Hiphop.js, la syntaxe abstraite définit deux types d'expressions.

- Les expressions *statiques* sont évaluées une seule fois, lors de la compilation du programme Hiphop.js. Sauf cas contraire explicitement mentionné, elles sont définies par une valeur passée à l'attribut `value` du nœud XML :

  `value=${`$expression_{JS}$`}`

  Par exemple :

  ```
  <hh.emit Alea value=${Math.rand()}}/>
  ```

  Une valeur numérique aléatoire est calculée lors de la compilation du programme Hiphop.js. Cette valeur est mémorisée et sera émise à chaque exécution de cette instruction.

- Les expressions *dynamiques* sont évaluées à l'exécution du programme Hiphop.js, comme celles de la syntaxe concrète. Elles sont définies par une fonction JavaScript. Sauf cas contraire explicitement mentionné, cette fonction est passée à l'attribut `apply` du nœud XML :

  `apply=${`$fonction_{JS}$`}`

  Par exemple :

  ```
  <hh.emit Alea apply=${function() {
     return Math.rand()
  }}/>
  ```

  La fonction JavaScript est appelée à l'exécution de l'instruction, et la valeur retournée correspond à la valeur de l'expression. Le signal `Alea` est donc émis avec une valeur numérique aléatoire à chaque instant où cette instruction est exécutée.

Lecture de signal

Soit un signal `s`. Une expression dynamique peut lire le statut et la valeur de `s` via les expressions JavaScript suivantes :

- `this.present.s` : retourne `vrai` si `s` est présent dans l'instant, `faux` sinon ;
- `this.prePresent.s` : retourne `vrai` si `s` était présent à l'instant précédent, `faux` sinon. À l'instant initial, cette expression retourne `faux` ;
- `this.value.s` : retourne la valeur de `s` dans l'instant ;
- `this.preValue.s` : retourne la valeur de `s` à l'instant précédent. À l'instant initial, cette expression retourne la valeur initiale du signal (`undefined` si le signal n'a pas de valeur initiale).

Ces expressions JavaScript ne peuvent être utilisées que littéralement comme ci-dessus, dans le corps de la fonction d'une expression dynamique Hiphop.js. Cette fonction ne peut pas être une *fonction fléchée* de JavaScript [ECMA, 2015]. Un premier contre-exemple est le code suivant :

```
<hh.emit v apply=${function() {
  const value = this.value;
  return value.s;
}/>
```

Ici, l'expression JavaScript de lecture de la valeur du signal `s` est invalide car l'expression n'est pas utilisée de façon littérale. Un second contre-exemple est le suivant :

```
<hh.emit v apply=${() => this.value.s}/>
```

Ici, l'expression JavaScript de lecture de la valeur de `s` est invalide car elle est utilisée dans une fonction fléchée. Enfin, un exemple valide est le suivant :

```
<hh.emit v apply=${function() {
  return this.value.s;
}}/>
```

Le signal `v` est émis avec pour valeur celle de `s` dans l'instant. Cet exemple est valide car l'expression JavaScript est utilisée de façon littérale et dans une fonction non fléchée.

Techniquement, ces restrictions sont liées au fait que Hiphop.js applique une analyse lexicale simple [2] sur le corps de la fonction lors de la compilation du programme afin de déterminer des dépendances de contrôle et de données de l'expression dynamique, voir section 6.4.3. Une autre raison est que la valeur du receveur `this` est construite et liée à `this` dynamiquement par Hiphop.js lors de l'appel à la fonction de l'expression dynamique. Les fonctions fléchées ne permettant pas de lier dynamiquement le receveur, il est impossible d'y injecter le statut et la valeur des signaux lus.

Délais

Les délais standards sont définis en utilisant les expressions statiques ou dynamiques de Hiphop.js. L'ensemble des exemples suivant utilisent l'instruction `<hh.await/>`, mais ils s'appliquent de façon similaire aux autres instructions utilisant un délai. Considérons un premier exemple :

```
<hh.await apply=${function() {
   return this.present.S
}}/>
```

Ici, la fonction d'expression dynamique sera exécutée à chaque instant suivant celui de démarrage de l'instruction `<hh.await/>`. À l'instant où le signal `s` est présent, l'expression `this.present.S` retourne `vrai`, ce qui entraîne la terminaison du délai et de l'instruction. Une forme plus compacte permet également de tester le statut de présence d'un signal dans l'instant et à l'instant précédent :

2. La raison est historique : initialement, Hiphop.js ne disposait pas de syntaxe concrète et donc pas d'analyseur lexical et syntaxique puisque le programme était directement écrit en XML.

```
<hh.await S/>
<hh.await pre S/>
```

Par ailleurs, un compteur est initialisé par une expression Hiphop.js statique en associant l'expression JavaScript à l'attribut `countValue`, ou par une expression Hiphop.js dynamique en associant la fonction JavaScript à l'attribut `countApply`. L'expression de test est une expression Hiphop.js statique ou dynamique :

```
<hh.await countValue=${3} apply=${function() {
   return this.present.S
}}/>
```

Le compteur est initialisé à 3 à chaque démarrage de l'instruction `<hh.await/>`, et se termine après trois instants où le signal `s` est présent. L'exemple suivant utilise une expression dynamique pour initialiser le compteur :

```
<hh.await
   countApply=${function() {
      return this.value.v
   }}
   apply=${function() {
      return this.present.s
   }}
/>
```

Le compteur est initialisé au démarrage de l'instruction `<hh.await/>` à la valeur $N > 0$ du signal `v` dans l'instant. L'environnement Hiphop.js vérifie dynamiquement que $N > 0$ après l'exécution de la fonction passée à `countApply`. Cet exemple s'écrit également de façon plus compacte :

```
<hh.await countApply=${function() {return this.value.V}} S/>
```

Enfin, les délais immédiats sont distingués des délais standards par la présence du mot clé `immediate` dans le nœud de l'instruction :

```
<hh.await immediate S/>
```

Le délai se termine immédiatement si `s` est présent à l'instant où l'instruction démarre. L'utilisation du mot-clé `immediate` avec un compteur est interdite.

La valeur `THIS` de `EXEC`

Dans la syntaxe concrète, l'ensemble des instructions JavaScript passées à `EXEC` acceptent la valeur `THIS`. Cela permet d'accéder à l'identifiant unique de l'ins-

tance de `EXEC`, de savoir si l'instance est préemptée et d'ajouter des propriétés, voir section 4.4.18. Dans la syntaxe abstraite, `THIS` correspond à l'expression `this.payload`. Ainsi :

- `this.payload.id` retourne l'identifiant unique de l'instance de `EXEC` ;
- `this.payload.killed` indique si l'instance de `EXEC` est préemptée.

Comme avec `THIS`, il est possible d'ajouter de nouveaux attributs quelconques à `this.payload`. L'ensemble des propriétés de `this` dans une expression dynamique Hiphop.js sont présentées en section 6.2.1.

6.1.3 Discussion

Historiquement, la syntaxe concrète de Hiphop.js n'existait pas et les programmes étaient directement écrits en utilisant la syntaxe abstraite. Ce choix était motivé par le fait que Hop.js disposait déjà d'un analyseur lexical et syntaxique XML, ce qui a permis de se concentrer sur l'implémentation de la compilation de Hiphop.js ainsi que sur certains aspects de l'intégration et de reporter à plus tard la question d'une syntaxe dédiée.

Puisque la syntaxe abstraite permet l'écriture de programmes Hiphop.js à un niveau plus proche de l'AST, elle facilite l'expérimentation de nouvelles constructions sans avoir à modifier l'analyseur lexical et syntaxique du préprocesseur, notamment par composition de constructions existantes. En revanche, cette syntaxe est peu abstraite et nécessite une compréhension fine de l'implémentation de Hiphop.js pour être utilisée. Pour cette raison, elle s'adresse d'avantage aux personnes désireuses de travailler sur le langage qu'aux utilisateurs.

6.2 Du code à l'AST

Afin de rendre les programmes Hiphop.js lisibles et exécutables par un environnement JavaScript, il est nécessaire de transformer les constructions en une forme purement JavaScript. Cette transformation se fait en deux étapes. La première étape repose sur un préprocesseur qui transforme les programmes Hiphop.js écrits avec la syntaxe concrète en programmes Hiphop.js écrits avec la syntaxe abstraite XML. La seconde étape repose sur une extension de Hop.js permettant la transformation de code XML en un emboîtement d'appel de fonctions JavaScript. Le code produit est alors conforme à la norme ECMAScript.

6.2.1 Préprocesseur

Un fichier JavaScript contenant du code Hiphop.js est analysé par un préprocesseur. Le préprocesseur crée un nouveau fichier JavaScript contenant le code source du fichier d'entrée dont les constructions Hiphop.js utilisant la syntaxe concrète sont remplacées en un code analogue utilisant la syntaxe abstraite. Les codes sources du préprocesseur se trouvent dans le dossier `preprocessor/`. Considérons par exemple le programme Hiphop.js suivant contenu dans un fichier `foo.js` :

```
module.exports = MODULE Foo (OUT bar) {
   LOOP {
      EMIT bar;
      PAUSE;
   }
}
```

Le préprocesseur peut-être invoqué de deux façons. La première et la plus usuelle consiste à importer le fichier JavaScript contenant du code Hiphop.js à l'exécution de Hop.js, comme n'importe quel module JavaScript :

```
require("hiphop");
const fooHh = require("./foo.js", "hiphop");
```

La première ligne permet d'importer l'environnement de compilation et d'exécution de Hiphop.js dans Hop.js. La seconde ligne permet d'importer le module JavaScript contenant le programme Hiphop.js dans Hop.js. Le second paramètre `hiphop` indique à Hop.js d'invoquer la fonction `Symbol.compiler` définie par Hiphop.js dans le fichier `lib/hiphop.js`. Cette fonction a pour effet d'invoquer le préprocesseur avec le fichier `foo.js` en entrée et de retourner à Hop.js le fichier généré. Le préprocesseur peut aussi être appelé depuis une invite de commande Unix. La commande est la suivante :

```
INPUT=foo.js OUTPUT=foo.out.js preprocessor/preprocessor.js
```

Une fois traité par le préprocesseur Hiphop.js, le programme est transformé en un programme analogue utilisant la syntaxe abstraite. Il correspond alors au code suivant qui est enregistré dans `foo.out.js` :

```
<hh.module __hh_debug_name__="Foo"
           bar=${{accessibility: hh.OUT}}>
  <hh.loop>
    <hh.emit bar/>
    <hh.pause/>
  </hh.loop>
</hh.module>
```

Le préprocesseur de Hiphop.js est composé de trois phases. Les deux premières sont implémentées dans `preprocessor/{lexer,parser}.js`. Il s'agit des analyses lexicale et syntaxique, où les grammaires de Hiphop.js et JavaScript sont reconnues (la spécification ECMAScript 5 est complète et ECMAScript 6 est partielle). La dernière phase est une génération de code du programme en utilisant la syntaxe abstraite. Elle est implémentée dans `preprocessor/gen.js`.

Cette transformation de code pose problème en cas d'erreur dans le code généré : l'erreur annoncée par Hop.js ou dans le navigateur correspond à une position dans le fichier généré et pas dans le fichier source. Ce problème est résolu par la création d'un fichier « source map » qui associe la position de chaque mot dans le fichier source à la position du mot correspondant dans le fichier destination [Lenz et Fitzgerald, 2011]. Ce fichier est utilisé par tous les navigateurs modernes ainsi que par Hop.js afin d'indiquer en cas d'erreur la position dans le fichier source.

Transformation des expressions et instructions Hiphop.js

Une expression Hiphop.js, une instruction `ATOM` et les instructions de la famille de `EXEC` sont implémentées sous forme de fonction JavaScript lors de la génération de la syntaxe abstraite. Différents cas sont possibles :

- Une expression Hiphop.js est transformée en une fonction JavaScript. La valeur de retour est calculée par une expression JavaScript sémantiquement équivalente à l'expression Hiphop.js.
- Le corps de `ATOM` est transformé en une fonction JavaScript. L'exécution de cette fonction est sémantiquement équivalente à l'exécution du corps de `ATOM`..
- Les instructions passées aux paramètres `ONSUSP`, `ONRES` et `ONKILL` de `EXEC` sont transformées en fonctions JavaScript. L'exécution de ces fonctions est sémantiquement équivalente à l'exécution des instructions respectives passées aux paramètres de `EXEC`.

La transformation d'une expression ou d'une instruction Hiphop.js en une fonction JavaScript consiste d'abord à substituer par une expression JavaScript chaque accesseur de signal et chaque construction spécifique à `EXEC`. Le code obtenu est ensuite textuellement inclus dans une fonction JavaScript. Les substitutions d'accesseurs et constructions spécifiques à `EXEC` sont les suivantes :

```
NOW(identifier):
        this.present.identifier

PRE(identifier):
        this.prePresent.identifier

VAL(identifier):
        this.value.identifier

PREVAL(identifier):
        this.preValue.identifier

DONE:
        this.done

DONEREACT:
        this.doneReact

THIS:
        this.payload
```

Par exemple, dans le code suivant :

```
EMIT s(VAL(v) + 1)
```

L'expression `VAL(v) + 1` est transformée en :

```
function() { return this.value.v + 1 }
```

Le nœud XML généré par le préprocesseur est donc :

```
<hh.emit s apply=${function() { return this.value.v + 1 }}/>
```

6.2.2 Transformation en code standard JavaScript

L'environnement Hop.js dispose d'une extension qui transforme automatiquement un arbre XML en un emboîtement de fonctions JavaScript lors de la lecture du code source. Considérons le programme Hiphop.js suivant, écrit avec la syntaxe abstraite :

```
<hh.module __hh_debug_name__="Foo"
           bar=${{accessibility: hh.OUT}}>
  <hh.loop>
    <hh.emit bar/>
    <hh.pause/>
  </hh.loop>
</hh.module>
```

Ce code est automatiquement transformé par Hop.js en code JavaScript standard [3] :

```
new hh.MODULE({
   __hh_debug_name__: "Foo",
   bar: {accessibility: hh.OUT}
}, [
   new hh.LOOP({}, [
      new hh.EMIT({signalName: "bar"}),
      new hh.PAUSE()
   ])
]);
```

L'AST du programme est construit pendant l'exécution du programme JavaScript, lorsque cet emboîtement de fonctions représentant le programme est exécuté. Dans le cas où le programme Hiphop.js est défini pour être exécuté sur le client, seule cette dernière représentation JavaScript est envoyée au client.

6.2.3 Construction de l'AST

L'exécution du programme vu en section 6.2.2 construit l'AST du programme. Chaque appel de fonction correspond à une instruction du programme et génère un objet JavaScript représentant un nœud de l'AST. Par exemple, `new hh.PAUSE` génère un nœud d'AST représentant l'instruction `PAUSE` et `new hh.MODULE` l'ensemble du module. Peu de vérifications sont appliquées à ce stade : seules des vérifications basiques comme la vérification du type des fils ou des attributs d'un nœud sont appliquées. Cela permet, par exemple, de s'assurer qu'un attribut `apply` reçoit bien une valeur fonctionnelle, ou qu'un nœud d'émission dispose d'un attribut identifiant le signal à émettre.

Les fonctions qui définissent les constructions de Hiphop.js sont définies dans `lib/lang.js`. Par exemple, l'implémentation simplifiée de la construction `EMIT` est :

```
const ast = require("./ast.js");
function EMIT(attrs={}) {
   const args = getEmitNodeArguments(attrs);
   checkNoChildren(arguments);
   return new ast.Emit(args.id, args.loc, args.noDebug,
                       args.signalNameList, args.func,
                       args.accessorList);
}
```

L'appel à `getEmitNodeArguments` permet de lire les attributs passés au nœud

3. La représentation peut varier légèrement en fonction du mode d'exécution de Hop.js.

XML, comme le nom des signaux émis, la fonction `apply`, etc. Ensuite, une fonction vérifie que le nœud ne contient aucun fils. Enfin le constructeur d'AST est appelé et le nœud d'AST est retourné. Les constructeurs d'AST de Hiphop.js sont définis dans `lib/ast.js`. Par exemple, l'implémentation simplifiée de la construction du nœud d'AST de `EMIT` est :

```
function Emit(id, loc, noDebug, signalNameList, func, accessorList) {
   ActionNode.call(this, id, loc, noDebug, func, accessorList);
   this.signalNameList = signalNameList;
}
```

Les nœuds d'AST sont organisés en une hiérarchie de classes en fonction du type d'instruction. `Emit` est une sous-classe de `ActionNode` qui contient une fonction `func` qui sera exécutée à l'exécution du programme Hiphop.js [4]. Ici, cette fonction permettra le calcul de la valeur éventuelle émise. Enfin, `accessorList` indique quels sont les signaux lus lors de l'exécution de `func`. Par exemple :

```
EMIT s(VAL(v) + 1)
```

Ici, `func` retourne la somme de 1 et de la valeur de `v` dans l'instant. La liste `accessorList` indique donc que la valeur courante de `v` est lue à l'exécution de `func`. La fonction `getAccessorList` analyse le contenu textuel de `func` et détermine quels sont les signaux lus en recherchant les expressions générées par la transformation des accesseurs. Elle est définie dans `src/lang.js`, et elle est appelée par le constructeur de `ActionNode`. Les informations fournies par `accessorList` sont nécessaires à la génération du circuit. Le mécanisme est détaillé à la section 6.4.3.

Enfin, `signalNameList` contient le nom des signaux émis par l'instruction ; dans l'exemple, il s'agit uniquement de `s`.

Construction dynamique de programme

Les transformations de code par le préprocesseur et par l'extension XML de Hop.js sont statiques car elles sont appliquées avant l'exécution du programme JavaScript environnant. En revanche, la construction d'un programme Hiphop.js et sa compilation sont dynamiques : elles se produisent pendant l'exécution du programme JavaScript environnant, voir le chapitre 5. Afin de présenter le mécanisme de construction dynamique de programme, reprenons l'exemple de la fonction `makePause` présentée en section 5.3.1 :

4. Dans le cas où `EMIT` émet une valeur dans le signal émis.

```
function makePause(n) {
   if (n <= 0) {
      return NOTHING;
   } else if (n == 1) {
      return PAUSE;
   } else {
      return SEQUENCE {
         PAUSE;
         ${makePause(n - 1)};
      } } }
```

Cette fonction construit une séquence Hiphop.js de `PAUSE`, dont le nombre est calculé à l'exécution du programme JavaScript. La construction `${`*js-expression*`}` étant en réalité une extension de la syntaxe XML de Hop.js (voir section 6.1), le préprocesseur ne modifie pas cette construction lors de la génération de la syntaxe XML :

```
function makePause(n) {
   if (n <= 0) {
      return <hiphop.nothing/>;
   } else if (n == 1) {
      return <hiphop.pause/>;
   } else {
      return <hiphop.sequence>
        <hh.pause/>
      ${makePause(n - 1)};
      </hh.sequence>;
   }
}
```

Une fois que les constructeurs XML sont transformés par Hop.js en appel de fonction JavaScript, la construction `${makePause(n - 1)}` devient un simple appel de fonction JavaScript et la fonction `makePause` qui est exécutée lors de la construction d'un programme Hiphop.js est :

```
function makePause(n) {
   if (n <= 0) {
      return hiphop.NOTHING();
   } else if (n == 1) {
      return hiphop.PAUSE();
   } else {
      return hiphop.SEQUENCE({}, [
         hiphop.PAUSE(),
         makePause(n - 1)
      ]);
   }
}
```

Cette fonction se résume donc à la construction récursive d'un arbre : les constructeurs Hiphop.js étant des valeurs JavaScript, ils sont manipulés à l'exécution du programme JavaScript comme n'importe quelle valeur du langage. Ainsi, que la syntaxe concrète ou la syntaxe abstraite soit utilisée, un programme Hiphop.js est toujours construit dynamiquement. Le langage ReactiveML est similaire à Hiphop.js en ce point : un programme ReactiveML est construit pendant l'exécution du programme OCaml environnant [Mandel et Pouzet, 2008].

6.3 De l'AST à la machine réactive

La dernière phase de compilation d'un programme Hiphop.js consiste à générer une machine réactive à partir de l'AST d'un module Hiphop.js. Cette phase est déclenchée lorsque le constructeur `new hh.ReactiveMachine` est invoqué par le programme JavaScript environnant. La fonction `ReactiveMachine` est définie dans `lib/machine.js`. Elle vérifie d'abord que le premier paramètre correspond bien à l'AST d'un programme Hiphop.js. Elle définit ensuite plusieurs propriétés qui seront initialisées par la compilation du programme, comme la liste des portes logiques du circuit généré ou la liste des signaux. Enfin, elle clone l'AST passé en paramètre, permettant de le garder vierge pour des compilations futures [5] et elle appelle la fonction principale de compilation nommée `compile` qui est définie dans `lib/compiler.js`. Cette fonction prend la machine réactive en paramètre et applique les procédures suivantes :

- Pour chaque nœud correspondant à l'instruction `RUN`, l'AST du module invoqué est copié. La copie de l'AST est ensuite ajoutée en tant que fils du nœud correspondant à `RUN`.
- Plusieurs analyses sont ensuite appliquées. Elles décorent l'AST par des paramètres utiles à la génération du modèle d'exécution, notamment à la génération des mécanismes associés à l'instruction `TRAP` et au phénomène de réincarnation [Berry, 2002]. De plus, certaines erreurs de programmation sont détectées ce qui permet de rejeter immédiatement un programme invalide. Par exemple, considérons le code suivant :

  ```
  EMIT sig(VAL(sig) + 1))
  ```

  Ce code est invalide car il y a un cycle de causalité : l'expression calculant la valeur à émettre dans le signal `sig` dépend de la valeur de `sig`. Une analyse

5. Ceci permet de générer plusieurs machines réactives à partir du même programme Hiphop.js.

vérifie que les expressions des instructions `EMIT` et `SUSTAIN` ne lisent pas la valeur du signal émis dans l'instant. Ici, l'émission de `sig` utilisant la valeur de `sig` dans l'instant, une erreur est levée. À noter que cette erreur est levée dès la compilation du programme, car il s'agit d'une simple analyse locale à l'expression de `EMIT`.

- Le modèle d'exécution, décrit à la section 6.4, est ensuite généré.
- Les optimisations décrites dans section 6.6 sont ensuite appliquées au circuit.
- Enfin, le circuit est associé à la machine réactive. À ce stade, la machine réactive est prête à enregistrer des gestionnaires d'évènements associés aux signaux de sortie, à recevoir l'émission de signaux d'entrée et à réagir.

À noter qu'aucune analyse de causalité n'est appliquée à la compilation du programme. Seul le cas trivial de l'émission d'un signal valué avec pour valeur celle du même signal dans l'instant est vérifié et rejeté.

6.4 Modèle d'exécution

Le modèle d'exécution de Hiphop.js est celui d'Esterel v5 et v7 [Berry, 2002, Potop-Butucaru *et al.*, 2007]. Il s'agit de circuits logiques booléens composés de portes logiques *ET*, *OU*, *NON* et de registres (mémoires de 1 bit) interconnectés. Une porte peut être associée à une action de calcul sur des valeurs de données du programme. Cette action est déclenchée lorsque la porte s'évalue à `vrai`. La réaction d'un programme Hiphop.js consiste à propager les valeurs `vrai` ou `faux` dans le circuit logique selon la *sémantique constructive* d'Esterel [Berry, 2002].

D'autres modèles implémentant la sémantique d'Esterel existent, par exemple [Edwards, 1999, Weil *et al.*, 2000, El Sibaïe, 2018]. Pour Hiphop.js, la compilation vers des circuits logiques booléens a été retenue pour trois raisons. La première raison est que cette technique est éprouvée car elle est utilisée par des compilateurs industriels d'Esterel v5 et v7. La seconde raison est que cette technique repose sur un isomorphisme entre une instruction et un circuit logique booléen. Cela simplifie la logique du compilateur : une unique fonction générant un circuit est définie pour chaque instruction. Enfin, la troisième raison est que cette technique facilite le développement du débogueur symbolique qui est décrit au chapitre 7.

6.4.1 Le graphe de portes associé à un programme

En Hiphop.js, le circuit logique booléen est simulé sous forme d'un graphe d'objets JavaScript. Chaque nœud du graphe est un objet qui simule une porte ou un registre en référençant ses entrées et ses sorties. Les signaux sont implémentés par une porte *OU*, l'ensemble des instructions Hiphop.js par des portes logiques ayant un nombre quelconque d'entrées ou de sorties, et les registres n'ont qu'une unique entrée. Par exemple, une instruction `PAUSE` est implémentée par un registre initialisé à `faux`. Les registres représentent la seule façon de passer des valeurs d'une réaction à la suivante : leur valeur de sortie à la réaction suivante correspond à leur valeur d'entrée à la réaction courante [Berry, 2002, Berry, 2018c].

Afin de comprendre l'exécution d'une réaction, que ce soit pour aider l'utilisateur à déboguer son programme ou un développeur de Hiphop.js à déboguer le compilateur, la machine réactive permet d'afficher la propagation des valeurs à travers les portes du circuit pendant une réaction. Par exemple, un extrait de la trace est le suivant :

```
propagate Module_a value:true loc:219
  receive Module_a_pre_reg value:true
  receive Await_Abort_testexpr_2 1/2 value:true dep:true loc:386
    value-before:-1
    value-after:-1
```

Cet extrait de trace, dont le détail est expliqué en section 6.4.3, indique que la porte implémentant le signal d'entrée `a` du programme « ABRO » implémenté dans `examples/abro/abro-hh.js` propage la valeur `vrai` à ses deux sorties.

Cette fonctionnalité est contrôlée par l'attribut `tracePropagation` de la machine réactive. Soit la machine réactive `m` :

- `m.tracePropagation = true` affiche la trace de la propagation des valeurs lors des prochaines réactions.
- `m.tracePropagation = false` n'affichera pas la trace de la propagation des valeurs lors des prochaines réactions.
- `new hh.ReactiveMachine(prg, {tracePropagation: true})` crée une machine réactive (implémentant le programme Hiphop.js `prg`) dont l'attribut `tracePropagation` est automatiquement mis à `vrai`.

6.4.2 Implémentation d'une réaction

Lors d'une réaction, la machine réactive simule l'exécution du circuit en propageant progressivement les valeurs des entrées du programme ainsi que les valeurs données par

l'état des registres vers les sorties du programme, selon la sémantique constructive des circuits qui implémente celle d'Esterel [Berry, 2002]. La valeur d'une porte est calculée et propagée, si c'est possible, lorsque qu'elle reçoit la valeur `vrai` ou `faux` d'une de ses entrées. On dit alors que la porte est *connue*.

- La porte *OU* propage `vrai` à ses sorties dès qu'elle reçoit `vrai` de l'une de ses entrées, alors qu'elle ne peut propager `faux` à ses sorties que lorsqu'elle a reçu `faux` depuis l'ensemble de ses entrées.
- Le comportement de la porte *ET* est dual : `faux` est propagé à ses sorties dès que `faux` est reçu depuis l'une de ses entrées, alors que `vrai` n'est propagé à ses sorties que lorsqu'elle a reçu `vrai` depuis l'ensemble de ses entrées.
- Enfin, la porte *NON* propage immédiatement la négation de la valeur qu'elle reçoit car elle ne dispose que d'une unique entrée.

Si la propagation des valeurs se termine en ayant calculé les valeurs de toutes les portes du circuit, alors le résultat est déterministe car indépendant de l'ordre des propagations élémentaires. Sinon, c'est qu'il y a une erreur causée par un cycle de causalité. Il s'agit d'une erreur de programmation, cette notion est étudiée en détail dans [Berry, 2002, Potop-Butucaru *et al.*, 2007]. Le code suivant est une implémentation simplifiée de la méthode `react` permettant de simuler l'exécution du circuit logique par la machine réactive. Cette méthode est définie dans `lib/machine.js`, c'est le point d'entrée d'une réaction :

```
ReactiveMachine.prototype.react = function() {
   let knownList = this.getKnown();
   let remain  = this.gateList.length();

   while (knownList.length) {
      let gate = knownList.shift();
      for (let i in gate.fanoutList) {
         let fanout = gate.fanoutList[i];
         let value = fanout.neg ? !gate.value : gate.value;
         if (fanout.net.receive(value, fanout.dep)) {
           knownList.push(fanout.net);
         }
      }
      remain--;
   }

   if (remain) {
      throw new Error("Causality error");
   }
}
```

Le receveur `this` correspond à la machine réactive. La variable `knownList` est une liste contenant les portes et les registres ayant un état connu au déclenchement de la réaction :

- La porte implémentant un signal d'entrée qui est émis par l'environnement avant le déclenchement de la réaction, ou qui n'est jamais émis depuis le programme Hiphop.js.
- Un registre auxiliaire interne permettant de démarrer la première réaction. Il est initialisé à `vrai` avant la première réaction puis maintenu à `faux` lors des réactions suivantes.
- Le registre de chaque instruction `PAUSE` présente dans le programme.
- Le registre de chaque signal dont la valeur correspond au statut du signal à l'instant précédent.

Toutes les autres portes du programme ont la valeur `undefined` en début de réaction, et ne sont donc pas dans `knownList`. L'algorithme suivant se répète ensuite tant que `knownList` n'est pas vide :

- La première porte de `knownList` est retirée de la liste et affectée à la variable `gate`;
- `gate` propage sa valeur à chacune de ses sorties via leur méthode `receive`. Si cette valeur suffit à déterminer celle d'une sortie, celle-ci est ajoutée à `knownList`. Le détail de l'appel à `receive` est expliqué en section 6.4.3.

Lorsque `knownList` est vide, la réaction est terminée. La variable `remain`, initialisée en début de réaction par le nombre de portes dans le circuit puis décrémentée à chaque fois qu'une porte propage sa valeur à ses sorties, permet de savoir en fin de réaction si toutes les portes ont bien calculé et propagé leur valeur. Ainsi, si `remain` n'est pas nulle lorsque `knownList` est vide, alors au moins une porte n'a pas calculé et propagé sa valeur. Cela signifie qu'un cycle de causalité existe dans le circuit et une erreur est levée. Autrement, les signaux de sortie émis par le programme Hiphop.js sont ceux dont la porte vaut `vrai`. Le nouvel état de contrôle du programme est donné par celui des registres : un registre de `PAUSE` valant `vrai` indique qu'une `PAUSE` est devenue active et que le contrôle passera à l'instruction suivante à la prochaine réaction.

Les sections suivantes détaillent les mécanismes mis en jeu pendant la réaction : la section 6.4.3 détaille la façon dont est calculée la valeur d'une porte et la section 6.4.4 détaille le mécanisme d'action associé à une porte.

6.4.3 Calcul de la valeur d'une porte

Le calcul de la valeur d'une porte pendant la réaction est décrit dans cette section. D'abord, une première partie décrit les types de dépendances d'une porte à l'autre. Ensuite, une seconde partie présente la structure de données utilisée pour représenter la connexion entre deux portes ou registres ainsi que la façon dont elle est utilisée pour l'invocation de `receive`, la méthode permettant à une porte de recevoir la valeur de l'une de ses entrées. Enfin, une implémentation de la méthode `receive` est présentée et expliquée.

Dépendance de contrôle et dépendance de donnée

Il existe deux types de dépendances entre les portes : les *dépendances de contrôle* et les *dépendances de données*.

Les dépendances de contrôle correspondent à la propagation du flot de contrôle et des valeurs booléennes dans le circuit. Elle permettent, par exemple, d'assurer le passage du contrôle d'une instruction à une autre en séquence, ou que l'instruction d'émission d'un signal dans une branche parallèle soit exécutée avant la lecture du statut du signal (via la primitive `NOW`) dans le même instant depuis une autre branche parallèle. La sémantique constructive des circuits correspondant aux instructions du langage repose sur ce type de dépendance [Berry, 2002].

Les dépendances de données sont utilisées lorsque des calculs sur des valeurs de signaux sont exécutées pendant la réaction, c'est-à-dire lorsque la valeur d'un signal est lue via la primitive `VAL`. Elles permettent, par exemple, d'assurer que la valeur d'un signal `s` n'est lue que lorsque toutes les instructions émettant `s` avec une valeur dans l'instant ont été exécutées. Ces dépendances ne font pas partie des circuits correspondant aux instructions et sont donc rajoutées en amont par les compilateurs Esterel v5, v7 et Hiphop.js. Considérons le programme suivant :

```
MODULE (OUT a, OUT b) {
   FORK {
     EMIT a;
     EMIT b(VAL(a));
   } PAR {
     EMIT a(34);
   }
}
```

À la première réaction de ce programme, l'instruction `FORK` est démarrée et démarre deux branches parallèles. La première branche émet le signal `a` puis émet le signal `b` avec pour valeur la valeur de `a`. La seconde branche émet le signal `a` avec pour valeur

34 (si `a` était un signal combiné et que la première émission comportait une valeur, alors la valeur de la seconde émission correspondrait au résultat de l'appel à la fonction de combinaison). Comme décrit à la section 4.4.8, l'exécution de l'instruction `EMIT b(VAL(a))` sera bloquée jusqu'à ce que l'instruction `EMIT a(34)` soit exécutée.

Afin que cet ordonnancement soit respecté, le compilateur ajoute une connexion entre la porte implémentant `EMIT a(34)` et celle implémentant l'évaluation de l'expression `(VAL(a))`. Cette connexion correspond à une dépendance de données entre l'émission de `a` avec valeur et la lecture de sa valeur. Enfin, notons que la primitive `PREVAL` n'entraîne pas la construction de dépendance de donnée : elle s'évalue en une valeur qui a été calculée à la réaction précédente, il n'y a donc pas besoin d'attendre pour la lire.

Connexion de portes et invocation de `receive`

Chaque porte ou registre dispose d'une liste de sorties appelée `fanoutList`. Cette liste référence des objets représentant une connexion avec une autre porte qui est appelée *netB*. Cette connexion est définie par un objet contenant trois attributs :

- `net` référence la porte réceptrice de la valeur propagée ;
- `neg` est un booléen indiquant si la négation logique doit être appliquée à la valeur lors de la propagation ;
- `dep` est un booléen. S'il est à `vrai` alors la connexion est une dépendance de données, sinon une dépendance de contrôle.

Cette structure de données est illustrée à la figure 6.2, où une porte *netA* propage sa valeur à la porte *netB*.

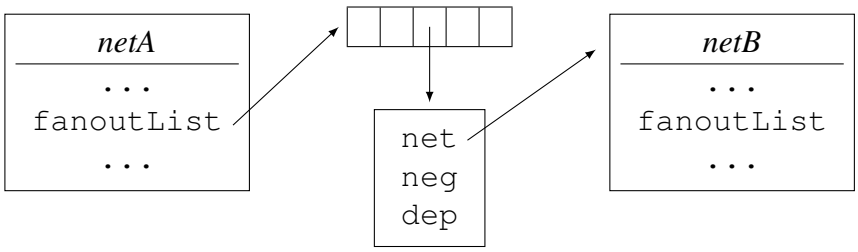

FIGURE 6.2 – Connexion entre deux portes ou registres.

Ainsi, comme le montre l'implémentation de la fonction de réaction `react`, la propagation de la valeur *v* d'une porte *netA* consiste à appliquer l'algorithme suivant pour chaque élément `fanout` de la `fanoutList` de *netA* :

- Calculer la valeur v_{comp} à propager : si l'attribut `neg` de *fan* est vrai, alors v_{comp} vaut la négation de v, sinon v_{comp} vaut v ;
- Appeler la méthode `receive` de la porte *netB* à qui la valeur est propagée via l'attribut `net` de *fan*. Le premier paramètre est v_{comp} et le second paramètre est la valeur de l'attribut `dep` de *fan*.

Exemple d'implémentation de `receive`

Cette partie permet de synthétiser les notions de propagation de valeurs et de dépendances. Le code présenté ici est l'implémentation de la méthode `receive` des portes logiques *ET* et *OU*, appelées `LogicalNet` dans le compilateur de Hiphop.js. Cette méthode est appelée par la porte qui reçoit une valeur de l'une de ses entrées. Ce code ainsi que l'implémentation de la logique de l'ensemble des portes est implémenté dans `lib/net.js`.

```
LogicalNet.prototype.receive = function(value, dep) {
   if (this.value !== undefined && this.depCount == 0) {
      return false;
   }

   if (dep) {
      this.depCount--;
   } else {
      this.inCount--;
   }

   if (!dep && this.neutral != value)) {
      this.value = value;
   }

   let ret = false;
   if (this.depCount == 0) {
      if (this.value !== undefined) {
         ret = true;
      } else if (this.inCount == 0) {
         this.value = this.neutral;
         ret = true;
      }
   }
}
```

Plusieurs caractéristiques sont à noter. D'abord, `LogicalNet` est une même classe implémentant à la fois les portes *ET* et les portes *OU*. Elles sont différenciées par l'attribut `neutral` : il est initialisé à l'élément neutre de l'opérateur booléen concerné :

`vrai` pour les portes *ET*, `faux` pour les portes *OU*. Ceci permet d'avoir un code commun implémentant ces deux portes logique tout en conservant le comportement respectif des portes *ET* et *OU* grâce à l'attribut `neutral`. Par ailleurs, les attributs `depCount` et `inCount` sont des compteurs respectivement initialisés en début de réaction par le nombre de dépendances de données et le nombre de dépendances de contrôle de la porte. Les tests logiques de cette fonction étant complexes, le comportement se résume ainsi :

- Si la valeur de la porte est déjà connue, alors la valeur reçue est ignorée.
- La valeur reçue est ignorée si elle provient d'une dépendance de données.
- Si la valeur reçue est différente de la valeur neutre de la porte, alors la valeur de la porte est fixée à la valeur reçue.
- Si le compteur de dépendances de données est nul et que la porte a fixé sa valeur, alors cette valeur est propagée. En revanche, si la porte n'a pas fixé sa valeur et que le compteur de dépendances de contrôle est nul, la valeur neutre de la porte est alors propagée.

Cette méthode retourne `faux` si la porte n'est pas connue ou si elle est déjà connue (elle a propagé sa valeur précédemment). Sinon, elle retourne `vrai` : la porte devient connue.

Il est maintenant possible d'expliquer l'extrait de la trace montrée en section 6.4.1. Elle est affichée lorsque l'attribut `tracePropagation` de la machine réactive est à `vrai` :

```
propagate Module_a value:true loc:219
  receive Module_a_pre_reg value:true
  receive Await_Abort_testexpr_2 1/2 value:true dep:true loc:386
    value-before:-1
    value-after:-1
```

L'extrait de cette trace correspond à la réaction d'un programme dont le signal d'entrée `a` est émis :

- La première ligne indique que la porte nommé `Module_a` (la porte implémentant le signal d'entrée `a`) propage la valeur `vrai` à ses sorties.
- La seconde ligne indique la réception de la valeur de `Module_a` par un registre nommé `Module_a_pre_reg` (le registre permettant de déterminer le statut de `a` à l'instant précédent). Le registre vaut maintenant `vrai` et propagera cette valeur à une porte auxiliaire au début de la réaction suivante [6].

6. Les expressions `PRE(a)` s'évalueront par la valeur de cette porte auxiliaire à la réaction suivante.

- La troisième ligne indique la réception de la valeur de `Module_a` par une porte logique nommée `Await_Abort_testexpr_2`.
  - Le suffixe `2` du nom de la porte indique son index de réincarnation, voir section 6.5.
  - `1/2` indique que cette porte a deux entrées dont une (`Module_a`) lui ayant propagé sa valeur.
  - `dep:true` indique que la connexion entre `Module_a` et `Await_Abort_testexpr_2` est une dépendance de donnée, la valeur reçue est donc ignorée.
  - `value-before` indique la valeur de la porte avant la réception et `value-after` après la réception. Ici, toutes deux valent -1, ce qui signifie que la porte n'est pas encore connue.

Enfin, le nombre préfixé par `loc:` correspond à la position dans le code source du programme Hiphop.js de l'instruction implémentée par cette porte.

6.4.4 Exécution de l'action associée à une porte

Le mécanisme permettant l'exécution d'une action associée à une porte pendant la réaction est décrit dans cette section. Ce mécanisme permet l'exécution d'expressions Hiphop.js, du bloc d'instructions de `ATOM`, ou encore des instructions paramétrant `EXEC`.

Le type de porte `ActionNet`, sous type de `LogicalNet`, correspond à une porte *ET* dont l'évaluation à `vrai` entraîne l'appel d'une fonction définie dans l'attribut `func`. Un autre attribut `accessorList` permet au compilateur de construire les dépendances assurant que le statut ou la valeur des signaux lus dans `func` ne seront pas ensuite modifiés au cours de la réaction, voir section 6.5. Le code suivant est une implémentation simplifiée de la méthode `receive` de `ActionNet` :

```
const signal = require("./signal.js");
ActionNet.prototype.receive = function(value, dep) {
   if(LogicalNet.prototype.receive.call(this, value, dep)) {
      if (this.value === true) {
         this.func.call(signal.generateThis(this));
      }
      return true;
   }
   return false;
}
```

La troisième ligne correspond à un appel à la méthode `receive` de la classe parente implémentant la porte logique *ET* calculant la valeur de la porte. Si cet appel retourne `vrai`, alors la porte devient connue. Si sa valeur est `vrai` alors l'action associée est immédiatement effectuée via l'appel de la fonction référencée par l'attribut `func`. La fonction `generateThis`, définie dans `lib/signal.js`, construit et retourne un objet contenant le statut ou la valeur de chaque signal utilisé dans la fonction `func`. Dans le cas de `EXEC`, les fonctions par lesquelles s'évaluent `DONE` et `DONEREACT` sont également créées et associées à cet objet.

Enfin, deux cas sont à noter :

- Dans le cas de l'émission de signaux par `EMIT` ou `SUSTAIN`, la valeur retournée par `func` soit être assignée au signal. La classe `ActionNet` est donc spécialisée par `SignalExpressionNet`, où `receive` est redéfinie afin de faire ce traitement.
- Dans le cas plus général des expressions Hiphop.js, par exemple l'expression d'un délai ou de `IF`, la valeur retournée par `func` doit être convertie en valeur booléenne et elle devient la valeur de la porte implémentant l'expression. Dans le cas de `IF`, si la condition est vraie, il faut propager :
  - La valeur `vrai` de la porte implémentant l'expression dans la porte *ET* qui connecte branche *alors*.
  - La valeur `faux` de la porte implémentant l'expression dans la porte *ET* qui connecte la branche *sinon*.

  La classe `ActionNet` est donc spécialisée par `TestExpressionNet`, où `receive` est redéfinie afin de faire ce traitement : appeler `func` si la porte reçoit `vrai` et fixer la valeur de la porte par le résultat de l'appel à `func`.

6.5 Génération du circuit

La sémantique d'Esterel peut-être définie sous forme d'équations logiques correspondant au comportement de chaque instruction du langage [Berry, 2002]. Ces équations sont directement transposables en circuits logiques booléens : le compilateur de Hiphop.js implémente la construction du circuit logique correspondant à chaque instruction du langage. Toutes les instructions sont implémentées dans `lib/compiler.js`. Par exemple, le code suivant implémente la construction du circuit de l'instruction `NOTHING` :

```
function makeNothing(astNode) {
   let goList = [];
   let kMatrix = [[]];

   for (let i = 0; i <= astNode.depth; i++) {
      let go = net.makeOr(astNode, ast.Nothing, "go", i);
      goList[i] = go;
      kMatrix[0][i] = go;
   }

   return new Interface(astNode, ast.Nothing, goList, null,
                        null, null, null, kMatrix);
}
```

La variable `goList` contient les portes par lesquelles le flot de contrôle arrive à l'instruction. Il s'agit d'une liste de portes afin d'implémenter la réincarnation qui consiste à dupliquer certaines portes du circuit et y accéder en fonction d'un *index de réincarnation* [Berry, 2018b, Berry, 2002]. La matrice `kMatrix` représente les portes par lesquelles le flot de contrôle en sort. L'objet de type `Interface` qui est retourné contient l'ensemble des connexions de ce circuit auquel d'autres circuits sont connectés. D'autres instructions plus complexes disposent de portes annexes, comme `ABORT` représentée à la figure 6.3 (sans réincarnation). Quatre portes logiques connectées au sous circuit (l'implémentation du corps de `ABORT`) permettent d'appliquer la préemption. L'ensemble du circuit est connecté à différents types de fils (`GO`, `RES`, `SUSP`, etc.) qui sont utilisés pour connecter l'instruction `ABORT` aux autres instructions du circuit via l'objet `Interface`.

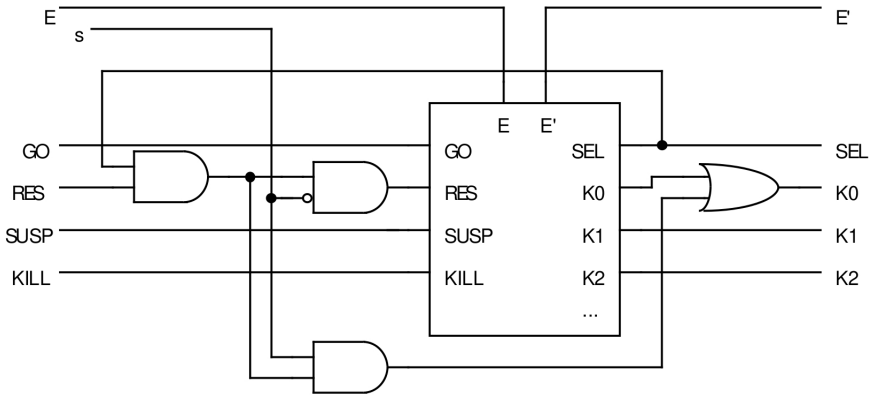

FIGURE 6.3 – Circuit de `ABORT`, figure empruntée à [Berry, 2002], page 121.

La construction globale du circuit d'un programme est de type « bottom-up » : les circuits correspondant aux feuilles de l'arbre sont d'abord construits, puis ceux des parents sont ensuite construits, et ainsi de suite jusqu'à la racine représentant le module. Ainsi,

lorsqu'un nœud parent est construit, il peut être directement connecté au circuit des nœuds fils via l'objet de type `Interface` retourné par la compilation de chaque instruction. Ce type de construction aide à la composition d'instructions complexes. Par exemple, l'instruction `AWAIT` est un raccourci syntaxique au code suivant :

```
ABORT(expression) {
   HALT;
}
```

La construction du circuit implémentant `AWAIT` se résume donc à la fonction suivante :

```
function makeAwait(astNode) {
   return makeAbort(astNode, makeHalt(astNode));
}
```

Le premier paramètre de `makeAbort` prend le nœud d'AST permettant de déterminer l'expression de délai de `ABORT`, et le second paramètre correspond au sous circuit du corps de l'instruction `ABORT`. Ici, il s'agit du circuit implémentant `HALT`.

Construction des dépendances

La section 6.2.3 a montré qu'une expression Hiphop.js, une instruction `ATOM` et les instructions de la famille de `EXEC` entraînent lors de la génération de la syntaxe abstraite la création d'une fonction sémantiquement équivalente et que ces expressions ou instructions sont réifiées en un nœud d'AST de type `ActionNode`. Lors de la génération du circuit, un nœud `ActionNode` entraîne la génération d'une porte de type `ActionNet`. Les dépendances sont automatiquement construites lors de la construction de la porte. Le code suivant est une version simplifiée du constructeur d'`ActionNet`. Il est implémenté dans `lib/net.js` :

```
const signal = require("./signal.js");
function ActionNet(astNode, type, debugName, func, accessorList) {
   LogicalNet.call(this, astNode, type, debugName, true);
   this.func = func;
   this.accessorList = accessorList;
   signal.runtimeSignalAccessor(this);
}
```

L'appel à la fonction `runtimeSignalAccessor`, définie dans `lib/signal.js` construit les dépendances de la porte. La version simplifiée est :

```
const net = require("./net.js");
function runtimeSignalAccessor(actionNet) {
   actionNet.accessorList.forEach(accessor => {
      let sig = compiler.getSignal(accessor.signalName);
      let lvl = lvl > sig.depth ? sig.depth : lvl;
      if (!accessor.getPre) {
        if (accessor.getValue) {
          sig.dependencyList[lvl].connectTo(actionNet, net.FAN.DEP);
        } else {
          sig.gateList[lvl].connectTo(actionNet, net.FAN.DEP);
        }
      }
   });
}
```

L'objet `sig` est un signal comprenant entre autres sa valeur dans l'instant et à l'instant précédent, sa *profondeur*, c'est-à-dire le nombre de réincarnations possibles du signal, la liste de portes *OU* `gateList` dont la valeur correspond au statut du signal (une pour chaque incarnation) et une liste de portes `dependencyList`. Chaque instruction d'émission avec valeur de `sig` est compilée en une porte avec une connexion de dépendance sur une porte D_{lvl} de `dependencyList`, où lvl correspond au niveau d'incarnation de `actionNet`. Ce mécanisme assure que lorsque D_{lvl} est connue la valeur de l'incarnation lvl de `sig` ne changera plus pendant la réaction, elle peut donc être lue. Les signaux sont définis dans `lib/signal.js`. La fonction `runtimeSignalAccessor` procède ainsi :

- L'incarnation `sig` à laquelle la dépendance doit être créée est calculée dans la variable `lvl`, conformément aux équations de [Berry, 2002].
- Si `actionNet.func` accède au statut dans l'instant de `sig`, une connexion de dépendance depuis la porte `sig.gateList[lvl]` vers `actionNet` est créée.
- Sinon, si `actionNet.func` accède à la valeur dans l'instant de `sig`, une connexion de dépendance est construite depuis `sig.dependency[lvl]`, assurant que la valeur est définitive dans l'instant.

6.6 Optimisations

Deux optimisations classiques sont appliquées sur le circuit généré [Sentovich *et al.*, 1992]. Elles sont implémentées dans la fonction `sweep` définie dans `lib/compiler.js`. Par défaut, elles sont appliquées sur le circuit une fois

qu'il est généré. Il est possible de les désactiver en passant l'attribut `sweep` à `faux` lors de la création de la machine réactive :

```
const hh = require("hiphop");
const SuspendableTimer = require("./timer.js");
new hh.ReactiveMachine(SuspendableTimer, {sweep: false});
```

Le reste de cette section détaille ces optimisations et montre ensuite le gain qu'elles apportent en termes de taille de circuit Hiphop.js.

Élimination des portes constantes

Le circuit généré contient des *portes constantes* : il s'agit de portes n'ayant aucune entrée, et propageant donc systématiquement la même valeur. Cette optimisation consiste donc à propager leur valeur neutre à leur sortie. Si la valeur neutre de la sortie est la même que celle de la porte constante, alors la connexion est supprimée. Une fois la connexion supprimée, la porte constante est aussi supprimée lorsqu'elle n'a plus de sortie.

Élimination des portes unaires

La technique de compilation utilisée dans Hiphop.js génère de nombreuses portes logiques inutiles. Il s'agit de portes ayant une unique entrée et propageant toujours la valeur reçue par l'entrée. Par exemple, l'instruction `NOTHING` génère une porte *OU* qui représente l'entrée du flot de contrôle de l'instruction et sa sortie :

```
EMIT a;
NOTHING;
EMIT b;
```

Ici, une porte *OU* est connectée entre la porte de sortie de `EMIT a` et la porte d'entrée de `EMIT b`. Cette porte est inutile : elle ne dispose que d'une unique entrée et propage simplement la valeur de son entrée. Elle est donc éliminée et la sortie de `EMIT a` est directement connectée à l'entrée de `EMIT b`. Supposons maintenant que le signal `b` n'ait qu'un unique émetteur dans le programme : l'instruction `EMIT b` de l'exemple. La porte logique implémentant l'instruction `EMIT b` est alors supprimée et la sortie de `EMIT a` est directement connectée sur la porte qui implémente le signal `b`.

À noter que les portes `ActionNet` n'ayant qu'une unique entrée ne doivent pas être supprimées. Par exemple, l'instruction `ATOM` est implémentée par une unique porte `ActionNet` : elle est à la fois l'entrée et la sortie de l'instruction, mais elle fait un effet de bord via l'appel à `func` si elle est évaluée à `vrai`. Un attribut booléen `noSweep`

est placé à `faux` par défaut sur l'ensemble des portes générées sauf les `ActionNet`. Ainsi, les portes ayant `noSweep` à `vrai` sont ignorées par `sweep` et ne sont pas supprimées.

Métriques

Le tableau 6.4 présente une comparaison de la taille du circuit en nombre de portes de différents programmes (tirés de `tests/` et de `examples/`), avant et après optimisations. Elles sont obtenues par l'appel à la fonction `stats` de la machine réactive [7]. En moyenne, les optimisations permettent de diminuer le nombre de portes du circuit d'un facteur 3.

Programme	Sans optimisations	Avec optimisations
ABRO [Berry, 2002]	234	73
ABCRO (variante de ABRO)	300	90
Formulaire (voir section 8.1)	325	141
P18 [Berry, 2002]	517	169
Wristwatch [Berry, 1989]	817	270
Minuterie (voir chapitre 3)	1328	329
Traducteur (voir section 8.2)	1419	578
Make -j4 (voir section 8.4)	2420	989
Hiphopfm (voir section 8.3)	2902	777
Prims (100 nombres, voir section 8.5)	14748	4379

FIGURE 6.4 – Portes générées par un programme Hiphop.js.

Le temps de compilation, d'exécution et la consommation mémoire sont des métriques difficiles à comparer à cause de la disparité des environnements JavaScript (clients ou serveurs) et de leurs optimisations (interprétation, puis compilation à la volée). À titre indicatif, le tableau 6.5 indique le temps moyen de réaction en fonction des optimisations de Hiphop.js pour les programmes vus précédemment. Les fonctions JavaScript `console.time` et `console.timeEnd` sont utilisées pour obtenir la durée d'une réaction.

7. Le numéro de « commit » de la distribution Hiphop.js correspondant à la version utilisée pour ces statistiques est `95df055ea52d2e7aa05296bb4ff29a00847fcc99`.

Programme	Sans optimisations	Avec optimisations
ABRO [Berry, 2002]	5	3
ABCRO (variante de ABRO)	5	4
Formulaire (voir section 8.1)	3	3
P18 [Berry, 2002]	2	1
Wristwatch [Berry, 1989]	14	15
Minuterie (voir chapitre 3)	3	2
Traducteur (voir section 8.2)	4	4
Make -j4 (voir section 8.4)	19	10
Hiphopfm (voir section 8.3)	9	6
Prims (100 nombres, voir section 8.5)	24	17

FIGURE 6.5 – Temps (millisecondes) d'une réaction par un programme Hiphop.js (moyenne sur 5 réactions).

Les programmes « P18 » et « jmake » sont exécutés sur le serveur (Hop.js 3.2), tandis que les autres sont exécutés sur le client (Firefox 60). Les optimisations de Hiphop.js semblent significatives lorsque la taille du programme est grande (notamment dans l'exemple « Prims »). De façon générale, ces chiffres montrent que le coût en temps d'exécution de l'implémentation de Hiphop.js par circuits est négligeable dans le contexte du web. Par exemple, une requête HTTP prendra plusieurs dizaines de millisecondes.

Afin de refaire ces expérimentations, les temps de compilation et de réaction peuvent être automatiquement calculés par Hiphop.js via l'API de la machine réactive. Par exemple :

```
const hh = require("hiphop");
const SuspendableTimer = require("./timer.js");
new hh.ReactiveMachine(SuspendableTimer,
                       { traceCompileDuration: true,
                         traceReactDuration: true });
```

Ici, une fois le programme compilé, le temps de compilation est affiché sur la console grâce à l'attribut `traceCompilationDuration`. Le temps de réaction est affiché sur la console après chaque réaction grâce à l'attribut `traceReactDuration`.

6.7 Conclusion

L'ensemble de la chaîne de compilation et d'exécution de Hiphop.js est écrite en JavaScript. La compilation du langage passe par une première étape où les programmes Hiphop.js sont textuellement transformés en un langage intermédiaire reposant sur un arbre XML. À l'exécution du programme JavaScript, cet arbre est évalué et génère l'AST du programme Hiphop.js. Le programme est ensuite compilé en reprenant le modèle de compilation d'Esterel v5 qui repose sur le principe des circuits logiques booléens. Enfin, diverses optimisations sont appliquées afin de contenir la taille du programme exécutable Hiphop.js.

Chapitre 7

Environnement de développement

Sommaire

Hiphop.js dispose d'un débogueur symbolique et distribué. Il permet la visualisation et l'exécution pas à pas d'un programme Hiphop.js quelque soit son environnement d'exécution : l'ordinateur du programmeur, un téléphone ou un objet connecté sans IHM. Dans une première partie la motivation et l'utilisation du débogueur de Hiphop.js sont présentées, elle s'adresse donc à tout programmeur Hiphop.js. Les éléments techniques internes au fonctionnement du débogueur sont ensuite détaillés dans une seconde partie qui s'adresse aux programmeurs désireux de modifier le débogueur et nécessite une compréhension fine des concepts détaillés au chapitre 6.

7.1 Intérêt et particularités

Le but du débogueur de Hiphop.js est de contrôler l'état du programme entre chaque réaction. Cela consiste à permettre la lecture des informations suivantes :

- Quelles étaient les instructions actives à la réaction précédente ?
- Quelles seront les instructions actives dès le début de la prochaine réaction ?
- Où en est le flot de contrôle du programme ?
- Quels sont les signaux (globaux et locaux) émis durant la réaction précédente et quelles sont leurs valeurs ?

Afin de rendre simples la visualisation et l'accès à ces informations, le débogueur est symbolique : le code source du programme est affiché selon un jeu de couleurs indiquant les différents états des instructions. Ensuite, un mode d'exécution pas à pas permet au programmeur de prendre le temps d'analyser le programme entre chaque réaction, notamment dans le contexte où de nombreuses réactions successives sont déclenchées par l'environnement. Enfin, un programme Hiphop.js peut être exécuté par des ordinateurs autres que celui sur lequel le programme est développé : client web distant, téléphone ou autre objet connecté sans IHM. Pour cette raison, le débogueur est distribué et accessible à distance via le web. L'ensemble de ces fonctionnalités sont détaillées dans la section 7.2.

7.2 Utilisation

7.2.1 Visualisation et navigation

Le point d'entrée du débogueur est accessible depuis un navigateur web à l'adresse `http://`*serveur*`/hop/`*nom*, où *serveur* correspond au serveur Hop.js diffusant l'application et *nom* correspond au nom du débogueur, voir section 7.2.3. La page web affiche une représentation du code source du module principal du programme qui indique le nom des instructions, signaux ainsi que les signaux lus dans les expressions [1]. Cet affichage correspond à l'état du programme lors de la dernière réaction, et il est automatiquement mis à jour une fois chaque réaction terminée. L'état des signaux et des instructions est représenté de la façon suivante :

- L'identifiant d'un signal émis à la réaction précédente est écrit en rouge. Celui d'un signal non émis est écrit en bleu. Cliquer sur le nom d'un signal affiche sa valeur si elle est définie.
- Une instruction active à la réaction précédente est écrite sur un fond vert.
- Une instruction active à la réaction précédente et qui n'est pas terminée au début de la prochaine réaction est écrite en couleur marron.

La figure 7.1 montre l'affichage d'un programme par le débogueur. Dans cet exemple, les instructions `MODULE`, `LOOPEACH`, `FORK` ainsi que la première branche du `FORK` sont sur fond vert et écrites en marron : le fond vert indique qu'elles étaient actives à la réaction précédente et la couleur marron indique qu'elles seront actives à la prochaine réaction [2]. En revanche, la seconde branche du `FORK` n'est pas active : `AWAIT(NOW(b))` s'est immédiatement terminée au début de la réaction précédente car le signal `b`, dont l'identifiant est affiché en rouge, était présent.

1. Lorsque le débogueur fût développé, Hiphop.js ne disposait que d'une syntaxe XML rendant la lecture du programme difficile. Une représentation différente était alors un avantage.
2. En termes d'implémentation, cela signifie qu'un registre implémentant une `PAUSE` est à `vrai`. Ici, la `PAUSE` à vrai est dans `AWAIT`.

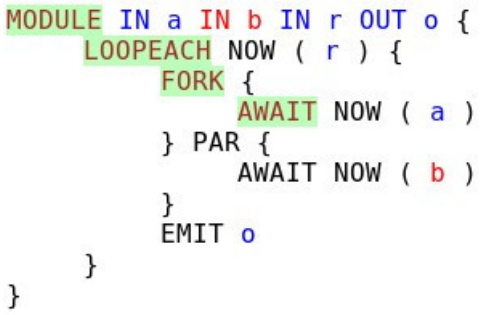

FIGURE 7.1 – Affichage du programme ABRO dans le débogueur.

L'instruction RUN affiche le nom du module appelé, voir la figure 7.2. Ce nom est cliquable : il permet d'ouvrir une nouvelle page affichant le code du module appelé, voir la figure 7.3. Afin de faciliter la navigation entre modules, la partie gauche de la page contient un arbre représentant la hiérarchie des appels de modules sur l'ensemble du programme. Le module visualisé dans la page courante est affiché en gras dans une taille supérieure. Un clic sur le nom d'un module permet de visualiser son corps.

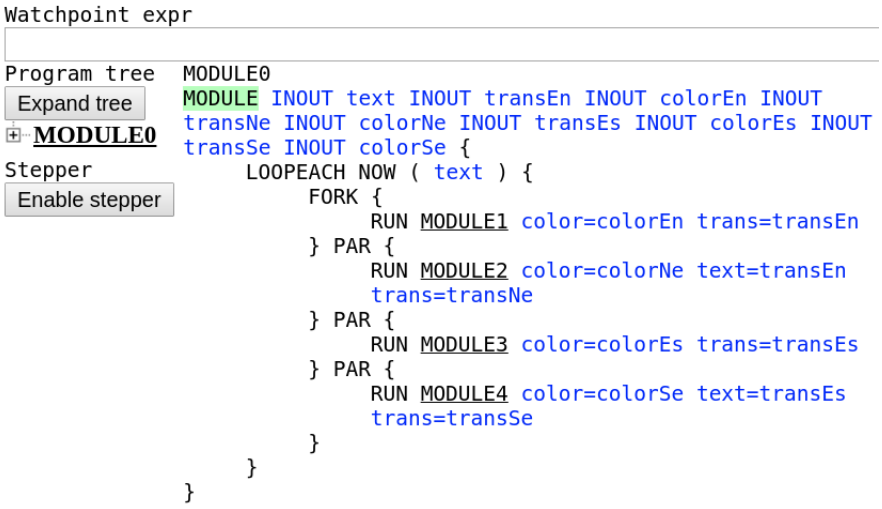

FIGURE 7.2 – Affichage du module principal d'une application appelant quatre autres modules.

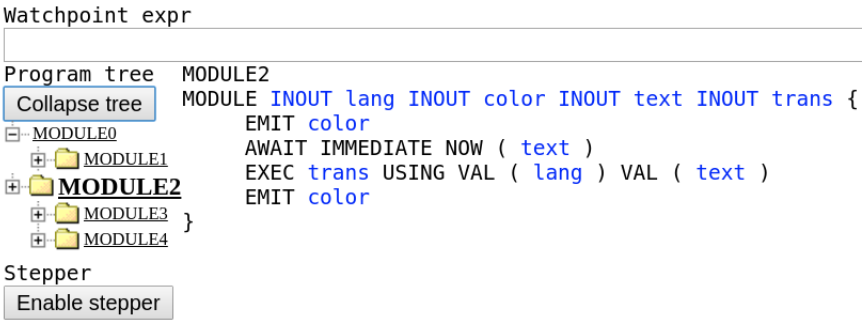

FIGURE 7.3 – Affichage de l'un des modules appelés depuis le module principal après un clic sur « MODULE2 ».

7.2.2 Mode d'exécution pas à pas

Le mode d'exécution pas à pas consiste à exécuter un *pas* sur demande de l'utilisateur. Un pas correspond à l'émission de zéro, de un ou de plusieurs signaux d'entrée depuis l'environnement ainsi qu'à la demande du déclenchement d'une réaction (appel de la méthode `react`, par exemple). Ce mode d'exécution ne concerne donc que le programme Hiphop.js et ne modifie pas l'exécution de l'environnement JavaScript. Lorsque le mode pas à pas est actif, chaque émission de signal depuis l'environnement est enregistrée dans un pas, mais le signal n'est pas émis dans la machine réactive. Ensuite, lorsque la demande de réaction est envoyée, le pas est ajouté dans une file d'attente, mais la réaction n'est pas déclenchée. Les émissions de signaux suivantes depuis l'environnement seront ensuite enregistrées dans un pas suivant. L'exécution d'un pas consiste à appliquer l'ensemble des émissions du pas dans la machine réactive et à déclencher immédiatement après une réaction.

Le mode d'exécution pas à pas peut être enclenché manuellement par un clic sur le bouton « Enable stepper ». Lorsque ce mode est enclenché, le bouton « Enable stepper » se transforme en un bouton « Disable stepper » pour désactiver ce mode, un champ texte numérique apparaît ainsi qu'un bouton « Next step » et deux valeurs :

- La valeur `remains` indique le nombre de pas dans la file.
- La valeur `steps` indique le nombre de pas exécutés.

Un clic sur le bouton « Next step » déclenche immédiatement et atomiquement le nombre de pas indiqué par le champ numérique. La figure 7.4 illustre cette fonctionnalité : au prochain clic sur « Next step », les deux pas dans la file seront exécutés.

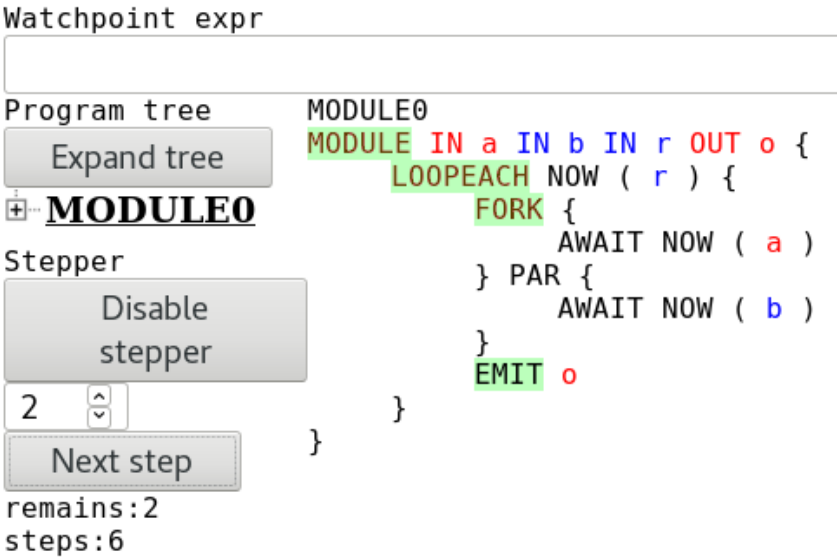

FIGURE 7.4 – IHM du débogueur avec le mode d'exécution pas à pas activé.

Déclenchement automatique du mode pas à pas

Le champ texte « Watchpoint expr » permet au programmeur d'entrer une expression Hiphop.js. Cette expression peut accéder à l'ensemble des signaux globaux du programme via les accesseurs habituels. Elle est évaluée après chaque réaction. Si elle est vraie, alors le mode « pas à pas » est automatiquement enclenché. Considérons le programme ABRO présenté en section 5.5.2 ainsi que l'expression suivante, une fois entrée dans le champ texte « Watchpoint expr » :

```
NOW(a) && NOW(o) && VAL(o) === 3
```

Ici, le mode d'exécution pas à pas sera automatiquement activé lorsque les signaux `a` et `o` sont émis et que la valeur de `o` est 3.

7.2.3 Interface de programmation

Le débogueur est activé de l'une des façons suivantes :

- À la création de la machine réactive :

```
const hh = require("hiphop");
const SuspensableTimer = require("./timer.js");
const m = new hh.ReactiveMachine(SuspendableTimer,
```

```
                                        {debuggerName: "timerDebug"});
```

- Lorsque la machine réactive `m` existe :

```
m.enableDebugger("timerDebug");
```

Dans les deux cas de figure, le débogueur du programme de minuteur présenté au chapitre 3 est alors accessible à l'adresse `http://`*serveur*`/hop/timerDebug`. Il est possible d'activer le débogueur sur plusieurs machines réactives depuis le même serveur, mais chaque débogueur doit avoir un nom différent. Enfin, le débogueur est désactivé par l'appel suivant :

```
m.disableDebugger();
```

7.3 Fonctionnement interne

Cette section présente les éléments importants au fonctionnement interne du débogueur, ils sont utiles à un programmeur souhaitant le modifier. L'architecture globale est d'abord présentée et certains aspects plus techniques sont ensuite détaillés.

7.3.1 Architecture globale

Le débogueur étant distribué, il est composé en trois parties indépendantes interagissant par l'échange de messages. La première partie est liée à la machine réactive. Son rôle consiste à :

- Informer le serveur de l'existence du débogueur sur cette machine réactive et détecter toute erreur provenant du serveur (coupure de connexion, conflit de nom) afin de désactiver le débogueur.
- Envoyer le nouvel état du programme après chaque réaction au serveur, le mécanisme est détaillé en section 7.3.2. À noter que la machine réactive garde une copie de l'AST du programme. Ceci permet d'une part de construire la représentation textuelle initiale du code source, d'autre part d'accéder aux portes implémentant chaque instruction afin de lire leur valeur (les portes sont référencées par le nœud d'AST d'une instruction).
- Gérer le mode d'exécution pas à pas.

La seconde partie est liée au serveur Hiphop.js. Son rôle consiste à :

- Gérer l'ensemble des débogueurs. Puisque chaque machine réactive peut activer un débogueur, cette partie constitue le point central répertoriant les débogueurs actifs. Considérons les machines réactives M_1 et M_2. M_1 active son débogueur nommé D et informe le serveur. Plus tard, M_2 active son débogueur avec le même nom D. Le serveur ordonne alors à M_1 de désactiver son débogueur. Ceci assure l'unicité de nom des débogueurs.
- Définir les services web permettant l'accès depuis un navigateur web au débogueur D depuis l'adresse `http://`*serveur*`/hop/`D.
- Gérer les client connectés à un débogueur. Lorsqu'une machine réactive envoie la mise à jour de l'état du programme, le serveur propage cette mise à jour à l'ensemble des clients connectés. Inversement, lorsqu'un client envoie une demande d'activation du mode d'exécution pas à pas ou une nouvelle expression de « watchpoint », ces messages sont propagés à la machine réactive et aux autres clients connectés.

À noter que comme la première partie est liée à la machine réactive, elle est exécutée là où le programme Hiphop.js est exécuté, donc soit sur le client, soit sur le serveur. Cela n'a aucun impact sur les communications avec le serveur car l'échange de messages repose exclusivement sur le réseau via le protocole WebSocket [Melnikov et Fette, 2011].

Enfin, une troisième partie correspond au programme envoyé sur le client web par le serveur lors de l'accès à `http://`*serveur*`/hop/`D. Elle gère l'affichage de la représentation du code source et des autres parties de l'IHM comme le contrôle du mode d'exécution pas à pas.

7.3.2 Mises à jours incrémentales

Le code source d'un module est représenté dans le débogueur par une liste d'objets JavaScript dont chaque élément représente un mot et regroupe diverses informations. Par exemple :

- Le mot qui doit s'afficher à l'écran, sa couleur et la couleur du fond.
- Une valeur éventuelle, si l'objet représente l'identifiant d'un signal ayant une valeur.
- Un lien, si l'objet représente le nom d'un module appelé dans une instruction `RUN`.

- Un numéro de ligne et une position dans cette ligne.

Lorsque la machine réactive active son débogueur, l'ensemble de cette liste est envoyée, pour chaque module du programme, au serveur. Après chaque réaction, certaines de ces informations doivent être mises à jour, par exemple pour changer la couleur du texte ou du fond d'un identifiant. Renvoyer l'ensemble des listes d'objets représentant le code de chaque module n'est pas raisonnable. En conséquence, seules les modifications à appliquer sur la liste de chaque module sont envoyées de la machine réactive au serveur, et seules les modifications à appliquer sur la liste du module visualisé par un client sont envoyées du serveur au client.

7.3.3 Limitations

L'implémentation actuelle du débogueur a deux limitations :

- L'ajout ou la suppression entre deux réactions de l'instruction `RUN` dans les branches d'un `FORK` n'est pas supportée. Une solution possible est d'étendre le mécanisme de mise à jour de la représentation du code de façon à supporter le renvoi complet de la représentation du programme lors de l'ajout ou du retrait d'une branche (et pas simplement la modification d'une partie du code).
- Les optimisations sur le circuit généré (actives par défaut) empêchent la détection de certaines instructions actives dans le débogueur. En conséquence, il est conseillé de désactiver les optimisations lorsque le débogueur est utilisé. C'est un problème d'implémentation qui pourra être ultérieurement résolu de la façon suivante : lorsqu'une porte est supprimée, il faut remplacer, dans le nœud d'AST de l'instruction, la référence de la porte supprimée par celle de son entrée.

7.4 Conclusion

Le langage Hiphop.js dispose d'un débogueur intégré et adapté à la programmation multitier grâce à son aspect distribué. Le débogueur permet de visualiser l'état d'un programme Hiphop.js entre deux réactions (état de chaque instruction, statut et valeur des signaux). Un mode d'exécution pas à pas permet au programmeur de prendre le temps d'étudier l'état du programme Hiphop.js après chaque réaction, même dans un contexte où de nombreuses réactions se succèdent.

Bien que fonctionnel, le débogueur pourrait être amélioré : d'une part pour afficher le code source réel du programme Hiphop.js (et non pas une simple représentation),

d'autre part pour supporter l'ajout et le retrait de branches d'un parallèle entre deux réactions. Notons cependant que la résolution de ces deux défauts lève un nouveau problème : puisque l'ajout et le retrait de branches d'un parallèle entre deux réactions est dynamique, le code source est inchangé. Il faut donc spécifier un mécanisme permettant de modifier l'affichage du code source afin d'y représenter l'ajout et le retrait dynamique de branches d'un parallèle.

Chapitre 8

Exemples d'applications Hiphop.js

Sommaire

Ce chapitre présente plusieurs applications Hiphop.js. Le but est d'illustrer par la pratique les différents concepts et constructions du langage et de montrer comment ils peuvent être utilisés pour résoudre différents problèmes. La lecture préalable des chapitres 3, 4 et 5 est nécessaire, ainsi que des connaissances HTML et CSS basiques. Le code source complet de chaque exemple se trouve dans le dossier `examples/` de la distribution Hiphop.js.

8.1 Validation automatique d'un formulaire

Le formulaire est un simple champ texte dont la bordure est initialement noire. Lorsque l'utilisateur modifie le texte, la bordure devient orange. Après un court délai où l'utilisateur n'a pas modifié le texte, le programme soumet le texte au serveur. Le serveur répond `right` si le texte est valide et la bordure devient verte. Autrement, le serveur répond `wrong` et la bordure devient rouge. Le code Hiphop.js est le suivant :

```
const hh = require("hiphop");
const timeoutModule = require("./timeoutModule.js");
const prg = MODULE (IN text, OUT status, OUT error) {
  IN text;
  OUT status, error;
  EVERY(NOW(text)) {
    EMIT status("checking");
    RUN(timeoutModule(500));
    PROMISE status, error check(VAL(text));
  }
};
const m = new hh.ReactiveMachine(prg);
```

Le code HTML de l'IHM est un simple champ texte :

```
<input type="text"
       class=~{m.value.status}
       oninput=~{m.inputAndReact('text', this.value)}/>
```

Lorsque la valeur du champ texte est modifiée, le signal d'entrée `text` est émis avec pour valeur le texte du champ et une réaction est déclenchée (attribut `oninput` du champ texte). Le corps de `EVERY` démarre, le signal de sortie `status` est émis avec la valeur « checking », une attente de 500 millisecondes est démarrée et la réaction se termine. La valeur émise dans `status` est le nom de la classe CSS `checking` définissant la bordure d'un champ texte de couleur orange. Le code CSS utilisé par l'IHM est :

```
input { border:5px solid black; }
.right { border-color: green; }
.checking { border-color: orange; }
.wrong { border-color: red; }
```

L'attribut `class` du champ texte est défini par la valeur que retourne le proxy réactif lié à `status`. Initialement, `status` est indéfini donc aucune classe CSS ne modifie la couleur de la bordure qui est noire par défaut. L'émission de `status` entraine la modification de la valeur de l'attribut `class` par `checking`. La couleur de la bordure devient donc automatiquement orange. Lorsque l'attente 500 millisecondes se termine une réaction est automatiquement déclenchée et `PROMISE` démarre en appelant la fonction JavaScript `check` :

```
function check(data) {
  const xhr = new XMLHttpRequest();
  const svc = "http://localhost:8080/hop/checkSvc?data=" + data;
  return new Promise((resolve, reject) => {
    xhr.onreadystatechange = () => {
      if (xhr.readyState == 4 && xhr.status == 200) {
        resolve(xhr.responseText);
      }
    };
    xhr.open("GET", svc, true);
    xhr.send();
  });
}
```

Une requête est envoyée à un serveur pour déterminer si la valeur du champ texte est valide et retourne une promesse. En attendant sa résolution ou son rejet, `PROMISE` agit comme `AWAIT` et la réaction se termine. Lorsque la réponse du serveur est reçue, la promesse est résolue (par l'appel à `resolve` dans `check`). Ceci a pour effet de déclencher une nouvelle réaction où `PROMISE` se termine en émettant `status` avec pour valeur la réponse du serveur. Le nom de la classe CSS utilisée par le champ texte est remplacé par la nouvelle valeur de `status`. Elle correspond à `right` si la valeur du champ est valide et la bordure devient de couleur verte. Autrement, elle correspond à `wrong` et la bordure devient de couleur rouge.

Que se passe-t-il lorsque l'utilisateur modifie le champ texte alors que le programme Hiphop.js est en train d'attendre l'écoulement des 500 millisecondes ou la réponse du serveur ? L'attente est interrompue par `EVERY`, c'est-à-dire que l'instruction `RUN` ou `PROMISE` est préemptée et le corps de `EVERY` redémarre une nouvelle itération. La terminaison de l'attente de 500 millisecondes ou la résolution d'une promesse associée à une ancienne itération de `EVERY` sont simplement ignorées par le programme. Il existe des cas où il faut interrompre l'opération asynchrone et pas simplement ignorer

sa réponse. Ces cas d'usage sont présentés aux sections 8.3 et 8.4.

Une implémentation purement JavaScript de cet exemple aurait nécessité l'utilisation d'une variable globale qui identifie de façon unique la dernière requête envoyée. Chaque réponse reçue et dont l'identifiant ne serait pas égal à celui de la dernière requête serait alors ignorée.

Ce type de programme est généralisable à une séquence d'opérations pouvant prendre du temps à s'exécuter et qui peut être annulée à tout instant. Il illustre quatre aspects importants de Hiphop.js :

- L'attente synchrone qui est bloquante pour le programme Hiphop.js lorsque le programme attend 500 millisecondes ou la réponse du serveur.
- L'intégration de Hiphop.js avec les promesses JavaScript, ce qui permet de gérer des mécanismes asynchrones à travers une simple instruction Hiphop.js non instantanée.
- La possibilité d'interrompre les attentes par l'instruction de préemption `EVERY`.
- L'intégration de Hiphop.js avec l'IHM d'une application grâce aux proxies réactifs de Hop.js.

Enfin, à noter que le signal de sortie `error` correspondant au cas de rejet de la promesse n'est pas utilisé. Il pourrait l'être, par exemple, pour signaler à l'utilisateur une erreur de communication avec le serveur dans une fenêtre « popup » :

```
m.addEventListener("error", evt => {
  alert("An error has occured: " + evt.signalValue);
});
```

Ici, à chaque réaction où le signal `error` est émis (en cas de rejet de la promesse), une alerte est affichée dans la fenêtre du navigateur. Cette alerte indique qu'une erreur s'est produite et affiche le message d'erreur (la valeur du signal `error`).

8.2 Traduction automatique et parallèle

L'application de traducteur présente un champ texte à l'utilisateur. À chaque modification du champ texte, deux requêtes sont envoyées en même temps à un serveur web. La première demande une traduction du français vers l'anglais, et la seconde du français vers l'espagnol. Lors de la réception de la réponse de chacune de ces requêtes, la traduction est affichée et une nouvelle requête de traduction inverse est envoyée au serveur afin

de retraduire la traduction vers le français. La retraduction vers le français est affichée dès réception. Il y a donc au total quatre résultats affichés à l'écran :

- La traduction du français vers l'anglais ;
- La retraduction de la traduction anglaise vers le français ;
- La traduction du français vers l'espagnol ;
- La retraduction de la traduction espagnole vers le français. ;

Lorsqu'une traduction est reçue, elle est affichée en vert. La figure 8.1 est une capture d'écran de l'application où toutes les traductions sont reçues. En revanche, si l'utilisateur modifie le texte d'entrée à traduire, toutes les traductions sont immédiatement affichées en rouge pour indiquer qu'elles ne correspondent plus au texte à traduire.

Par exemple, dans la figure 8.2 le suffixe « on » a été retiré de « maison ». La traduction anglaise et espagnole ont été reçues mais pas les retraductions françaises. Enfin, la réception d'une traduction correspondant à une ancienne version du texte est ignorée, notamment dans le cas où l'utilisateur modifie le texte à traduire avant que toutes les traductions ne soient connues.

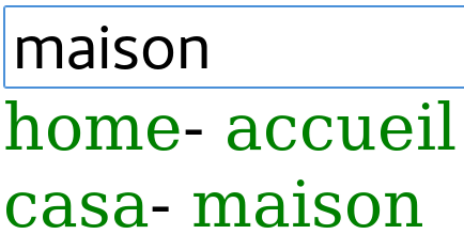

FIGURE 8.1 – Les traductions correspondent au mot à traduire.

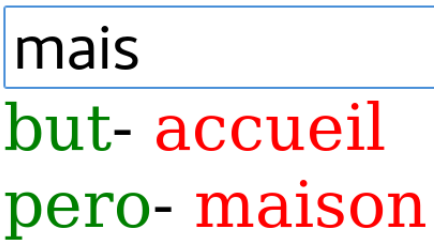

FIGURE 8.2 – Les retraductions françaises ne correspondent pas au mot à traduire.

Commençons par implémenter un sous ensemble de la spécification. Le code d'un module gérant une unique traduction avec le changement de sa couleur est :

```
function promiseColor(langPair) {
  return MODULE (IN text, OUT color, OUT trans, OUT error) {
    EMIT color("red");
    AWAIT IMMEDIATE(NOW(text));
    PROMISE trans, error translate(langPair, VAL(text));
    EMIT color("green");
  }
}
```

Le signal d'entrée `text` correspond au texte à traduire, les signaux de sorties `color` et `trans` correspondent respectivement à la couleur et au texte de la traduction. À l'exécution de ce module, la couleur d'une éventuelle traduction existante devient rouge et `AWAIT` agit comme une pause jusqu'à ce que le texte à traduire soit modifié. À noter l'utilisation de `IMMEDIATE` permettant à `AWAIT` de se terminer immédiatement s'il démarre et que `text` est présent. À l'instant où `AWAIT` se termine, une requête est envoyée au serveur par l'intermédiaire de `PROMISE` et de la fonction `translate`. Cette fonction prend deux paramètres : `langPair` indique la langue source et destination, par exemple « fr|en » pour traduire du français vers l'anglais. Le second paramètre est le texte à traduire. Quand la traduction est reçue, la promesse retournée par `translate` est résolue : `PROMISE` se termine en émettant la traduction dans `trans` et la traduction devient de couleur verte. Ce module est construit à l'appel de la fonction `promiseColor` en spécifiant la paire de langue source et destination. Ceci rend le module générique, ce qui permet ensuite de le construire et l'inclure quatre fois dans un autre module gérant les quatre traductions :

```
const hh = require("hiphop");
const prg = MODULE (IN text,
                    OUT transEn, OUT colorEn,
                    OUT transNe, OUT colorNe,
                    OUT transEs, OUT colorEs,
                    OUT transSe, OUT colorSe) {
  EVERY IMMEDIATE(NOW(text)) {
    FORK {
      RUN(promiseColor("fr|en"), color=colorEn, trans=transEn);
    } PAR {
      RUN(promiseColor("en|fr"), color=colorNe, text=transEn,
                                 trans=transNe);
    } PAR {
      RUN(promiseColor("fr|es"), color=colorEs, trans=transEs);
    } PAR {
      RUN(promiseColor("es|fr"), color=colorSe, text=transEs,
                                 trans=transSe);
    }
  }
}
const m = new hh.ReactiveMachine(prg);
```

Ce module est le point d'entrée Hiphop.js de l'application de traduction. Le signal d'entrée `text` correspond au texte à traduire, les signaux de sortie sont :

- `transEn` et `colorEn`, ils correspondent à la traduction anglaise et sa couleur ;
- `transNe` et `colorNe`, ils correspondent à la retraduction de l'anglais vers le français et sa couleur ;
- `transEs` et `colorEs`, ils correspondent à la traduction espagnole et sa couleur :
- `transSe` et `colorSe`, ils correspondent à la retraduction de l'espagnol vers le français et sa couleur.

L'instruction `FORK` définit quatre branches parallèles, chacune gérant une traduction différente. Les branches sont composées d'une unique instruction `RUN` dont le premier paramètre est un module construit à la compilation du programme par l'appel à `promiseColor`. La première branche gère la traduction du français vers l'anglais :

- Le paramètre passé à `promiseColor` est `fr|en`.
- Le signal `text` du module appelé est implicitement lié à `text` du module global. Il correspond donc au texte (français) à traduire.
- Le signal `trans` du module appelé est explicitement lié à `transEn`, donc toute émission de `trans` dans le module appelé émet également `transEn`. Même remarque pour `color` du module appelé et `colorEn`.

La seconde branche retraduit la traduction anglaise, lorsqu'elle est reçue, vers le français. Contrairement à la première branche, le signal `text` du module appelé est explicitement lié à `transEn`. En effet, ce module ne doit pas traduire le texte entré par l'utilisateur, mais la traduction anglaise correspondant à la valeur de `transEn`. Ainsi, à l'instant où la traduction anglaise est connue, `transEn` est émis, l'instruction `AWAIT IMMEDIATE(NOW(text))` du second module se termine et la requête de retraduction de l'anglais vers le français est envoyée au serveur. La troisième et quatrième branche sont analogues à la première et seconde, mais pour la langue espagnole. Enfin, toutes les branches sont démarrées à chaque réaction où le texte à traduire est modifié. La préemption de `EVERY` assure par construction que chaque traduction reçue correspond au texte à traduire. Il s'agit là d'un apport important de Hiphop.js : les réceptions de traductions correspondant à une ancienne version du texte à traduire sont automatiquement ignorées lorsque `PROMISE` est préemptée par `EVERY`.

Deux remarques permettent de comprendre l'utilisation de l'option `IMMEDIATE` avec `EVERY` et `AWAIT` :

- Si le signal `text` est présent dès la première réaction du programme (avec un texte à traduire comme valeur), il faut immédiatement démarrer le corps de `EVERY` car autrement la première traduction ne sera pas déclenchée. Pour cette raison le mot clé `IMMEDIATE` est utilisé : cela permet l'évaluation de l'expression du délai dès le démarrage de `EVERY` et permet donc la terminaison du délai si `text` est présent, voir section 4.4.4.

- Dans le cas du module retourné par `promiseColor`, `AWAIT` permet d'attendre la traduction anglaise et espagnole avant de démarrer les retraductions vers le français dans la deuxième et quatrième branche. En revanche, lorsque la première et la troisième branche sont démarrées, c'est parce que `text` est présent : il y a donc un nouveau texte à traduire et il faut démarrer immédiatement les traductions vers l'anglais et l'espagnol. Pour cette raison, `IMMEDIATE` permet la terminaison immédiate de `AWAIT`.

Une séquence existe entre la traduction française/anglaise et la traduction inverse anglaise/française (ainsi que pour les traductions avec l'espagnol) puisqu'il faut attendre que la traduction se termine avant de démarrer la traduction inverse. Pourtant, au lieu de démarrer deux branches parallèles (une pour les traductions avec l'anglais, une pour les traductions avec l'espagnol) dont chacune contient deux `RUN` en séquence (chacun exécutant `promiseColor`), l'implémentation démarre quatre branches parallèles et la séquence entre une traduction et sa traduction inverse est visible au niveau du `AWAIT` du `promiseColor` de la traduction inverse. Démarrer les quatre modules `promiseColor` est indispensable : la première chose que fait le module est de mettre en rouge l'affichage existant de la traduction qu'il gère. Si le module gérant la traduction inverse est démarré après celui de la traduction initiale, alors l'affichage existant de la traduction inverse restera de couleur verte lorsque le texte à traduire change et ne deviendra rouge que lorsque la traduction sera reçue, juste avant de démarrer la traduction inverse. Le choix de gérer les changements de couleur et la requête au service web de traduction dans un même module rend le programme plus composable : il suffit d'ajouter une unique instruction `RUN` pour ajouter une traduction dans le programme. Autrement, les modifications pour ajouter une nouvelle traduction seraient plus invasives : il faudrait modifier une partie du code pour gérer les changements de couleur d'une nouvelle traduction et modifier une autre partie du code pour gérer la requête au service web pour la nouvelle traduction.

Enfin, l'implémentation de l'IHM du traducteur est :

```
<input oninput=~{m.inputAndReact("text", this.value)}/>
<div>
  <span style=~{`color: ${m.value.colorEn}`}>
    <react>~{m.value.transEn}</react>
  </span> -
  <span style=~{`color: ${m.value.colorNe}`}>
    <react>~{m.value.transNe}</react>
  </span>
</div>
<div>
  <span style=~{`color: ${m.value.colorEs}`}>
    <react>~{m.value.transEs}</react>
  </span> -
  <span style=~{`color: ${m.value.colorSe}`}>
    <react>~{m.value.transSe}</react>
  </span>
</div>
```

Ce code illustre l'utilisation de proxies réactifs dans les littéraux de modèles [1] (souvent appelés *template literals*). Dans le contexte des littéraux de modèles, la construction `${`*js-expression*`}` permet, au moment de la construction de la chaîne de caractères, d'y injecter la valeur retournée par l'évaluation *js-expression*. Ici, cette construction est utilisée pour injecter un proxy réactif. Grâce à ce proxy réactif, chaque changement de la valeur d'un signal indiquant la couleur d'une traduction entraînera la reconstruction automatique de la chaîne de caractères définissant le style (et donc la couleur) de la traduction.

Cette application de traducteur, comme celle du formulaire, illustre plusieurs principes de Hiphop.js comme l'attente synchrone, la préemption et l'intégration avec l'IHM. Elle a pour but d'illustrer plus précisément la construction de modules génériques, leur réutilisation dans des modules avec `RUN` ainsi que l'utilisation massive de code parallèle synchrone, où les branches se synchronisent automatiquement et dynamiquement par l'attente et l'émission de signaux.

8.3 Module de lecture audio ou vidéo

La bibliothèque standard de JavaScript dispose d'objets `Audio` et `Video` permettant la lecture de fichiers audio et vidéo par le navigateur. Ces objets disposent d'une API asynchrone permettant de charger un fichier, de le jouer, de sauter à une position précise, de savoir si une erreur s'est produite, de connaître la durée du média, etc. Il est possible d'encapsuler ce type d'objet dans un module Hiphop.js assez générique pour être utilisé

1. https://www.ecma-international.org/ecma-262/6.0/#sec-template-literals

comme une mini bibliothèque de lecture audio. Le code du module est :

```
const timeoutModule = require("./timeoutModule.js");
MODULE (IN src, IN pausePlay, IN seekTo,
        OUT loaded, OUT error,
        OUT paused, OUT trackEnded,
        OUT position, OUT duration) {
  TRAP Terminated {
    PROMISE loaded, error load(VAL(src));
    IF (NOW(error)) {
      EXIT Terminated;
    }
    FORK {
      SUSPEND TOGGLE(NOW(pausePlay))
              EMITWHENSUSPENDED paused {
        FORK {
          PROMISE trackEnded, error start()
                  ONRES audio.play()
                  ONSUSP audio.pause()
                  ONKILL audio.pause();
          EXIT Terminated;
        } PAR {
           LOOP {
             RUN(timeoutModule(1000));
             EMIT position(audio.currentTime);
             EMIT duration(audio.duration);
           }
         }
      }
    } PAR {
      EVERY(NOW(seekTo)) {
        ATOM {
          audio.currentTime = VAL(seekTo);
        }
        EMIT position(audio.currentTime);
      }
    }
  }
}
```

La valeur du signal d'entrée `src` correspond au nom du fichier à lire. Elle doit être connue au démarrage du module. La réception de `pausePlay` permet de mettre en pause la lecture et de la redémarrer, la réception de `seekTo` permet de sauter à une position du fichier (la position est la valeur du signal). Le signal de sortie `loaded` est émis lorsque le fichier audio est chargé par le navigateur et donc prêt à être joué, `error` est émis lorsqu'une erreur se produit lors du chargement ou de la lecture du fichier, `paused` est émis à chaque réaction où la lecture est en pause, `trackEnded`

est émis lorsque la lecture est terminée. Enfin, `position` et `duration` indiquent respectivement la position actuelle et la durée du fichier en cours de lecture. Lorsque le module démarre, une fonction `load` (détaillée par la suite) est appelée via `PROMISE`. Cette fonction déclenche le chargement du fichier audio à lire et retourne une promesse qui est résolue lorsque le fichier est correctement chargé par le navigateur. Dans le cas d'une erreur, la promesse est rejetée et l'instruction `PROMISE` ayant exécuté `load` se termine en émettant le signal `error` : l'instruction `EXIT` est alors démarrée ce qui permet la terminaison de l'ensemble du module. Dans le cas d'un chargement avec succès, plusieurs mécanismes parallèles sont alors démarrés :

- Une fonction `start` (détaillée par la suite) est appelée via `PROMISE`. Cette fonction démarre immédiatement la lecture du morceau et retourne une promesse qui est résolue lorsque la lecture est terminée (fin de morceau). À cet instant, l'instruction `PROMISE` qui a démarré la lecture se termine et le démarrage de l'instruction `EXIT` permet la terminaison de l'ensemble du module.
- Une boucle déclenche chaque seconde une réaction dans laquelle la position courante et la durée du morceau (lues depuis l'objet `Audio`, introduit par la suite) sont émises dans les signaux de sortie `position` et `duration`. Ces émissions permettent, par exemple, d'ajuster un curseur dans une IHM.
- Une boucle attend l'émission du signal d'entrée `seekTo` pour sauter à un position du morceau. Si c'est le cas, la position est modifiée dans l'objet `Audio` et le signal `position` est émis pour correspondre à la nouvelle position.

Par ailleurs, une instruction `SUSPEND` englobe l'instruction `PROMISE` gérant la lecture ainsi que la boucle mettant à jour la position et la durée du morceau. Ceci garantit par construction que lorsque le signal `pausePlay` est émis, la lecture est mise en pause. Le signal de sortie `paused` est alors émis à chaque réaction où la lecture est en pause grâce à l'option `EMITWHENSUSPENDED`, par exemple pour modifier l'IHM en conséquence.

À noter que la boucle permettant de mettre à jour la position de lecture, lorsque `seekTo` est émis, n'est pas incluse dans la suspension. Ainsi, il est possible de changer la position même quand la lecture est suspendue. Il s'agit d'un choix arbitraire, mais il serait très simple d'empêcher le changement de position lorsque la lecture est suspendue : il suffirait de déplacer la boucle `EVERY(NOW(seekTo))` dans le corps de `SUSPEND`. Ceci illustre les garanties temporelles apportées par les constructions de Hiphop.js.

Afin de masquer l'objet `Audio` au programmeur et de lier automatiquement ses gestionnaires d'évènements au programme Hiphop.js, le module est construit dans la fonction suivante :

```
function audioModule() {
  const audio = new Audio();
  let loadedListener = undefined;
  let endedListener = undefined;
  let errorListener = undefined;

  function load(src) {
    audio.src = src;
    return new Promise((resolve, reject) => {
      if (loadedListener) {
        audio.removeEventListener("onloadeddata", loadedListener);
      }
      loadedListener = () => resolve();
      audio.addEventListener("loadedListener", loadedListener);

      if (errorListener) {
        audio.removeEventListener("error", errorListener);
      }
      errorListener = (errorMsg) => reject(errorMsg);
      audio.addEventListener("error", errorListener);
    });
  }

  function start() {
    audio.play()
    return new Promise((resolve, reject) => {
      if (endedListener) {
        audio.removeEventListener("ended", endedListener);
      }
      endedListener = () => resolve();
      audio.addEventListener("ended", endedListener);
    });
  }

  return MODULE (IN src, IN pausePlay, IN seekTo,
                 OUT loaded, OUT error,
                 OUT paused, OUT trackEnded,
                 OUT position, OUT duration) {
    ...
  }
}
```

La fonction `audioModule` est appelée lors de la construction d'un programme Hiphop.js. Par exemple :

```
MODULE simpleAudioPlayer (IN src) {
  EVERY(NOW(src)) {
    RUN(audioModule())
  }
}
```

À la compilation du programme, la fonction `audioModule` est appelée : une instance de l'objet `Audio` JavaScript est créée ainsi que le module Hiphop.js précédemment défini. Le signal d'entrée `src` de `simpleAudioPlayer` est implicitement lié au signal d'entrée `src` du module retourné par l'appel à `audioModule`.

Ce programme est un lecteur audio très simple : à chaque fois que le signal d'entrée `src` est émis, le corps du `EVERY` est démarré, donc le chargement puis la lecture d'un fichier audio est démarrée. À cet instant, les gestionnaires d'évènements de terminaison de lecture ou d'erreur de chargement du fichier sont dynamiquement créés et connectés à l'objet audio via les méthodes `load` et `start`. Les gestionnaires d'évènements créés lors d'une précédente exécution du corps de `EVERY` sont inutiles, il est donc préférable de les enlever : cela évite de multiplier des appels de fonctions depuis la boucle d'évènements et permet aussi de libérer la mémoire. À noter cependant que la suppression de ces anciens gestionnaires d'évènements n'est pas obligatoire pour garantir la correction du programme :

- Soit la promesse (dans `load` ou `start`) ayant créé et connecté ces gestionnaires d'évènements est terminée, les gestionnaire d'évènements ne seront donc jamais appelés.
- Soit `PROMISE` est préemptée par `TRAP` ou par la préemption `EVERY` qui englobe `RUN(audioModule())` : la résolution ou le rejet de la promesse encapsulée par `PROMISE` est donc ignorée par Hiphop.js (et n'a donc aucun effet sur le programme).

Cet exemple illustre deux aspects de Hiphop.js. D'abord, il montre comment Hiphop.js masque entièrement l'API asynchrone d'un objet par encapsulation et fournit au programmeur un module Hiphop.js standard et réutilisable. Ensuite, il montre que les garanties temporelles qu'offre le langage par construction sont importantes : ici, il est possible de modifier le comportement d'une fonctionnalité par simple déplacement syntaxique d'une instruction, sans influence sur le comportement global de l'application (déplacement de la boucle `EVERY(NOW(seekTo))` dans le corps de `SUSPEND` pour empêcher le changement de la position de lecture lorsque la lecture est suspendue).

8.4 Une version simplifiée de `make -j`

La commande `make -j`N parallélise sur N processeurs les commandes appelées depuis un $Makefile$, si c'est possible. Cette application Hiphop.js reprend le même principe, appliqué à la compilation de fichiers C. Une liste de fichiers à compiler est donnée ainsi que le nombre de compilations parallèles possibles. Chaque `.c` est compilé si le `.o` associé n'existe pas ou si sa date de modification est inférieure à celle du `.c`. Lorsque tous les `.o` sont générés sans erreurs, l'édition de liens est appliquée. En cas d'erreur, toutes les compilations en cours sont immédiatement arrêtées. Le point d'entrée est une fonction JavaScript `make` :

```
function make(files, nCPU) {
  const nextFile = (() => {
    let nextId = 0;
    return () => files[nextId++];
  })();

  const m = new hh.ReactiveMachine(
    MODULE (OUT stdout, OUT stderr) {
      WEAKABORT(NOW(stderr)) {
        ${compileN(nCPU)}
        PROMISE stdout, stderr link(files);
      }
    }
  );

  m.addEventListener("stdout", evt => console.log(evt.signalValue));
  m.addEventListener("stderr", evt => console.error(evt.signalValue));
}
```

Cette fonction construit un programme Hiphop.js et génère une machine réactive. Le programme définit deux signaux de sorties dont les émissions sont respectivement affichées sur la sortie standard et la sortie d'erreur. Le paramètre `files` représente la liste des fichiers à compiler et `nCPU` le nombre maximal de compilations parallèles. Un appel à `make` peut être :

```
make([
       {src: "a.c", obj: "a.o"},
       {src: "b.c", obj: "b.o"},
       {src: "c.c", obj: "c.o"}
     ], 2);
```

Dans cet exemple, trois fichiers `a.c`, `b.c` et `c.c` sont à compiler, leur compilation générant respectivement les fichiers objets `a.o`, `b.o` et `c.o`. Comme `nCPU = 2`, `a.c` et `b.c` sont compilés en parallèle, puis lorsque l'une des deux compilations se

termine `c.c` est compilé. Si une erreur se produit, `stderr` est émis et l'instruction `WEAKABORT` interrompt les compilations actives, empêche les compilations suivantes ainsi que l'édition de liens. La fonction `compileN` injecte dans le module principal les instructions Hiphop.js gérant les compilations :

```
function compileN(nCPU) {
  if (nCPU > 0) {
    return FORK {
      FOR(mustCompile, file(nextFile());
          VAL(file);
          EMIT file(nextFile())) {
        PROMISE mustCompile, stderr cmpDate(VAL(file).src,
                                            VAL(file).obj);
        IF (VAL(mustCompile)) {
          PROMISE stdout, stderr compile(VAL(file), THIS)
                  ONKILL THIS.child.kill("SIGKILL");
        }
      }
    } PAR {
      ${compileN(nCPU - 1)}
    }
  } else {
    return NOTHING;
  }
}
```

Ce code récursif construit une instruction `FORK` ayant `nCPU` branches. Chaque branche est identique, il s'agit d'une boucle `FOR` qui gère la compilation d'un seul fichier à la fois, jusqu'à ce que tous les fichiers soient compilés. Dans la première partie du `FOR`, deux signaux locaux `mustCompile` et `file` sont créés et `file` est initialisé avec l'éventuel prochain fichier à compiler. La seconde partie de `FOR` est ensuite évaluée : si il y a un fichier à compiler, alors `VAL(file)` n'est pas `undefined`, le corps de `FOR` est donc exécuté. La date de modification du `.c` est alors comparée avec sa version objet éventuelle par la fonction `cmpDate` :

```
const fs = require("fs");
function cmpDate(fx, fy) {
  return new Promise((resolve, reject) => {
    fs.stat(fx, (err, sx) => {
      if (err) {
        reject(err);
      } else {
        fs.stat(fy, (err, sy) => {
          resolve(!!err || (sx.mtime > sy.mtime));
        });
      }
    }); }); }
```

Cette fonction accède au système de fichier afin de comparer les dates du fichier `.c` et du `.o` éventuel via des appels d'une fonction de bibliothèque JavaScript qui est asynchrone. L'ensemble est encapsulé dans une promesse afin de pouvoir l'appeler directement depuis Hiphop.js via `PROMISE`. Si la promesse est rejetée, c'est que le fichier source n'existe pas : `stderr` est émis, l'erreur est affichée et le programme s'arrête. Si la promesse est résolue, `mustCompile` est émis avec pour valeur le résultat booléen de la comparaison de date. Dans le cas où la date de modification du fichier `.c` est supérieure à celle du fichier `.o` (ou si le `.o` n'existe pas), la compilation est démarrée par la fonction `compile` appelée via `PROMISE`. Puisque `PROMISE` agit comme une pause jusqu'à la terminaison de la compilation, le corps de `FOR` ne se termine qu'une fois la compilation terminée. À cet instant, la troisième partie est exécutée en émettant `file` avec pour valeur un éventuel fichier à compiler. L'itération continue jusqu'à ce que `file` soit `undefined` ou que la préemption englobante de `WEAKABORT` l'interrompe ainsi que l'ensemble des autres branches de `FORK`. À noter l'utilisation de `WEAKABORT` : puisque la préemption est appliquée lorsque `stderr` est émis depuis le corps de `WEAKABORT`, `ABORT` ne peut pas être utilisée puisqu'elle empêcherait l'exécution de son corps et donc l'émission de `stderr`. Cela causerait un cycle de causalité.

La compilation d'un fichier `.c` nécessite l'appel d'une application externe. La bibliothèque JavaScript dispose de la fonction asynchrone `spawn` permettant d'exécuter un processus externe parallèle. Des gestionnaires d'évènements permettent de rediriger la sortie standard et d'erreur du processus externe ainsi que de notifier sa terminaison. Puisque `spawn` est aussi utilisé pour l'édition de lien, la fonction `exec` l'encapsule dans une promesse générique utilisable pour exécuter n'importe quel programme depuis une instruction `PROMISE` :

```
const spawn = require("child_process").spawn;
function exec(cmd, opts, thisPromise=undefined) {
  return new Promise((resolve, reject) => {
    const child = spawn(cmd, opts);
    let log = "";
    child.stdout.on("data", data => log += data);
    child.stderr.on("data", data => log += data);
    child.on("close", status =>
                      status ? reject(log) : resolve(log));
    if (thisPromise) {
      thisPromise.child = child;
    }
  });
}
```

Le gestionnaire d'évènement `close` est appelé à la terminaison du processus. La variable `status` retourne le code de terminaison du processus, permettant de résoudre la promesse s'il est nul (terminaison avec succès) ou de la rejeter (terminaison avec

erreur). La variable `thisPromise` correspond à l'objet `THIS`. Elle est utilisée pour accéder à l'objet `child` représentant le processus de compilation depuis l'instruction `ONKILL` de `PROMISE` : lorsqu'une erreur (émission de `stderr`) se produit, toutes les branches du parallèle sont préemptées, y compris l'instruction `PROMISE` contrôlant le processus externe de compilation. L'instruction `ONKILL` est appelée lors de la préemption de `PROMISE`. Ici, elle permet l'interruption du processus externe via le signal Posix `SIGKILL`. La fonction `compile` est :

```
function compile(file, thisPromise) {
  return exec("gcc", ["-c", file.src, "-o", file.obj], thisPromise);
}
```

Enfin, lorsque tous les fichiers sont compilés sans erreur, les branches parallèles se terminent toutes et `FORK` termine. L'édition de liens est alors appliquée par un processus externe, comme pour la compilation. La fonction `link` est :

```
function link(files) {
  return exec("gcc", files.map(file => file.obj));
}
```

Cet exemple permet d'illustrer l'intégration poussée de Hiphop.js avec des mécanismes asynchrones comme l'accès et le contrôle de fonctions systèmes par l'intermédiaire de `PROMISE` et de bibliothèques JavaScript. Cela montre aussi l'intérêt de pouvoir passer des valeurs dans les différentes instructions de `PROMISE` par l'intermédiaire de `THIS`. L'exemple illustre par ailleurs la nécessité de construire un programme Hiphop.js dynamiquement, en fonction de valeurs connues uniquement lors de l'exécution du programme JavaScript. Dans cet exemple, les valeurs connues lors de l'exécution du programme sont la liste des fichiers à compiler et le nombre de compilations en parallèle.

8.5 Les nombres premiers

Cette application est une implémentation de la *machine chimique* [Berry et Boudol, 1992, Boussinot et Laneve, 1995, Mandel et Pouzet, 2008]. Il s'agit d'une animation où des nombres se déplacent constamment dans un cadre. Initialement seul « 2 » est présent. Des boutons permettent d'ajouter un ou plusieurs nombres dans le cadre. La valeur de chaque nombre ajouté est incrémentée en fonction de celle du nombre précédemment ajouté. Les nombres divisibles par d'autres nombres présents sont appelés « proies » et sont affichés en rouge pour aider à suivre l'exécution. Les nombres se déplacent de façon aléatoire, mais lorsqu'un diviseur est à proximité d'une proie il se met à la suivre. La proie est supprimée quand le diviseur la touche, la taille d'affichage du diviseur est alors grossie pour indiquer qu'il a « mangé » une

proie. Après un moment d'exécution, seuls les nombres premiers sont donc affichés et ils se déplacent aléatoirement. La figure 8.3 est une capture d'écran de l'application où les proies « 4 » et « 6 » attirent les diviseurs « 2 » et « 3 ». La figure 8.4 montre des diviseurs dont la taille d'affichage a augmenté après avoir « attrapé » quelques proies.

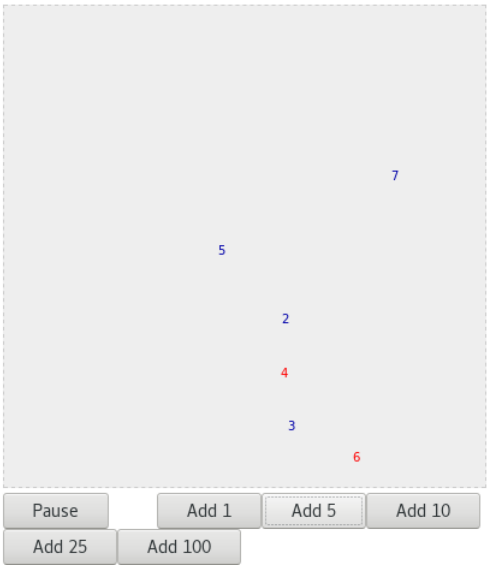

FIGURE 8.3 – L'application en début d'exécution après un clic sur « Add 5 ».

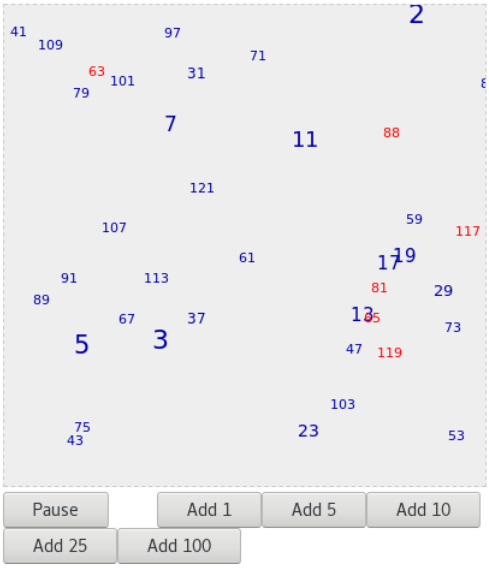

FIGURE 8.4 – L'application quelques instants après un clic sur « Add 100 ».

Le point d'entrée du programme Hiphop.js orchestrant l'application est :

```
function reduce(acc, n) {
  if (n) {
    acc.push(n);
  }
  return acc;
}

MODULE (OUT numbers COMBINE (reduce),
        OUT killn COMBINE (reduce)) {
  FORK par {
    LOOP {
      ATOM {
        clearRect();
      }
      EMIT numbers([]);
      EMIT killn([]);
      PAUSE;
    }
  } PAR {
    ${numberBranch(2)}
  }
}
```

L'interface du programme est composée de deux signaux de sortie : `numbers` contient les nombres se déplaçant dans le cadre et `killn` indique quelles sont les « proies attrapées » durant la réaction. Ces signaux sont combinés, permettant l'émission de multiples valeurs pendant la réaction. La fonction de combinaison `reduce` ajoute la valeur `n` retournée par l'expression d'émission dans le tableau `acc` et le retourne. Ceci suppose que la première valeur émise dans ces signaux à chaque réaction est un tableau.

Le corps du module est initialement composé de deux branches parallèles. À noter que l'instruction `FORK` est nommée et identifiable par `par`. La première branche consiste, pour chaque réaction, à réinitialiser l'affichage (appel à la fonction `clearReact`) puis à l'émission d'un tableau vierge dans `numbers` et `killn`. La seconde branche correspond à l'unique nombre initial, « 2 ». Elle est construite et retournée par la fonction `numberBranch` permettant l'ajout d'un nombre `n` passé en paramètre :

```
function numberBranch(n) {
  let num = new Num(n);
  let trap = TRAP Exit {
    LOOP {
      AWAIT(NOW(numbers) && NOW(killn));
      EMIT numbers(num);
      EMIT killn(prey(VAL(numbers)));
      IF (VAL(killn).indexOf(num) >= 0) {
        ATOM {
          num.dead = true;
          m.getElementById("par").removeChild(trap);
        }
        EXIT Exit;
      } ELSE {
        IF (num.prey && VAL(killn).indexOf(num.prey) >= 0) {
          ATOM {
            num.init();
          }
        }
        ATOM {
          num.move();
        }
      }
    }
  }
  return trap;
}
```

Le code attend d'abord que `numbers` et `killn` soient présents. Ce mécanisme de synchronisation assure que les premiers émetteurs de ces signaux dans la réaction sont ceux de la boucle vue précédemment. Ainsi, `numbers` et `killn` sont toujours initialisées par un tableau vierge en début de réaction. Ensuite, le nombre correspondant à la branche est ajouté dans le tableau de `numbers`. Une fois que toutes les branches ont émis leur nombre, la fonction `prey` est appelée par chaque branche, recherchant une proie. Si une proie est trouvée et attrapée, elle est émise dans `killn`. À noter un second mécanisme de synchronisation : l'expression de `EMIT killn` permettant la recherche d'une proie lit la valeur de `numbers`. Or, elle n'est connue que lorsque tous les émetteurs de `numbers` ont émis leur valeur dans ce signal. L'instruction `EMIT killn` est donc exécutée par chaque branche une fois que toutes les branches ont émis leur nombre dans `numbers`. La suite du code supprime la branche si son nombre courant a été attrapé :

- `m` est une variable globale représentant la machine réactive ;
- `getElementById("par")` retourne une représentation de l'instruction

FORK identifiée par par dans le programme ;

- removeChild(trap) supprime la branche correspondant à trap dans le parallèle. Comme trap représente la branche courante, cela revient à supprimer le nombre attrapé.

En revanche, si le nombre a attrapé une proie, l'appel à num.init() augmente la taille de son affichage. Enfin, il est déplacé et affiché dans le cadre par l'appel à num.move(). Le corps de la boucle se termine et il est immédiatement redémarré. L'instruction AWAIT n'étant pas instantanée, son expression n'est pas évaluée avant la réaction suivante, après que les signaux numbers et killn soient réinitialisés. Ce code est créé et ajouté au programme entre deux réactions après un clic sur l'un des boutons pour ajouter des nombres. La fonction d'ajout d'un nombre est la suivante :

```
function addNumber(n) {
  m.getElementById("par").appendChild(numberBranch(n));
}
```

Le mécanisme est similaire au retrait d'une branche mis à part la méthode appendChild permettant l'ajout du code Hiphop.js passé en paramètre. Le code permettant l'affichage et la recherche des proies est purement JavaScript et synchrone, il n'est donc pas présenté ici. Le lecteur intéressé peut le lire dans le fichier examples/prims/client.js de la distribution Hiphop.js. Enfin, ce programme met à jour l'IHM directement dans du code JavaScript appelé pendant la réaction dans les instructions ATOM (fonction clearRect et méthode move).

Cette application repose sur trois mécanismes de Hiphop.js. D'abord, cela illustre le mécanisme de création et de suppression d'une branche d'une instruction FORK entre deux réactions à chaque fois qu'un nombre apparaît ou est attrapé. Ce mécanisme permet de manipuler directement, depuis les constructeurs du langage, la création et le comportement temporel d'objets JavaScript. Ensuite, cela montre l'utilisation de signaux combinés, avec par exemple le signal killn correspondant aux nombres attrapés et dont la valeur est modifiée à plusieurs reprises pendant la réaction. Enfin, la synchronisation de branches parallèles en fonction de l'état de signaux est utilisée à deux reprises : pour attendre que numbers et killn soit réinitialisés à chaque réaction et pour attendre que numbers soit rempli par l'ensemble des nombres existants.

Chapitre 9

Perspectives

Sommaire

Ce chapitre présente de façon informelle plusieurs problèmes ouverts ainsi que des questions prospectives sur les évolutions futures de Hiphop.js. Les constructions et les exemples de code présentés ici ne sont pas implémentés et ne servent qu'à illustrer les concepts introduits.

9.1 Extensions du langage

9.1.1 Émission de signaux globaux depuis EXEC

Hiphop.js permet de contrôler le démarrage, la suspension ou la terminaison d'une tâche asynchrone. Il peut être nécessaire de connaître, entre son démarrage et sa terminaison, l'état de la tâche asynchrone. Par exemple, le module `audioModule` vu en section 8.3 met continuellement à jour la position courante du fichier audio dans le signal `position` lorsque la lecture est démarrée. Reprenons une partie du code de `audioModule` :

```
FORK {
  PROMISE trackEnded, error start()
          ONRES audio.play()
          ONSUSP audio.pause()
          ONKILL audio.pause();
  EXIT Terminated;
} PAR {
  LOOP {
    RUN(timeoutModule(1000));
    EMIT position(audio.currentTime);
    EMIT duration(audio.duration);
  }
}
```

Ce code parallèle démarre la lecture et attend sa terminaison dans une première branche. Dans une seconde branche, il émet chaque seconde la valeur courante de la position de lecture et de la durée du morceau. Ces deux émissions dépendent de l'état du lecteur audio géré par `PROMISE` et devraient donc être émises directement par le lecteur audio à chaque changement de position ou de durée. De façon générale, si l'action gérée par `PROMISE` a des changements d'état irréguliers, il est raisonnable d'attendre que l'état de la tâche asynchrone change pour faire réagir le programme, plutôt que de faire réagir le programme de façon régulière pour lire l'état de la tâche. En effet, si l'état de la tâche n'a pas changé entre deux lectures, la réaction est inutile.

Une première solution est de créer des gestionnaires d'évènements qui font réagir la machine réactive à chaque changement de position de lecture et de durée du lecteur

audio. Le code suivant est ajouté dans la fonction `start` vue à la section 8.3 :

```
audio.ondurationchange = evt =>
   m.inputAndReact("position", evt.target.currentTime);

audio.ontimeupdate = evt =>
   m.inputAndReact("duration", evt.target.duration);
```

Les signaux `position` et `duration` doivent alors être des signaux d'entrée-sortie puisqu'ils sont maintenant émis depuis l'environnement. Le parallèle n'est alors plus nécessaire :

```
PROMISE trackEnded start()
        ONRES audio.play()
        ONSUSP audio.pause()
        ONKILL audio.pause()
EXIT Terminated;
```

Le code Hiphop.js est simplifié car il ne lit plus les changements d'états du lecteur de musique chaque seconde. En revanche, cette solution pose un grave problème : le nom des signaux `position` et `duration` sont statiquement inscrits dans le code de `start`. Puisque ce module peut-être appelé dans d'autres programmes Hiphop.js, ces signaux peuvent être liés à des signaux globaux ayant un nom différent, ce qui empêchera de les émettre depuis l'environnement. L'introduction de l'instruction `MANGLE` est une solution possible à ce problème.

L'instruction `MANGLE` permettrait de résoudre le nom d'un signal lors de la compilation du programme Hiphop.js. Par exemple, si `audioModule` est appelé dans un autre module en liant explicitement le signal `position` à un signal global `positionAux`, alors la construction `MANGLE(position)` s'évalue par la valeur `"positionAux"` à la compilation du programme. L'appel à `PROMISE` devient :

```
PROMISE trackEnded start(MANGLE(position),
                         MANGLE(duration))
        ONRES audio.play()
        ONSUSP audio.pause()
        ONKILL audio.pause()
```

La fonction `start` devient :

```
function start(src, position, duration) {
   ...
   audio.ondurationchange = evt =>
      m.inputAndReact(position, evt.target.currentTime);
   audio.ontimeupdate = evt =>
      m.inputAndReact(duration, evt.target.duration); }
```

Le module est maintenant réutilisable : le nom des signaux `position` et `duration` n'est pas inscrit statiquement dans le code de `start`, mais calculé à la compilation du programme et la valeur obtenue est passée à `start` lors de son appel.

Un autre problème demeure : si `audioModule` est appelé par un module Hiphop.js sans liaison avec le signal `position`, alors `position` devient local à `audioModule`. Une solution possible est la suivante. Lorsque `MANGLE` est utilisé pour résoudre le nom d'un signal local, ce signal devient implicitement un signal global d'entrée dont identifiant n'est connu que par l'évaluation de `MANGLE` : il n'est donc pas accessible en dehors du code JavaScript appelé par `PROMISE`.

9.1.2 Variables locales

Le langage Esterel permet l'utilisation de variables. Une variable Esterel peut changer de valeur plusieurs fois pendant la réaction. Dans du code parallèle, soit plusieurs branches parallèles peuvent lire une variable x, soit une unique branche peut lire et écrire dans x. En Hiphop.js, les variables ne sont pas implémentées. Comme les programmes Hiphop.js sont construits et exécutés par JavaScript qui dispose de fermetures, il est possible de déclarer des variables locales dans une fonction JavaScript et de les utiliser dans le programme Hiphop.js construit et retourné par cette fonction. Il est même possible de définir la portée des variables dans le programme Hiphop.js :

```
function makeHHSubProgram(init) {
  let v = init;
  return LOOP {
    ATOM {
      v++;
    }
    PAUSE;
    ATOM {
      console.log("v subprogram =", v);
    }
  }
}
```

La fonction `makeHHSubProgram` retourne un sous-programme Hiphop.js qui boucle à chaque réaction en incrémentant une variable locale `v`. Cette variable est initialisée à la construction du programme, mais elle pourrait tout aussi bien être initialisée dans une instruction `ATOM`. Cette variable n'est connue et accessible que par le sous-programme construit et retourné par `makeHHSubProgram`. Un exemple de programme complet est :

```
function makeHHProgram() {
  return MODULE () {
    FORK {
      ${makeHHSubProgram(1)}
    } PAR {
      ${makeHHSubProgram(100)}
    }
  }
}
```

Ici, deux variables locales `v` existent et ont une valeur différente, mais le programme retourné par `makeHHProgram` n'y a pas accès. Cette approche a l'avantage de ne pas rendre le langage ou le compilateur plus complexe, mais elle a deux limites. D'abord, considérons le cas suivant :

```
function makeShared() {
  let shared;
  return FORK {
    ...
  } PAR {
    ...
  }
}
```

Comme Hiphop.js ne connaît pas les variables JavaScript il est impossible de vérifier que dans le cas où la variable `shared` est modifiée dans l'instant, elle n'est lue que par la branche qui l'a modifiée. Sans cette garantie, la sémantique d'Esterel n'est pas respectée, ce qui rend le programme non déterministe.

Un autre problème concerne l'initialisation des variables :

```
function makeHHPreemptProgram() {
  return MODULE (IN reset) {
    LOOPEACH(NOW(reset)) {
      RUN(makeHHProgram);
    }
  }
}
```

Ici, le programme retourné par `makeHHProgram` est inclus dans le corps de l'instruction de préemption `LOOPEACH`. Le corps de ce programme est composé de deux sous-programmes construits par deux appels à `makeHHSubProgram`. Ces deux sous-programmes sont préemptés et redémarrés à chaque réaction où le signal `reset` est présent. Puisque ces deux sous-programmes définissent chacun une variable `v`, la valeur de cette variable doit être réinitialisée respectivement à 1 et 100 (valeurs passées en paramètre lors de l'appel à `makeHHSubProgram`) pour chacun des sous-programmes

lors du redémarrage du corps de LOOPEACH. Malheureusement ceci n'est pas automatiquement fait par Hiphop.js puisque le langage n'a pas conscience de ces variables. Il faut que le programmeur le fasse explicitement, par exemple :

```
function makeHHSubProgram(init) {
  let v;
  return SEQUENCE {
    ATOM {
      v = init;
    }
    LOOP {
      ATOM {
        v++;
      }
      PAUSE;
      ATOM {
        console.log("v subprogram =", v);
      }
    }
  }
}
```

La variable v est maintenant initialisée à l'exécution par ATOM avant d'entrer dans la boucle. Ainsi, si ce code est préempté et redémarré, l'instruction ATOM précédant la boucle sera exécutée et réinitialisera la valeur de v. Cette manipulation n'est pas intuitive et alourdit le code.

Afin de pallier ces deux limitations, il serait envisageable d'implémenter le modèle du langage SCEst dans Hiphop.js [Rathlev *et al.*, 2015], ce qui permettrait de fusionner variables et signaux tout en conservant la sémantique constructive d'Esterel.

9.1.3 Une machine réactive *itérable*

Lors de son appel, une fonction JavaScript est toujours exécutée du début de son code jusqu'à complétion (fin de code ou instruction return) ou jusqu'à ce qu'une exception non rattrapée soit levée. Il existe cependant un cas particulier : un *générateur* est un type spécial de fonction pouvant retourner à l'appelant plusieurs valeurs intermédiaires. Lors de son premier appel, l'exécution commence au début du code du générateur. Si le flot de contrôle exécute une instruction yield, alors l'exécution du générateur s'arrête et le contrôle est rendu à l'appelant, avec une valeur éventuelle. Si le générateur est réinvoqué, le code *après* yield sera exécuté. Techniquement, un générateur est invoqué via la méthode prédéfinie next qui retourne une paire comprenant une valeur optionnelle (passée à yield) ainsi qu'un booléen indiquant si le générateur s'est terminé (le flot de contrôle est arrivé à la fin du code). Par ailleurs, JavaScript définit aussi

des *objets itérables*. Les tableaux sont des exemples d'objets itérables. De façon plus générale, n'importe quel objet est itérable s'il définit une méthode spécifique [1] et que cette méthode est un générateur.

Une machine réactive ressemble à un générateur : elle dispose d'une méthode `react` dont l'appel déclenche un calcul et retourne des valeurs intermédiaires via les signaux de sorties. L'instruction `PAUSE` de Hiphop.js rappelle l'instruction `yield` d'un générateur : ces instructions indiquent la terminaison d'un calcul et le point de départ du calcul suivant. La machine réactive devient un itérable JavaScript par le code suivant :

```
ReactiveMachine.prototype[Symbol.iterator] = function*() {
  this.react();
  if (!this.cinterface.k0.value) {
    yield;
  }
};
```

L'instruction `yield` est appelée si la dernière instruction du programme ne s'est pas terminée dans l'instant, cela signifiant que le programme Hiphop.js ne s'est pas encore terminé. Cela semble être une nouvelle approche d'intégration avec JavaScript : construire un programme retournant un itérable standard dans lequel il est possible d'itérer jusqu'à la terminaison du programme Hiphop.js, par exemple via la forme syntaxique JavaScript `for...of`.

L'implémentation d'une machine réactive itérable est directe puisqu'elle se résume au code précédent. Cependant, cela pose plusieurs questions sur l'interface entre l'émission de signaux d'entrée ou la réception de signaux de sortie et la méthode `next`. Passer un paramètre à l'appel à `next` doit-il émettre un signal d'entrée avec une valeur ? Si oui, quel signal ? De façon analogue, puisque `yield` peut retourner une valeur à l'appelant, l'émission d'un signal de sortie doit-elle retourner une valeur à l'appelant via `yield` ? Faut-il définir un signal global d'entrée-sortie spécifique aux générateurs pour gérer ces cas particuliers ? Ceci ouvre donc la voie à des explorations sur une intégration fine entre l'API de Hiphop.js et les évolutions récentes de JavaScript.

9.2 Version distribuée de Hiphop.js

L'architecture de Hiphop.js est de type *Globally Asynchronous Locally Synchronous* (GALS) : un système réactif synchrone central s'interface avec un environnement asynchrone. Des techniques permettent de distribuer des systèmes synchrones communiquant entre eux via le réseau [Caspi et Girault, 1995, Santos *et al.*, 2017].

1. La méthode doit être accessible via `Symbol.iterator`.

Ce mécanisme ne nécessite pas de modifier le langage. Il nécessite d'étendre l'API des machines réactives afin de « connecter » des signaux globaux d'une machine réactive à ceux d'une autre machine réactive. Techniquement, puisque les deux machines réactives peuvent ne pas être exécutées dans le même environnement d'exécution (une peut être sur un client et l'autre sur un serveur), cette API peut utiliser en interne des WebSocket pour la communication sous-jacente.

Chapitre 10

Conclusion générale

Le web, historiquement un moyen d'échange de documents statiques, est devenu aujourd'hui une plate-forme universelle pour écrire des applications de tous types. Ces applications sont souvent riches en interactions avec l'utilisateur via des IHM complexes et dynamiques, mais aussi avec des serveurs distants. Au niveau de l'implémentation ces interactions se traduisent par des évènements asynchrones. JavaScript, le langage majoritairement utilisé pour implémenter les applications web, dispose d'outils limités pour gérer des évènements asynchrones ce qui rend les programmes difficiles à implémenter et à maintenir lorsque la combinatoire d'évènements et d'états à orchestrer est grande, en particulier quand il faut préempter des opérations asynchrones en cours d'exécution.

L'orchestration d'évènements asynchrones est un problème connu et largement étudié dans le domaine des systèmes réactifs. Ces études ont permis la conception de nombreux langages réactifs synchrones qui permettent de faire interagir plusieurs composants entre eux, de façon parallèle, déterministe et vérifiable. Ces technologies se prêtent originellement à des environnements statiques, où les différents composants qui interagissent ensemble sont connus lorsque le programme est implémenté.

L'adaptation d'un langage réactif synchrone dans un contexte dynamique a été exploré avec le langage ReactiveML [Mandel et Pouzet, 2008], qui permet de construire et compiler un programme réactif synchrone puis d'y associer des évènements de l'environnement pendant l'exécution du programme environnant. En revanche, l'intégration fine avec le web et donc avec le langage JavaScript n'avait encore jamais été explorée. Il s'agit de la contribution majeure de cette thèse : la conception et l'implémentation d'un langage réactif synchrone pour le web.

Le langage réactif synchrone Hiphop.js est le fruit de cette thèse. Hiphop.js étend le langage JavaScript en y incorporant la sémantique et les constructions d'Esterel v5. Dans un premier temps, l'implémentation JavaScript des circuits booléens d'Esterel et une syntaxe XML ont permis d'explorer l'orchestration d'évènements asynchrones dans le web en se connectant à JavaScript via des gestionnaires d'évènements. Ceci a aussi permis de manipuler l'arbre XML de programmes Hiphop.js comme des valeurs JavaScript et de construire dynamiquement le programme, mais aussi de modifier dynamiquement le programme réactif en ajoutant ou retirant des branches parallèles entre deux instants. L'intégration est ensuite poussée en dotant Hiphop.js d'une syntaxe concrète dédiée et plus proche de celle de JavaScript. L'intégration de Hiphop.js et de cette nouvelle syntaxe avec Hop.js permet d'exécuter des programmes JavaScript contenant du code Hiphop.js sans pré-compilation apparente par l'utilisateur, mais aussi d'envoyer, de compiler et d'exécuter des programmes Hiphop.js dans le navigateur de façon transparente.

Une part importante de la programmation web consiste en la programmation d'IHM qui, dans le contexte interactif des applications web, est constamment mise à jour pour refléter des changements d'états dans le programme. L'API du DOM étant de trop bas

niveau, de nombreuses technologies et recherches proposent des solutions pour rendre l'accès au DOM plus abstrait. Il s'agit d'une piste également explorée par Hiphop.js : une solution est apportée à ce problème en permettant l'injection dans le DOM de l'état et de la valeur des signaux de programmes Hiphop.js. Ainsi, après chaque réaction d'un programme Hiphop.js, les nœuds du DOM référençant des signaux du programme sont automatiquement mis à jour. Ceci offre une couche d'abstraction au dessus du DOM et d'uniformisation entre le programme réactif et l'IHM.

Par ailleurs, les applications web sont distribuées : au moins une partie s'exécute dans le navigateur web (ou application mobile) et une autre partie sur un serveur. Afin de faciliter le développement d'applications Hiphop.js, un débogueur symbolique distribué a été implémenté. Il permet de déboguer une application Hiphop.js s'exécutant depuis n'importe quel périphérique via une interface web accessible depuis l'ordinateur du développeur.

Enfin, Hiphop.js est actuellement utilisé à l'Inria dans un contexte de production musicale interactive. Le but de ces travaux est d'écrire, à travers un programme Hiphop.js, l'enchaînement musical d'un spectacle tel que le ferait un compositeur avec une partition. La différence est qu'ici, la musique n'est pas figée dans la partition, mais va être dynamiquement proposée aux spectateurs. La musique est jouée en fonction du choix des spectateurs et de certaines contraintes imposées par le compositeur. En d'autres termes, il s'agit d'*orchestrer* un spectacle. Le terme orchestration correspond à la fois au chef d'orchestre qui dirige ses musiciens, mais aussi au programme Hiphop.js qui propose à chaque utilisateur et de façon dynamique des sons à jouer au cours des différentes scènes. Les constructions temporelles de Hiphop.js sont particulièrement adaptées à ce type de programme puisqu'il s'agit, par exemple, de proposer à certains spectateurs de jouer le son d'un instrument un certain nombre de fois ou jusqu'à ce qu'un autre utilisateur décide de faire jouer un autre instrument. On retrouve donc les notions de séquence, de parallélisme et de préemption. Ces travaux ont donné lieu à deux concerts à Nice dans le cadre du festival MANCA en décembre 2017 [1]. Plus de 200 personnes ont participé au spectacle en sélectionnant différents types de sons à jouer au cours des différentes scènes [2]. Pour interagir avec le spectacle, chaque spectateur s'est connecté via son téléphone mobile à une application web écrite en Hop.js/Hiphop.js.

1. `http://www.cirm-manca.org/fiche-oeuvre.php?oe=427`
2. `https://www.youtube.com/watch?v=lCkWSWFxk7w`

Annexes

Annexe A

Carte du langage Hiphop.js

Module et signaux globaux

hh-module:
MODULE *identifier*$_{opt}$ (*signal-list*$_{opt}$) {
hh-statement
}

signal-list:
signal
signal-list, *signal*

signal:
direction identifier combine$_{opt}$
direction identifier(*expression*) *combine*$_{opt}$

direction:
IN
OUT
INOUT

combine:
COMBINE(*functional-value*)

Expressions et délais

hh-expression:
js-expression
NOW(*identifier*)
PRE(*identifier*)
VAL(*identifier*)
PREVAL(*identifier*)

delay:
(*hh-expression*)
COUNT(*hh-eprexssion*, *hh-expression*)
IMMEDIATE(*hh-expression*)

Instructions

hh-statement:
basic-statement
sequence-statement
parallel-statement
emit-statement
await-statement
cond-statement
preemption-statement
suspension-statement
local-statement
atom-statement
loop-statement
trap-statement
exec-statement
run-statement

Flot de contrôle basique

basic-statement:
NOTHING
PAUSE
HALT

Séquence et parallélisme

sequence-statement:
hh-statement; *hh-statement*
SEQUENCE { *hh-statement* }

parallel-statement:
FORK *identifier*$_{opt}$ { *hh-statement* } *branch*$_{opt}$

branch:
PAR { *hh-statement* } *branch*$_{opt}$

Émission de signal

emit-statement:
EMIT *emit-list*
SUSTAIN *emit-list*

emit-list:
emit-signal
emit-list, *emit-signal*

emit-signal:
identifier
identifier(*hh-expression*)

Attente et test conditionnel

await-statement:
AWAIT *delay*

cond-statement:
IF (*hh-expression*) { *hh-statement* } *else*$_{opt}$

else:
ELSE { *hh-statement* }
ELSE *cond-statement*

Préemption et suspension

preemption-statement:
ABORT *delay* { *hh-statement* }
WEAKABORT *delay* { *hh-statement* }

suspension-statement:
SUSPEND *delay* { *hh-statement* }
SUSPEND FROM *delay* **TO** *delay emit-suspended*$_{opt}$ { *hh-statement* }
SUSPEND TOOGLE *delay emit-suspended*$_{opt}$ { *hh-statement* }

emit-suspended:
EMITWHENSUSPENDED *identifier*

Exécution atomique de code JavaScript pendant la réaction

atom−statement:
ATOM { *hiphopjs* }

Signaux locaux

local-statement:
LOCAL *local-signal-list* { *hh-statement* }

local-signal-list:
local-signal
local-signal-list, *local-signal*

local-signal:
identifier combine$_{opt}$
identifier(*hh-expression*) *combine*$_{opt}$

Boucles

loop-statement:
LOOP { *hh-statement* }
WHILE (*hh-expression*) { *hh-statement* }
FOR (*signal-list*$_{opt}$; *hh-expression*$_{opt}$; *hh-statement*$_{opt}$) { *hh-statement* }
LOOPEACH *delay* { *hh-statement* }
EVERY *delay* { *hh-statement* }

Trappes

trap-statement:
TRAP *identifier* { *hh-statement* }
EXIT *identifier*

Contrôle d'exécution JavaScript asynchrone

exec-statement:
EXEC *js-bridge exec-params*$_{opt}$
EXECEMIT *identifier js-bridge exec-params*$_{opt}$
PROMISE *identifier*$_{then}$, *identifier*$_{catch}$ *js-bridge exec-params*$_{opt}$

js-bridge:
js-instruction
THIS
DONE
DONEREACT

exec-params:
ONSUSP *js-bridge*
ONRES *js-bridge*
ONKILL *js-bridge*

Réutilisation de modules

run-statement:
RUN (*hh-expression*)
RUN (*hh-expression*, *signal-assoc-list*)

signal-assoc-list:
identifier = *identifier*
signal-assoc-list, *identifier* = *identifier*

Annexe B

Grammaire de la syntaxe abstraite de Hiphop.js

hh-module:
        <hh.module *name*$_{opt}$ *signal-list*$_{opt}$> *hh-statement* </hh.module>

name:
        __hh_debug_name__="*identifier*"

signal-list:
        signal *signal-list*$_{opt}$

signal:
        identifier
        identifier=${*js-expression*}

hh-statement:
        basic-statement
        sequence-statement
        parallel-statement
        emit-statement
        await-statement
        cond-statement
        preemption-statement
        suspension-statement
        atom-statement
        local-statement
        loop-statement
        trap-statement
        exec-statement
        run-statement

basic-statement:
        <hh.nothing/>
        <hh.pause/>
        <hh.halt/>

sequence-statement:
        <hh.sequence> *hh-statement* </hh.sequence>

parallel-statement:
        <hh.parallel id="*identifier*"> *branch* </hh.parallel>

branch:
        hh-statement *branch*$_{opt}$

hh-expression:
apply=${*js-expression*}
value=${*js-expression*}

hh-counter:
countApply=${*js-expression*}
countValue=${*js-expression*}

hh-delay:
immediate$_{opt}$ *identifier* *hh-counter*$_{opt}$
pre$_{opt}$ *identifier* *hh-counter*$_{opt}$
immediate$_{opt}$ *hh-expression* *hh-counter*$_{opt}$

emit-statement:
<hh.emit *identifier* *hh-expression*/>
<hh.sustain *identifier* *hh-expression*/>

await-statement:
<hh.await *hh-delay*/>

cond-statement:
<hh.if *hh-delay*> *hh-statement*$_{then}$ *hh-statement*$_{else_{opt}}$ </hh.if>

preemption-statement:
<hh.abort *hh-delay*> *hh-statement* </hh.abort>
<hh.weakabort *hh-delay*> *hh-statement* </hh.weakabort>

suspend-statement:
<hh.suspend *suspend-params*> *hh-statement* </hh.suspend>

suspend-params:
hh-delay
from=*identifier* to=*identifier* *when-susp*$_{opt}$
fromApply=*js-expression* toApply=*js-expression* *when-susp*$_{opt}$
toggleSignal=*identifier* *when-susp*$_{opt}$
toogleApply=*js-expression* *when-susp*$_{opt}$

when-susp:
emitwhensuspended="*identifier*"

atom-statement:
<hh.atom apply=${*js-expression*}/>

local-statement:
        <hh.local *signal-list*> *hh-statement* </hh.local>

loop-statement:
        <hh.loop> *hh-statement* </hh.loop>
        <hh.loopeach *hh-delay*> *hh-statement* </hh.loopeach>
        <hh.every *hh-delay*> *hh-statement* </hh.every>

trap-statement:
        <hh.trap *identifier*> *hh-statement* </hh.trap>
        <hh.exit *identifier*/>

exec-statement:
        <hh.exec $identifier_{opt}$ apply=${*js-expression*} *exec-params*/>

exec-params:
        $susp\text{-}param_{opt}$
        $res\text{-}param_{opt}$
        $kill\text{-}param_{opt}$

susp-param:
        susp=${*js-expression*}

res-param:
        res=${*js-expression*}

kill-param
        kill=${*js-expression*}

run-statement:
        <hh.run module=${*js-expression*} $signal\text{-}mapping_{opt}$/>

signal-mapping:
        "*identifier*"="*identifier*" $signal\text{-}mapping_{opt}$

Annexe C

Architecture globale de l'implémentation du compilateur Hiphop.js

Le contenu de la racine de la distribution de Hiphop.js contient :

- `doc/` : documentation au format Markdown du langage ;
- `docker/` : fichiers permettant la création d'une image Docker comprenant un système minimal GNU/Linux avec Hop.js et Hiphop.js pré-installés ;
- `examples/` : exemples de programmes Hiphop.js ;
- `lib/` : compilation vers les circuits Esterel et environnement d'exécution de programmes Hiphop.js ;
- `node_modules/` : bibliothèques tierces, actuellement seule `source-map` [1] est utilisée par le préprocesseur Hiphop.js ;
- `package.json` : méta-informations sur la distribution Hiphop.js. Il contient une information nécessaire pour que Hop.js puisse exporter l'environnement Hiphop.js sur le client ;
- `preprocessor/` : implémentation du préprocesseur de Hiphop.js ;
- `tests/` : tests de non régression de Hiphop.js. La commande `./TEST` lance l'ensemble des tests, `./TEST abro` lance le test implémenté par les fichiers `abro.js` et `abro.out`. Enfin, l'option `-g` lance Hop.js en mode de développement pour l'exécution des tests ;
- `ulib/` : bibliothèques utilisateur de Hiphop.js expérimentales.

Détails de `lib/`

Le dossier `lib/` contient les fichiers suivants :

- `ast.js` : définit les différentes classes de l'AST de Hiphop.js. Cet AST est construit par Hop.js lors de l'évaluation des instructions Hiphop.js ;
- `batch.js` : définit un environnement permettant de lancer une invite de commandes d'une machine réactive. La fonction `batch` exportée par la distribution Hiphop.js prend une machine réactive en paramètre et démarre une invite de commandes. La syntaxe est similaire à l'invite de commandes Esterel [Berry, 2000a] ;
- `compiler.js` : définit la transformation de chaque instruction Hiphop.js (représentée par un nœud d'AST) en circuit Esterel ;

1. `https://github.com/mozilla/source-map`

- `debugger-common.js` : constantes et fonctions communes aux différentes parties du débogueur ;
- `debugger.js` : définit la partie du débogueur qui est attaché à une machine réactive ;
- `debugger-server.js` : définit la partie serveur du débogueur, c'est-à-dire la gestion des différentes instances de débogueurs et gestion de la partie cliente du débogueur ;
- `error.js` : définit les différentes erreurs pouvant être levées lors de la compilation ou l'exécution de programmes Hiphop.js ;
- `hiphop.js` : fichier d'entrée de la distribution Hiphop.js. Il exporte l'ensemble des fonctionnalités de Hiphop.js ;
- `inheritance.js` : définitions de fonctions facilitant le chaînage de prototypes JavaScript afin de simuler du sous-typage simple. Ce fichier est utilisé pour construire les différentes classes d'AST ;
- `lang.js` : définition de la syntaxe abstraite de Hiphop.js.
- `machine.js` : permet de construire une machine réactive à partir de l'AST d'un programme Hiphop.js passé en paramètre. La gestion de la réaction, la gestion de `EXEC`, la liaison des signaux avec l'environnement, les proxies réactifs, l'activation du débogueur et l'accès aux métriques sont définis dans ce fichier ;
- `net.js` : constructeurs génériques permettant la génération des différents types de portes logiques Esterel. Ce fichier est utilisé par `compiler.js` pour construire les circuits correspondant à chaque instruction ;
- `signal.js` : constructeurs génériques permettant la création des signaux ainsi que leur gestion lors de l'exécution d'une machine réactive.

Références bibliographiques

[Abramov, 2015] ABRAMOV, D. (2015). Redux. https://redux.js.org/.

[Berry, 1989] BERRY, G. (1989). Programming a digital watch in Esterel v3. Rapport technique RR-1032, INRIA.

[Berry, 2000a] BERRY, G. (2000a). The Esterel v5 Language Primer Version v5_91. ftp://ftp.inrialpes.fr/pub/synalp/reports/esterel-primer.pdf.gz.

[Berry, 2000b] BERRY, G. (2000b). The Foundations of Esterel. *In Proof, Language, and Interaction*, pages 425–454.

[Berry, 2002] BERRY, G. (2002). The Constructive Semantics of Pure Esterel. http://www-sop.inria.fr/members/Gerard.Berry/Papers/EsterelConstructiveBook.pdf.

[Berry, 2007] BERRY, G. (2007). SCADE : Synchronous design and validation of embedded control software. *In Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems*, pages 19–33. Springer.

[Berry, 2008] BERRY, G. (2008). The Esterel v7 Reference Manual : Version v7_60 for Esterel Studio 6.1. https://www.college-de-france.fr/media/gerard-berry/UPL6812476055586711 66_Esterelv7.60_ReferenceManual.pdf.

[Berry, 2018a] BERRY, G. (2018a). Esterel de A à Z : 1. Principes, idées et styles pour la programmation réactive. https://www.college-de-france.fr/media/gerard-berry/

UPL1960279709430924348_UPL8987634546892388239_2018_02_07_Berry_Cours2_EsterelGe__ne__ral.compressed_1_.pdf.

[Berry, 2018b] BERRY, G. (2018b). Esterel de A à Z : Réincarnation statique des boucles en Esterel pour les compilations matérielles et logicielles efficaces. https://www.college-de-france.fr/site/gerard-berry/course-2018-03-14-16h00.htm.

[Berry, 2018c] BERRY, G. (2018c). Esterel de A à Z : Traduction des programmes Esterel en circuits digitaux pour la conception des circuits électroniques. https://www.college-de-france.fr/site/gerard-berry/course-2018-03-07-16h00.htm.

[Berry et Boudol, 1992] BERRY, G. et BOUDOL, G. (1992). The chemical abstract machine. *Theoretical computer science*, 96(1):217–248.

[Berry et Gonthier, 1992] BERRY, G. et GONTHIER, G. (1992). The ESTEREL Synchronous Programming Language : Design, Semantics, Implementation. *Sci. Comput. Program.*, 19(2):87–152.

[Berry *et al.*, 2011] BERRY, G., NICOLAS, C. et SERRANO, M. (2011). Hiphop : a synchronous reactive extension for Hop. *In Proceedings of the 1st ACM SIGPLAN international workshop on Programming Language and Systems Technologies for Internet Clients (PLASTIC)*, pages 49–56. ACM.

[Bjornson *et al.*, 2010] BJORNSON, J., TAYANOVSKYY, A. et GRANICZ, A. (2010). Composing Reactive GUIs in F# Using WebSharper. *In Proceedings of Symposium on Implementation and Application of Functional Languages (IFL)*, pages 203–216. Springer.

[Bloomberg, 2016] BLOOMBERG (2016). Bucklescript. https://bucklescript.github.io/en/.

[Boussinot, 1991] BOUSSINOT, F. (1991). Reactive C : An extension of C to program reactive systems. *Software : Practice and Experience*, 21(4):401–428.

[Boussinot et Laneve, 1995] BOUSSINOT, F. et LANEVE, C. (1995). Two Semantics for a Language of Reactive Objects. Rapport technique RR-2511, INRIA.

[Caspi et Girault, 1995] CASPI, P. et GIRAULT, A. (1995). Execution of distributed reactive systems. *In Proceedings of European Conference on Parallel Processing (Euro-Par)*, pages 13–26. Springer.

[Caspi *et al.*, 1987] CASPI, P., PILAUD, D., HALBWACHS, N. et PLAICE, J. (1987). LUSTRE : A declarative language for programming synchronous systems. *In Proceedings of the 14th Annual ACM Symposium on Principles of Programming Languages (POPL)*, volume 178. ACM.

[Chlipala, 2015] CHLIPALA, A. (2015). Ur/Web : A simple model for programming the web. *In Proceedings of the 42nd Annual ACM Symposium on Principles of Programming Languages (POPL)*, volume 50, pages 153–165. ACM.

[Chudnov et Naumann, 2015] CHUDNOV, A. et NAUMANN, D. A. (2015). Inlined Information Flow Monitoring for JavaScript. *In Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 629–643. ACM.

[Colaço *et al.*, 2005] COLAÇO, J.-L., PAGANO, B. et POUZET, M. (2005). A conservative extension of synchronous data-flow with state machines. *In Proceedings of the 5th ACM international conference on Embedded Software (EMSOFT)*, pages 173–182. ACM.

[Cooper *et al.*, 2008] COOPER, E., LINDLEY, S., WADLER, P. et YALLOP, J. (2008). The essence of form abstraction. *In Proceedings of Asian Symposium on Programming Languages and Systems (APLAS)*, pages 205–220. Springer.

[Cooper et Krishnamurthi, 2004] COOPER, G. H. et KRISHNAMURTHI, S. (2004). FrTime : Functional reactive programming in PLT Scheme. *Computer science technical report. Brown University. CS-03-20*.

[Czaplicki et Chong, 2013] CZAPLICKI, E. et CHONG, S. (2013). Asynchronous functional reactive programming for GUIs. *In Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, volume 48, pages 411–422.

[Dahl, 2009] DAHL, R. (2009). Node.js. https://nodejs.org/.

[Doeraene, 2013] DOERAENE, S. (2013). Scala.js : Type-directed interoperability with dynamically typed languages. Rapport technique EPFL-REPORT-190834, École polytechnique fédérale de Lausanne.

[ECMA, 2015] ECMA (2015). ECMAScript language specification. http://www.ecma-international.org/ecma-262/6.0/.

[Edwards, 1999] EDWARDS, S. A. (1999). Compiling Esterel into Sequential Code. *In Proceedings of the Seventh International Workshop on Hardware/Software Codesign (CODES)*, pages 147–151. IEEE.

[El Sibaïe et Chailloux, 2016] EL SIBAÏE, R. et CHAILLOUX, E. (2016). Synchronous-reactive web programming. *In Proceedings of Workshop on Reactive and Event-Based Languages and Systems (REBELS)*, pages 9–16. ACM.

[El Sibaïe, 2018] EL SIBAÏE, R. (2018). *Programmation Web Réactive dans un cadre typé statiquement pour l'orchestration de contenus multimédia riches*. Thèse de doctorat, Sorbonne Université, Paris.

[Elliott et Hudak, 1997] ELLIOTT, C. et HUDAK, P. (1997). Functional Reactive Animation. *In Proceedings of International Conference on Functional Programming (ICFP)*, pages 263–273. ACM.

[Facebook, 2013] FACEBOOK (2013). React. https://reactjs.org/.

[Facebook, 2014a] FACEBOOK (2014a). Flow. https://flow.org/.

[Facebook, 2014b] FACEBOOK (2014b). Flux. https://facebook.github.io/flux/.

[Facebook, 2014c] FACEBOOK (2014c). JSX. https://facebook.github.io/jsx/.

[Facebook, 2015] FACEBOOK (2015). React Native. https://facebook.github.io/react-native/.

[Friedman et Wise, 1977] FRIEDMAN, D. P. et WISE, D. S. (1977). Aspects of applicative programming for file systems (preliminary version). *SIGSOFT Softw. Eng. Notes*, 2(2):41–55.

[GitHub, 2013] GITHUB (2013). Electron. https://electronjs.org/.

[Google, 2008] GOOGLE (2008). V8. https://developers.google.com/v8/.

[Google, 2011] GOOGLE (2011). Dart. https://www.dartlang.org/.

[Google, 2016] GOOGLE (2016). Angular. https://angular.io/.

[Ha *et al.*, 2009] HA, J., HAGHIGHAT, M. R., CONG, S. et MCKINLEY, K. S. (2009). A Concurrent Trace-based Just-In-Time Compiler for Single-threaded JavaScript. *In Proceedings of the Workshop on Parallel Execution of Sequential Programs on Multicore Architectures (PESPMA)*, pages 47–54.

[Hanxleden *et al.*, 2014] HANXLEDEN, R. V., MENDLER, M., AGUADO, J., DUDERSTADT, B., FUHRMANN, I., MOTIKA, C., MERCER, S., O'BRIEN, O. et ROOP, P. (2014). Sequentially Constructive Concurrency—A conservative extension of the synchronous model of computation. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(4s):144.

[Harel, 1987] HAREL, D. (1987). Statecharts : A visual formalism for complex systems. *Science of computer programming*, 8(3):231–274.

[Harel et Pnueli, 1985] HAREL, D. et PNUELI, A. (1985). On the Development of Reactive Systems. *In Logics and Models of Concurrent Systems*, pages 477–498. Springer.

[Hewitt *et al.*, 1973] HEWITT, C., BISHOP, P. et STEIGER, R. (1973). A Universal Modular ACTOR Formalism for Artificial Intelligence. *In Proceedings of the 3rd International Joint Conference on Artificial Intelligence (IJCAI)*, pages 235–245. Morgan Kaufmann Publishers Inc.

[Jeffrey et Van Cutsem, 2015] JEFFREY, A. et VAN CUTSEM, T. (2015). Functional Reactive Programming with nothing but Promises (Implementing Push/Pull FRP using JavaScript Promises). *In Proceedings of Workshop on Reactive and Event-based Languages & Systems (REBELS)*.

[Kambona *et al.*, 2013] KAMBONA, K., BOIX, E. G. et DE MEUTER, W. (2013). An evaluation of reactive programming and promises for structuring collaborative web applications. *In Proceedings of the 7th Workshop on Dynamic Languages and Applications (Dyla)*, page 3.

[Kernighan et Ritchie, 1978] KERNIGHAN, B. W. et RITCHIE, D. M. (1978). *The C Programming Language*. Prentice Hall.

[Kiczales *et al.*, 1997] KICZALES, G., LAMPING, J., MENDHEKAR, A., MAEDA, C., LOPES, C., LOINGTIER, J.-M. et IRWIN, J. (1997). Aspect-Oriented Programming. *In Proceedings of European Conference on Object-Oriented Programming (ECOOP)*, pages 220–242. Springer.

[Kitchin *et al.*, 2009] KITCHIN, D., QUARK, A., COOK, W. R. et MISRA, J. (2009). The Orc Programming Language. *In Proceedings of Formal Techniques for Distributed Systems, Joint 11th IFIP WG 6.1 International Conference FMOODS 2009 and 29th IFIP WG 6.1 International Conference FORTE 2009 on Formal Techniques for Distributed Systems*, pages 1–25.

[LeGuernic *et al.*, 1991] LEGUERNIC, P., GAUTIER, T., LE BORGNE, M. et LE MAIRE, C. (1991). Programming real-time applications with SIGNAL. *Proceedings of the IEEE*, 79(9):1321–1336.

[Lenz et Fitzgerald, 2011] LENZ, J. et FITZGERALD, N. (2011). Source Map. https://sourcemaps.info/spec.html.

[Mandel et Pouzet, 2008] MANDEL, L. et POUZET, M. (2008). ReactiveML : un langage fonctionnel pour la programmation réactive. *Technique et Science Informatiques (TSI)*.

[Melnikov et Fette, 2011] MELNIKOV, A. et FETTE, I. (2011). The WebSocket Protocol. RFC 6455.

[Meyerovich *et al.*, 2009] MEYEROVICH, L. A., GUHA, A., BASKIN, J., COOPER, G. H., GREENBERG, M., BROMFIELD, A. et KRISHNAMURTHI, S. (2009). Flapjax : a programming language for Ajax applications. *In Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications (OOPSLA)*, volume 44, pages 1–20. ACM.

[Microsoft, 2012] MICROSOFT (2012). C# Asynchronous Programming. https://docs.microsoft.com/en-us/dotnet/csharp/async.

[Mikkonen et Taivalsaari, 2008] MIKKONEN, T. et TAIVALSAARI, A. (2008). Web applications : Spaghetti code for the 21st century. *In Proceedings of the 6th International Conference on Software Engineering - Research, Management & Applications (SERA)*. IEEE Computer Society.

[Milicevic *et al.*, 2013] MILICEVIC, A., JACKSON, D., GLIGORIC, M. et MARINOV, D. (2013). Model-based, event-driven programming paradigm for interactive web applications. *In Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software (Onward!)*, pages 17–36. ACM.

[Mozilla, 2011] MOZILLA (2011). IonMonkey. https://wiki.mozilla.org/IonMonkey.

[Mozilla *et al.*, 2015] MOZILLA, GOOGLE, MICROSOFT, APPLE et W3C (2015). Webassembly. https://webassembly.org/.

[Myter *et al.*, 2016] MYTER, F., SCHOLLIERS, C. et DE MEUTER, W. (2016). Many spiders make a better web : A unified web-based actor framework. *In Proceedings of the 6th International Workshop on Programming Based on Actors, Agents, and Decentralized Control*, AGERE 2016, pages 51–60. ACM.

[O'Reilly, 2003] O'REILLY, T. (2003). What Is Web 2.0. https://www.oreilly.com/pub/a//web2/archive/what-is-web-20.html.

[Paananen, 2012] PAANANEN, J. (2012). Bacon.js. https://baconjs.github.io/.

[Paris *et al.*, 1993] PARIS, J.-P., BERRY, G., MIGNARD, F., COURONNÉ, P., CASPI, P., HALBWACHS, N., SOREL, Y., BENVENISTE, A., GAUTIER, T., LE GUERNIC, P., DUPONT, F. et LE MAIRE, C. (1993). Projet SYNCHRONE : les formats communs des langages synchrones. Research Report RT-0157, INRIA.

[Philips *et al.*, 2016] PHILIPS, L., DE KOSTER, J., DE MEUTER, W. et DE ROOVER, C. (2016). Dependence-driven Delimited CPS Transformation for JavaScript. *In Proceedings of the 2016 ACM SIGPLAN International Conference on Generative Programming : Concepts and Experiences (GPCE)*, pages 59–69. ACM.

[Pnueli, 1986] PNUELI, A. (1986). Applications of temporal logic to the specification and verification of reactive systems : a survey of current trends. *In Current trends in Concurrency*, pages 510–584. Springer.

[Podwysocki *et al.*, 2013] PODWYSOCKI, M., DE SMET, B. et MEIJER, E. (2013). RxJS. http://reactivex.io/rxjs/.

[Potop-Butucaru *et al.*, 2007] POTOP-BUTUCARU, D., EDWARDS, S. A. et BERRY, G. (2007). *Compiling Esterel*. Springer.

[Proietti, 2006] PROIETTI, V. (2006). Mootools. https://mootools.net/.

[Rastogi *et al.*, 2015] RASTOGI, A., SWAMY, N., FOURNET, C., BIERMAN, G. et VEKRIS, P. (2015). Safe & Efficient Gradual Typing for TypeScript. *In Proceedings of the 42nd Annual ACM Symposium on Principles of Programming Languages (POPL)*, pages 167–180. ACM.

[Rathlev *et al.*, 2015] RATHLEV, K., SMYTH, S., MOTIKA, C., von HANXLEDEN, R. et MENDLER, M. (2015). SCEst : Sequentially Constructive Esterel. *In Proceedings of Formal Methods and Models for Codesign (MEMOCODE)*, pages 10–19. IEEE.

[Resig, 2006] RESIG, J. (2006). jQuery. https://jquery.com/.

[Reynders *et al.*, 2014] REYNDERS, B., DEVRIESE, D. et PIESSENS, F. (2014). Multi-tier functional reactive programming for the web. *In Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward!)*, pages 55–68. ACM.

[Santos *et al.*, 2017] SANTOS, R. C., SANT'ANNA, F. et RODRIGUEZ, N. (2017). A GALS Approach for Programming Distributed Interactive Multimedia Applications. *In XVII Workshop on Ongoing Thesis and Dissertations*.

[Sentovich *et al.*, 1992] SENTOVICH, E., SINGH, K., LAVAGNO, L., MOON, C., MURGAI, R., SALDANHA, A., SAVOJ, H., STEPHAN, P., BRAYTON, R. K. et SANGIOVANNI-VINCENTELLI, A. L. (1992). SIS : A System for Sequential Circuit Synthesis. Rapport technique UCB/ERL M92/41, Berkeley.

[Serrano *et al.*, 2006] SERRANO, M., GALLESIO, E. et LOITSCH, F. (2006). HOP : a language for programming the Web 2.0. *In Proceedings of the First Dynamic Languages Symposium (DLS)*. ACM.

[Serrano et Prunet, 2016] SERRANO, M. et PRUNET, V. (2016). A glimpse of Hopjs. *In Proceedings of International Conference on Functional Programming (ICFP)*, pages 180–192. ACM.

[Syme *et al.*, 2011] SYME, D., PETRICEK, T. et LOMOV, D. (2011). The F# Asynchronous Programming Model. *In International Symposium on Practical Aspects of Declarative Languages (PADL)*, pages 175–189. Springer.

[Tardieu, 2004] TARDIEU, O. (2004). *De la sémantique opérationnelle à la spécification formelle de compilateurs : l'exemple des boucles en Esterel*. Thèse de doctorat, École nationale supérieure des mines, Paris.

[Tilkov et Vinoski, 2010] TILKOV, S. et VINOSKI, S. (2010). Node.js : Using javascript to build high-performance network programs. *IEEE Internet Computing*, 14(6):80–83.

[Van Cutsem et Miller, 2010] VAN CUTSEM, T. et MILLER, M. S. (2010). Proxies : design principles for robust object-oriented intercession APIs. *In Proceedings of the 6th Symposium on Dynamic Languages (DLS)*, pages 59–72. ACM.

[Van den Vonder *et al.*, 2017] Van den VONDER, S., MYTER, F., DE KOSTER, J. et DE MEUTER, W. (2017). Enriching the Internet By Acting and Reacting. *In Proceedings of Companion to the first International Conference on the Art, Science and Engineering of Programming (Programming)*. ACM.

[Vouillon et Balat, 2014] VOUILLON, J. et BALAT, V. (2014). From bytecode to JavaScript : the Js_of_ocaml compiler. *Software : Practice and Experience*, 44(8):951–972.

[W3C, 2004] W3C (2004). Document Object Model (DOM) Level 3 Core Specification. https://www.w3.org/TR/DOM-Level-3-Core/.

[W3C, 2017] W3C (2017). HTML 5.2. https://www.w3.org/TR/html52/.

[W3Tech, 2018] W3TECH (2018). Usage of JavaScript for websites. https://w3techs.com/technologies/details/cp-javascript/all/all.

[Wanstrath, 2009] WANSTRATH, C. (2009). Mustache. http://mustache.github.io/mustache.5.html.

[Weil *et al.*, 2000] WEIL, D., BERTIN, V., CLOSSE, E., POIZE, M., VENIER, P. et PULOU, J. (2000). Efficient compilation of Esterel for real-time embedded systems. *In Proceedings of International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 2–8. ACM.

[Welc *et al.*, 2010] WELC, A., HUDSON, R. L., SHPEISMAN, T. et ADL-TABATABAI, A.-R. (2010). Generic Workers : Towards Unified Distributed and Parallel JavaScript Programming Model. *In Proceedings of Programming Support Innovations for Emerging Distributed Applications (PSI EtA)*. ACM.

[You, 2014] YOU, E. (2014). Vue.js. https://fr.vuejs.org/.

[Zakai, 2011] ZAKAI, A. (2011). Emscripten : An LLVM-to-JavaScript Compiler. *In Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, pages 301–312. ACM.