Loops in Esterel: From Operational Semantics to Formally Specified Compilers

Olivier Tardieu

▶ To cite this version:

Olivier Tardieu. Loops in Esterel: From Operational Semantics to Formally Specified Compilers. domain_other. École Nationale Supérieure des Mines de Paris, 2004. English. NNT : 2004ENMP1237 . pastel-00001336

HAL Id: pastel-00001336
https://pastel.hal.science/pastel-00001336v1

Submitted on 26 Jul 2005

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

École des Mines de Paris
Centre de Mathématiques Appliquées
Sophia Antipolis

Thèse

pour obtenir le grade de
Docteur de l'École des Mines de Paris
Spécialité: Informatique Temps Réel, Robotique et Automatique

présentée et soutenue publiquement
par

Olivier Tardieu

le 24 septembre 2004

De la sémantique opérationnelle à la spécification formelle de compilateurs : l'exemple des boucles en Esterel

---

Loops in Esterel: From Operational Semantics to Formally Specified Compilers

---

Jury :

MM.	Jean-Jacques Lévy	Président	INRIA Rocquencourt
	Gérard Berry	Directeur de thèse	Esterel Technologies
	Nicolas Halbwachs	Rapporteur	CNRS / Vérimag
	David B. MacQueen	Rapporteur	University of Chicago
	Stephen A. Edwards	Examinateur	Columbia University
	Robert de Simone	Examinateur	INRIA Sophia Antipolis

à mes parents

Je tiens à remercier :

- Jean-Jacques Lévy qui après m'avoir convaincu de faire une thèse m'a fait le plaisir et l'honneur d'en présider le jury,
- Gérard Berry qui a dirigé mes travaux et dont l'intelligence et le charisme ne cessent de m'émerveiller,
- Robert de Simone qui m'a accueillis dans l'équipe TICK à l'INRIA, a encadré mes travaux au quotidien et supervisé la rédaction de ce mémoire, me prodiguant ses conseils et supportant avec patience mon esprit de contradiction,
- Frédéric Boussinot pour sa disponibilité inégalable, son humour et son énergie, à qui le contenu de cette thèse aussi bien sur la forme que sur le fond doit énormément,
- Dave MacQueen, rapporteur, venu spécialement de Chicago pour la soutenance,
- Nicolas Halbwachs rapporteur lui aussi,
- Stephen Edwards pour sa participation au jury de soutenance,
- Dumitru Potop-Butucaru qui a trouvé le temps de m'apprendre Esterel alors même qu'il rédigeait sa thèse,
- Frédéric Dabrowski, Florian Loitsch et Fabrice Peix pour leur relecture attentive du manuscrit et les innombrables corrections qui en sont le résultat,
- Marie-Solange Tissier et Benoit Legait pour m'avoir donné carte blanche pendant trois ans,
- Catherine Juncker, Évelyne Largeteau, Sophie Honnorat de l'INRIA et Dominique Micollier de l'École des Mines pour leur support tout au long de la thèse,
- ainsi que tous ceux de l'équipe TICK/AOSTE, chez MIMOSA, au CMA et ailleurs qui m'ont accompagné pendant ces trois années de travail et de loisir.

Résumé

De plus en plus souvent, en voiture ou en avion par exemple, la sécurité des personnes repose sur le bon fonctionnement de systèmes informatiques. Pour concevoir des systèmes ayant le niveau de fiabilité requis, il est nécessaire d'employer des méthodes et des outils spécifiques, en commençant par des formalismes de description appropriés.

Les langages de programmation synchrones, dont Esterel [BdS91], Scade/Lustre [HCRP91] ou encore Signal [GBGM91], concilient parallélisme et déterminisme. Ils sont donc bien adaptés à la description de systèmes embarqués critiques, pour lesquels prédictibilité et reproductibilité du comportement sont hautement souhaitables, voire impératifs. Fondés sur un modèle mathématique à la fois simple et puissant de discrétisation du temps, ces langages facilitent la vérification et la certification des programmes et de leurs implémentations.

Dans ce document nous nous intéressons à la génération de code optimisé et certifié (par une preuve mathématique) pour le langage Esterel. A cette double exigence de performance et de rigueur, nous en ajoutons une troisième : la réutilisabilité. Il existe en effet tout une gamme de compilateurs Esterel, industriels ou académiques, qui permettent à partir d'un même code source Esterel de générer des automates de contrôle (machines de Mealy), des circuits digitaux, du code C, VHDL, Verilog, etc. Nous nous concentrons sur les difficultés communes à l'élaboration de ces différents compilateurs, c'est à dire celles dues au langage Esterel lui-même plutôt que spécifiques à un type de traduction donné, avec l'ambition de spécifier et prouver des algorithmes à la fois performants et génériques.

Notre contribution consiste d'une part à clarifier des difficultés connues, formalisant et prouvant des algorithmes dérivés de compilateurs existants, d'autre part à proposer de nouveaux formalismes et algorithmes que nous avons implémenté dans notre propre compilateur Esterel.

Après une réflexion sur la sémantique du langage Esterel qui nous conduit à une révision technique de celle-ci (dans sa forme), nous nous attachons à comprendre et traiter les problèmes posés par la compilation des boucles en Esterel, c'est à dire l'exécution répétée d'un même bloc de code. L'idée apparemment banale d'itération soulève en effet dans le langage Esterel et ses variantes des problèmes spécifiques ardus nécessitant une attention particulière [Ber99], non seulement pour détecter les usages incorrects des structures de boucles, mais aussi pour compiler efficacement les boucles correctes.

Pour obtenir un schéma de compilation à la fois efficace, prouvé et portable nous proposons d'étendre le langage par la définition d'une nouvelle primitive "`gotopause`" permettant d'exprimer des branchements à la fois non locaux et non instantanés. Grâce à cette instruction originale nous pouvons décrire la compilation des boucles comme une réécriture des programmes, consistant à encoder progressivement les boucles à l'aide d'instructions "`gotopause`". Nous parvenons ainsi à traiter les difficultés posées par les boucles par un prétraitement des programmes (*preprocessing*) indépendant du type de traduction ultérieure choisi.

Après une brève présentation du langage Esterel et une introduction à la compilation certifiée, nous décrivons ici les grandes lignes de notre travail dans ses trois composantes: la sémantique d'Esterel, l'extension du langage et la vérification et la compilation des boucles.

Esterel: le modèle réactif synchrone

Esterel est un langage de programmation de haut niveau dédié à la spécification de systèmes réactifs. A l'inverse des systèmes classiques (dit transformationnels) qui reçoivent des données puis calculent un résultat et se terminent, un système réactif maintient une interaction continue avec son environnement. Il est soumis en permanence à des sollicitations extérieures auxquelles il doit répondre de façon appropriée dans des délais imposés. Les systèmes d'exploitation temps réel, les circuits digitaux, les protocoles de communication, ou encore les interfaces homme-machine sont des exemples de tels systèmes.

Dans l'approche synchrone [BB91, BCE+03] adoptée par Esterel pour la programmation des systèmes réactifs, tout calcul est supposé instantané:

- Le temps s'écoule en une suite d'instants logiques appelée horloge.
- A chaque instant a lieu une réaction qui calcule les sorties du programme en fonction de ses entrées et modifie s'il y a lieu son état interne.
- Chaque réaction est instantanée. Toutes les grandeurs mises en jeu par le programme basculent de leur ancienne vers leur nouvelle valeur simultanément et instantanément. En particulier les sorties sont disponibles sans retard sur les entrées.

Esterel est un langage concurrent. Dans un programme, de multiples composants peuvent s'exécuter en parallèle. Les communications entre composants parallèles sont supposées être instantanées:

- Les composants parallèles d'un programme partagent la même horloge
- Les composants parallèles d'un programme communiquent par l'intermédiaire de signaux booléens: un signal est présent si émis dans l'instant, absent par défaut.
- La présence comme l'absence d'un signal est diffusée instantanément. En d'autre termes, l'information de présence/absence est accessible de façon cohérente et instantanée par tous les composants parallèles du programme.

Esterel est un langage impératif. Des comportements complexes peuvent être construits au moyen de structures de contrôle classiques telles que compositions en séquence ou en parallèle, boucles, instructions conditionnelles, exceptions, etc. Par exemple, le programme ci-dessous émet le signal `O` dès que les signaux `A` et `B` (attendus en parallèle) sont reçus, à condition que `R` ne soit pas reçu avant:

```
abort
  [ await A || await B ];
  emit O
when R
```

Exprimée au moyen de règles de sémantique à la Plotkin [Plo81], la sémantique comportementale logique [BG92, Ber93b] d'Esterel donne une description simple et non ambiguë du langage, qui respecte les principes énoncés plus haut. Elle spécifie ainsi pour l'exemple précédent que si `A`, `B` et `R` sont reçus simultanément alors `O` n'est pas émis, la construction "`abort ... when R`" ayant priorité sur l'instruction "`emit O`".

Le langage Esterel et la sémantique comportementale logique sont présentés et formellement définis au Chapitre 2, où sont introduits par ailleurs les outils et les techniques formelles utilisés dans la suite du document (notions d'occurrence, contexte, bisimulation, équivalence de programmes, etc.).

Compilation certifiée

Un compilateur traduit un programme depuis un langage source vers un langage cible [ASU86]. On dit qu'il est correct s'il préserve le comportement des programmes. Prouver la correction d'un compilateur exige donc de disposer de:

- une spécification de la structure et du comportement des programmes, c'est à dire une définition de la syntaxe et de la sémantique des deux langages considérés,
- un critère pour la comparaison des comportements source et cible,
- une description de la traduction.

Une fois ces éléments réunis, il est possible de chercher une preuve mathématique de la correction de la traduction.

L'idée de certifier ainsi un compilateur n'est pas nouvelle [MP67] et a fait l'objet de nombreuses publications (cf. [Dav03]), mais sa mise en oeuvre reste en général extrêmement complexe. Le langage Esterel, et plus généralement les langages synchrones, nous semble constituer de ce point de vue une cible idéale et atteignable. De part leur champ d'application – systèmes à forte criticité – et parce qu'ils sont soigneusement formalisés autour d'un petit nombre de structures primitives, nous pensons que le jeu en vaut la chandelle.

Aussi le travail présenté dans ce document peut-il être vu comme un prolongement d'efforts antérieurs (cf. Chapitre 1), visant à fournir la spécification complète d'un compilateur Esterel accompagnée de la preuve de sa correction, sans pour autant sacrifier la performance.

Une sémantique déterministe

Les programmes Esterel imposent des dépendances entre les signaux. L'instruction suivante par exemple émet le signal `B` si et seulement si le signal `A` est absent:

```
present A else emit B end
```

Bien que dépendante de l'absence de `A`, l'émission de `B` a lieu au même instant que le test sur `A` et non après (un instant plus tard au moins). Cette simultanéité peut conduire à des incohérences si `A` dépend de `B` par ailleurs. Considérons par exemple le programme suivant, où la construction "`signal S in` p `end`" déclare le signal local `S` pour le code p:

```
signal S in
  present S else emit S end
end
```

Le signal `S` est-il présent ou absent lors de l'exécution de ce programme?

- Si `S` est supposé présent alors il n'est pas émis et donc ne peut pas être présent, par définition de la présence/absence d'un signal. Contradiction.
- Si au contraire `S` est supposé absent alors il est émis donc présent. Contradiction[1].

Bref, `S` ne peut être ni présent ni absent. En d'autres termes, ce programme n'admet aucun comportement cohérent. Conformément à cette analyse informelle, la sémantique formelle du langage ne définit pour lui aucun comportement. Il est incorrect.

[1] Le test sur `S` et l'émission éventuelle de `S` se produisent au cours d'un même instant, donc non seulement l'émission dépend du résultat du test mais le résultat du test lui-même dépend de l'émission éventuelle de `S`.

De façon duale, la sémantique comportementale logique définit deux comportements possibles pour le programme:

```
signal S in
  present S then emit S else emit O end
end
```

Si `S` est supposé présent alors il est émis et `O` ne l'est pas. Si `S` est supposé absent alors il n'est pas émis, mais `O` l'est. Ces deux comportements sont conformes aux hypothèses de synchronie énoncées plus haut et formalisées par les règles de la sémantique comportementale logique.

Cependant cette forme de non déterminisme caché – `O` peut être émis ou ne pas l'être – n'est pas en pratique jugée satisfaisante. Un tel programme est donc déclaré incorrect. En général on dit d'un programme qu'il est logiquement correct [Gon88] si, à tout moment de son exécution et pour tout input, il admet un et un seul comportement possible.

Dans ce document, nous proposons de reformuler la sémantique comportementale logique d'Esterel de façon à assurer qu'elle ne définit jamais plus d'un comportement possible pour un programme et une séquence d'inputs donnés, de façon à ce que, par construction, tous les comportements définis par la sémantique révisée soient logiquement corrects.

Concrètement, plutôt que de définir un nombre quelconque de comportements qu'il faut ensuite dénombrer, il s'agit d'assurer localement, au fur et à mesure, c'est à dire à chaque déclaration de signal local, qu'une et une seule alternative – présence ou absence du signal – est non contradictoire.

Dans la sémantique que nous proposons, pour le code "`signal S in` p `end`", le signal `S` est:

- présent si à la fois:
  - il est émis par p s'il est supposé présent dans p,
  - il est émis par p même s'il est supposé absent dans p (i.e. contradiction),
- absent si à la fois:
  - il n'est pas émis par p s'il est supposé absent dans p,
  - il n'est pas émis par p même s'il est supposé présent dans p (i.e. contradiction).

Remarquons qu'il ne s'agit pas ici de modifier la sémantique sur le fond, c'est à dire de revoir la définition du comportement des programmes corrects, mais de la transformer dans sa forme en simplifiant l'analyse de correction et la manipulation des programmes incorrects.

Cette sémantique déterministe est décrite au Chapitre 3.

Un langage étendu

Avec le développement de générateurs de code C rapide pour le langage Esterel [CPP+02, Pot02, EKH04], nous constatons que les techniques de compilation employées diffèrent fortement de celles préalablement utilisées pour la synthèse efficace de circuits. Par conséquent nous avons souhaité, autant que possible, fonder notre réflexion sur le code source Esterel plutôt que sur tout autre format de représentation dérivé, de façon à assurer la généralité de nos contributions.

Force est de constater néanmoins que pour compiler les boucles il faut les remplacer par des structures de plus bas niveau. C'est pourquoi nous avons choisi d'étendre le langage Esterel à l'aide d'une nouvelle construction "`gotopause`". Elle permet de sauter d'un point à un autre du programme sans contrainte de localité, mais en consommant un instant logique. L'exécution ne reprend son cours au point de destination qu'à l'instant suivant le saut.

Dans l'exemple suivant, le signal `A` est émis au premier instant d'exécution, puis `C` dans un deuxième instant. Le signal `B` n'est jamais émis.

```
emit A;
gotopause 1;
emit B;
1: pause;
emit C
```

Parce qu'elle comporte un délai, "`gotopause`" est une instruction simple. En particulier, au contraire des boucles "`loop` ... `end`" (voir ci-dessous) qui n'imposent aucun délai entre itérations, sa compilation en C comme en circuits ne pose aucune difficulté.

Chapitre 6, nous détaillons la formalisation du langage étendu, donnant sa sémantique et démontrant qu'elle étend la sémantique du langage original. Pour ce faire, nous avons recours à la notion d'état [Mig94] pour décrire l'avancement de l'exécution d'un programme en lieu et place de celle de résidu utilisée traditionnellement pour formaliser la sémantique comportementale logique d'Esterel. Concrètement, il s'agit de remplacer le calcul direct du programme à exécuter au prochain instant par un calcul indirect reposant sur la définition des points de contrôle actifs du programme. Nous montrons la correction de cette substitution, puis spécifions l'instruction "`gotopause`" comme une simple modification de l'état : ajout d'un point de contrôle actif correspondant à la cible du saut.

Au delà des problèmes liés à la compilation des boucles, nous pensons que cette nouvelle instruction permettra à l'avenir d'exprimer à l'aide de transformations de programmes d'autres optimisations qui ne sont pas exprimables en Esterel non étendu. De plus elle permet d'encoder naturellement des automates en Esterel, palliant ainsi à un manque évident du langage original.

La compilation des boucles

En Esterel, grâce à l'instruction "`loop` ... `end`", il est possible de répéter l'exécution d'un même bloc de code. Par exemple, le programme ci-dessous incrémente la valeur de la variable `V` à chaque instant d'exécution:

```
loop
  V := V+1;
  pause
end
```

L'instruction "`pause`" indique que l'exécution doit être interrompue – la réaction en cours se termine – pour reprendre à l'instant suivant.

Boucles instantanées

Malheureusement il est très facile d'écrire des boucles incorrectes. L'oubli de l'instruction "`pause`" dans l'exemple précédent conduit au programme absurde suivant, avec un instant qui ne se termine pas:

```
loop
  V := V+1
end
```

De façon à éviter ce genre d'erreur – appelée boucle instantanée – il semble raisonnable d'imposer de terminer chaque corps de boucle par une instruction "`pause`". Mais cette solution

n'est pas tenable en pratique, comme illustré Chapitre 1. Plutôt que d'imposer des restrictions syntaxiques fortes sur la structure des corps de boucles, Esterel fait le choix d'une vérification d'ordre sémantique en interdisant qu'une itération commencée dans l'instant ne se termine dans le même instant.

Cette interdiction est, par nature, dynamique. Le programme suivant se comporte correctement jusqu'à la réception du signal `I`:

```
await I;
loop
  V := V+1
end
```

Aussi faut-il vérifier à la compilation que de telles erreurs ne pourront pas se produire à l'exécution. Chapitre 4, nous proposons une caractérisation des boucles instantanées et un algorithme efficace pour les détecter à la compilation en utilisant des techniques d'analyse statique. Nous apportons la preuve de la correction de cet algorithme et procédons à une analyse de ses performances.

Schizophrénie

Une fois les boucles instantanées éliminées, il reste à compiler les boucles correctes. Cette opération se révèle particulièrement délicate [Ber99] car, bien qu'une seule itération puisse être complétée à chaque instant, une même réaction peut englober plusieurs itérations partielles et en particulier faire référence à plusieurs instances distinctes d'un même signal local (même déclaration).

```
loop
  signal S in
    present S then emit O end;
    pause;
    emit S
  end
end
```

Dans cet exemple, la déclaration du signal `S` est locale à la boucle. A chaque itération correspond une nouvelle instance du signal `S`. Au deuxième instant d'exécution, la première itération se termine avec l'émission de `S` et la deuxième commence avec l'exécution du test "`present S then emit O end`". Pourtant `O` n'est pas émis : le test porte sur une nouvelle instance du signal `S` qui n'a aucun lien avec l'instance de `S` émise précédemment. Par conséquent, ce programme ne doit pas être confondu avec le programme suivant, où la déclaration du signal `S` est placée à l'extérieur de la boucle, qui se comporte différemment:

```
signal S in
  loop
    present S then emit O end;
    pause;
    emit S
  end
end
```

En résumé, lorsqu'un signal est local à une boucle, il peut être nécessaire de distinguer au cours d'une même réaction plusieurs instances de ce signal. Un tel signal est dit schizophrène.

Chapitre 5 nous expliquons pourquoi les signaux schizophrènes compliquent fortement la compilation des programmes qui en comportent. Puis nous proposons une définition générale de la schizophrénie fondée sur la pose de marqueurs dans le code.

En Esterel, la sémantique de la boucle se définit par dépliage dynamique. Une boucle est une séquence infinie dont les composants sont obtenus par copie du corps de boucle initial, au fur et à mesure des itérations. En utilisant des marqueurs préservés par l'opération de copie pour identifier les déclarations de signaux locaux, il est possible de localiser les signaux schizophrènes: il suffit pour cela de comptabiliser le nombre d'occurrences d'un même marqueur rencontrées en un instant.

Enfin nous développons un algorithme efficace pour identifier les cas de schizophrénie à la compilation (par analyse statique).

Réincarnation

La schizophrénie peut être éliminée par une simple réécriture des programmes [Mig94]: la duplication des corps de boucles. Pour l'exemple précédent, le résultat de cette duplication est:

```
loop
  signal S in
    present S then emit O end;
    pause;
    emit S
  end
end
```

⟹

```
loop
  signal S in
    present S then emit O end;
    pause;
    emit S
  end;
  signal S in
    present S then emit O end;
    pause;
    emit S
  end
end
```

Aucun des deux signaux locaux du programme résultant n'est schizophrène. Alors que le programme initial a besoin de deux instances simultanées d'un même signal `S` pour s'exécuter, le second utilise deux signaux `S`, mais ne nécessite qu'une seule instance pour chacun. Evidemment, les deux programmes se comportent de la même façon.

Cette technique, parfaitement correcte et tout à fait générale, n'est pourtant pas satisfaisante. En effet, si plusieurs boucles sont imbriquées, il faut alors dupliquer récursivement les corps de boucles, ce qui peut conduire à une croissance exponentielle de la taille du programme. Nous détaillons ces différents points au Chapitre 7.

Traduction de "`loop`" en "`gotopause`"

Grâce à l'instruction "`gotopause`", nous décrivons Chapitre 7 une réécriture des programmes Esterel qui permet d'éliminer les instructions "`loop` ... `end`", par exemple:

```
loop
  signal S in
    present S then emit O end;
    pause;
    emit S
  end
end
```

⟹

```
signal S in
  present S then emit O end;
  1: pause;
  emit S
end;
signal S in
  present S then emit O end;
  gotopause 1;
end
```

Les raisons qui motivent cette transformation sont les suivantes:

- L'utilisation de l'instruction "`gotopause`" ne génère pas de boucle instantanée.
- L'utilisation de l'instruction "`gotopause`" ne génère pas de schizophrénie[2].
- La croissance de la taille du code est quadratique au plus.

Bref, cette transformation guérit la schizophrénie sans croissance exponentielle de la taille du code. Nous formalisons l'algorithme correspondant et prouvons sa correction: il préserve la sémantique du programme initial.

Enfin, en appliquant cette transformation de façon sélective et non systématique, c'est à dire en mettant à profit la détection de la schizophrénie décrite au Chapitre 5, nous obtenons un algorithme final, qui, tout en conservant les bénéfices précédemment énoncés, se révèle en pratique quasi-linéaire.

Nous démontrons le bien fondé de notre méthode par la preuve mathématique de sa correction ainsi que par une implémentation dont nous mesurons les performances (Chapitre 8).

Plan

- Nous décrivons le contexte et les motivations de notre travail (Chapitre 1).
- Nous présentons le langage Esterel et la sémantique comportementale logique (Chapitre 2).
- Nous revisitons la sémantique comportementale logique pour éliminer le non déterminisme à la source et mieux comprendre les erreurs dans les programmes (Chapitre 3).
- Nous définissons formellement les boucles instantanées (Chapitre 4) et la schizophrénie (Chapitre 5).
- Nous étendons le langage avec une nouvelle construction "`gotopause`" entièrement formalisée (Chapitre 6).
- Nous spécifions un prétraitement efficace des programmes Esterel qui détecte les boucles instantanées (Chapitre 4) et guérit la schizophrénie (Chapitre 7).
- Nous prouvons la correction de notre algorithme (Chapitres 4, 5, 6 et 7).
- Nous l'implémentons (Chapitre 8).

[2]Plus généralement, l'instruction "`gotopause`" ne génère pas problème de même nature que les boucles instantanées ou la schizophrénie. Même si elle autorise des sauts en "arrière", elle ne permet pas d'exécuter le même code plus d'une fois par instant.

Contents

1 Introduction **15**
1.1 Esterel and the Synchronous Paradigm . . . 17
1.2 Loops in Esterel . . . 19
1.3 Provably Correct Compilers . . . 22
1.4 Contributions . . . 24
1.5 Structure of the Document . . . 25

2 Pure Esterel **29**
2.1 Syntax and Intuitive Semantics . . . 29
2.2 Exceptions . . . 33
2.3 Input+Output Signals . . . 34
2.4 Logical Behavioral Semantics . . . 37
2.5 Logical Correctness . . . 43
2.6 Subterms and Occurrences . . . 45
2.7 Bisimulations and Observational Equivalence . . . 48

3 Reactive Deterministic Semantics **51**
3.1 Deterministic Semantics . . . 51
3.2 Proper Statements . . . 54
3.3 Reactive Deterministic Semantics . . . 57
3.4 Proper Subterms . . . 62

4 Instantaneous Loops **65**
4.1 Loop Safety . . . 66
4.2 (Non-)Instantaneous Statements . . . 69
4.3 Static Analysis . . . 70

5 Schizophrenia **77**
5.1 Beeps . . . 77
5.2 Instantly Reentered Subterms . . . 82
5.3 Schizophrenic Programs . . . 84
5.4 Static Analysis . . . 87

6 Esterel* **93**
6.1 A New Primitive Instruction: gotopause . . . 93
6.2 Esterel State Semantics . . . 94
6.3 Esterel* State Semantics . . . 100
6.4 Loop Safety and Schizophrenia . . . 102

7 Reincarnation **105**
7.1 Exponential Reincarnation 105
7.2 Quadratic Reincarnation 109
7.3 Quasi-Linear Reincarnation 113

8 Implementation **119**
8.1 Full Esterel* Support 119
8.2 A Prototype Compiler 121
8.3 Experiments 122

9 Conclusion **123**
9.1 Results 123
9.2 Future Work 125

A Proof of Theorem 7.21 **127**

Bibliography **131**

Index **137**

Chapter 1

Introduction

Nowadays, more and more embedded computer systems are being used for safety-critical applications [Sto96], resulting in an ever-increasing demand for reliable software and hardware design methods and tools. From nuclear power plants, to planes, cars, or medical appliances, the correct operation of computer systems is vital to ensure the safety of the public and the environment.

In addition to functional and timing requirements, designers of safety-critical embedded systems have to cope with non-functional constraints such as limited resources or predefined architectures. Many formalisms, languages, and tools have been developed or updated to support (part of) these complex modeling, specification, and programming tasks. For instance, numerous profiles (i.e. extensions) of the UML [OMG03] modeling language have been proposed to deal with real-time intensive applications [Dou97, LGT98, SR98]. As another example, the AAA methodology (Algorithm-Architecture Adequation) and SynDEx tool [GLS99] developed in the AOSTE research team in INRIA – our team – aim at providing ways to specify and automatically synthesize mappings of algorithms onto custom heterogeneous distributed architectures.

The paradigm of synchrony [BB91, BCE+03] has emerged as a simple and mathematically-sound foundation for the design of systems expressing a high level of concurrency, while maintaining deterministic, thus predictable and reproducible system behaviors. Time is assumed as divided into discrete instants. Concurrent threads run in lockstep (to one or several clocks). Communications are instantaneous. Several programming languages have adopted the synchronous approach, in particular the three French pioneers: Esterel [BdS91], Lustre [HCRP91], and Signal [GBGM91].

In contrast with traditional thread-based or event-based concurrency models that embed no precise or deterministic scheduling policy in the design formalism itself, synchronous language semantics and compilers take care of all scheduling decisions, synthesis, and validation steps. As a consequence, there can be no ambiguity about the behaviors and interactions of concurrent threads. In particular, programs are guaranteed to behave the same whatever the execution platform, which is more uncommon than thought, but obviously very convenient (and sometimes mandatory) for the design of safety-critical applications.

It should be noted here that, because of the instantaneous communication hypothesis, purely synchronous formalisms or languages are not well suited for dealing with distributed architectures, including multiprocessor systems. In general, more powerful abstractions are required, such as the globally asynchronous locally synchronous (GALS) paradigm [Cha85]. But synchronous components still occur there as fundamental building blocks.

The solid mathematical framework common to all synchronous languages facilitates validation and certification, as it enables formal reasoning about programs and implementations. A lot of works have indeed been concerned with the testing and model checking of synchronous systems, for example for Esterel in [Bou98], for Lustre in [HLR92, BCPD99], or Signal in

[KNT00, MRBS01], to only cite a few. However, whereas code generation has been extensively considered from the point of view of language features, such as arrays [Mor02], or performance, latch optimization for instance [STB96], not many efforts have explicitly aimed at establishing the formal correctness of the corresponding code generators, apart from a few noticeable exceptions which we shall discuss later in this introduction.

Nevertheless, a designer wants not only to convince himself that his design is correct, but also that the software or hardware automatically generated from the design is correct. As a result, a formalized or even certified code generator (by means of a mathematical proof) is a must-have utility in a design chain for safety-critical applications.

Automatically deriving provably correct interpreters or compilers for formally specified programming languages is one way to go. It typically consists in defining first a meta-language for the description of some family of programming languages. This includes frameworks for the description of the syntax (i.e. grammar) and semantics (i.e. interpretation) of these languages. Then, a meta-translator of such descriptions into interpreters or compilers is defined. Finally, a proof of the correctness of this translator must be completed. This approach led for instance to the Centaur system [BCD+88], which can be used to automatically derive language specific environments, including interpreters, from formal language specifications (for some subclass of natural semantics [Kah87] specifications). It has been applied to Esterel in [Ber90]. Similarly, provably correct compiler generators have been built, for example in [Pal92].

Of course, as the family of supported languages or language features gets wider, or when the number of code generation and optimization techniques considered goes up, then the definition and proof of the translation become increasingly difficult. Therefore, focusing on a single language, or even a unique compiler, makes sense as well. In fact, it is still unclear whether a compiler both competitive and formally certified can be built for a mainstream language such as C or Java.

In this document, we concentrate on the synchronous language Esterel. Its syntax is imperative, fit for the design of safety-critical embedded systems where the control-handling aspects prevail. In addition to traditional control-flow operators, it defines suspension and preemption mechanisms compatible with concurrency while preserving determinism [Ber93a]. It allows threads to instantly react to absent signals as well as received signals, thanks to instant-based causality principles. Esterel enjoys a full-fledged formal semantics, expressed with structural operational semantics rules [Plo81], which, in our view, is both simple enough to make formal reasoning about programs and code generation tractable, and rich and mature enough to be worth the realistic effort.

Historically, Esterel was conceived as a high-level description language for control automata. Then, a wide variety of compilers have been implemented for Esterel, both in academic and industrial contexts. In addition to FSM (finite state machine [Kle56]) synthesis [BG92], some of these compilers now natively support gate-level logic synthesis [Ber92], fast C code generation [CPP+02, Pot02, EKH04], as well as hardware synthesis via VHDL or Verilog backend synthesizers [ET03]. Over the years, dozens of pieces of these code generation schemes have been discussed, documented, and sometimes formalized and proved. Nevertheless, several complex algorithms have not been dealt with yet in a satisfactory way, if at all. As a consequence, even if the language itself is formally specified, as well as a few models used for intermediate representation levels, actual compilers are not.

In this work, we address the problem of efficient and provably correct code generation for Esterel, starting with precise compiler specification. Our contribution partly consists in clarifying already identified issues, formalizing and proving algorithms derived from existing compiler implementations, but we also propose new formalisms, new static analysis techniques, and new program transformations to enable the development of improved code generation algorithms,

which we have implemented in a new prototype compiler. Our goal is to achieve performances at least as good as existing compilers, while proving the correctness of our methods. Moreover, rather than focusing on any particular target language or compiler, we aim at providing algorithms that are not only (*i*) efficient and (*ii*) provably correct, but also (*iii*) that can easily benefit existing Esterel compilers, targeting both hardware and software code.

We concentrate on the `loop` construct of the language, which is the only primitive construct that makes it possible to branch back to the beginning of a piece of code, thus enabling repeated executions of this piece of code. As a result, many important issues only occur in programs containing loops. Here, we precisely focus on these issues, that we have found to be critical to the quality of the generated software or hardware code. Current loop-handling techniques differ in nature and performance from one compiler to another, with severe drawbacks, as we shall see.

While specific in its aim, this work seems to us highly generic in its demonstration of how transformational steps involved in the code generation process can be derived mathematically from the language semantics.

We now briefly present an overview of synchronous languages and Esterel in Section 1.1, discuss loops in Esterel in Section 1.2, and provably correct compilers in Section 1.3. We sketch our contributions in Section 1.4, and provide the structure of the document in Section 1.5.

1.1 Esterel and the Synchronous Paradigm

Esterel is a high-level imperative synchronous programming language for the specification of control-oriented reactive systems. Esterel was born in the nineteen eighties [BC84] and evolved since then. In this work, we consider the Esterel v5 dialect of Esterel [Ber00a], endorsed by current academic compilers from Columbia University [EKH04] and INRIA [Ber00b].

Reactive Systems

Reactive systems [Hal93, HP85] are required to continuously react to input events with matching output events within appropriate time frames. Examples of reactive systems include real time operating systems, digital circuits, communication protocols, man-machine interfaces, monitoring systems, etc. Typical reactive systems are made of interconnected control and data parts. For instance, a microprocessor pilots an arithmetic and logic unit with an instruction decoder. In Esterel, the emphasis is put on events and communication rather than calculus, for the programming of reactive systems in which the control-handling aspects prevail.

Synchronous Paradigm

In the synchronous approach [BB91, BCE+03], computation and communication delays are neglected, so that it becomes possible, in a first phase of the development at least, to work with a simple but powerful abstraction of time:

- Time flows in a discrete manner, being made of a succession of well-identified instants.
- At each instant, a computation occurs, called reaction, which computes the outputs and the next state of the program from the inputs and the current state of the program.
- This reaction is computed in zero time. All quantities of the system, including inputs and outputs, switch from their old to their new values instantly and simultaneously. In particular, outputs are available as soon as inputs are.
- Threads sharing the same clock (i.e. time reference), can communicate via instantly broadcast signals. On the other hand, threads having unrelated clocks cannot communicate.

The synchronous paradigm has been especially advocated for the design of digital circuits [Ber92], embedded systems [Edw00], and real-time applications [Ber89]. A number of synchronous programming languages have been developed, including imperative languages such as Esterel and Quartz [Sch01b], data-flow languages like Scade/Lustre [HCRP91] and Signal [GBGM91], or Statecharts [Har87] like graphical design languages, SyncCharts [And95, And96] and Argos [Mar91] for example. Attempts at mixing various programming styles have been reported as well [PMM+98].

Signals

Esterel deals with signals. Signals have a Boolean status, which obeys the signal coherence law: a signal is absent by default, present if emitted in the current instant. Thanks to the synchronous hypothesis, both absence and presence are instantly broadcast, and simultaneously available in a consistent fashion to all threads of execution (there is single clock in Esterel).

Imperative Style

Esterel programming style is imperative. Sophisticated control-flow patterns can be built by sequential and parallel compositions of behaviors, tests, loops, preemption and suspension mechanisms. For example, the following program emits the signal `O` as soon as both the signals `A` and `B` have occurred – they are awaited in parallel – provided that `R` does not occur first:

```
abort
  [ await A || await B ];
  emit O
when R
```

Formal Semantics

Expressed with structural operational semantics rules [Plo81], the logical behavioral semantics [BG92, Ber93b] of Esterel, formalizes the signal coherence law and precisely defines the behavior of programs. In the previous example for instance, this semantics specifies that if `A`, `B` and `R` occur in the same instant, then `O` is not emitted, as the outer `abort` construct has priority over the inner `emit` construct.

Causality

Programs express relations between signals. For instance,

```
present A else emit B end
```

emits `B` if `A` is absent. Because of the synchronous hypothesis, the status of `B` is available no later than the status of `A` is. In fact, they are evaluated simultaneously. In general, the consequences of a choice are simultaneous to the choice itself, and may contribute to it. This distinctive feature of synchronous formalisms, called instantaneous feedback, may lead to trouble. For example,

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

admits no possible interpretation conforming with the signal coherence law, provided that `S` is a local signal not emitted elsewhere:

- If we suppose `S` present, then it is not emitted. Contradiction.
- If we suppose `S` absent, then it is emitted. Contradiction.

Over the years, many solutions have been proposed to deal with this issue, referred to as causality in synchronous formalisms [Gon88]. They range from non-instantaneous reaction to signal absence as in the SL language [BdS95], to quasi-synchronous hypotheses typical of hardware design languages, which reintroduce "time" within instants by means of δ-delays [IEE94].

For Esterel, causality analysis has been refined many times, aiming at accepting the largest possible set of programs of "reasonable" behavior. The two most significant attempts at such a definition in our view, are the logical correctness criterion and the constructive semantics of Esterel:

- A program is said to be logically correct[1] [BG92] iff at any step of any execution (i.e. for any sequence of inputs), there exists exactly one valuation of its signals compatible with the signal coherence law. Basically, logical correctness guarantees a deterministic deadlock-free execution. But the cost of the computation of this behavior can be prohibitive (NP-hard).

- The constructive semantics [Ber99], inspired from digital circuits and three-valued logic, ensures by rejecting more "unreasonable" programs that program behaviors can be unambiguously computed without "speculation". Such programs are said to be constructive. For instance, if `S` is a local signal not emitted elsewhere,

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

  is logically correct – `S` is present – but not constructive, as, intuitively, the status of `S` must be guessed prior to its emission.

The logical correctness criterion and the constructive semantics are thus two possible ways of defining "reasonable" behaviors, that is to say causal programs. In each case, causality analysis consists in deciding whether or not a program is causal.

In practice, both criteria are too expensive to check exactly for large programs. Therefore, nowadays compilers implement several alternative procedures for defining and checking causality, of differing powers and costs. All these procedures enforce logical correctness, which is the agreed minimal correctness criterion for Esterel programs. But beyond logical correctness, diverging approaches coexist.

The question raised here is not to decide which behavior a program should have – this is already achieved by the logical behavioral semantics for all logically correct programs – but whether this behavior should be considered to be "reasonable" or not, and how it should be computed in practice.

In this work, we are interested in loop-related issues. It turns out that these issues are essentially independent from causality. Anyway, they are typically dealt with before causality analysis in the code generation process. As a consequence, we shall not detail the constructive semantics of Esterel, but rather stick with the logical behavioral semantics of Esterel, and take into account all logically correct programs, in order to provide techniques and tools applicable whatever the causality analysis.

1.2 Loops in Esterel

Thanks to loops, it is possible to repeat behaviors over time. Moreover, because in Esterel there is neither recursion nor a jump instruction, this is the only way repeated behaviors can be specified, that is to say the only way to go "back".

[1] Technically, we shall say it is strongly correct (cf. Chapter 2).

```
loop
  present I then
    pause;
    emit O
  else
    pause
  end
end
```

instant	0	1	2	3	4
input I	×	×		×	
output O		×	×		×

Figure 1.1: Computing the Previous Status of a Signal

For example, in order to increment the variable V in each instant of execution, one may write:

```
loop
  V := V+1;
  pause
end
```

Remark the pause instruction. It specifies that the current reaction is finished, and requires the execution to be restarted in the next instant.

Similarly, in each instant, the program of Figure 1.1 emits O if I was present in the previous instant, thus providing the previous status of the signal I.

Loops are responsible for the introduction of two complex issues in Esterel: instantaneous loops and schizophrenia, which we discuss below.

Instantaneous Loops

First, incorrect programs can be easily written because of loops. Let us suppose we forgot the pause instruction in our first loop example, and wrote instead:

```
loop
  V := V+1
end
```

Then, infinitely many iterations have to be completed within a single reaction, that is to say during the same instant, which Esterel does not admit. Such "instantaneous" loops have to be rejected as incorrect.

In order to avoid such errors, we could think of constraining each loop body to end with a pause instruction for instance. But then, the behavior of our second example (Figure 1.1) becomes very difficult to reproduce, requiring not less than three loops (in the kernel language of Chapter 2), as illustrated in Figure 1.2.

Rather than imposing strong syntactic restrictions on the loop construct, Esterel semantics verifies that each iteration of a loop retains the control for at least one instant, so that at most one iteration can be completed per instant, thus effectively bounding the amount of computation needed for one reaction. This restriction is of dynamic nature, as it is imposed in each reaction, and may lead to runtime errors. For example, the following program fails upon the reception of I:

```
await I;
loop
  V := V+1
end
```

```
loop                                 trap T in
  present I then                       loop
    pause;                               present I then exit T end;
    emit O                               pause
  else                                 end
    pause                            end;
  end                 becomes        pause;
end                                  loop
                                       emit O;
                                       trap T in
                                         loop
                                           present I then exit T end;
                                           pause
                                         end
                                       end;
                                       pause
                                     end
```

Figure 1.2: Programming with "`loop ...; pause end`"

In the context of embedded or safety-critical systems design however, runtime errors cannot be tolerated. Esterel compilers have to predict and prevent them, and replace a runtime check by a compile time filtering of programs. In this work, we want to specify and verify such a compile time analysis of programs.

Schizophrenia

Second, even if non-instantaneous, loops remain complex structures, difficult to compile correctly, because a reaction may spread across two iterations of the same loop, and in fact more as we shall see. Let us consider an example:

```
loop
  signal S in
    present S then emit O end;
    pause;
    emit S
  end
end
```

Since the signal `S` is local to the loop in this program, each iteration refers to a fresh signal `S`, so that `O` is never emitted. In each reaction (starting from the second one), one iteration finishes and another one starts. The declaration of `S` is left and instantly reentered, so that two statuses of `S` are computed per instant. In general, because of the instantaneous feedback mentioned before, these statuses have to be computed simultaneously, rather than sequentially. In other words, two instances of `S` are simultaneously alive, which compilers have to carefully distinguish. We say that the signal `S` here, hence the program itself, are schizophrenic.

The programming style advocated by Esterel – local declarations plus imperative loops – naturally leads to schizophrenic specifications [CI89]. So, compilers cannot afford to reject such program patterns. In this work, we would like to grasp the ins and outs of schizophrenia, and, once and for all, get rid of it, by automatically rewriting schizophrenic programs into equivalent non-schizophrenic programs, the latter more easily compiled, optimized, and debugged.

In fact, this has been proposed before. A simple solution, first described by Mignard [Mig94], consists in recursively unfolding loop bodies once, that is to say recursively duplicating loop bodies, thus producing for the above example the equivalent program:

```
loop
  signal S in
    present S then emit O end;
    pause;
    emit S
  end;
  signal S in
    present S then emit O end;
    pause;
    emit S
  end
end
```

In the rewritten program, neither the first signal declaration block, nor the second one can be left and instantly reentered. At most one instance of each declared signal is needed per instant.

This rewriting technique is called reincarnation as it explicitly distributes the several simultaneous instances of S, that is to say "incarnations", into several distinct "bodies". But it can be exponential in case of nested loops. As a consequence, more efficient program transformations are necessary, which we shall investigate in this work.

1.3 Provably Correct Compilers

A compiler translates programs in a source language into programs in a target language [ASU86]. It is said to be correct iff it translates any source program to a target program having the same behavior. In order to prove the correctness of a compiler, one needs:

- a definition of program structures and behaviors, that is to say of the syntax and semantics of both languages,
- a means of comparing these behaviors, that is to say some sort of equivalence criterion,
- a description of the code generation itself.

Provided that all required elements are precisely specified, it becomes possible to look for a mathematical proof of the correctness of the code generation. Today, the ultimate goal of such an agenda is to complete all the steps of this reasoning process using a theorem prover/proof assistant, in order to mechanically guarantee (i.e. certify) the correctness of the proof.

It should be noted at this point that formal compiler verification has basically nothing to do with the so-called "certification" processes favored for instance by aircraft manufacturers, such as the Federal Aviation Administration "RTCA DO-178B" standard. In this document, we think of certified compilers as formally verified compilers.

Attempts at proving the correctness of a compiler can be traced back to the nineteen sixties with the work of McCarthy and Painter [MP67]. In November 2003, M.A. Dave gathered a hundred references under the title "Compiler verification: a bibliography" [Dav03]. Rather than going through the whole compiler verification history for general purpose languages, we shall focus here on synchronous languages and formalisms. We encourage the reader to refer to the afore mentioned bibliography, to the pioneering 1981 book by W. Polak [Pol81], or for instance

to the more recent book by S. Stepney [Ste93], for an in-depth discussion of the theory and practice of compiler verification.

Concurrent languages are reputedly more complex than sequential languages. This can be observed when it comes to proving compiler correctness [Gla94]. But, because they reconcile concurrency and determinism, synchronous languages are especially convenient amongst concurrent languages. Moreover, synchronous languages have been introduced for software and hardware reliability. Thus, formalizing synchronous systems is an active research area.

Of course, Esterel is not the only synchronous language that has attracted attention. Various axiomatizations of synchronous languages have been reported, for instance for Signal and Lustre there are:

- In [NBT98, Now99], D. Nowak et al. formalize the trace semantics of Signal within the proof assistant Coq [CDT01].
- In [BCPD99], S. Bensalem et al. describe the principles of a translation of Lustre programs to the PVS specification and verification system [SORS99].

Most often, these reports are only concerned with the verification of program properties, rather than compiler properties. Nevertheless, a few focus on compiler correctness:

- In [PSS98a], A. Pnueli et al. introduce the idea of translation validation. Rather than trying to prove in advance the correctness of a compiler for all input programs, they advocate for the validation of individual runs of the compiler. By appending a validation phase at the end of the code generation process, they ensure for each run of the compiler that the code generated correctly implements the submitted source program. In [PSS99], they apply this idea to the translation of Signal programs into C code, by means of a code validation tool (CVT) [PSS98b].
- In [BH01], S. Boulmé and G. Hamon describe the embedding in Coq of the Lucid-Synchrone programming language [CP99]. Lucid-Synchrone programs must satisfy some non-trivial static properties to be valid. Thanks to this embedding, part of the required analyses can be performed by the Coq type checker, being automatically derived from the language specification.

Let us now concentrate on Esterel:

- First, it should be noted that the numerous works that have contributed to formalizing Esterel semantics can be viewed as part the collective effort toward the design and proof of a certified compiler. There are many, which we have already referred to, or will in due time, and we shall not discuss them again here.

  Similarly, all (partial) descriptions of code generation schemes for Esterel contribute to the effort. There are lots of them. As mentioned before, available techniques range from FSM synthesis, to logic synthesis (also called Boolean equation synthesis or circuit synthesis), and more recently fast C code generation, as well as VHDL and Verilog synthesis.
- In [Ber99], Berry completed the definition of a semantics of Esterel (the constructive semantics), a semantics of digital circuits (constructive circuits), and the description of a formal translation of Esterel programs into digital circuits. As a result, he can express a formal equivalence theorem, meaning that the constructive circuit obtained from this translation exactly implements the constructive semantics of the source Esterel programs[2].

[2] Technically, Berry's theorem holds for loop-safe programs only (cf. Chapter 4).

However, the formal proof of the "theorem" still remains to be completed. Moreover, the translation formalized here is far from the circuit synthesis implemented in [Ber00b]. Because of its better understanding of schizophrenia, the real compiler can produce circuits quadratically smaller than the formalized translation in the best case, and significantly smaller in practice.

- In [Ter00], D. Terrasse formalized part of the translation of the Esterel language into constructive circuits, within the proof assistant Coq. Then, she formally established that the outputs of the first instant of execution of the generated circuit are correct.

  Unfortunately, her approach was limited to combinatorial Esterel programs, that is to say programs without `pause` instructions. As a result, she could not consider `loop` constructs either, as loops only make sense in non-instantaneous programs.

- More recently, in a series of papers [Sch01a, Sch01b, SW01, SBS04], Schneider et al. completed the formal verification of a logical[3] circuit synthesis for a variant of the Esterel language, using the HOL theorem prover [GM93].

  This is a great achievement. But this translation remains to be optimized, being basically just as efficient as Berry's formal translation.

Our work is in line with this research. In essence, we are trying to design (*i*) the algorithms and (*ii*) the proof techniques that are still lacking in Berry's and Schneider's works, and would make it possible to formally verify an optimizing hardware compiler for Esterel, that is to say efficiently taking care of schizophrenia, and including a realistic compile time filtering of instantaneous loops.

Moreover, we observe with the development of C code generators for Esterel, that efficient software synthesis and efficient hardware synthesis obey very different constraints and objectives [CPP+02, Pot02, EKH04]. As a consequence, in this work, we shall try to stick to source code analyses and source to source transformations of Esterel programs, applicable to the efficient and provably correct synthesis of both software and hardware.

1.4 Contributions

While instantaneous loops and schizophrenia are not new issues in Esterel and have been extensively discussed and already dealt with within previous compiler implementations, we believe that they are not yet fully understood. Loop-handling techniques differ from one compiler to another, so that:

- More or fewer programs are rejected because of "potentially instantaneous loop" errors, directly impacting users.

- More or less code or circuit replication is carried out to deal with schizophrenia. Current fast C code generators, as well as Berry's formal translation and Schneider's certified compiler, have an expansion ratio that can be quadratically worse than the optimized digital circuit synthesis implemented in [Ber00b].

Our main contributions are the following:

- We significantly revise the logical behavioral semantics of Esterel, in order to formally define and reason about loop errors and schizophrenia.

[3]In contrast with Berry and Terrasse, Schneider considers a logical semantics of Esterel rather than a constructive semantics, thus "logical" rather than constructive circuits.

- We extend Esterel with a non-instantaneous jump instruction, easily compiled, and making new program transformations possible. We fully formalize the extended language: Esterel*.
- We specify and implement a preprocessor that rewrites any Esterel program into a semantically equivalent, loop-error-free, non-schizophrenic Esterel* program, making the efficiency of the hardware-centric algorithms implemented in [Ber00b] available to all compilers.
- We complete a hand-written proof of the correctness of our preprocessor for the pure Esterel subset [Ber93a] of Esterel (see below).

These achievements will form the content of this PhD thesis.

1.5 Structure of the Document

This document is organized into the following chapters:

Chapter 2: Pure Esterel

Pure Esterel is the fragment of the Esterel language where data variables and data-handling primitives are abstracted away. As our main concern is with control-flow primitives, we first focus on the pure Esterel language. In this chapter, we describe the pure Esterel language, formalize its logical behavioral semantics and logical correctness. We return to full Esterel in Chapter 8.

Here, our only contribution is the definition of "input+output" signals (cf. Section 2.3) instead of traditional inputoutput signals [Ber00a], thanks to which we can formally identify programs and statements (i.e. pieces of programs), thus ending with a description of the semantics more compact than usual.

In addition, we introduce various formal techniques and tools, which we extensively rely on in the sequel. We show how to keep track of occurrences of statements in programs along the execution using contexts and tags. We also define observationally equivalent programs as well as observationally equivalent semantics, using bisimulations.

Chapter 3: Reactive Deterministic Semantics

In the course of this research, we came to the conclusion that the logical behavioral semantics and the logical correctness criterion described in Chapter 2, are not the best starting point for the definition of errors in Esterel programs, especially loop errors. The logical behavioral semantics may define for a given piece of code too few (zero) or two many (two or more) behaviors compatible with the signal coherence law. These errors may cancel each other, so that, in the end, exactly one behavior remains, and the program is found to be logically correct.

This observation motivates the introduction of a revised logical behavioral semantics. In this chapter, we describe a "deterministic semantics", which defines either zero or one behavior for any piece of code, thus avoiding the risk of hidden errors. We define "proper" programs as deadlock-free programs w.r.t. the deterministic semantics. Since the deterministic semantics reveals more errors than the logical behavioral semantics, logically correct programs are not always proper. Reciprocally, we show that proper programs are logically correct.

We further extend this semantics with explicit error-handling rules, producing a "reactive deterministic semantics". It defines exactly one behavior for any piece of code, but marks some behaviors as improper. This last semantics lets us precisely identify errors in improper programs. In particular, we can classify the loops of a given program into proper and improper loops, which comes in very handy in Chapter 4.

Chapter 4: Instantaneous Loops

As explained before, Esterel semantics dynamically verifies that executed loops are not instantaneous. We say that a program is loop safe iff this dynamic check is not necessary. We establish that whereas logically correct programs are not loop safe in general, proper programs always are. More precisely, programs with proper loops are loop safe. As announced, the logical behavioral semantics is not an adequate framework for the analysis of incorrect loops. On the other hand, the deterministic semantics (or reactive deterministic semantics) is fine.

In order to guarantee loop safety, we define non-instantaneous statements as statements whose execution cannot terminate instantly, whatever the context of execution. We verify that programs with non-instantaneous loop bodies, have proper loops, hence are loop safe.

We then formalize the compile time decision procedure that identifies potentially instantaneous loop bodies in [Ber00b]. Using abstract interpretation techniques [CC77], we prove that it is correct, that is to say catches all loop unsafe programs. We analyze its trade-offs, and study its computational complexity.

Chapter 5: Schizophrenia

We thoroughly discuss the reasons for instantly reentered blocks of code to raise specific difficulties in Esterel-like languages. Using a "`beep`" instruction, we demonstrate than instantly reentered signal declarations and instantly reentered parallel statements lead to complex behaviors, requiring ad hoc code generation techniques. We say they are schizophrenic constructs.

Esterel semantics defines loops by dynamic unfolding of the loop body: a loop is an infinite sequence composed of an infinite number of copies of the loop body, made on demand (lazily). Using tags preserved in this unfolding, we can keep track of occurrences originating from the same statement. By checking whether a tag is encountered more than once in an instant or not, we can decide whether a statement is instantly reentered or not, therefore whether a program contains schizophrenic constructs or not, e.g. is schizophrenic or not.

We further extend the decision procedure of Chapter 4 to detect schizophrenic program patterns, using similar conservative abstraction mechanisms to preserve efficiency.

Chapter 6: Esterel*

We have seen that schizophrenia can be cured by source-level program rewriting techniques. Such techniques, however, are inherently inefficient. In order to enable more efficient program transformations, we propose to extend the Esterel language with a new "`gotopause`" primitive, which behaves as a non-instantaneous jump instruction compatible with Esterel synchronous concurrency. In other words, `gotopause` enables delayed branching to remote locations in the program. In this chapter, we build the extended language, which we name Esterel*.

As jumps disregard the program structure, we first have to reformulate the logical behavioral semantics of Esterel into a logical state semantics, which makes it possible to formalize jumps easily. We prove this state semantics to be observationally equivalent to the logical behavioral semantics. We remark that syntactic restrictions have to be imposed on `gotopause` occurrences for them to make sense. Formally, we define well-formed Esterel* programs and fully formalize their semantics.

We discuss loop safety and schizophrenia in Esterel*.

Chapter 7: Reincarnation

Using the new primitive of Esterel* and the analysis of schizophrenia specified in Chapter 5, we build a very efficient algorithm for reincarnation, which we prove to be semantics-preserving

and complete, that is to say it transforms any program into a non-schizophrenic observationally equivalent program.

Technically, we start from the original exponential rewriting technique due to Mignard. We then achieve a quadratic program transformation by expanding all `loop` constructs using `gotopause` constructs. We finally make this transformation quasi-linear in practice, by replacing the systematic unfolding of loops with a selective unfolding limited to schizophrenic signal declarations and parallel statements.

Chapter 8: Implementation

In Chapter 7 and before, we concentrate on formalizing and proving program analyses and transformations on a kernel pure Esterel language. Full Esterel adds to pure Esterel the ability to manipulate data: private variables, shared values, counters, registers, etc. In this chapter, we briefly sketch how our methods can be extended to cope with data. We describe our prototype implementation of a full Esterel v5 [Ber00a] compiler based on this extended preprocessing, and discuss early experiments.

Chapter 2

Pure Esterel

Without loss of generality, we focus in this work on a kernel language inspired from Berry [Ber99], which retains just enough of the pure Esterel language to attain its full expressive power. We define our kernel language and describe its primitive constructs in Section 2.1, formalize its logical behavioral semantics in Section 2.2 to Section 2.4, and discuss program correctness in Section 2.5. We consider occurrences of terms in statements and their reductions in Section 2.6, and finally define observational equivalence and dead code in Section 2.7.

2.1 Syntax and Intuitive Semantics

Figure 2.1 describes the grammar of our kernel language, as well as the intuitive behavior of its primitive constructs. The non-terminals p and q denote *statements*, S *signals*, and T *exceptions*. In this work, the words *statement* and *program* are synonymous.

The infix "`;`" operator binds tighter than "`||`". Brackets "`[`" and "`]`" may be used to group statements in arbitrary ways. In a `present` statement, the `then` or `else` branch may be omitted. For example, "`present` S `else` p `end`" is a shortcut for "`present` S `then nothing else` p `end`".

Signals and exceptions are lexically scoped and respectively declared within statements by the constructs "`signal` S `in` ... `end`" and "`trap` T `in` ... `end`".

In this first section, we suppose that all occurrences of exceptions are bound by a declaration, that is to say that each "`exit` T" instruction is in the scope of a "`trap` T `in` ... `end`" construct. We shall consider free exceptions in Section 2.2.

p, q	::=	`nothing`	does nothing and terminates instantly
		`pause`	stops the execution till next instant
		p`;` q	executes p followed by q if/when p terminates
		p `||` q	executes p in parallel with q
		`[`p`]`	executes p
		`loop` p `end`	repeats p forever
		`signal` S `in` p `end`	declares signal S in p
		`emit` S	emits signal S
		`present` S `then` p `else` q `end`	executes p if S is present, q otherwise
		`trap` T `in` p `end`	declares and catches exception T in p
		`exit` T	raises exception T
S, T	::=	*identifier*	

Figure 2.1: Primitive Pure Esterel Constructs

In addition, we distinguish amongst free signals:

- *input signals* which only occur in `present` statements,
- *output signals* which only occur in `emit` statements,
- *input+output*[1] *signals* which occur both in `present` and `emit` statements

Let us consider the following statement:

```
present I then emit S end;
present S then emit O end
```

Here, the signal `I` is an input signal, `O` is an output signal, `S` is an input+output signal. In this first section, we suppose free signals to be either input or output signals. We shall discuss input+output signals in Section 2.3.

Instants and Reactions

An Esterel statement runs in steps called *reactions* in response to the *ticks* of a *global clock*. Each reaction takes one *instant*. Primitive constructs execute in zero time except for the `pause` instruction.

When the clock ticks, a reaction occurs, which computes the output signals and the new state of the program from the input signals and the current state of the program. It may either finish the execution instantly or delay part of it till the next instant, because it reached at least one `pause` instruction. In the latter case, the execution is resumed when the clock ticks again from the locations of the `pause` instructions reached in the previous instant. And so on.

The execution of the statement "`emit A; pause; emit B; emit C; pause; emit D`" emits the signal `A` in the first instant of its execution, then emits `B` and `C` in the second instant, finally emits `D` and terminates in the third instant. It takes three instants to complete, that is to say proceeds by three reactions. The signals `B` and `C` are emitted *simultaneously*, as their emissions occur in the same instant of execution. In particular, "`emit B; emit C`" and "`emit C; emit B`" cannot be distinguished in Esterel.

Synchronous Concurrency

Concurrency in Esterel is synchronous. One reaction of the parallel composition "p `||` q" is made of exactly one reaction of each non-terminated branch (p and q, or p, or q), until the termination of all branches:

```
[
  pause; emit A; pause; emit B
||
  emit C; pause; emit D
];
emit E
```

This statement emits `C` in the first instant of its execution, then emits `A` and `D` in the second instant. At this point the second branch terminates. In the third instant, `B` and `E` are emitted, and the execution terminates. Again, `A` and `D` are emitted simultaneously, `B` and `E` are emitted simultaneously.

[1] Input+output signals are not the inputoutput signals of [Ber99, Ber00a], hence the "+" (cf. Section 2.3).

Exceptions

Exceptions are lexically scoped, declared, and caught by the "`trap` T `in` ... `end`" construct, raised by the "`exit` T" instruction. In sequential code (no parallel statement), the `exit` statement behaves as a "goto" to the end of the matching `trap` block:

```
trap T in
  emit A; pause; emit B; exit T; emit C
end;
emit D
```

This statement emits `A` in the first instant, then `B` and `D` and terminates in the second instant. Signal `C` is never emitted.

An exception occurring in a parallel context causes all parallel branches to terminate instantly, while finishing their current instantaneous computations:

```
trap T in
  emit A; pause; emit B; exit T; emit C
||
  emit E; pause; emit F; pause; emit G
end;
emit D
```

The signals `A` and `E` are emitted in the first instant, then `B`, `F`, and `D` in the second and final one. Neither `C` nor `G` is emitted. Exceptions implement *weak preemption*: "`exit T`" in the first branch does not prevent `F` to be simultaneously emitted in the second one.

Exception declarations may be nested:

```
trap T in
  trap U in
    exit T || exit U
  end;
  emit A
end
```

The signal `A` is never emitted. The outermost exception, `T` here, has always priority over inner ones, `U` in this example.

Loops

The execution of "`loop emit S; pause end`" emits `S` at each instant, and never terminates. Finitely iterated loops may be obtained by combining `loop`, `trap` and `exit` statements, as in the kernel expansions of "`await` S" and "`await_not` S":

$$\texttt{await}\ S \stackrel{def}{=} \texttt{trap T in loop pause; present}\ S\ \texttt{then exit T end end end}$$
$$\texttt{await_not}\ S \stackrel{def}{=} \texttt{trap T in loop pause; present}\ S\ \texttt{else exit T end end end}$$

Loop bodies should not be *instantaneous*. For example "`loop emit S end`" is not a correct program. Such a pattern would prevent the reaction to reach completion, and the instant to end. Therefore, loop bodies are required to raise an exception or retain the control for at least one instant, that is to say execute at least a `pause` or an `exit` statement in each iteration.

Signals

The instruction "`signal` S `in` p `end`" declares the *local* signal S in p. In each instant,

- A local or output signal S is *emitted* iff at least one "`emit` S" statement is executed in this instant. Input signals are never emitted, since they never occur in `emit` statements.
- The *status* of a local or input signal S is either *present* or *absent*. If S is present then all "`present` S `then` p `else` q `end`" statements executed in this instant, execute their "`then` p" branch in this instant; if S is absent they all execute their "`else` q" branch.
  - A local signal is present iff it is emitted in this instant.
  - An input signal is present iff it is provided by the *environment* of execution in this instant.

In this work, we consider output signals, which never occur in `present` statements, to have no status at all (cf. Section 2.3). In each instant,

- The environment provides the status of the input signals.
- The reaction occurs, deciding which local and output signals are emitted.
- The environment observes the emitted output signals.

For example,

- ```
  signal A in
    present A then emit B end
  ||
    emit A
  end
  ```
  Both the local signal `A` and the output signal `B` are emitted. The local signal `A` is present.
- ```
  signal S in
    present I then emit S end
  end
  ```
  The status of `I` depends on the environment, and the status of `S` follows from that of `I`.
- ```
  signal S in
    emit S;
    pause;
    present S then emit O end
  end
  ```
  The signal `S` is emitted in the first instant of execution only, thus `O` is not emitted by this statement, as `S` is absent at the time of the "`present S then emit O end`" test.

Suspension and Abortion

Our kernel instructions are those of Berry [Ber99] except for the "`suspend` p `when` S" statement. It can nevertheless be encoded by substituting all `pause` instructions of p by "`await_not` S" instructions:

$$\texttt{suspend}\ p\ \texttt{when}\ S \stackrel{def}{=} p[\texttt{await\_not}\ S/\texttt{pause}]$$
  ```

Thanks to `trap` and `suspend` statements, we can also encode the constructs for *weak* and *strong abortion* defined in Esterel v5 [Ber00a]:

$$\texttt{weak abort } p \texttt{ when } S \stackrel{def}{=} \texttt{trap T in } p\texttt{; exit T || await } S\texttt{; exit T end}$$
$$\texttt{abort } p \texttt{ when } S \stackrel{def}{=} \texttt{weak abort suspend } p \texttt{ when } S \texttt{ when } S$$

where `T` is a fresh exception name.

2.2 Exceptions

In the informal semantics of the previous section, the behavior of a statement depends on the relative priorities of exceptions. For example,

- ```
  trap T in
    trap U in
      exit T || exit U
    end
  end
  ```

  The inner parallel statement "`exit T || exit U`" raises the exception `T`, since `T` has priority over `U`.

- ```
  trap U in
    trap T in
      exit T || exit U
    end
  end
  ```

  The inner parallel statement "`exit T || exit U`" raises the exception `U`, since `U` has priority over `T`.

In other words, the behavior of "`exit T || exit U`" depends on its context of occurrence. In order to specify the semantics of `exit` statements independently from `trap` statements, and more generally enable the definition of a *structural* semantics for pure Esterel, we may:

- either say that "`exit T || exit U`" raises both `T` and `U`, and deal with multiple exceptions,
- or encode priority levels into `exit` statements.

We shall retain the second approach, as it is closer to implementation. We decorate `exit` statements with the respective *depths* of exceptions, which we note "`exit` T_d". The depth $d \in \mathbb{N}$ of "`exit` T" is the number of exception declarations that T will traverse before being caught:

- If "`exit` T_d" is enclosed in a declaration of T, then d is be the number of exception declarations that have to be traversed before reaching the declaration of T.
- If "`exit` T_d" is not enclosed in a declaration of T, then d must be greater or equal to the number of exception declarations enclosing this `exit` statement, as T will traverse all these declarations at least.

For example,

```
trap T in
  trap U in
    exit T1      %has depth 1 because of the declaration of U.
  ||
    exit U0      %has depth 0.
  ||
    exit V3      %could have any depth greater or equal to 2.
  end;
  exit T0        %has depth 0.
end
```

Thanks to this encoding, the behavior of `exit` statements can be described apart from matching `trap` constructs. In particular,

- An unmatched exception aborts the execution. For instance, in "`exit` T_3`; emit O`", the signal `O` is not emitted, as the execution is aborted by the exception `T`.
- Concurrent `exit` statements raise the exception of greater depth. For example, the statement "`exit` T_1 `|| exit` U_0 `|| exit` V_3" raises the exception `V` of depth 3.
- An exception of depth greater than 0 traverses the first `trap` statement it reaches, but looses one depth unit in the traversal. For instance, "`trap U in exit` T_3 `|| exit` U_0 `end`" raises the exception `T` of remaining depth 2.
- An exception of depth 0 is caught by the first `trap` statement it reaches.

In particular, exceptions bound by declarations behave as described in the previous section. Such a "De Bruijn" encoding [dB72] of exceptions for Esterel was first advocated in [Gon88]. As usual, we shall hide depths as often as possible.

2.3 Input+Output Signals

For simplicity, we supposed up to now that the free signals of a statement are either input or output signals, that is to say either occur in `emit` statements or `present` statements, never both. We would like to formalize the semantics of pure Esterel in a structural operational style, and for instance, derive the semantics of:

```
signal S in
  emit S;
  present S then emit O end
end
```

from that of:

```
emit S;
present S then emit O end
```

in which `S` is neither an input nor an output signal, but an input+output signal. Therefore, we have to consider free signals that occur both in `emit` and `present` statements.

We define the behavior of input+output signals by combining the behaviors of input and output signals. In each instant,

- an output or input+output signal S is emitted iff at least one "`emit` S" statement is executed in this instant, as we earlier specified for output signals.

```
emit S;                              %output signal S
present S then nothing end;          %input signal S
signal S in
  emit S;                            %local signal S number 1
  signal S in
    emit S;                          %local signal S number 2
    present S then nothing end       %local signal S number 2
  end;
  present S then nothing end         %local signal S number 1
end;
emit S;                              %output signal S
present S then nothing end           %input signal S
```

Figure 2.2: Local, Input, and Output Signals

- an input or input+output signal S is present iff it is provided by the environment in this instant, as we earlier specified for input signals.

In contrast with local (i.e. bound) signals, there is *no* relation between emission and status for input+output signals. For example,

- "`emit O`" emits `O`, but the status of `O` depends on the environment.
- "`present I then emit O end`" emits `O` iff `I` is provided by the environment.
- "`emit S; present S then emit O end`" emits `S`. Moreover, `O` is emitted iff `S` is provided by the environment, that is to say emitted if `S` is provided by the environment, not emitted if `S` is not provided by the environment, independently from the *local emission* of `S`.

In particular, an input+output signal may be, in the same instant, emitted and absent (if not provided by the environment). An input+output signal S is truly a combination of two distinct signals, an input signal S and an output signal S, which do not interact even if they share a common name:

- "`emit` S" statements refer to the output signal S.
- "`present` S `then` p `else` q `end`" statements refer to the input signal S.

From now on, we shall only speak of local, input, and output signals, implicitly referring to the input parts and output parts of input+output signals, as illustrated by Figure 2.2.

Thanks to this definition of input+output signals, in the sequel, we shall be able to formally derive the reactions of "`signal` S `in` p `end`" from that of p. Intuitively, enclosing a statement p in a declaration of S consists in:

- a *restriction* isolating the signal S from the environment:
  - The status of S in p inside "`signal` S `in` p `end`" does not depend on the environment.
  - The statement "`signal` S `in` p `end`" does not output S, even if S is locally emitted.
- a *signal coherence law* making the status of S correspond to the emission of S:
  - If S is emitted by p then S is present in p.
  - If S is not emitted by p then S is absent in p.

As a result, if p reacts to inputs I, with outputs O, and either S is both in I and O ($S \in I \cap O$) or S is neither in I nor in O ($S \notin I \cup O$), then "`signal` S `in` p `end`" reacts to inputs J with outputs $O \backslash \{S\}$, if $J \backslash \{S\} = I \backslash \{S\}$, that is to say if J and I agree on all signals but possibly S.

InputOutput Signals

The input+output signal interpretation is at first glance unintuitive, as such a signal may be absent even if locally emitted. Therefore, the Esterel v5 standard [Ber00a] defines "inputoutput" signals as the following:

- an inputoutput signal S is emitted iff at least one "`emit` S" statement is executed in this instant.
- an inputoutput signal is present iff provided by the environment in this instant *or locally emitted*.

Inputoutput signals however are more complex than input+output signals, as they combine the features of local and input+output signals. In our view, an inputoutput signal is both:

- a means of local communication: present if emitted (local signal),
- a means of communication with the environment:
  - present if provided by the environment (input signal),
  - observed by the environment if emitted (output signal).

Indeed, inputoutput signals can be encoded into combinations of input+output and local signals. If we want `S` to behave as an inputoutput signal for p, we may write the following:

```
                                signal Sin in
                                  signal Sout in
                                    trap T in
                                      subst(p); exit T
                                    ||
                                      loop
                                        present S then emit Sin end
inputoutput S in p end  =def          ||
                                        present Sout then emit Sin; emit S end
                                      ||
                                        pause
                                      end
                                    end
                                  end
                                end
```

where *subst*(p) is obtained by replacing in p all occurrences of "`emit S`" with "`emit Sout`" statements, and all occurrences of "`present S ... end`" with "`present Sin ... end`" statements.

In summary, input+output signals are more easily formalized and as powerful as inputoutput signals. Therefore, in this work, in which we are especially interested with formal reasoning about the language, we shall interpret free signals as input+output signals rather than inputoutput signals.

Thanks to the above encoding, we shall avoid the cumbersome distinction between programs and statements required by Berry's logical behavioral semantics [Ber99], which considers free signals of programs to be inputoutput signals whereas free signals of statements are (implicitly) considered to be input+output signals.

2.4 Logical Behavioral Semantics

The logical behavioral semantics of Esterel formalizes the intuitive semantics of the previous sections. It expresses a *reaction* of the statement p as a labeled transition of the form:

$$p \xrightarrow[I]{O,k} p'$$

where:

- the *input set* I, written below the arrow, lists the signals provided by the environment for the reaction, that is to say the *inputs* of the reaction;
- the *output set* O, written above the arrow, lists the output signals of p emitted by the reaction, that is to say the *outputs* of the reaction;
- the integer $k \in \mathbb{N}$ is the *completion code* [Gon88] of the reaction;
- the statement p' is the *residual* of the reaction.

Completion Code and Residual

The completion code k and the residual p' encode the *state* of the execution:

- If $k = 1$ then this reaction does not complete the execution of p.
  It has to be resumed by the execution of p' in the next instant.
- If $k \neq 1$ then this reaction ends the execution of p, and the residual p' is `nothing`:
  - $k = 0$ if the execution completes *normally*, that is to say without exception;
  - $k = d + 2$ if an exception of depth d *escapes* from p.

In particular, for any set of inputs I,

$$\texttt{nothing} \xrightarrow[I]{\emptyset,0} \texttt{nothing} \qquad \text{(nothing)}$$

$$\texttt{pause} \xrightarrow[I]{\emptyset,1} \texttt{nothing} \qquad \text{(pause)}$$

$$\texttt{exit } T_d \xrightarrow[I]{\emptyset,d+2} \texttt{nothing} \qquad \text{(exit)}$$

Execution

An *execution* of the statement p is a potentially infinite *chain* of reactions, such that all completion codes are equal to 1, except for the last one in the finite case:

- finite execution: $p \xrightarrow[I_0]{O_0,1} p_1 \xrightarrow[I_1]{O_1,1} \ldots \xrightarrow[I_n]{O_n,k}$ `nothing`, with $k \neq 1$, for some $n \in \mathbb{N}$.
- infinite execution: $p \xrightarrow[I_0]{O_0,1} p_1 \xrightarrow[I_1]{O_1,1} \ldots \xrightarrow[I_n]{O_n,1} \ldots$

We say that $\vec{I} = (I_0, I_1, ..., I_n)$ in the finite case and $\vec{I} = (I_n)_{n\in\mathbb{N}}$ in the infinite case is the *sequence of inputs* of the execution. Similarly, $\vec{O}$ is the *sequence of outputs*.

For example, the first reaction of `pause` does not terminate its execution, for any inputs. It has to be continued by the execution of `nothing` in the next instant, which terminates instantly:

$$\texttt{pause} \xrightarrow[I_0]{\emptyset,1} \texttt{nothing} \xrightarrow[I_1]{\emptyset,0} \texttt{nothing}$$

Inputs and Outputs

The set O lists the emitted output signals of the reaction. In particular,

$$\texttt{emit } S \xrightarrow[I]{\{S\},0} \texttt{nothing} \tag{emit}$$

The set I lists the signals provided by the environment for the reaction. Therefore, it defines the status of the free signals of the statement. So,

- If $S \in I$ and $p \xrightarrow[I]{O,k} p'$ then $\texttt{present } S \texttt{ then } p \texttt{ else } q \texttt{ end} \xrightarrow[I]{O,k} p'$.
- If $S \notin I$ and $q \xrightarrow[I]{O,k} q'$ then $\texttt{present } S \texttt{ then } p \texttt{ else } q \texttt{ end} \xrightarrow[I]{O,k} q'$.

Thus the deduction rules in Gentzen style notation:

$$\frac{S \in I \qquad p \xrightarrow[I]{O,k} p'}{\texttt{present } S \texttt{ then } p \texttt{ else } q \texttt{ end} \xrightarrow[I]{O,k} p'} \tag{present+}$$

$$\frac{S \notin I \qquad q \xrightarrow[I]{O,k} q'}{\texttt{present } S \texttt{ then } p \texttt{ else } q \texttt{ end} \xrightarrow[I]{O,k} q'} \tag{present−}$$

Remark that I may contain arbitrary signals, including signals that are not input signals of p. They are, therefore, not relevant to the reaction.

Exceptions

If p in "$\texttt{trap } T \texttt{ in } p \texttt{ end}$" reacts with completion code:

- 0, i.e. terminates instantly without exception, so does "$\texttt{trap } T \texttt{ in } p \texttt{ end}$",
- 1, i.e. pauses, so does "$\texttt{trap } T \texttt{ in } p \texttt{ end}$",
- 2, i.e. raises the exception of depth 0, that is to say T, then it is caught by this `trap` statement, which instantly terminates.
- k, for $k > 2$, i.e. raises the exception of depth $k-2$, then "$\texttt{trap } T \texttt{ in } p \texttt{ end}$" raises the exception of depth $k-3$, thus the completion code $k-3+2 = k-1$.

As a consequence, the completion code "$\downarrow k$" of "$\texttt{trap } T \texttt{ in } p \texttt{ end}$" is computed from the completion code k of p as follows:

$$\downarrow k = \begin{cases} 0 & \text{if } k = 0 \text{ or } k = 2 \\ 1 & \text{if } k = 1 \\ k-1 & \text{if } k > 2 \end{cases}$$

We define:

$$\frac{p \xrightarrow[I]{O,k} p'}{\texttt{trap } T \texttt{ in } p \texttt{ end} \xrightarrow[I]{O,\downarrow k} \delta_1^k(\texttt{trap } T \texttt{ in } p' \texttt{ end})} \tag{trap}$$

On the one hand, as p' may still refer to T if the execution has to be continued in the next instant ($k = 1$), the declaration of T must be preserved in the residual. On the other hand, if the execution of p terminates instantly ($k \neq 1$), we expect the residual of the reaction to be `nothing`. Therefore, we construct the residual using function:

$$\delta_1^k(p) = \begin{cases} \texttt{nothing} & \text{if } k \neq 1 \\ p & \text{if } k = 1 \end{cases}$$

For example,

$$\texttt{trap T in trap U in exit } \texttt{T}_1 \texttt{ || exit } \texttt{V}_5 \texttt{ end end} \xrightarrow[\emptyset]{\emptyset, 3} \texttt{nothing}$$

Sequence

The behavior of "$p\texttt{;}\ q$" depends on the behavior of p:

- If the execution of p terminates instantly and normally, it has to be instantly continued by the execution of q.

$$\frac{p \xrightarrow[I]{O, 0} p' \qquad q \xrightarrow[I]{O', k} q'}{p\texttt{;}\ q \xrightarrow[I]{O \cup O', k} q'} \qquad \text{(sequence-0)}$$

- If the execution of p does not terminate instantly, p' remaining to be executed, then the execution of the sequence has to be continued by the execution of "$p'\texttt{;}\ q$" in the next instant.

$$\frac{p \xrightarrow[I]{O, 1} p'}{p\texttt{;}\ q \xrightarrow[I]{O, 1} p'\texttt{;}\ q} \qquad \text{(sequence-1)}$$

- If the execution of p raises an exception, then it aborts the execution of the whole sequence.

$$\frac{p \xrightarrow[I]{O, k} p' \qquad k \geq 2}{p\texttt{;}\ q \xrightarrow[I]{O, k} \texttt{nothing}} \qquad \text{(sequence-2)}$$

In the sequel, we shall merge the (sequence-1) and (sequence-2) rules into a single rule:

$$\frac{p \xrightarrow[I]{O, k} p' \qquad k \neq 0}{p\texttt{;}\ q \xrightarrow[I]{O, k} \delta_1^k(p'\texttt{;}\ q)} \qquad \text{(sequence-k)}$$

Loops

Iterative behaviors are defined by unfolding, with the following deduction rule:

$$\frac{p \xrightarrow[I]{O, k} p' \qquad k \neq 0}{\texttt{loop } p \texttt{ end} \xrightarrow[I]{O, k} \delta_1^k(p'\texttt{; loop } p \texttt{ end})} \qquad \text{(loop)}$$

As announced, this rule requires the body p to execute either a `pause` instruction ($k = 1$) or raise an exception ($k \geq 2$). No reaction is defined for "`loop nothing end`" for instance (cf. Section 2.5).

Synchronous parallelism

Thanks to completion codes, the behavior of "p `||` q" can be specified with a simple "max" operator:

$$\frac{p \xrightarrow[I]{O,k} p' \quad q \xrightarrow[I]{O',l} q' \quad m = \max(k,l)}{p \texttt{ || } q \xrightarrow[I]{O \cup O', m} \delta_1^m(p' \texttt{ || } q')} \quad \text{(parallel)}$$

- If $k = 0$ and $l = 0$, both the execution of p and q terminate instantly, and no exception is raised. So does "p `||` q".
- If [$k = 0$ and $l = 1$] or [$k = 1$ and $l = 0$] or [$k = 1$ and $l = 1$], then again no exception is raised. In addition, one branch at least does not terminate instantly. Thus, the parallel composition of p and q does not either. The completion code of the reaction is 1, and the residual is "p' `||` q'". For simplicity, we retain the parallel construct even if one branch terminates. It is of no consequence, since p' and "p' `|| nothing`" for instance have the same semantics.
- If $k \geq 2$ or $l \geq 2$ then one exception at least escapes from one parallel branch, thus from "p `||` q". Moreover, if both branches raise an exception, then the exception of higher priority corresponds to the greater depth, thus the greater completion code.

For example, consider the following proof tree:

$$\frac{\dfrac{\dfrac{\texttt{exit T}_1 \xrightarrow[\emptyset]{\emptyset,3} \texttt{nothing} \quad \texttt{exit U}_0 \xrightarrow[\emptyset]{\emptyset,2} \texttt{nothing}}{\texttt{exit T}_1 \texttt{ || exit U}_0 \xrightarrow[\emptyset]{\emptyset,3} \texttt{nothing}}}{\texttt{trap U in exit T}_1 \texttt{ || exit U}_0 \texttt{ end} \xrightarrow[\emptyset]{\emptyset,2} \texttt{nothing}} \text{ since } \downarrow 3 = 2}{\texttt{trap T in trap U in exit T}_1 \texttt{ || exit U}_0 \texttt{ end end} \xrightarrow[\emptyset]{\emptyset,0} \texttt{nothing}} \text{ since } \downarrow 2 = 0$$

Local Signals

As discussed in Section 2.3, in "`signal` S `in` p `end`", S is present in p iff it is emitted by p, that is to say either:

- if S is supposed present in p (inputs $I \cup \{S\}$), then S is emitted by p (i.e. $S \in O$):

$$\frac{p \xrightarrow[I \cup \{S\}]{O,k} p' \quad S \in O}{\texttt{signal } S \texttt{ in } p \texttt{ end} \xrightarrow[I]{O \backslash \{S\}, k} \delta_1^k(\texttt{signal } S \texttt{ in } p' \texttt{ end})} \quad \text{(signal+)}$$

  The restriction "$O \backslash \{S\}$" prevents S to escape its scope of definition[2].

- if S is supposed absent in p (inputs $I \backslash \{S\}$), then S is not emitted by p (i.e. $S \notin O$):

$$\frac{p \xrightarrow[I \backslash \{S\}]{O,k} p' \quad S \notin O}{\texttt{signal } S \texttt{ in } p \texttt{ end} \xrightarrow[I]{O, k} \delta_1^k(\texttt{signal } S \texttt{ in } p' \texttt{ end})} \quad \text{(signal−)}$$

[2]There may be nested definitions for the name S. This is taken care of by the (signal+) and (signal−) rules, which ensure separation of the local and global signals S. For instance, for (signal+), remark $I \cup \{S\}$ is the same whatever the status of S in I, and similarly for $O \backslash \{S\}$.

For example, if $I = \{\texttt{A}, \texttt{B}\}$,

- $$\frac{\texttt{emit S} \xrightarrow[\{\texttt{A,B,S}\}]{\{\texttt{S}\},0} \texttt{nothing} \qquad \texttt{S} \in \{\texttt{S}\}}{\texttt{signal S in emit S end} \xrightarrow[\{\texttt{A,B}\}]{\emptyset,0} \texttt{nothing}}$$ using (signal+)

- $$\frac{\texttt{pause} \xrightarrow[\{\texttt{A,B}\}]{\emptyset,1} \texttt{nothing} \qquad \texttt{S} \notin \emptyset}{\texttt{signal S in pause end} \xrightarrow[\{\texttt{A,B}\}]{\emptyset,1} \texttt{signal S in nothing end}}$$ using (signal−)

The resulting semantics is not *deterministic*, as it may define several behaviors for a given statement. In "`signal S in present S then emit S else pause end end`", both hypotheses on `S` are valid, leading to two distinct behaviors, for instance for the set of inputs $\{\texttt{A}\}$:

$$\frac{\dfrac{\texttt{S} \notin \{\texttt{A}\} \qquad \texttt{pause} \xrightarrow[\{\texttt{A}\}]{\emptyset,1} \texttt{nothing}}{\texttt{present S then emit S else pause end} \xrightarrow[\{\texttt{A}\}]{\emptyset,1} \texttt{nothing}} \qquad \texttt{S} \notin \emptyset}{\texttt{signal S in present S then emit S else pause end end} \xrightarrow[\{\texttt{A}\}]{\emptyset,1} \texttt{signal S in nothing end}}$$

$$\frac{\dfrac{\texttt{S} \in \{\texttt{A},\texttt{S}\} \qquad \texttt{emit S} \xrightarrow[\{\texttt{A,S}\}]{\{\texttt{S}\},0} \texttt{nothing}}{\texttt{present S then emit S else pause end} \xrightarrow[\{\texttt{A,S}\}]{\{\texttt{S}\},0} \texttt{nothing}} \qquad \texttt{S} \in \{\texttt{S}\}}{\texttt{signal S in present S then emit S else pause end end} \xrightarrow[\{\texttt{A}\}]{\emptyset,0} \texttt{nothing}}$$

Moreover, there are statements that have no possible behavior. No reaction is defined for "`signal S in present S else emit S end end`", whatever the inputs. Both hypotheses on `S` are contradictory:

- if `S` is supposed to be absent then it is emitted. Contradiction.
- if `S` is supposed to be present then it is not emitted. Contradiction.

We shall further discuss these issues in Section 2.5.

Logical Behavioral Semantics

In summary, the logical behavioral semantics of Esterel specifies that the statement p reacts to inputs I with outputs O, completion code k, and residual p' iff, using the deduction rules of Figure 2.3, it can be shown that:

$$p \xrightarrow[I]{O,k} p'$$

Our logical behavioral semantics differs from Berry's semantics [Ber99] in two design choices:

- As proposed by Mignard in [Mig94], we ensure that the residual of the reaction is `nothing` if its completion code is not 1, that is to say if no further reaction is necessary. Of course, this does not change executions at all, but lets us compare semantics more easily in the sequel.
- We do not require or enforce O to be a subset of I, as we consider free signals to be input+output signals rather than inputoutput signals, as discussed in Section 2.3.

$$\texttt{nothing} \xrightarrow[I]{\emptyset,0} \texttt{nothing} \tag{nothing}$$

$$\texttt{pause} \xrightarrow[I]{\emptyset,1} \texttt{nothing} \tag{pause}$$

$$\texttt{exit } T_d \xrightarrow[I]{\emptyset,d+2} \texttt{nothing} \tag{exit}$$

$$\texttt{emit } S \xrightarrow[I]{\{S\},0} \texttt{nothing} \tag{emit}$$

$$\frac{p \xrightarrow[I]{O,k} p' \qquad k \neq 0}{\texttt{loop } p \texttt{ end} \xrightarrow[I]{O,k} \delta_1^k(p'\texttt{; loop } p \texttt{ end})} \tag{loop}$$

$$\frac{p \xrightarrow[I]{O,k} p' \qquad q \xrightarrow[I]{O',l} q' \qquad m = \max(k,l)}{p \texttt{ || } q \xrightarrow[I]{O \cup O',m} \delta_1^m(p' \texttt{ || } q')} \tag{parallel}$$

$$\frac{S \in I \qquad p \xrightarrow[I]{O,k} p'}{\texttt{present } S \texttt{ then } p \texttt{ else } q \texttt{ end} \xrightarrow[I]{O,k} p'} \tag{present+}$$

$$\frac{S \notin I \qquad q \xrightarrow[I]{O,k} q'}{\texttt{present } S \texttt{ then } p \texttt{ else } q \texttt{ end} \xrightarrow[I]{O,k} q'} \tag{present−}$$

$$\frac{p \xrightarrow[I]{O,k} p'}{\texttt{trap } T \texttt{ in } p \texttt{ end} \xrightarrow[I]{O,\downarrow k} \delta_1^k(\texttt{trap } T \texttt{ in } p' \texttt{ end})} \tag{trap}$$

$$\frac{p \xrightarrow[I]{O,0} p' \qquad q \xrightarrow[I]{O',k} q'}{p\texttt{; } q \xrightarrow[I]{O \cup O',k} q'} \tag{sequence-0}$$

$$\frac{p \xrightarrow[I]{O,k} p' \qquad k \neq 0}{p\texttt{; } q \xrightarrow[I]{O,k} \delta_1^k(p'\texttt{; } q)} \tag{sequence-k}$$

$$\frac{p \xrightarrow[I \cup \{S\}]{O,k} p' \qquad S \in O}{\texttt{signal } S \texttt{ in } p \texttt{ end} \xrightarrow[I]{O \setminus \{S\},k} \delta_1^k(\texttt{signal } S \texttt{ in } p' \texttt{ end})} \tag{signal+}$$

$$\frac{p \xrightarrow[I \setminus \{S\}]{O,k} p' \qquad S \notin O}{\texttt{signal } S \texttt{ in } p \texttt{ end} \xrightarrow[I]{O,k} \delta_1^k(\texttt{signal } S \texttt{ in } p' \texttt{ end})} \tag{signal−}$$

Figure 2.3: Logical Behavioral Semantics

2.5 Logical Correctness

In the previous section, we have seen that there are programs for which the logical behavioral semantics defines no behavior, and also programs with two or more behaviors. Indeed, a statement p may admit zero, one or several reactions for a given set of inputs I. Moreover, a given reaction of p may result from more than one proof tree. For example, for $I = \{\mathtt{A}\}$,

	reactions	proofs
`nothing`	one	one
`loop nothing end`	zero	zero
`signal S in present S else emit S end end`	zero	zero
`signal S in present S then emit S end end`	one	two
`signal S in present S then emit S else pause end end`	two	two

Let us focus on "`signal S in present S then emit S end end`". The semantics defines exactly one reaction for inputs $I = \{\mathtt{A}\}$, but with two possible proofs:

- $$\dfrac{\dfrac{\mathtt{S} \in \{\mathtt{A},\mathtt{S}\} \qquad \mathtt{emit\ S} \xrightarrow[\{\mathtt{A},\mathtt{S}\}]{\{\mathtt{S}\},0} \mathtt{nothing}}{\mathtt{present\ S\ then\ emit\ S\ end} \xrightarrow[\{\mathtt{A},\mathtt{S}\}]{\{\mathtt{S}\},0} \mathtt{nothing}} \qquad \mathtt{S} \in \{\mathtt{S}\}}{\mathtt{signal\ S\ in\ present\ S\ then\ emit\ S\ end\ end} \xrightarrow[\{\mathtt{A}\}]{\emptyset,0} \mathtt{nothing}} \text{ using (signal+)}$$

- $$\dfrac{\dfrac{\mathtt{S} \notin \{\mathtt{A}\} \qquad \mathtt{nothing} \xrightarrow[\{\mathtt{A}\}]{\emptyset,0} \mathtt{nothing}}{\mathtt{present\ S\ then\ emit\ S\ end} \xrightarrow[\{\mathtt{A}\}]{\emptyset,0} \mathtt{nothing}} \qquad \mathtt{S} \notin \emptyset}{\mathtt{signal\ S\ in\ present\ S\ then\ emit\ S\ end\ end} \xrightarrow[\{\mathtt{A}\}]{\emptyset,0} \mathtt{nothing}} \text{ using (signal}-\text{)}$$

We say that the *internal behavior* of "`signal S in present S then emit S end end`" is not deterministic, as the local signal `S` can be both present or absent. Its *observed behavior* is nevertheless deterministic.

We expect programs to have deterministic deadlock-free executions. So, programs with no or too many possible behaviors should be discarded as "incorrect". In this section, we formalize such a correctness criterion.

Generic Definitions

In the sequel, we shall consider various semantics of Esterel expressing reactions as transition relations of the form:

$$p \xmapsto[I]{O,k,\vec{\alpha}} p'$$

where:

- p and p' are typically statements, and in general elements of the *domain* $\mathcal{D}_{\mapsto}$ of the transition relation; in other words, $\mathcal{D}_{\mapsto}$ is the set of objects the semantics applies to.
- I and O are respectively the sets of inputs and outputs of the reaction.
- k is the completion code of the reaction.
- $\vec{\alpha}$ represents additional labels that may be defined, as for instance in Section 2.6.
  If so, we write $p \xmapsto[I]{O,k} p'$ iff there exists $\vec{\alpha}$ such that $p \xmapsto[I]{O,k,\vec{\alpha}} p'$.

For any such transition relation $\mapsto$, including the current one, we define:

- p is $\mapsto$*reactive w.r.t.* I iff there exists at least one tuple (O, k, p') such that $p \overset{O,k}{\underset{I}{\longmapsto}} p'$.

  p is $\mapsto$*reactive* iff p is reactive w.r.t. I for all I.

- p is $\mapsto$*deterministic w.r.t.* I iff there is at most one tuple (O, k, p') such that $p \overset{O,k}{\underset{I}{\longmapsto}} p'$.

  p is $\mapsto$*deterministic* iff p is deterministic w.r.t. I for all I.

- p is $\mapsto$*strongly-deterministic w.r.t.* I iff p is deterministic w.r.t. I and for all O, k, and p' the proof of $p \overset{O,k}{\underset{I}{\longmapsto}} p'$ is unique if it exists.

  p is $\mapsto$*strongly-deterministic* iff p is strongly deterministic w.r.t. I for all I.

- $p \mapsto q$ iff there exist inputs I and outputs O such that $p \overset{O,1}{\underset{I}{\longmapsto}} q$.

- q is $\mapsto$*reachable from* p iff $p \overset{*}{\mapsto} q$, where $\overset{*}{\mapsto}$ is the reflexive transitive closure of $\mapsto$, that is to say iff $p \mapsto ... \mapsto ... \mapsto q$ for some number $n \in \mathbb{N}$ of transitions, including zero.

- p is $\mapsto$*correct* iff for all q $\mapsto$reachable from p, q is $\mapsto$reactive and $\mapsto$deterministic.

- p is $\mapsto$*strongly-correct* iff for all q $\mapsto$reachable from p, q is $\mapsto$reactive and $\mapsto$strongly-deterministic.

Determinism ensures that the observed behavior of a statement is deterministic. Strong determinism guarantees that its internal behavior is deterministic, too. A statement is reactive iff it is not stuck. Reactivity combined with (strong) determinism ensures that there exists a unique reaction (with a unique proof) for this statement, whatever the inputs.

The statement q is reachable from p iff the execution of p may lead to the execution of q. Correctness characterizes statements that have deterministic deadlock-free executions for any sequence of inputs. In addition, strong correctness ensures strong determinism. Of course, strong correctness implies correctness. Strong correctness becomes a concern as soon as side effects or debugging have to be taken into account, as both may expose the internal behavior of a program.

Logical Correctness

As usual, we say that p is *logically correct* iff it is $\rightarrow$correct, where $\rightarrow$ is the transition relation defined by the logical behavioral semantics, *strongly correct* iff it is $\rightarrow$strongly-correct. As expected, not all statements are logically or strongly correct. For example,

- "`loop pause end`" is strongly correct, thus logically correct.
- "`loop nothing end`" is not logically correct.
- "`signal S in present S then pause end end`" is strongly correct.
- "`signal S in emit S; present S then pause end end`" is strongly correct.
- "`signal S in present S then emit S end end`" is logically but not strongly correct.
- "`signal S in present S then emit S else pause end end`" is not logically correct.
- "`signal S in present S else emit S end end`" is not logically correct.
- "`signal S in present S then emit S else emit S end end`" is strongly correct.

$$
\begin{array}{lll}
C[\,] & ::= & [\,] \\
 & & C[\,]\texttt{;}\ q \\
 & & p\texttt{;}\ C[\,] \\
 & & p\ \texttt{||}\ C[\,] \\
 & & C[\,]\ \texttt{||}\ q \\
 & & \texttt{loop}\ C[\,]\ \texttt{end} \\
 & & \texttt{signal}\ S\ \texttt{in}\ C[\,]\ \texttt{end} \\
 & & \texttt{present}\ S\ \texttt{then}\ C[\,]\ \texttt{else}\ q\ \texttt{end} \\
 & & \texttt{present}\ S\ \texttt{then}\ p\ \texttt{else}\ C[\,]\ \texttt{end} \\
 & & \texttt{trap}\ T\ \texttt{in}\ C[\,]\ \texttt{end}
\end{array}
$$

Figure 2.4: Contexts

2.6 Subterms and Occurrences

In this section, we provide tools to deal with *subterms* of programs, that is to say statements within statements. We first formalize occurrences of subterms and positions using contexts. We then introduce tags in order to keep track of occurrences through reactions.

Contexts

Following standard notation [Bar81], we say that a *context* is a statement with a single *hole* "[ ]". For instance, "`loop pause;` [ ] `end`" is a context. Contexts are recursively built in Figure 2.4:

- [ ] is the empty context,
- if $C[\,]$ is a context and q is a statement then "$C[\,]$`;` q" is a context,
- etc. for all constructs of pure Esterel.

As usual, $C[x]$ denotes the statement (respectively context) obtained by replacing the hole [ ] of $C[\,]$ by the statement (respectively context) x, without any renaming of signals or exceptions. For example, if $C[\,]$ = "`loop pause;` [ ] `end`" then $C[$`emit S`$]$ = "`loop pause; emit S end`".

If $p = C[q]$ then $C[\,]$ characterizes an *occurrence* of q in p: we say that q *occurs* at *position* $C[\,]$ in p. For example, `pause` has two occurrence in "`loop pause; pause end`" with positions "`loop pause;` [ ] `end`" and "`loop` [ ]`; pause end`".

Using contexts, we may for instance formally express that loop bodies are preserved by the semantics, that is to say that no fresh loop bodies are introduced by the execution:

Lemma 2.1. *If* $p \xrightarrow{*} p'$ *and* $p' = C'[\texttt{loop}\ b\ \texttt{end}]$ *then there exists* $C[\,]$ *s.t.* $p = C[\texttt{loop}\ b\ \texttt{end}]$.

Proof. By recurrence on the length of $p \xrightarrow{*} p'$, we may suppose that $p \rightarrow p'$. The only rule of the logical behavioral semantics (Figure 2.3) that builds `loop` subterms in residuals is the rule:

$$\frac{p \xrightarrow[I]{O,k} p' \qquad k \neq 0}{\texttt{loop}\ p\ \texttt{end} \xrightarrow[I]{O,k} \delta_1^k(p'\texttt{;}\ \texttt{loop}\ p\ \texttt{end})} \quad \text{(loop)}$$

The body p of the residual loop is copied from the initial loop. □

Tags

In order to relate subterms of the residual p' of a reaction to subterms of the initial statement p, we extend the syntax of pure Esterel with *tags*. We tag all pure Esterel constructs with integers:

- $\texttt{signal}^{tag}$ ... `in` ... `end`
- ... $\texttt{||}^{tag}$ ...
- etc. for all constructs of pure Esterel.

The tag of a statement p is the tag of its root construct. For example, "$\texttt{pause}^1\texttt{;}^2\ \texttt{pause}^3$" has tag 2. If q of tag n has position $C[\ ]$ in p, we note $p = C[q^n]$. We say that a statement p is *well tagged* iff its constructs are tagged with pairwise distinct tags. Tags of well-tagged statements unambiguously characterize occurrences:

Lemma 2.2. *If p is well tagged and q is a subterm of p of tag n then there exists a unique $C[\ ]$ such that $p = C[q^n]$.*

Proof. Structural induction. □

Then, we instrument the logical behavioral semantics as follows:

- We preserve tags in the residual of a reaction if its execution has to be continued later on ($k = 1$).
- We add an extra component M to transitions:

$$p \xrightarrow[I]{O,k,M} p'$$

  We call M the *multiset of tags* of the reaction. It is obtained by collecting the tags of the occurrences *reduced* in the course of the reaction. For example, if p reacts with the multiset M and q with N then "$p\ \texttt{||}^n\ q$" produces the multiset $M \uplus N \uplus \{n\}$:

$$\frac{p \xrightarrow[I]{O,k,M} p' \qquad q \xrightarrow[I]{O',l,N} q' \qquad m = \max(k,l)}{p\ \texttt{||}^n\ q \xrightarrow[I]{O \cup O', m, M \uplus N \uplus \{n\}} \delta_1^m(p'\ \texttt{||}^n\ q')} \quad \text{(parallel)}$$

For example,

$$\texttt{pause}^1\texttt{;}^2\ \texttt{pause}^3 \xrightarrow[I_0]{\emptyset,1,\{1,2\}} \texttt{nothing}^1\texttt{;}^2\ \texttt{pause}^3 \xrightarrow[I_1]{\emptyset,1,\{1,2,3\}} \texttt{nothing}^3 \xrightarrow[I_2]{\emptyset,0,\{3\}} \texttt{nothing}$$

Loop unfolding may *replicate* subterms, thus tags, and introduce *new* tags. For instance,

$$\texttt{loop}^1\ \texttt{pause}^2\ \texttt{end} \xrightarrow[I_0]{\emptyset,1,\{1,2\}} \texttt{nothing}^2\texttt{;}^3\ \texttt{loop}^1\ \texttt{pause}^2\ \texttt{end} \xrightarrow[I_1]{\emptyset,1,\{1,2,2,3\}} \dots$$

where 3 is a fresh tag. Of course, this replication is the reason for using multisets rather than sets. We shall further discuss this property of loops in Chapter 5.

Figure 2.5 shows the *logical behavioral semantics with tags* derived from the logical behavioral semantics of Figure 2.3.

$$\texttt{nothing}^n \xrightarrow[I]{\emptyset,0,\{n\}} \texttt{nothing} \qquad \text{(nothing)}$$

$$\texttt{pause}^n \xrightarrow[I]{\emptyset,1,\{n\}} \texttt{nothing}^n \qquad \text{(pause)}$$

$$\texttt{exit}^n\ T_d \xrightarrow[I]{\emptyset,d+2,\{n\}} \texttt{nothing} \qquad \text{(exit)}$$

$$\texttt{emit}^n\ S \xrightarrow[I]{\{S\},0,\{n\}} \texttt{nothing} \qquad \text{(emit)}$$

$$\frac{p \xrightarrow[I]{O,k,M} p' \quad k \neq 0}{\texttt{loop}^n\ p\ \texttt{end} \xrightarrow[I]{O,k,M\uplus\{n\}} \delta_1^k(p'\,\texttt{;}^{\textit{(new tag)}}\ \texttt{loop}^n\ p\ \texttt{end})} \qquad \text{(loop)}$$

$$\frac{p \xrightarrow[I]{O,k,M} p' \quad q \xrightarrow[I]{O',l,N} q' \quad m = \max(k,l)}{p\ \texttt{||}^n\ q \xrightarrow[I]{O\cup O',m,M\uplus N\uplus\{n\}} \delta_1^m(p'\ \texttt{||}^n\ q')} \qquad \text{(parallel)}$$

$$\frac{S \in I \quad p \xrightarrow[I]{O,k,M} p'}{\texttt{present}^n\ S\ \texttt{then}\ p\ \texttt{else}\ q\ \texttt{end} \xrightarrow[I]{O,k,M\uplus\{n\}} p'} \qquad \text{(present+)}$$

$$\frac{S \notin I \quad q \xrightarrow[I]{O,k,M} q'}{\texttt{present}^n\ S\ \texttt{then}\ p\ \texttt{else}\ q\ \texttt{end} \xrightarrow[I]{O,k,M\uplus\{n\}} q'} \qquad \text{(present}-\text{)}$$

$$\frac{p \xrightarrow[I]{O,k,M} p'}{\texttt{trap}^n\ T\ \texttt{in}\ p\ \texttt{end} \xrightarrow[I]{O,\downarrow k,M\uplus\{n\}} \delta_1^k(\texttt{trap}^n\ T\ \texttt{in}\ p'\ \texttt{end})} \qquad \text{(trap)}$$

$$\frac{p \xrightarrow[I]{O,0,M} p' \quad q \xrightarrow[I]{O',k,N} q'}{p\,\texttt{;}^n\ q \xrightarrow[I]{O\cup O',k,M\uplus N\uplus\{n\}} q'} \qquad \text{(sequence-0)}$$

$$\frac{p \xrightarrow[I]{O,k,M} p' \quad k \neq 0}{p\,\texttt{;}^n\ q \xrightarrow[I]{O,k,M\uplus\{n\}} \delta_1^k(p'\,\texttt{;}^n\ q)} \qquad \text{(sequence-k)}$$

$$\frac{p \xrightarrow[I\cup\{S\}]{O,k,M} p' \quad S \in O}{\texttt{signal}^n\ S\ \texttt{in}\ p\ \texttt{end} \xrightarrow[I]{O\setminus\{S\},k,M\uplus\{n\}} \delta_1^k(\texttt{signal}^n\ S\ \texttt{in}\ p'\ \texttt{end})} \qquad \text{(signal+)}$$

$$\frac{p \xrightarrow[I\setminus\{S\}]{O,k,M} p' \quad S \notin O}{\texttt{signal}^n\ S\ \texttt{in}\ p\ \texttt{end} \xrightarrow[I]{O,k,M\uplus\{n\}} \delta_1^k(\texttt{signal}^n\ S\ \texttt{in}\ p'\ \texttt{end})} \qquad \text{(signal}-\text{)}$$

Figure 2.5: Logical Behavioral Semantics with Tags

In addition to tags, we may also collect the names of the rules *used* in the reaction, thus pairs $(\mathit{rule}, \mathit{tag})$. For instance, the (signal+) rule now becomes:

$$\frac{p \xrightarrow[I \cup \{S\}]{O, k, M} p' \qquad S \in O}{\texttt{signal}^n\ S\ \texttt{in}\ p\ \texttt{end} \xrightarrow[I]{O \backslash \{S\}, k, M \uplus \{(\text{signal}+, n)\}} \delta_1^k(\texttt{signal}^n\ S\ \texttt{in}\ p'\ \texttt{end})} \quad \text{(signal+)}$$

This last semantics precisely reports whether a given local signal S of tag n is:

- present in the reaction: $(\text{signal}+, n) \in M$,
- absent in the reaction: $(\text{signal}-, n) \in M$,
- out of the "scope" of the reaction: there exists no rule Θ such that $(\Theta, n) \in M$.

All these variants of the logical behavioral semantics define the same reactions, thus executions, but return additional information about the proofs of the reactions. Therefore, we refer to all of them as *the* logical behavioral semantics of Esterel, without distinction.

Reducibility and Relevance

In the sequel, we shall instrument various semantics with tags. For any statement transition relation $\mapsto$, for any well-tagged statement p, we say that:

- the occurrence of tag n is *reduced by rule* Θ *in the reaction* $p \xmapsto[I]{O, k, M} p'$ if $(\Theta, n) \in M$.
- the rule Θ is *used in the reaction* $p \xmapsto[I]{O, k, M} p'$ if $(\Theta, n) \in M$ for some tag n.
- the occurrence of tag n is $\mapsto$*reducible by rule* Θ *in the execution of* p iff there exists q, I, O, k, M, and q' such that $p \overset{*}{\mapsto} q$ and $q \xmapsto[I]{O, k, M} q'$ and $(\Theta, n) \in M$.
- the occurrence of tag n is $\mapsto$*reducible in the execution of* p iff there exists Θ such that n is $\mapsto$reducible by rule Θ in the execution of p.
- the rule Θ is $\mapsto$*relevant to the execution of* p iff there exists an occurrence of tag n in p that is $\mapsto$reducible by Θ.

Intuitively, an occurrence is reducible in the execution of p, a rule is relevant to the execution of p, iff they may "participate" in the execution of p. Remark that reducibility is not a property of statements in general but a property of occurrences of subterms with respect to a given statement.

2.7 Bisimulations and Observational Equivalence

In order to compare programs and semantics, we define observational equivalence by means of bisimulations[3]. As a first application, we discuss dead code in the logical behavioral semantics.

[3] In this work, we only consider *strong* bisimulations [Mil89]. Moreover, as we want to compare semantics of various domains, we require bisimulations to be total surjective relations rather than reflexive relations. Finally, as we shall only consider normalizing semantics ($k \neq 1 \Rightarrow p' = \texttt{nothing}$), we do not need the custom 1-bisimulations of [Tar04b].

Bisimulation – Observational Equivalence

Let $\mapsto$ and $\hookrightarrow$ be two transition relations. We say that the binary relation $\sim$ of domain $\mathcal{D}_{\mapsto} \times \mathcal{D}_{\hookrightarrow}$ is a *bisimulation between* $\mapsto$ *and* $\hookrightarrow$ iff:

- relation $\sim$ is *total*: for all p in $\mathcal{D}_{\mapsto}$ there exists q in $\mathcal{D}_{\hookrightarrow}$ such that $p \sim q$;
- relation $\sim$ is *surjective*: for all q in $\mathcal{D}_{\hookrightarrow}$ there exists p in $\mathcal{D}_{\mapsto}$ such that $p \sim q$;
- if $p \sim q$ then for all (I, O, k):
  - if $p \xmapsto[I]{O,k} p'$ then there exists q' such that $q \xhookrightarrow[I]{O,k} q'$ and $p' \sim q'$;
  - if $q \xhookrightarrow[I]{O,k} q'$ then there exists p' such that $p \xmapsto[I]{O,k} p'$ and $p' \sim q'$.

We say that $\mapsto$ and $\hookrightarrow$ are *observationally equivalent* iff there exists such a bisimulation. We say that p and q are *bisimilar w.r.t.* $\mapsto$ *and* $\hookrightarrow$ iff there exists a bisimulation $\sim$ between $\mapsto$ and $\hookrightarrow$ such that $p \sim q$. In particular, we say that p and q are $\mapsto$*bisimilar* iff they are bisimilar w.r.t. $\mapsto$ and $\mapsto$.

Dead Code

For any statement transition relation $\mapsto$, we say that q at position $C[\,]$ in p is $\mapsto$*dead in* p iff for any statement r, p and $C[r]$ are bisimilar. For example, the `pause` instruction is $\rightarrow$dead in:

```
trap T in
  exit T;
  pause
end
```

Remark that `pause` is also non-$\rightarrow$reducible. In fact,

Theorem 2.3. *If q at position $C[\,]$ in p is $\rightarrow$dead in p then q is not $\rightarrow$reducible in p.*

Proof (Sketch). If q at position $C[\,]$ in p is $\rightarrow$reducible in p then p and $C[$`emit O;` $q]$ where `O` is a fresh signal are not $\rightarrow$bisimilar. Hence, q is not $\rightarrow$dead in p. □

But a statement may be both non-$\rightarrow$reducible and non-$\rightarrow$dead. For example, `pause` is neither $\rightarrow$reducible nor $\rightarrow$dead in:

```
signal S in
  present S then pause end
end
```

as this statement and the following one are not bisimilar:

```
signal S in
  present S then emit S; emit O end
end
```

Intuitively, reactions not only depend on the subterms occurring in their proofs, that is to say $\rightarrow$reducible subterms, but also on subterms that prevent proofs to be completed. In our example, `pause` prevents `S` to be present.

On the contrary, the deterministic semantics we shall introduce in Chapter 3 eliminates this unintuitive mismatch, by making such dependencies explicit.

SUMMARY

We have described the syntax of the pure Esterel language and formalized its logical behavioral semantics as a set of SOS rules. In a way largely independent from the precise rules of the semantics, we have defined reactivity, (strong) determinism, and (strong) correctness. We have further instrumented the syntax and semantics of the language with tags in order to keep track of occurrences of subterms through reactions. Finally, we have introduced bisimulations, so we can compare program behaviors, and define observationally equivalent semantics.

Chapter 3

Reactive Deterministic Semantics

We derive from the logical behavioral semantics a deterministic semantics in Section 3.1. In contrast with the logical behavioral semantics, the execution of a program is fully decided by the sequence of inputs in the deterministic semantics. This semantics leads to an updated correctness criterion for Esterel programs called properness defined in Section 3.2. In Section 3.3, we extend the deterministic semantics with explicit error-handling rules, thus defining a reactive deterministic semantics. We generalize properness to subterms (i.e. pieces of programs) in Section 3.4.

3.1 Deterministic Semantics

The logical behavioral semantics provides a very compact, structural formalization of the behavior of Esterel programs, which makes formal reasoning about the language tractable. It founds reactivity and determinism, the agreed minimal correctness criteria for Esterel programs.

However, working with these criteria can be tedious. While, reactivity may be attested with a simple proof tree, establishing (strong-)determinism is more complex and formally requires a proof about proof trees (proof of uniqueness). Moreover, defining first many (proofs of) reactions for non-(strongly)-deterministic statements, which we then discard because there are too many, seems utterly inefficient.

Therefore, we propose to rewrite the rules (signal−) and (signal+) for signal declarations:

$$\frac{p \xrightarrow[I \cup \{S\}]{O,k} p' \qquad S \in O}{\texttt{signal } S \texttt{ in } p \texttt{ end} \xrightarrow[I]{O \backslash \{S\},k} \delta_1^k(\texttt{signal } S \texttt{ in } p' \texttt{ end})} \quad \text{(signal+)}$$

$$\frac{p \xrightarrow[I \backslash \{S\}]{O,k} p' \qquad S \notin O}{\texttt{signal } S \texttt{ in } p \texttt{ end} \xrightarrow[I]{O,k} \delta_1^k(\texttt{signal } S \texttt{ in } p' \texttt{ end})} \quad \text{(signal−)}$$

as the following (where k^+, k^-, etc. are just convenient names):

$$\frac{p \overset{O^-,k^-}{\underset{I \backslash \{S\}}{\circ\!\!\longrightarrow}} p^- \qquad S \in O^- \qquad p \overset{O^+,k^+}{\underset{I \cup \{S\}}{\circ\!\!\longrightarrow}} p^+ \qquad S \in O^+}{\texttt{signal } S \texttt{ in } p \texttt{ end} \overset{O^+ \backslash \{S\},k^+}{\underset{I}{\circ\!\!\longrightarrow}} \delta_1^{k^+}(\texttt{signal } S \texttt{ in } p^+ \texttt{ end})} \quad \text{(signal++)}$$

$$\frac{p \overset{O^-,k^-}{\underset{I \backslash \{S\}}{\circ\!\!\longrightarrow}} p^- \qquad S \notin O^- \qquad p \overset{O^+,k^+}{\underset{I \cup \{S\}}{\circ\!\!\longrightarrow}} p^+ \qquad S \notin O^+}{\texttt{signal } S \texttt{ in } p \texttt{ end} \overset{O^-,k^-}{\underset{I}{\circ\!\!\longrightarrow}} \delta_1^{k^-}(\texttt{signal } S \texttt{ in } p^- \texttt{ end})} \quad \text{(signal−−)}$$

We call the resulting semantics the *deterministic semantics*, and denote the corresponding reactions with the transition symbol "$\circ\!\!\rightarrow$". Figure 3.1 shows the complete semantics. Intuitively, it consists in enforcing in each signal rule that the other one does not apply, without introducing negative premises [Gro90] such as the negation of the hypothesis of rule (signal+):

$$S, p, I, O, k, p' \text{ are } \textit{not} \text{ such that } p \xrightarrow[I\cup\{S\}]{O,k} p' \text{ and } S \in O$$

Rather than negating the whole precondition, we only swap the binary decision $S \in O$ for $S \notin O$, and vice versa. In the logical behavioral semantics, we had:

- (signal+): if S is supposed present in p then it is emitted by p.
- (signal−): if S is supposed absent in p then it is not emitted by p.

In our deterministic semantics, the rule for the `signal` construct are:

- (signal++):
  - if S is supposed present in p then it is emitted.
  - if S is supposed absent in p then it is *still* emitted.
- (signal−−):
  - if S is supposed absent in p then it is not emitted.
  - if S is supposed present in p then it is not emitted *either*.

For example, the deterministic semantics produces the same reactions as the logical behavioral semantics in the following two cases (cf. Chapter 2):

- $$\frac{\texttt{emit S} \overset{\{\texttt{S}\},0}{\underset{\{\texttt{A}\}}{\circ\!\!\longrightarrow}} \texttt{nothing} \quad \texttt{S} \in \{\texttt{S}\} \quad \texttt{emit S} \overset{\{\texttt{S}\},0}{\underset{\{\texttt{A,S}\}}{\circ\!\!\longrightarrow}} \texttt{nothing} \quad \texttt{S} \in \{\texttt{S}\}}{\texttt{signal S in emit S end} \overset{\emptyset,0}{\underset{\{\texttt{A}\}}{\circ\!\!\longrightarrow}} \texttt{nothing}} \text{ by (signal++)}$$
- $$\frac{\texttt{pause} \overset{\emptyset,1}{\underset{\{\texttt{A}\}}{\circ\!\!\longrightarrow}} \texttt{nothing} \quad \texttt{S} \notin \emptyset \quad \texttt{pause} \overset{\emptyset,1}{\underset{\{\texttt{A,S}\}}{\circ\!\!\longrightarrow}} \texttt{nothing} \quad \texttt{S} \notin \emptyset}{\texttt{signal S in pause end} \overset{\emptyset,1}{\underset{\{\texttt{A}\}}{\circ\!\!\longrightarrow}} \texttt{signal S in nothing end}} \text{ by (signal−−)}$$

Whatever I, the deterministic semantics defines no reaction for:

- the non-$\longrightarrow$reactive statement:

  "`signal S in present S else emit S end end`"

- the non-$\longrightarrow$deterministic statement:

  "`signal S in present S then emit S else pause end end`"

- the non-$\longrightarrow$strongly-deterministic statement:

  "`signal S in present S then emit S end end`"

In other words, all these statements are non-$\circ\!\!\rightarrow$reactive.

$$\texttt{nothing} \xrightarrow[I]{\emptyset, 0} \texttt{nothing} \tag{nothing}$$

$$\texttt{pause} \xrightarrow[I]{\emptyset, 1} \texttt{nothing} \tag{pause}$$

$$\texttt{exit } T_d \xrightarrow[I]{\emptyset, d+2} \texttt{nothing} \tag{exit}$$

$$\texttt{emit } S \xrightarrow[I]{\{S\}, 0} \texttt{nothing} \tag{emit}$$

$$\frac{p \xrightarrow[I]{O, k} p' \quad k \neq 0}{\texttt{loop } p \texttt{ end} \xrightarrow[I]{O, k} \delta_1^k(p'\texttt{; loop } p \texttt{ end})} \tag{loop}$$

$$\frac{p \xrightarrow[I]{O, k} p' \quad q \xrightarrow[I]{O', l} q' \quad m = \max(k, l)}{p \texttt{ || } q \xrightarrow[I]{O \cup O', m} \delta_1^m(p' \texttt{ || } q')} \tag{parallel}$$

$$\frac{S \in I \quad p \xrightarrow[I]{O, k} p'}{\texttt{present } S \texttt{ then } p \texttt{ else } q \texttt{ end} \xrightarrow[I]{O, k} p'} \tag{present+}$$

$$\frac{S \notin I \quad q \xrightarrow[I]{O, k} q'}{\texttt{present } S \texttt{ then } p \texttt{ else } q \texttt{ end} \xrightarrow[I]{O, k} q'} \tag{present−}$$

$$\frac{p \xrightarrow[I]{O, k} p'}{\texttt{trap } T \texttt{ in } p \texttt{ end} \xrightarrow[I]{O, \downarrow k} \delta_1^k(\texttt{trap } T \texttt{ in } p' \texttt{ end})} \tag{trap}$$

$$\frac{p \xrightarrow[I]{O, 0} p' \quad q \xrightarrow[I]{O', k} q'}{p\texttt{; } q \xrightarrow[I]{O \cup O', k} q'} \tag{sequence-0}$$

$$\frac{p \xrightarrow[I]{O, k} p' \quad k \neq 0}{p\texttt{; } q \xrightarrow[I]{O, k} \delta_1^k(p'\texttt{; } q)} \tag{sequence-k}$$

$$\frac{p \xrightarrow[I \setminus \{S\}]{O^-, k^-} p^- \quad S \in O^- \quad p \xrightarrow[I \cup \{S\}]{O^+, k^+} p^+ \quad S \in O^+}{\texttt{signal } S \texttt{ in } p \texttt{ end} \xrightarrow[I]{O^+ \setminus \{S\}, k^+} \delta_1^{k^+}(\texttt{signal } S \texttt{ in } p^+ \texttt{ end})} \tag{signal++}$$

$$\frac{p \xrightarrow[I \setminus \{S\}]{O^-, k^-} p^- \quad S \notin O^- \quad p \xrightarrow[I \cup \{S\}]{O^+, k^+} p^+ \quad S \notin O^+}{\texttt{signal } S \texttt{ in } p \texttt{ end} \xrightarrow[I]{O^-, k^-} \delta_1^{k^-}(\texttt{signal } S \texttt{ in } p^- \texttt{ end})} \tag{signal−−}$$

Figure 3.1: Deterministic Semantics

Determinism

The new semantics is *globally deterministic* in the following sense:

Lemma 3.1. *For every statement p, p is $\circ\!\!\rightarrow$deterministic:*

$$\forall p, \forall I : \left[p \overset{O_0,k_0}{\underset{I}{\circ\!\!\longrightarrow}} p_0 \wedge p \overset{O_1,k_1}{\underset{I}{\circ\!\!\longrightarrow}} p_1\right] \Rightarrow [O_0 = O_1 \wedge k_0 = k_1 \wedge p_0 = p_1]$$

Lemma 3.2. *For every statement p, p is $\circ\!\!\rightarrow$strongly-deterministic. In particular,*

$$\forall p, \forall I, \forall O, \forall k, \forall p' : \textit{the proof of } p \overset{O,k}{\underset{I}{\circ\!\!\longrightarrow}} p' \textit{ is unique if it exists.}$$

Proof. Structural induction. □

Corollary 3.3. *p is $\circ\!\!\rightarrow$strongly-correct iff p is $\circ\!\!\rightarrow$correct iff* $\left[\forall q : p \overset{*}{\circ\!\!\rightarrow} q \Rightarrow q \textit{ is } \circ\!\!\rightarrow\textit{reactive}\right]$.

There is no need to count reactions or proof trees in the deterministic semantics. Therefore, $\circ\!\!\rightarrow$correctness is much easier to manipulate than logical correctness as it reduces to $\circ\!\!\rightarrow$reactivity. Moreover, $\circ\!\!\rightarrow$strong-correctness comes for free with $\circ\!\!\rightarrow$correctness.

3.2 Proper Statements

We say that p is *proper* iff p is $\circ\!\!\rightarrow$correct. We now precisely relate the logical behavioral and the deterministic semantics, in particular strong correctness and properness.

Properness implies Strong Correctness

Lemma 3.4. *If $p \overset{O,k}{\underset{I}{\circ\!\!\longrightarrow}} p'$ then $p \overset{O,k}{\underset{I}{\longrightarrow}} p'$.*

Proof. Any deduction valid in the deterministic semantics is translatable into a valid deduction in the logical behavioral semantics (by erasing premises). □

Lemma 3.5. *If $p \overset{O_0,k_0}{\underset{I}{\circ\!\!\longrightarrow}} p_0$ and $p \overset{O_1,k_1}{\underset{I}{\longrightarrow}} p_1$ then $O_0 = O_1, k_0 = k_1, p_0 = p_1$.*

Lemma 3.6. *If $p \overset{O_0,k_0}{\underset{I}{\circ\!\!\longrightarrow}} p_0$ then the proof of $p \overset{O_0,k_0}{\underset{I}{\longrightarrow}} p_0$ is unique.*

Proof. Structural induction on p. Let us consider the case $p =$ "`signal` S `in` q `end`". By hypothesis, $p \overset{O_0,k_0}{\underset{I}{\circ\!\!\longrightarrow}} p_0$. As (signal++) or (signal−−) must be used to define this reaction, there exist $O_0^-, k_0^-, q_0^-, O_0^+, k_0^+, q_0^+$ such that:

$$q \overset{O_0^-,k_0^-}{\underset{I\backslash\{S\}}{\circ\!\!\longrightarrow}} q_0^- \quad \text{and} \quad q \overset{O_0^+,k_0^+}{\underset{I\cup\{S\}}{\circ\!\!\longrightarrow}} q_0^+$$

Then, using Lemma 3.1,

- either $S \notin O_0^-, S \notin O_0^+, O_0 = O_0^-, k_0 = k_0^-, p_0 = \delta_1^{k_0^-}$ (`signal` S `in` q_0^- `end`),
- or $S \in O_0^-, S \in O_0^+, O_0 = O_0^+\backslash\{S\}, k_0 = k_0^+, p_0 = \delta_1^{k_0^+}$ (`signal` S `in` q_0^+ `end`).

Let us focus on the first case; the second one is similar. By induction hypothesis:

- if $q \xrightarrow[I\backslash\{S\}]{O_1^-, k_1^-} q_1$ then $O_0^- = O_1^-, k_0^- = k_1^-, q_0^- = q_1^-$ and the proof of this reaction is unique.
- if $q \xrightarrow[I\cup\{S\}]{O_1^+, k_1^+} q_1$ then $O_0^+ = O_1^+, k_0^+ = k_1^+, q_0^+ = q_1^+$ and the proof of this reaction is unique.

Therefore,

- No reaction can be defined for p using rule (signal+) as $S \notin O_0^+$.
- By rule (signal−), if $p \xrightarrow[I\backslash\{S\}]{O_1, k_1} p_1$ then $O_0 = O_1, k_0 = k_1, p_0 = p_1$.

Moreover, the proof of this reaction is unique. □

In summary, by writing $p \overset{O,k}{\underset{I}{\circ\!\!\longrightarrow}} p'$, we express that:

- p *may* react to inputs I, with outputs O, completion code k, and residual p' in the deterministic semantics, thus in the logical behavioral semantics as well (Lemma 3.4),
- p *must* react this way in both semantics (Lemma 3.1 and Lemma 3.5)
- the internal behavior of p is deterministic (Lemma 3.2 and Lemma 3.6).

In particular, the drawing of a single proof tree of the deterministic semantics for the statement p with inputs I, establishes that p is →reactive and →strongly-deterministic w.r.t. I, so that:

Theorem 3.7. *If p is proper then p is strongly correct (i.e. →strongly-correct).*

Proof. Let p be a proper statement, and q be such that $p \xrightarrow{*} q$. By Lemma 3.5, if u is ∘→reactive and $u \rightarrow v$ then $u \circ\!\!\rightarrow v$. Thus, by recurrence on the number of transitions in the chain $p \xrightarrow{*} q$, we obtain that $p \overset{*}{\circ\!\!\rightarrow} q$. By definition of properness, q is ∘→reactive. By Lemma 3.4, q is →reactive. By Lemma 3.5, q is →deterministic. By Lemma 3.6, q is →strongly-deterministic. □

Strong Correctness does not imply Properness

Reciprocally however, a strongly correct statement is not necessarily proper, as →reactivity combined with →strong-determinism does not imply ∘→reactivity. Let us consider two examples:

- ```
  signal S in
    present S then loop nothing end end
  end
  ```

  For all inputs I, the logical behavioral semantics defines the following unique proof tree:

  $$\dfrac{\dfrac{\mathtt{S} \notin I\backslash\{\mathtt{S}\} \qquad \mathtt{nothing} \xrightarrow[I\backslash\{\mathtt{S}\}]{\emptyset, 0} \mathtt{nothing}}{\mathtt{present\ S\ then\ loop\ nothing\ end\ end} \xrightarrow[I\backslash\{\mathtt{S}\}]{\emptyset, 0} \mathtt{nothing}} \qquad \mathtt{S} \notin \emptyset}{\mathtt{signal\ S\ in\ present\ S\ then\ loop\ nothing\ end\ end\ end} \xrightarrow[I]{\emptyset, 0} \mathtt{nothing}}$$

  The deterministic semantics however defines no reaction for this statement, whatever I. Neither (signal++) nor (signal−−) applies, as "`present S then loop nothing end end`" is not ∘→reactive w.r.t. $I \cup \{\mathtt{S}\}$, whatever I.
  ```

- ```
  loop
    signal S in
      present S then emit S else pause end
    end
  end
  ```

  The body "`signal S in present S then emit S else pause end end`" may reacts in two possible ways in the logical behavioral semantics, whatever I, with respective completion codes 0 and 1:

  $$\texttt{signal S in present S then emit S else pause end end} \xrightarrow[I]{\emptyset, 0} \texttt{nothing}$$

  $$\texttt{signal S in present S then emit S else pause end end} \xrightarrow[I]{\emptyset, 1} \texttt{signal S in nothing end}$$

  Since exactly one of these two reactions admits a non-zero completion code, the whole loop statement is both →reactive and →strongly-deterministic. On the other hand, the deterministic semantics defines no reaction for the body, thus no reaction for the loop.

Strongly-Correct Non-Proper Statements

In the logical behavioral semantics, non-determinism may compensate for non-reactivity, or the other way around, so that a piece of incorrect (e.g. "`loop nothing end`") code may be embedded into a strongly correct program. More precisely,

Theorem 3.8. *If p is →reactive and →strongly-deterministic but not ⊸→reactive then there exists a subterm q of p such that q is not →reactive or not →strongly-deterministic.*

Proof. By structural induction on p, if p and all its subterms are →reactive and →strongly-deterministic then p is ⊸→reactive.

Let us consider the only non-trivial induction $p =$ "`signal` S `in` q `end`". By hypothesis, p and all its all subterms are →reactive and →strongly-deterministic. As a consequence, q and all its subterms are →reactive and →strongly-deterministic. By induction hypothesis, q is ⊸→reactive.

Let us choose the set of inputs I. There exists $(k^-, O^-, q^-, k^+, O^+, q^+)$ such that:

$$q \overset{O^+, k^+}{\underset{I \cup \{S\}}{\circ\!\!\longrightarrow}} q^+ \quad \text{and} \quad q \overset{O^-, k^-}{\underset{I \backslash \{S\}}{\circ\!\!\longrightarrow}} q^-$$

There are four cases:

- $S \in O^+$, $S \in O^-$, then p is ⊸→reactive w.r.t. I by rule (signal++).
- $S \notin O^+$, $S \notin O^-$, then p is ⊸→reactive w.r.t. I by rule (signal−−).
- $S \in O^+$, $S \notin O^-$, then p is not →strongly-deterministic w.r.t. I, since both (signal+) and (signal−) are applicable. Contradiction.
- $S \notin O^+$, $S \in O^-$, then p is not →reactive w.r.t. I, since neither (signal+) nor (signal−) is applicable. Contradiction.

In all valid cases, p is ⊸→reactive. □

If p is →reactive and →strongly-deterministic but one subterm q of p is not →reactive or not →strongly-deterministic, then q behaves "well" in p only because of its context of occurrence, which constrains the execution of q from the outside, making sure the non-reactive or non-strongly-deterministic behaviors of q are never triggered. Intuitively, q could be simplified while preserving the behavior of p.
  ```

Let us consider our two examples in this new light:

- ```
  signal S in
    present S then loop nothing end end
  end
  ```

  The subterm "`present S then loop nothing end end`" is not $\rightarrow$reactive because of its `then` branch, but never used with `S` present. Therefore, it can be replaced by its implicit `else` branch, that is to say `nothing`, leading to the program "`signal S in nothing end`", which is equivalent to the initial one (i.e. $\rightarrow$bisimilar) and proper.

- ```
  loop
    signal S in
      present S then emit S else pause end
    end
  end
  ```

  The body of the loop is not $\rightarrow$deterministic, but the enclosing loop, by forbidding the reaction of completion code $k = 0$, enforces the signal `S` to be absent. Again, the subterm "`present S then emit S else pause end`" can simplified, by discarding the case `S` present. The final program "`loop signal S in pause end end`" is equivalent and proper.

Therefore, there is something "wrong" with these programs, even if strong and logical correctness are not sensitive to it. In any case, they are intricate constructions with no practical purpose.

To conclude, the deterministic semantics does not change the semantics of "reasonable" programs, while defining at most one execution for all programs and all input sequences. Therefore, it provides a much better starting point for formal reasoning about programs than the logical behavioral semantics, and for instance:

Theorem 3.9. *If p is proper, then q at position $C[\,]$ in p is either $\circ\!\!\rightarrow$reducible or $\circ\!\!\rightarrow$dead in p.*

Proof (Sketch). If q at position $C[\,]$ in p is $\circ\!\!\rightarrow$reducible in p then p and $C[$`emit O;` $q]$ where `O` is a fresh signal are not $\circ\!\!\rightarrow$bisimilar. Hence, q is not $\circ\!\!\rightarrow$dead in p.

Let us now suppose that p is proper and q is not $\circ\!\!\rightarrow$reducible in p. Any execution of p is also an execution of $C[r]$, whatever r. Moreover, thanks to global determinism, $C[r]$ admits no additional execution, as p, thus $C[r]$, already admit a possible execution for each input sequence. Therefore, p and $C[r]$ are $\circ\!\!\rightarrow$bisimilar, whatever r. In other words, q is $\circ\!\!\rightarrow$dead in p. □

In the sequel, we shall provide further arguments for preferring properness over strong or logical correctness.

3.3 Reactive Deterministic Semantics

The existence of a single proof tree of the deterministic semantics establishes that a statement is $\circ\!\!\rightarrow$reactive with respect to a given set of inputs. Establishing non-$\circ\!\!\rightarrow$reactivity however still requires universally quantified proofs about proof trees. In this section, we further transform the semantics, so that non-$\circ\!\!\rightarrow$reactivity may be shown with a single proof tree as well. We shall use the symbol "$\bullet\!\!\rightarrow$" for the transitions of the resulting semantics.

In order to reason about non-$\circ\!\!\rightarrow$reactivity within the rules of the semantics themselves, we have to encode non-$\circ\!\!\rightarrow$reactivity somehow: we reuse the existing exception propagation mechanism of the logical behavioral and deterministic semantics to handle non-$\circ\!\!\rightarrow$reactivity.

We introduce the completion code "$+\infty$" to represent non-$\circ\!\!\rightarrow$reactivity. It obeys the obvious arithmetic relations:

$$\forall k \in (\mathbb{N} \cup \{+\infty\}) : \max(k, +\infty) = +\infty \qquad\qquad \downarrow\!+\infty = +\infty$$

Our goal is to achieve the following properties:

- If $p \circ\!\!\xrightarrow[I]{O,k} p'$ then $p \bullet\!\!\xrightarrow[I]{O,k} p'$.

  The reactions of the deterministic semantics must be preserved.

- If $p \bullet\!\!\xrightarrow[I]{O,k} p'$ and $k \neq +\infty$ then $p \circ\!\!\xrightarrow[I]{O,k} p'$.

  Added reactions must have the completion code $+\infty$.

- If $p \bullet\!\!\xrightarrow[I]{O,k} p'$ and $k = +\infty$ then $O = \emptyset$ and $p' = \texttt{nothing}$.

  Added reactions must have residual `nothing` and no outputs.

- If p is not $\circ\!\!\rightarrow$reactive w.r.t. I then $p \bullet\!\!\xrightarrow[I]{\emptyset,+\infty} \texttt{nothing}$.

  If p is not $\circ\!\!\rightarrow$reactive w.r.t. I then a reaction of completion code $+\infty$ must be defined.

- If $p \bullet\!\!\xrightarrow[I]{\emptyset,+\infty} \texttt{nothing}$ then p is not $\circ\!\!\rightarrow$reactive w.r.t. I.

  If p is $\circ\!\!\rightarrow$reactive w.r.t. I then no reaction should be added.

In summary, a statement p must react to inputs I either as it does in the deterministic semantics if p is $\circ\!\!\rightarrow$reactive w.r.t. I, or with completion code $+\infty$, residual `nothing`, and no outputs, if it is not.

A tentative proof by structural induction that all statements are $\circ\!\!\rightarrow$reactive fails for the following reasons:

- Even if $p \circ\!\!\xrightarrow[I]{O,k} p'$, if $k = 0$ then "`loop` p `end`" is not $\circ\!\!\rightarrow$reactive.

- Even if $p \circ\!\!\xrightarrow[I \cup \{S\}]{O^+,k^+} p^+$ and $p \circ\!\!\xrightarrow[I \backslash \{S\}]{O^-,k^-} p^-$:

  - If $S \in O^+$ and $S \notin O^-$ then "`signal` S `in` p `end`" is not $\circ\!\!\rightarrow$reactive.
  - If $S \notin O^+$ and $S \in O^-$ then "`signal` S `in` p `end`" is not $\circ\!\!\rightarrow$reactive.

In a first attempt at the definition of the "$\bullet\!\!\rightarrow$" statement transition relation, we may add the following three rules to those of the deterministic semantics, corresponding to the three problems we have just identified:

$$\frac{p \bullet\!\!\xrightarrow[I]{O,k} p' \qquad k = 0}{\texttt{loop}\ p\ \texttt{end} \bullet\!\!\xrightarrow[I]{\emptyset,+\infty} \texttt{nothing}} \qquad \text{(loop-error)}$$

$$\frac{p \bullet\!\!\xrightarrow[I \backslash \{S\}]{O^-,k^-} p^- \qquad S \in O^- \qquad p \bullet\!\!\xrightarrow[I \cup \{S\}]{O^+,k^+} p^+ \qquad S \notin O^+}{\texttt{signal}\ S\ \texttt{in}\ p\ \texttt{end} \bullet\!\!\xrightarrow[I]{\emptyset,+\infty} \texttt{nothing}} \qquad \text{(signal}+-\text{)}$$

$$\frac{p \bullet\!\!\xrightarrow[I \backslash \{S\}]{O^-,k^-} p^- \qquad S \notin O^- \qquad p \bullet\!\!\xrightarrow[I \cup \{S\}]{O^+,k^+} p^+ \qquad S \in O^+}{\texttt{signal}\ S\ \texttt{in}\ p\ \texttt{end} \bullet\!\!\xrightarrow[I]{\emptyset,+\infty} \texttt{nothing}} \qquad \text{(signal}-+\text{)}$$

Intuitively,

- (loop-error) would report *instantaneous* loop bodies,
- (signal+−) would report *non-reactive* signals (neither (signal+) nor (signal−)),
- (signal−+) would report *non-strongly-deterministic* signals (both (signal+) and (signal−)).

But the resulting semantics lacks the intended properties. For example, the following statement is non-$\circ\!\!\rightarrow$reactive, but admits a reaction with finite completion code in this extended semantics:

```
signal S in
  present S then
    loop nothing end
  end
end
```

On the one hand, because of the implicit "`else nothing`" branch,

$$\frac{\mathtt{S} \notin I\backslash\{\mathtt{S}\} \qquad \mathtt{nothing} \bullet\!\xrightarrow[I\backslash\{\mathtt{S}\}]{\emptyset,0} \mathtt{nothing}}{\mathtt{present\ S\ then\ loop\ nothing\ end\ end} \bullet\!\xrightarrow[I\backslash\{\mathtt{S}\}]{\emptyset,0} \mathtt{nothing}} \quad \text{using (present−)}$$

On the other hand,

$$\frac{\mathtt{S} \in I \cup \{\mathtt{S}\} \qquad \dfrac{\mathtt{nothing} \bullet\!\xrightarrow[I\cup\{\mathtt{S}\}]{\emptyset,0} \mathtt{nothing}}{\mathtt{loop\ nothing\ end} \bullet\!\xrightarrow[I\cup\{\mathtt{S}\}]{\emptyset,+\infty} \mathtt{nothing}}\ \text{using (loop-error)}}{\mathtt{present\ S\ then\ loop\ nothing\ end\ end} \bullet\!\xrightarrow[I\cup\{\mathtt{S}\}]{\emptyset,+\infty} \mathtt{nothing}} \quad \text{using (present+)}$$

Thus, by rule (signal−−),

$$\mathtt{signal\ S\ in\ present\ S\ then\ loop\ nothing\ end\ end\ end} \bullet\!\xrightarrow[I]{\emptyset,0} \mathtt{nothing}$$

Indeed, the (signal−−) rule may define a reaction for "`signal` S `in` p `end`" with a completion code $k^- \in \mathbb{N}$, even if k^+ is $+\infty$. In order to avoid such patterns, we shall further constrain the (signal−−) rule as the following:

$$\frac{p \bullet\!\xrightarrow[I\backslash\{S\}]{O^-,k^-} p^- \qquad S \notin O^- \qquad p \bullet\!\xrightarrow[I\cup\{S\}]{O^+,k^+} p^+ \qquad S \notin O^+ \qquad \max(k^-,k^+) < +\infty}{\mathtt{signal}\ S\ \mathtt{in}\ p\ \mathtt{end} \bullet\!\xrightarrow[I]{O^-,k^-} \delta_1^{k^-}(\mathtt{signal}\ S\ \mathtt{in}\ p^-\ \mathtt{end})} \quad \text{(signal−−)}$$

The "$\max(k^-,k^+) < +\infty$" hypothesis ensures that all deductions are obtained from premises with finite completion codes. We update rules (signal++), (signal+−), and (signal−+) as well:

$$\frac{p \bullet\!\xrightarrow[I\backslash\{S\}]{O^-,k^-} p^- \qquad S \in O^- \qquad p \bullet\!\xrightarrow[I\cup\{S\}]{O^+,k^+} p^+ \qquad S \in O^+ \qquad \max(k^-,k^+) < +\infty}{\mathtt{signal}\ S\ \mathtt{in}\ p\ \mathtt{end} \bullet\!\xrightarrow[I]{O^+\backslash\{S\},k^+} \delta_1^{k^+}(\mathtt{signal}\ S\ \mathtt{in}\ p^+\ \mathtt{end})} \quad \text{(signal++)}$$

$$\frac{p \xrightarrow[I\backslash\{S\}]{O^-,k^-} p^- \quad S \in O^- \quad p \xrightarrow[I\cup\{S\}]{O^+,k^+} p^+ \quad S \notin O^+ \quad \max(k^-,k^+) < +\infty}{\texttt{signal } S \texttt{ in } p \texttt{ end} \xrightarrow[I]{\emptyset,+\infty} \texttt{nothing}} \quad \text{(signal+−)}$$

$$\frac{p \xrightarrow[I\backslash\{S\}]{O^-,k^-} p^- \quad S \notin O^- \quad p \xrightarrow[I\cup\{S\}]{O^+,k^+} p^+ \quad S \in O^+ \quad \max(k^-,k^+) < +\infty}{\texttt{signal } S \texttt{ in } p \texttt{ end} \xrightarrow[I]{\emptyset,+\infty} \texttt{nothing}} \quad \text{(signal−+)}$$

We expect, however, reactions with infinite completion codes to have zero outputs:

$$S \in O^{\pm} \Rightarrow k^{\pm} \in \mathbb{N}$$

Therefore, we can discard half of these extra constraints, and retain the equivalent set of rules:

$$\frac{p \xrightarrow[I\backslash\{S\}]{O^-,k^-} p^- \quad S \notin O^- \quad p \xrightarrow[I\cup\{S\}]{O^+,k^+} p^+ \quad S \notin O^+ \quad \max(k^-,k^+) < +\infty}{\texttt{signal } S \texttt{ in } p \texttt{ end} \xrightarrow[I]{O^-,k^-} \delta_1^{k^-}(\texttt{signal } S \texttt{ in } p^- \texttt{ end})} \quad \text{(signal−−)}$$

$$\frac{p \xrightarrow[I\backslash\{S\}]{O^-,k^-} p^- \quad S \in O^- \quad p \xrightarrow[I\cup\{S\}]{O^+,k^+} p^+ \quad S \in O^+}{\texttt{signal } S \texttt{ in } p \texttt{ end} \xrightarrow[I]{O^+\backslash\{S\},k^+} \delta_1^{k^+}(\texttt{signal } S \texttt{ in } p' \texttt{ end})} \quad \text{(signal++)}$$

$$\frac{p \xrightarrow[I\backslash\{S\}]{O^-,k^-} p^- \quad S \in O^- \quad p \xrightarrow[I\cup\{S\}]{O^+,k^+} p^+ \quad S \notin O^+ \quad k^+ \neq +\infty}{\texttt{signal } S \texttt{ in } p \texttt{ end} \xrightarrow[I]{\emptyset,+\infty} \texttt{nothing}} \quad \text{(signal+−)}$$

$$\frac{p \xrightarrow[I\backslash\{S\}]{O^-,k^-} p^- \quad S \notin O^- \quad k^- \neq +\infty \quad p \xrightarrow[I\cup\{S\}]{O^+,k^+} p^+ \quad S \in O^+}{\texttt{signal } S \texttt{ in } p \texttt{ end} \xrightarrow[I]{\emptyset,+\infty} \texttt{nothing}} \quad \text{(signal−+)}$$

One last rule must be introduced to propagate $+\infty$ through local signal declarations:

$$\frac{p \xrightarrow[I\backslash\{S\}]{O^-,k^-} p^- \quad p \xrightarrow[I\cup\{S\}]{O^+,k^+} p^+ \quad \max(k^-,k^+) = +\infty}{\texttt{signal } S \texttt{ in } p \texttt{ end} \xrightarrow[I]{\emptyset,+\infty} \texttt{nothing}} \quad \text{(signal-}\infty\text{)}$$

We call the resulting semantics the *reactive deterministic semantics*. Its rules are gathered in Figure 3.2. It has all the intended properties, in particular:

Lemma 3.10. *If* $p \xrightarrow[I]{O,k} p'$ *then* $k = +\infty$ *iff* (loop-error), (signal+−) *or* (signal−+) *is* $\bullet\!\!\rightarrow$*used by the reaction.*

Proof. Structural induction. The rules (loop-error), (signal+−), and (signal−+) generate $+\infty$. The other rules may only propagate $+\infty$. No rule can hide $+\infty$, i.e. if $+\infty$ occurs in a proof tree, then it is propagated down to the conclusion. □

Lemma 3.11. *If* $p \xrightarrow[I]{O,k} p'$ *and* $k < +\infty$ *then rule* (signal-∞) *is not* $\bullet\!\!\rightarrow$*used by the reaction.*

Proof. Structural induction. □

$$\texttt{nothing} \bullet\!\xrightarrow[I]{\emptyset,\,0} \texttt{nothing} \qquad \text{(nothing)}$$

$$\texttt{pause} \bullet\!\xrightarrow[I]{\emptyset,\,1} \texttt{nothing} \qquad \text{(pause)}$$

$$\texttt{exit}\ T_d \bullet\!\xrightarrow[I]{\emptyset,\,d+2} \texttt{nothing} \qquad \text{(exit)}$$

$$\texttt{emit}\ S \bullet\!\xrightarrow[I]{\{S\},\,0} \texttt{nothing} \qquad \text{(emit)}$$

$$\frac{p \bullet\!\xrightarrow[I]{O,\,0} p' \qquad q \bullet\!\xrightarrow[I]{O',\,k} q'}{p\texttt{;}\ q \bullet\!\xrightarrow[I]{O \cup O',\,k} q'} \qquad \text{(sequence-0)}$$

$$\frac{p \bullet\!\xrightarrow[I]{O,\,k} p' \qquad q \bullet\!\xrightarrow[I]{O',\,l} q' \qquad m = \max(k, l)}{p\ \texttt{||}\ q \bullet\!\xrightarrow[I]{O \cup O',\,m} \delta_1^m(p'\ \texttt{||}\ q')} \qquad \text{(parallel)}$$

$$\frac{S \in I \qquad p \bullet\!\xrightarrow[I]{O,\,k} p'}{\texttt{present}\ S\ \texttt{then}\ p\ \texttt{else}\ q\ \texttt{end} \bullet\!\xrightarrow[I]{O,\,k} p'} \qquad \text{(present+)}$$

$$\frac{S \notin I \qquad q \bullet\!\xrightarrow[I]{O,\,k} q'}{\texttt{present}\ S\ \texttt{then}\ p\ \texttt{else}\ q\ \texttt{end} \bullet\!\xrightarrow[I]{O,\,k} q'} \qquad \text{(present−)}$$

$$\frac{p \bullet\!\xrightarrow[I]{O,\,k} p'}{\texttt{trap}\ T\ \texttt{in}\ p\ \texttt{end} \bullet\!\xrightarrow[I]{O,\,\downarrow k} \delta_1^k(\texttt{trap}\ T\ \texttt{in}\ p'\ \texttt{end})} \qquad \text{(trap)}$$

$$\frac{p \bullet\!\xrightarrow[I]{O,\,k} p' \qquad k \neq 0}{p\texttt{;}\ q \bullet\!\xrightarrow[I]{O,\,k} \delta_1^k(p'\texttt{;}\ q)} \qquad \text{(sequence-}k\text{)}$$

$$\frac{p \bullet\!\xrightarrow[I]{O,\,k} p' \qquad k \neq 0}{\texttt{loop}\ p\ \texttt{end} \bullet\!\xrightarrow[I]{O,\,k} \delta_1^k(p'\texttt{;}\ \texttt{loop}\ p\ \texttt{end})} \qquad \text{(loop)}$$

$$\frac{p \bullet\!\xrightarrow[I]{O,\,k} p' \qquad k = 0}{\texttt{loop}\ p\ \texttt{end} \bullet\!\xrightarrow[I]{\emptyset,\,+\infty} \texttt{nothing}} \qquad \text{(loop-error)}$$

$$\frac{p \bullet\!\xrightarrow[I\backslash\{S\}]{O^-,\,k^-} p^- \qquad S \in O^- \qquad p \bullet\!\xrightarrow[I\cup\{S\}]{O^+,\,k^+} p^+ \qquad S \in O^+}{\texttt{signal}\ S\ \texttt{in}\ p\ \texttt{end} \bullet\!\xrightarrow[I]{O^+\backslash\{S\},\,k^+} \delta_1^{k^+}(\texttt{signal}\ S\ \texttt{in}\ p'\ \texttt{end})} \qquad \text{(signal++)}$$

$$\frac{p \bullet\!\xrightarrow[I\backslash\{S\}]{O^-,\,k^-} p^- \qquad S \notin O^- \qquad p \bullet\!\xrightarrow[I\cup\{S\}]{O^+,\,k^+} p^+ \qquad S \notin O^+ \qquad \max(k^-, k^+) < +\infty}{\texttt{signal}\ S\ \texttt{in}\ p\ \texttt{end} \bullet\!\xrightarrow[I]{O^-,\,k^-} \delta_1^{k^-}(\texttt{signal}\ S\ \texttt{in}\ p'\ \texttt{end})} \qquad \text{(signal−−)}$$

$$\frac{p \bullet\!\xrightarrow[I\backslash\{S\}]{O^-,\,k^-} p^- \qquad S \in O^- \qquad p \bullet\!\xrightarrow[I\cup\{S\}]{O^+,\,k^+} p^+ \qquad S \notin O^+ \qquad k^+ \neq +\infty}{\texttt{signal}\ S\ \texttt{in}\ p\ \texttt{end} \bullet\!\xrightarrow[I]{\emptyset,\,+\infty} \texttt{nothing}} \qquad \text{(signal+−)}$$

$$\frac{p \bullet\!\xrightarrow[I\backslash\{S\}]{O^-,\,k^-} p^- \qquad S \notin O^- \qquad k^- \neq +\infty \qquad p \bullet\!\xrightarrow[I\cup\{S\}]{O^+,\,k^+} p^+ \qquad S \in O^+}{\texttt{signal}\ S\ \texttt{in}\ p\ \texttt{end} \bullet\!\xrightarrow[I]{\emptyset,\,+\infty} \texttt{nothing}} \qquad \text{(signal−+)}$$

$$\frac{p \bullet\!\xrightarrow[I\backslash\{S\}]{O^-,\,k^-} p^- \qquad p \bullet\!\xrightarrow[I\cup\{S\}]{O^+,\,k^+} p^+ \qquad \max(k^-, k^+) = +\infty}{\texttt{signal}\ S\ \texttt{in}\ p\ \texttt{end} \bullet\!\xrightarrow[I]{\emptyset,\,+\infty} \texttt{nothing}} \qquad \text{(signal-}\infty\text{)}$$

Figure 3.2: Reactive Deterministic Semantics

Lemma 3.12. *For every statement p, p is $\bullet\!\!\rightarrow$reactive and $\bullet\!\!\rightarrow$strongly-deterministic.*

Proof. Structural induction. □

The reactive deterministic semantics defines exactly one execution per sequence of inputs:

Corollary 3.13. *For every statement p, p is $\bullet\!\!\rightarrow$strongly-correct.*

with completion code $+\infty$ iff there is an error:

Lemma 3.14. *p is not $\circ\!\!\rightarrow$reactive w.r.t. I iff there exists O and p' such that $p \xrightarrow[I]{O,+\infty}_{\bullet} p'$.*

Proof. All deductions of the deterministic semantics remain valid in the reactive deterministic semantics. Reciprocally, by Lemma 3.10 and Lemma 3.11, a proof of a reaction of the reactive deterministic semantics with a finite completion code never uses rule (loop-error), (signal+−), (signal−+), or (signal-∞). □

Theorem 3.15. *p is proper iff the rules* (loop-error)*,* (signal+−)*, and* (signal−+) *are not $\bullet\!\!\rightarrow$relevant to the execution of p.*

Proof. By recursion, using Lemma 3.14, p is proper iff no execution of p ends with completion code $+\infty$, that is to say iff these rules are not $\bullet\!\!\rightarrow$relevant to the execution of p, by Lemma 3.10. □

3.4 Proper Subterms

Compilers not only reject programs because they are incorrect, but they give precise error messages about the reasons and locations of errors. Thanks to the explicit handling of errors provided by the reactive deterministic semantics, we shall now easily formalize *non-proper occurrences of subterms* within non-proper statements, thus formally identifying errors.

In Chapter 2, we have described tags, and shown how occurrences of subterms within statements can be identified using tags. Applying the same construction to the reactive deterministic semantics, we can define for any well-tagged (i.e. pairwise distinct tags) statement p:

- The loop "$\texttt{loop}^n$ b `end`" in p is *not proper* iff it is $\bullet\!\!\rightarrow$reducible by rule (loop-error) in the execution of p, *proper* otherwise.
- The signal declaration "$\texttt{signal}^n$ S `in` b `end`" in p is *not proper* iff it is $\bullet\!\!\rightarrow$reducible by rule (signal+−) or (signal−+) in the execution of p, *proper* otherwise.

Corollary 3.16. *A statement p is proper iff its loops and signal declarations are proper.*

For example, the subterm "`loop nothing end`" is not proper in context "`pause;` [ ]", that is to say in program:

```
pause;
loop nothing end
```

As a result, the program itself is not proper.

However, "`loop nothing end`" is proper in context "`trap T in exit T; [ ] end`", that is to say in program:

```
trap T in
  exit T;
  loop nothing end
end
```

In general, the properness of a loop or local signal declaration depends on its position. With the definition of non-instantaneous loops in Chapter 4, we shall ensure that loops are proper, in a position-independent manner.

SUMMARY

We have defined a deterministic semantics for pure Esterel, leading to an updated correctness criterion, which we call properness. This semantics, while preserving the behaviors of "reasonable" programs (i.e. proper programs), eliminates the non-determinism of the logical behavioral semantics. We have further extended this semantics into a reactive deterministic semantics, by embedding the ability to deal with improper programs into the semantics itself. As a result, we were able to express properness as the conjunction of two predicates respectively concerned with loops and local signal declarations. In other words, we can now precisely identify errors within programs and classify them into loop errors and local signal declarations errors.

Chapter 4

Instantaneous Loops

The logical behavioral semantics provides a unique rule for `loop` statements:

$$\frac{p \xrightarrow[I]{O,k} p' \quad k \neq 0}{\texttt{loop}\ p\ \texttt{end} \xrightarrow[I]{O,k} \delta_1^k(p'\texttt{;}\ \texttt{loop}\ p\ \texttt{end})} \quad \text{(loop)}$$

The side condition $k \neq 0$ requires loop bodies to:

- either execute a `pause` instruction ($k = 1$).
- or raise an exception, which aborts the loop ($k \geq 2$).

As a result, at most one iteration can be completed in each instant[1]. Reciprocally, programs like "`loop nothing end`" are non-reactive, thus logically incorrect.

This requirement is of dynamic nature, as it is enforced in each reaction, and may be responsible for runtime errors at any instant. For example, "`await I; loop nothing end`" fails upon the reception of signal `I`.

In critical systems however, runtime errors cannot be tolerated, as they cannot be corrected on the fly. As a consequence, Esterel compilers have to prevent them. Intuitively, they may only compile *loop safe* programs, that is to say programs for which the runtime check $k \neq 0$ can be safely ignored. Reciprocally, *loop unsafe* programs must be rejected.

Compilers must replace a runtime check by a compile time analysis of programs. We believe that this issue is a typical example of a well known complexity of Esterel, already solved by existing implementations, but not yet fully understood. For example, depending your choice of compiler, the following program may be (correctly) compiled or rejected:

```
loop
  trap T in
    trap U in
      trap V in
        exit U || exit V
      end;
      exit T
    end;
    pause
  end
end
```

[1] Nevertheless, many iterations of the same loop may be started in a single instant (cf. Chapter 5).

In order to establish the logical correctness of this program, one has to take into account the relative priority of `U` over `V`, which shows that exception `T` is never raised, which in turn implies that the `pause` instruction is executed in each instant. The SAXO-RT [CPP+02] compiler for Esterel, for instance, cannot complete this chain of deductions, as its internal representation of programs abstracts away exception priority levels.

Our goal in this chapter is to specify such a conservative filtering algorithm for the rejection of unsafe programs:

- in a formal, provably correct fashion,
- precisely enough,
- with a reasonable computational complexity.

We shall formalize the algorithm implemented in the Esterel compiler of Berry et al. [Ber00b], and partly described in [Ber99], which, in our view, provides a reasonable trade-off between precision and efficiency, and matches user intuition well enough. In particular, it handles the above example.

In order to prove the correctness of this algorithm, we first formalize loop safety in Section 4.1. Then, in Section 4.2, we define (non-)instantaneous statements, and establish that programs with non-instantaneous loop bodies are loop safe and have proper loops (cf. Chapter 3). Finally, in Section 4.3, we specify the decision procedure for the acceptance/rejection of programs w.r.t. loop safety. We prove it to be correct.

4.1 Loop Safety

We define loop safety w.r.t. the logical behavioral semantics and the deterministic semantics. We relate loop safety to the various correctness criteria we previously defined: logical correctness, strong correctness, properness, and proper loops.

Logical Behavioral Semantics

Ignoring the runtime check $k \neq 0$ for loop bodies means replacing the (loop) rule of the logical behavioral semantics with the rule:

$$\frac{p \xrightarrow[I]{O,k} p'}{\texttt{loop}\ p\ \texttt{end} \xrightarrow[I]{O,k} \delta_1^k(p'\texttt{;}\ \texttt{loop}\ p\ \texttt{end})} \qquad \text{(unsafe-loop)}$$

or equivalently keeping the initial (loop) rule and adding to the semantics the rule:

$$\frac{p \xrightarrow[I]{O,k} p' \qquad k = 0}{\texttt{loop}\ p\ \texttt{end} \xrightarrow[I]{O,0} \texttt{nothing}} \qquad \text{(unsafe-loop-0)}$$

We choose the second option. Remark that (unsafe-loop-0) differs from the (loop-error) rule introduced in Chapter 3 in the definition of the completion code and outputs of the reaction: (loop-error) reports an error, whereas (unsafe-loop-0) ignores the error.

We say that p is *→loop-safe* iff (unsafe-loop-0) is not relevant to the execution of p in the logical behavioral semantics extended with rule (unsafe-loop-0), meaning that the removal of the check $k \neq 0$ changes nothing to the internal and external behavior of p.

Strongly correct programs are not →loop-safe in general. As a consequence, the check $k \neq 0$ cannot be safely ignored for strongly correct programs, and even more so for logically correct programs.

For instance,

```
signal S in
  present S then
    loop emit S end
  else
    pause
  end
end
```

is strongly correct but not $\rightarrow$loop-safe, as the extended semantics defines reactions for it using rule (unsafe-loop-0). Whatever I,

$$\dfrac{\dfrac{\mathtt{S} \in I \cup \{\mathtt{S}\} \qquad \dfrac{\mathtt{emit\ S} \xrightarrow[I\cup\{\mathtt{S}\}]{\{\mathtt{S}\},0} \mathtt{nothing}}{\mathtt{loop\ emit\ S\ end} \xrightarrow[I\cup\{\mathtt{S}\}]{\{\mathtt{S}\},0} \mathtt{nothing}} \text{ using (unsafe-loop-0)}}{\mathtt{present\ S\ then\ loop\ emit\ S\ end\ end} \xrightarrow[I\cup\{\mathtt{S}\}]{\{\mathtt{S}\},0} \mathtt{nothing} \qquad \mathtt{S} \in \{\mathtt{S}\}}}{\mathtt{signal\ S\ in\ present\ S\ then\ loop\ emit\ S\ end\ else\ pause\ end\ end} \xrightarrow[I]{\emptyset,0} \mathtt{nothing}}$$

Therefore, $\rightarrow$loop-safety requires its own analysis.

Deterministic Semantics

Similarly, we may extend the deterministic semantics with the same rule:

$$\dfrac{p \overset{O,k}{\underset{I}{\circ\!\!\longrightarrow}} p' \qquad k = 0}{\mathtt{loop}\ p\ \mathtt{end} \overset{O,0}{\underset{I}{\circ\!\!\longrightarrow}} \mathtt{nothing}} \qquad \text{(unsafe-loop-0)}$$

and say that p is $\circ\!\!\rightarrow$*loop-safe* iff (unsafe-loop-0) is not relevant to the execution of p in the extended deterministic semantics.

Lemma 4.1. *The extended deterministic semantics remains globally deterministic. Every statement p is strongly deterministic w.r.t. the extended deterministic semantics.*

Proof. Structural induction. □

Lemma 4.2. *All proper statements are $\circ\!\!\rightarrow$loop-safe and $\rightarrow$loop-safe.*

Proof. By Lemma 4.1, if p is $\circ\!\!\rightarrow$reactive w.r.t. the unextended deterministic semantics, then the extended deterministic semantics cannot define additional reactions for p. Therefore, if p is proper, the extended deterministic semantics does not define additional executions for p, that is to say (unsafe-loop-0) is not relevant to the execution of p in the extended deterministic semantics.

By Lemma 3.5, which remains valid for the extended semantics, (unsafe-loop-0) is not relevant to the execution of p in the extended logical behavioral semantics. □

In other words, $\circ\!\!\rightarrow$loop-safety is one component of properness. This is again a reason for preferring the deterministic semantics over the logical behavioral semantics, and properness over logical or strong correctness.

In Chapter 3, we defined proper loops and proper signal declarations, so that properness is the conjunction of the two. As one might expect, there is no need to decide whether local signal declarations are proper or not in order to guarantee $\circ\!\!\!\rightarrow$loop-safety.

Theorem 4.3. *If all loops of p are proper then p is $\circ\!\!\!\rightarrow$loop-safe.*

> *Proof (Sketch).* If some reaction of p uses rule (unsafe-loop-0) in the extended deterministic semantics, then a reaction of p may be defined in the reactive deterministic semantics that uses rule (loop-error) instead of (unsafe-loop-0). □

Comparison

There is no simple connection between $\rightarrow$loop-safety and $\circ\!\!\!\rightarrow$loop-safety (or proper loops).

- $\rightarrow$loop-safety does not imply $\circ\!\!\!\rightarrow$loop-safety (even for strongly correct programs).

```
signal S in
  present S then loop nothing end end
end
```

  is strongly correct (cf. Chapter 3) and $\rightarrow$loop-safe, since `S` cannot be present in the (extended) logical behavioral semantics, but not $\circ\!\!\!\rightarrow$loop-safe, as (unsafe-loop-0) is relevant to its execution in the extended deterministic semantics:

$$\dfrac{\dfrac{\texttt{S} \in I \cup \{\texttt{S}\} \qquad \dfrac{\texttt{nothing} \xrightarrow[I\cup\{\texttt{S}\}]{\emptyset,0}\!\circ\; \texttt{nothing}}{\texttt{loop nothing end} \xrightarrow[I\cup\{\texttt{S}\}]{\emptyset,0}\!\circ\; \texttt{nothing}}\ \text{using (unsafe-loop-0)}}{\texttt{present S then loop nothing end end} \xrightarrow[I\cup\{\texttt{S}\}]{\emptyset,0}\!\circ\; \texttt{nothing}} \qquad \texttt{S} \notin \emptyset \qquad \ldots}{\texttt{signal S in present S then loop nothing end end end} \xrightarrow[I]{\emptyset,0}\!\circ\; \texttt{nothing}}$$

- $\circ\!\!\!\rightarrow$loop-safety does not imply $\rightarrow$loop-safety (even for strongly correct programs).

```
loop
  signal S in
    present S then emit S else pause end
  end
end
```

  is strongly correct (cf. Chapter 3) and $\circ\!\!\!\rightarrow$loop-safe, as the extended deterministic semantics defines no reaction for it. But the program is not $\rightarrow$loop-safe, because of reaction:

$$\dfrac{\dfrac{\dfrac{\texttt{S} \in I \cup \{\texttt{S}\} \qquad \texttt{emit S} \xrightarrow[I\cup\{\texttt{S}\}]{\{\texttt{S}\},0} \texttt{nothing}}{\texttt{present S then emit S else pause end} \xrightarrow[I\cup\{\texttt{S}\}]{\{\texttt{S}\},0} \texttt{nothing}} \qquad \texttt{S} \in \{\texttt{S}\}}{\texttt{signal S in present S then emit S else pause end end} \xrightarrow[I]{\emptyset,0} \texttt{nothing}}}{\texttt{loop signal S in present S then emit S else pause end end end} \xrightarrow[I]{\emptyset,0} \texttt{nothing}}$$

As a result, in the rest of this chapter, we shall deal with $\rightarrow$loop-safety as well as proper loops (thus $\circ\!\!\!\rightarrow$loop-safety). For simplicity, we say that p is loop safe iff p is both $\rightarrow$loop-safe and $\circ\!\!\!\rightarrow$loop-safe.

4.2 (Non-)Instantaneous Statements

In Chapter 3, we observed that the properness of a loop depends its position in the program. In a first step toward an efficient analysis of loop safety, we shall abstract away such dependencies in a conservative manner. Using universal quantifiers, we first define:

- *p is instantaneous* iff $\forall I, \forall O, \forall k, \forall p' : p \xrightarrow[I]{O,k} p' \Rightarrow k = 0$.
- *p is non-instantaneous* iff $\forall I, \forall O, \forall k, \forall p' : p \xrightarrow[I]{O,k} p' \Rightarrow k \neq 0$.

Remark that there are statements that are neither instantaneous nor non-instantaneous, as they behavior may be context-dependent. For instance,

- "`nothing`" is instantaneous.
- "`pause`" is non-instantaneous.
- "`exit` T" is non-instantaneous ($k \geq 2$).
- "`present` S `then pause end`" is neither instantaneous nor non-instantaneous.

We choose to define (non-)instantaneous statements using the logical behavioral semantics. Fortunately, there is no need for alternate definitions w.r.t. to the deterministic semantics, since all behaviors defined by the deterministic semantics also exist in the logical behavioral semantics:

Lemma 4.4. *If p is instantaneous then $\forall I, \forall O, \forall k, \forall p' : p \circ\!\xrightarrow[I]{O,k} p' \Rightarrow k = 0$.*

Lemma 4.5. *If p is non-instantaneous then $\forall I, \forall O, \forall k, \forall p' : p \circ\!\xrightarrow[I]{O,k} p' \Rightarrow k \neq 0$.*

Proof. By Lemma 3.4, if $p \circ\!\xrightarrow[I]{O,k} p'$ then $p \xrightarrow[I]{O,k} p'$. □

We now establish that non-instantaneous loop bodies ensure loop safety and proper loops:

Theorem 4.6. *If every loop body occurring in p is non-instantaneous then p is loop safe.*

Proof. Let us suppose a proof uses rule (unsafe-loop-0) in either extended semantics for a loop of body b. Then b reacts to some inputs with completion code 0. Therefore, b is not non-instantaneous. □

Theorem 4.7. *If p is non-instantaneous then "*`loop` *p* `end`*" is proper in $C[\,]$, whatever $C[\,]$.*

Proof. Similar the previous proof, replacing (unsafe-loop-0) with (loop-error). □

Loop safe programs may nevertheless contain loop bodies that *may be instantaneous*, that is to say are not non-instantaneous ($\neq$ instantaneous). For example,

```
trap T in
  exit T;
  loop nothing end
end
```

is loop safe, as "`exit T`" prevents the execution of "`loop nothing end`".

However, relying on such externally enforced correctness properties is useless and error-prone. Therefore, forbidding loop bodies that may be instantaneous is a conservative approach which makes sense both from the compiler's and from the user's perspectives.

From now on, we say that:

- a loop is *potentially instantaneous* iff its body may be instantaneous.
- a loop is *non-instantaneous* iff its body is non-instantaneous.

In the sequel, we want to detect and reject potentially instantaneous loops.

4.3 Static Analysis

Deciding whether a statement may be instantaneous or not requires one to take into account the 2^n possible valuations of its n input signals, which is unreasonable for large programs. In fact, SAT (Boolean satisfiability in propositional logic) can be expressed in terms of instantaneous termination in Esterel, via a polynomial reduction. For instance,

$$(A \vee \neg B \vee C) \wedge (\neg A \vee C \vee \neg D) \wedge (\neg B \vee \neg C \vee D) \text{ is satisfiable}$$

$$\Updownarrow$$

```
present A else present B then present C else pause end end end;
present A then present C else present D then pause end end end;
present B then present C then present D else pause end end end
```

may be instantaneous

Therefore, compilers rely on conservative static analysis techniques for the detection of potentially instantaneous loops, such as the one we formalize below.

Abstract Semantics

By making abstraction [CC77] of signals (inputs I and outputs O) and residuals (p') in the logical behavioral semantics of Esterel we obtain the abstract semantics of Figure 4.1, where:

$$p \xrightarrow[I]{O,k} p' \quad \text{is abstracted into:} \quad p \xrightarrow[\cdot]{\cdot,k} \cdot \quad \text{which we abbreviate as:} \quad p \hookrightarrow k$$

Remark that the abstract rules corresponding to (signal+) and (signal−) are the same.

Lemma 4.8. *If* $p \xrightarrow[I]{O,k} p'$ *then* $p \hookrightarrow k$.

Proof. All deductions of the logical behavioral semantics remain valid in the abstract semantics. □

We define the set Γ_p of the *potential completion codes of* p as $\Gamma_p = \{k \in \mathbb{N} \text{ s.t. } p \hookrightarrow k\}$.

Corollary 4.9. *For any statement* p, $\left\{k \in \mathbb{N} \text{ s.t. } \exists I, \exists O, \exists p' : p \xrightarrow[I]{O,k} p'\right\} \subset \Gamma_p$

Corollary 4.10. *If* $0 \notin \Gamma_b$ *then* b *is non-instantaneous.*

Figure 4.2 derives a recursive algorithm for the computation of Γ from the abstract semantics of Figure 4.1, by collecting the completion codes obtained using all deduction rules that may apply to each Esterel construct. For example, $k \in \Gamma_{p;\ q}$ iff:

- either $0 \in \Gamma_p$ and $k \in \Gamma_q$ (sequence-0),
- or $k \in \Gamma_p$ and $k \neq 0$ (sequence-k).

Similarly, $k \in \Gamma_{\texttt{loop}\ p\ \texttt{end}}$ iff $k \in \Gamma_p$ and $k \neq 0$. Hence, $\Gamma_{\texttt{loop}\ p\ \texttt{end}} = \Gamma_p \backslash \{0\}$.

$$\texttt{nothing} \hookrightarrow 0 \quad \text{(nothing)}$$

$$\texttt{pause} \hookrightarrow 1 \quad \text{(pause)}$$

$$\texttt{exit } T_d \hookrightarrow d+2 \quad \text{(exit)}$$

$$\texttt{emit } S \hookrightarrow 0 \quad \text{(emit)}$$

$$\frac{p \hookrightarrow k \quad k \neq 0}{\texttt{loop } p \texttt{ end} \hookrightarrow k} \quad \text{(loop)}$$

$$\frac{p \hookrightarrow k \quad q \hookrightarrow l}{p \texttt{ || } q \hookrightarrow \max(k,l)} \quad \text{(parallel)}$$

$$\frac{p \hookrightarrow k}{\texttt{present } S \texttt{ then } p \texttt{ else } q \texttt{ end} \hookrightarrow k} \quad \text{(present+)}$$

$$\frac{q \hookrightarrow k}{\texttt{present } S \texttt{ then } p \texttt{ else } q \texttt{ end} \hookrightarrow k} \quad \text{(present−)}$$

$$\frac{p \hookrightarrow k}{\texttt{trap } T \texttt{ in } p \texttt{ end} \hookrightarrow \downarrow k} \quad \text{(trap)}$$

$$\frac{p \hookrightarrow 0 \quad q \hookrightarrow k}{p\texttt{; } q \hookrightarrow k} \quad \text{(sequence-0)}$$

$$\frac{p \hookrightarrow k \quad k \neq 0}{p\texttt{; } q \hookrightarrow k} \quad \text{(sequence-k)}$$

$$\frac{p \hookrightarrow k}{\texttt{signal } S \texttt{ in } p \texttt{ end} \hookrightarrow k} \quad \text{(signal+)}$$

$$\frac{p \hookrightarrow k}{\texttt{signal } S \texttt{ in } p \texttt{ end} \hookrightarrow k} \quad \text{(signal−)}$$

Figure 4.1: Abstract Semantics

p	Γ_p
`nothing`	$\{0\}$
`pause`	$\{1\}$
`exit` T_d	$\{d+2\}$
`emit` S	$\{0\}$
`loop` p `end`	$\Gamma_p \backslash \{0\}$
p `||` q	$\{m \in \mathbb{N} \ s.t. \ \exists k \in \Gamma_p, \exists l \in \Gamma_q, m = \max(k,l)\}$ i.e. $\max(\Gamma_p, \Gamma_q)$
`present` S `then` p `else` q `end`	$\Gamma_p \cup \Gamma_q$
`trap` T `in` p `end`	$\{l \in \mathbb{N} \ s.t. \ \exists k \in \Gamma_p, l = \downarrow k\}$ i.e. $\downarrow\Gamma_p$
p`;` q	if $0 \in \Gamma_p$ then $(\Gamma_p \backslash \{0\}) \cup \Gamma_q$ else Γ_p
`signal` S `in` p `end`	Γ_p

Figure 4.2: Potential Completion Codes of Reactions

Decision Procedure

In order to ensure that a program p is loop safe, we can compute Γ_b for every loop body b of p, and check that none of these sets Γ_b contains zero. This check can be easily embedded within the computation of Γ itself by replacing:

$$\Gamma_{\texttt{loop}\ p\ \texttt{end}} = \Gamma_p \backslash \{0\}$$

with the following definition:

$$\Gamma_{\texttt{loop}\ p\ \texttt{end}} = \text{if } 0 \in \Gamma_p \text{ then ERROR else } \Gamma_p$$

Theorem 4.11. *If the computation of* Γ_q *completes without* ERROR *for every subterm* q *of the statement* p *then* p *is loop safe and all its loops are proper.*

Proof. This includes computing Γ_b for each loop body b of p. □

This leads to the following decision procedure for any program p:

- compute Γ for all subterms of p in one exhaustive recursive traversal of p,
- if the computation completes normally (no ERROR): accept program (loop safe).
- if the computation fails (ERROR): reject program (potentially loop unsafe).

This procedure can be further tuned to report the location of the potential problem (if any). Formally, we shall attach the tag of the incriminated loop to the exception:

$$\Gamma_{\texttt{loop}^n\ p\ \texttt{end}} = \text{if } 0 \in \Gamma_p \text{ then ERROR}(n) \text{ else } \Gamma_p$$

Since this procedure ensures that all loops of p are proper, the only errors that may be left in p are improper signal declarations, in the sense of Chapter 3.

We shall further extend this procedure in Chapter 5 for the detection of schizophrenia.

Complexity

Let us choose a statement p and define:

- N is the size of p, that is to say the number of primitive constructs in p.
- D is the maximum depth of exceptions in p ($D = -1$ in the absence of exceptions).

For example, $N = 5$ and $D = 1$ for the program:

```
trap T in
  trap U in
    exit T1 || exit U0
  end
end
```

of abstract syntax tree:

```
        trap T
          |
        trap U
          |
          ||
        /    \
exit T1      exit U0
```

The number of steps, that is to say operations on sets of completion codes, in the computation of Γ_p is at most equal to N. Under the assumption of atomic set operations, the analysis is linear in the size of the statement.

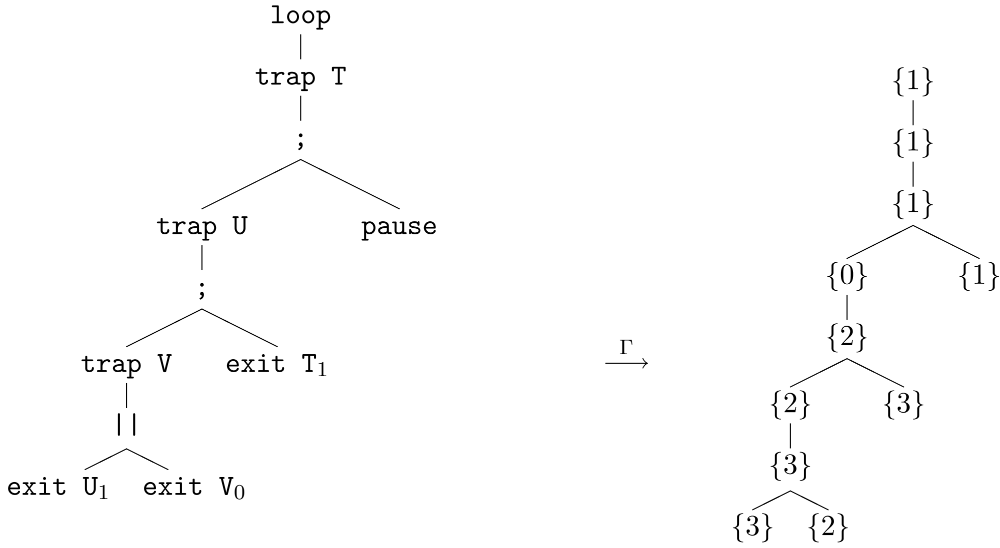
More precisely, the sets of completion codes considered in the computation of Γ_p are subsets of $\{0, ..., D+2\}$ of size $D+3$. All required set operations (including "max") have a complexity at most linear in the size of the sets they apply to. Therefore, provided that exception declarations are not nested beyond some fixed bound ($D \leq c^{\text{te}}$), this analysis is linear in the size of the statement.

Examples

Using this analysis we can establish that our initial example is loop safe.

```
loop
  trap T in
    trap U in
      trap V in
        exit U1 || exit V0
      end;
      exit T1
    end;
    pause
  end
end
```

We obtain:



The computation proceeds as follows:

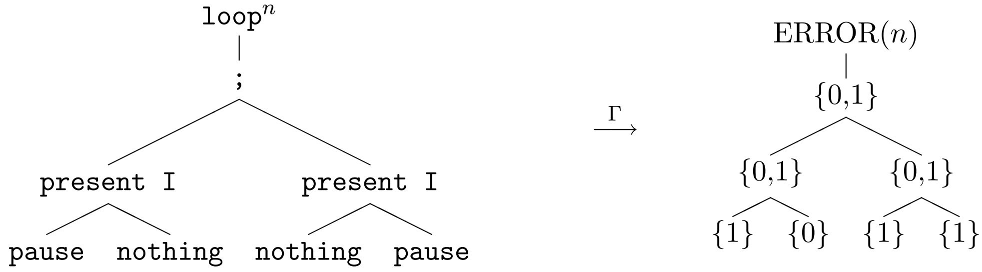
$$
\begin{aligned}
\Gamma_{\texttt{exit U}_1} &= \{3\} \\
\Gamma_{\texttt{exit V}_0} &= \{2\} \\
\Gamma_{\texttt{exit U}_1\texttt{ || exit V}_0} &= \{3\} \\
\Gamma_{\texttt{trap V in exit U}_1\texttt{ || exit V}_0\texttt{ end}} &= \{2\} \\
\Gamma_{\texttt{exit T}_1} &= \{3\} \\
\Gamma_{\texttt{trap V in exit U}_1\texttt{ || exit V}_0\texttt{ end; exit T}_1} &= \{2\} \\
\Gamma_{\texttt{trap U in trap V in exit U}_1\texttt{ || exit V}_0\texttt{ end; exit T}_1\texttt{ end}} &= \{0\} \\
\Gamma_{\texttt{pause}} &= \{1\} \\
\Gamma_{\texttt{trap U in trap V in exit U}_1\texttt{ || exit V}_0\texttt{ end; exit T}_1\texttt{ end; pause}} &= \{1\} \\
\Gamma_{\texttt{trap T in trap U in trap V in exit U}_1\texttt{ || exit V}_0\texttt{ end; exit T}_1\texttt{ end; pause end}} &= \{1\} \\
\Gamma_{\texttt{loop trap T in trap U in trap V in exit U}_1\texttt{ || exit V}_0\texttt{ end; exit T}_1\texttt{ end; pause end end}} &= \{1\}
\end{aligned}
$$

Since the computation completes normally, this program is loop safe. In addition, we verify that it cannot terminate instantaneous or raise an exception.

Unfortunately, since our analysis abstracts signals away, it cannot verify that the following program is loop safe for instance:

```
loop^n
  present I then pause end;
  present I else pause end
end
```

The analysis proceeds as follows:



$$\begin{aligned}
\Gamma_{\texttt{pause; emit O}} &= \{1\}\\
\Gamma_{\texttt{present I then pause end}} &= \{0,1\}\\
\Gamma_{\texttt{present I else pause end}} &= \{0,1\}\\
\Gamma_{\texttt{present I then pause end; present I else pause end}} &= \{0,1\}\\
\Gamma_{\texttt{loop}^n\texttt{ present I then pause end; present I else pause end end}} &= \text{ERROR}(n)
\end{aligned}$$

A generic workaround for such issues consists in adding a `pause` statement in parallel to the body of the incriminated `loop` (as reported by the analysis). In our example, we obtain:

```
loop
  present I then pause end;
  present I else pause end
||
  pause
end
```

Provided that the original loop is non-instantaneous, this simple program transformation preserves the semantics, while ensuring a successful analysis:

$$\begin{aligned}
\Gamma_{\texttt{present I then pause; emit O end; present I else pause end || pause}} &= \max(\{0,1\},\{1\}) &= \{1\}\\
\Gamma_{\texttt{loop present I then pause; emit O end; present I else pause end || pause end}} & &= \{1\}
\end{aligned}$$

This transformation however leads to less efficient code generation[2]. Therefore, by precisely taking into account exceptions in our analysis, we require less user thinking and intervention, and enable a better code generation than analyses that ignore priority levels.

Nevertheless, because the exact analysis would be of exponential complexity, false positives cannot be totally eliminated.

[2]In particular, this transformation introduces schizophrenic parallel statements (cf. Chapter 5).

SUMMARY

We have defined loop safety and formalized a decision procedure for the rejection/acceptance of Esterel programs w.r.t. loop safety. We have established its correctness: all loop unsafe programs are rejected. From now on, we shall only consider programs for which this decision procedure completes successfully. We say they are *safe* programs. Safe programs have *obviously non-instantaneous* loops, that is to say loops that are non-instantaneous in a way that the decision procedure recognizes.

Chapter 5

Schizophrenia

In the previous chapter, we have encountered a first source of complexity due to loops in Esterel: loops are not always correct. We have dealt with it by rejecting at compile time potentially instantaneous loops. In this chapter, we shall see that non-instantaneous loops still require our attention, as several traversals of the same piece of code may take place simultaneously in Esterel. In order to motivate the definition of schizophrenia, we first discuss "beeps" in Esterel in Section 5.1. Then, in Section 5.2, we establish that instantly reentered local signal declarations and parallel subterms are the program patterns we have to focus on. We thoroughly discuss these patterns and define schizophrenia in Section 5.3. Finally, we develop static analysis techniques to detect schizophrenia in Section 5.4. In this chapter, we focus on the logical behavioral semantics. But the results for the deterministic semantics would be the same (and the proofs slightly easier).

5.1 Beeps

In contrast with the full language, pure Esterel is side-effect free. In particular, even if it is possible to emit a signal several times in a reaction, there is no way to count these emissions.

Lemma 5.1. *If p is $\rightarrow$deterministic and instantaneous, then, for all inputs, the statements "p", "p; p" and "p || p" are $\rightarrow$bisimilar.*

Proof. Let us choose p both $\rightarrow$deterministic and instantaneous, and I.

- If p is not $\rightarrow$reactive w.r.t. I, then neither is "p; p", nor "p || p".
- If $p \xrightarrow[I]{O_0, k_0} p'_0$ and $p \xrightarrow[I]{O_1, k_1} p'_1$, then $O_0 = O_1$, $k_0 = k_1 = 0$, $p'_0 = p'_1 = \texttt{nothing}$. As a result, there exists a unique reaction defined for "p; p" w.r.t. I:

$$\frac{p \xrightarrow[I]{O_0, 0} \texttt{nothing} \qquad p \xrightarrow[I]{O_0, 0} \texttt{nothing}}{p\texttt{;}\ p \xrightarrow[I]{O_0 \cup O_0, 0} \texttt{nothing}} \quad \text{(sequence-0)}$$

  Similarly, there exists a unique reaction defined for "p || p" w.r.t. I:

$$\frac{p \xrightarrow[I]{O_0, 0} \texttt{nothing} \qquad p \xrightarrow[I]{O_0, 0} \texttt{nothing} \qquad 0 = \max(0,0)}{p \ \texttt{||}\ p \xrightarrow[I]{O_0 \cup O_0, 0} \delta_1^0(\texttt{nothing || nothing}) = \texttt{nothing}} \quad \text{(parallel)}$$

In all cases, $\forall O, \forall k, \forall p' : p \xrightarrow[I]{O,k} p'$ iff $p\texttt{;}\ p \xrightarrow[I]{O,k} p'$ iff $p \ \texttt{||}\ p \xrightarrow[I]{O,k} p'$. □

In oder to make schizophrenia easier to grasp, we first extend the pure Esterel language with a `beep` instruction of obvious behavior, so that "`beep`" and "`beep; beep`" for instance behave differently. Formally, we add to the transition relation a new integer label "b" which reports the number of beeps occurring in the reaction:

$$p \xrightarrow[I]{O,k,b} p'$$

Then, `beep` is defined as the following:

$$\texttt{beep} \xrightarrow[I]{\emptyset,0,1} \texttt{nothing} \qquad \text{(beep)}$$

And for instance,

$$\frac{p \xrightarrow[I]{O,0,b} p' \qquad q \xrightarrow[I]{O',k,b'} q'}{p\texttt{;}\ q \xrightarrow[I]{O\cup O',k,b+b'} q'} \qquad \text{(sequence-0)}$$

The complete logical behavioral semantics of the extended language is given in Figure 5.1.

In the first instant of execution of a logically correct program, control propagates from left to right, from top to bottom only. No instruction can be executed twice. Formally, let *filter* be the function that recursively removes `loop` constructs from programs:

$$\begin{aligned}
\mathit{filter}(\texttt{beep}) &\stackrel{def}{=} \texttt{beep}\\
\mathit{filter}(\texttt{nothing}) &\stackrel{def}{=} \texttt{nothing}\\
\mathit{filter}(\texttt{pause}) &\stackrel{def}{=} \texttt{pause}\\
\mathit{filter}(p\texttt{;}\ q) &\stackrel{def}{=} \mathit{filter}(p)\texttt{;}\ \mathit{filter}(q)\\
\mathit{filter}(p\ \texttt{||}\ q) &\stackrel{def}{=} \mathit{filter}(p)\ \texttt{||}\ \mathit{filter}(q)\\
\boxed{\mathit{filter}(\texttt{loop}\ p\ \texttt{end}) \stackrel{def}{=} \mathit{filter}(p)}\\
\mathit{filter}(\texttt{signal}\ S\ \texttt{in}\ p\ \texttt{end}) &\stackrel{def}{=} \texttt{signal}\ S\ \texttt{in}\ \mathit{filter}(p)\ \texttt{end}\\
\mathit{filter}(\texttt{emit}\ S) &\stackrel{def}{=} \texttt{emit}\ S\\
\mathit{filter}(\texttt{present}\ S\ \texttt{then}\ p\ \texttt{else}\ q\ \texttt{end}) &\stackrel{def}{=} \texttt{present}\ S\ \texttt{then}\ \mathit{filter}(p)\ \texttt{else}\ \mathit{filter}(q)\ \texttt{end}\\
\mathit{filter}(\texttt{trap}\ T\ \texttt{in}\ p\ \texttt{end}) &\stackrel{def}{=} \texttt{trap}\ T\ \texttt{in}\ \mathit{filter}(p)\ \texttt{end}\\
\mathit{filter}(\texttt{exit}\ T) &\stackrel{def}{=} \texttt{exit}\ T
\end{aligned}$$

Lemma 5.2. *If p is safe then* $\forall I, \forall O, \forall k, \forall b : \left[\exists p' : p \xrightarrow[I]{O,k,b} p'\right] \Leftrightarrow \left[\exists p'_0 : \mathit{filter}(p) \xrightarrow[I]{O,k,b} p'_0\right]$.

Proof. Structural induction. □

The instantaneous behavior (i.e. outputs, completion code, and beeps) of a safe program does not depend on its loops.

By instrumenting the semantics with tags in a way similar to Chapter 2, we also establish that:

Lemma 5.3. *If p is well tagged and $p \xrightarrow[I]{O,k,b,M} p'$ then no tag is repeated in M.*

Proof. Structural induction. □

In particular, each `beep` instruction of a program can be executed (reduced) at most once in its first reaction. In other words, there is at most one beep per `beep` instruction.

$$\texttt{beep} \xrightarrow[I]{\emptyset, 0, 1} \texttt{nothing} \tag{beep}$$

$$\texttt{nothing} \xrightarrow[I]{\emptyset, 0, 0} \texttt{nothing} \tag{nothing}$$

$$\texttt{pause} \xrightarrow[I]{\emptyset, 1, 0} \texttt{nothing} \tag{pause}$$

$$\texttt{exit } T_d \xrightarrow[I]{\emptyset, d+2, 0} \texttt{nothing} \tag{exit}$$

$$\texttt{emit } S \xrightarrow[I]{\{S\}, 0, 0} \texttt{nothing} \tag{emit}$$

$$\frac{p \xrightarrow[I]{O, k, b} p' \qquad k \neq 0}{\texttt{loop } p \texttt{ end} \xrightarrow[I]{O, k, b} \delta_1^k(p'\texttt{; loop } p \texttt{ end})} \tag{loop}$$

$$\frac{p \xrightarrow[I]{O, k, b} p' \qquad q \xrightarrow[I]{O', l, b'} q' \qquad m = \max(k, l)}{p \texttt{ || } q \xrightarrow[I]{O \cup O', m, b+b'} \delta_1^m(p' \texttt{ || } q')} \tag{parallel}$$

$$\frac{S \in I \qquad p \xrightarrow[I]{O, k, b} p'}{\texttt{present } S \texttt{ then } p \texttt{ else } q \texttt{ end} \xrightarrow[I]{O, k, b} p'} \tag{present+}$$

$$\frac{S \notin I \qquad q \xrightarrow[I]{O, k, b} q'}{\texttt{present } S \texttt{ then } p \texttt{ else } q \texttt{ end} \xrightarrow[I]{O, k, b} q'} \tag{present−}$$

$$\frac{p \xrightarrow[I]{O, k, b} p'}{\texttt{trap } T \texttt{ in } p \texttt{ end} \xrightarrow[I]{O, \downarrow k, b} \delta_1^k(\texttt{trap } T \texttt{ in } p' \texttt{ end})} \tag{trap}$$

$$\frac{p \xrightarrow[I]{O, 0, b} p' \qquad q \xrightarrow[I]{O', k, b'} q'}{p\texttt{; } q \xrightarrow[I]{O \cup O', k, b+b'} q'} \tag{sequence-0}$$

$$\frac{p \xrightarrow[I]{O, k, b} p' \qquad k \neq 0}{p\texttt{; } q \xrightarrow[I]{O, k, b} \delta_1^k(p'\texttt{; } q)} \tag{sequence-k}$$

$$\frac{p \xrightarrow[I \cup \{S\}]{O, k, b} p' \qquad S \in O}{\texttt{signal } S \texttt{ in } p \texttt{ end} \xrightarrow[I]{O \setminus \{S\}, k, b} \delta_1^k(\texttt{signal } S \texttt{ in } p' \texttt{ end})} \tag{signal+}$$

$$\frac{p \xrightarrow[I \setminus \{S\}]{O, k, b} p' \qquad S \notin O}{\texttt{signal } S \texttt{ in } p \texttt{ end} \xrightarrow[I]{O, k, b} \delta_1^k(\texttt{signal } S \texttt{ in } p' \texttt{ end})} \tag{signal−}$$

Figure 5.1: Logical Behavioral Semantics of Pure Esterel + “`beep`”

Because of non-instantaneous loops, one might expect that no `beep` instruction can be instantly executed twice, at any stage of the execution. Consider, for instance, the program:

```
loop
  present ALARM then beep end;
  pause
end
```

There is at most one beep per instant. However, this intuition is wrong:

```
loop
  present I then pause end;
  beep
||
  pause
end
```

In this program, if the input signal `I` is present in the first instant of execution, then absent, two beeps are produced in the second instant of execution. Indeed, the first reaction is:

```
loop
  present I then pause end;
  beep
||
  pause
end
```

$\xrightarrow[\{\mathtt{I}\}]{\emptyset, 1, 0}$

```
[ nothing; beep || nothing ];
loop
  present I then pause end;
  beep
||
  pause
end
```

Then, the second reaction is:

```
[ nothing; beep || nothing ];
loop
  present I then pause end;
  beep
||
  pause
end
```

$\xrightarrow[\emptyset]{\emptyset, 1, 2}$

```
[ nothing || nothing ];
loop
  present I then pause end;
  beep
||
  pause
end
```

As the semantics of loops involves unfolding, instructions may be duplicated by the reduction of a loop. Then, two copies of the same instruction, `beep` in this example, may be simultaneously executed at some later stage of the execution.

Such a behavior can be observed for inputless, local-signal-free programs, too:

```
loop
  trap T in
    pause;
    exit T
  ||
    loop
      beep;
      pause
    end
  end
end
```

$\xrightarrow[I_0]{\emptyset, 1, 1}$

```
trap T in
  nothing;
  exit T
||
  nothing;
  loop
    beep;
    pause
  end
end;
loop
  ...
end
```

$\xrightarrow[I_1]{\emptyset, 1, 2}$

```
trap T in
  nothing;
  exit T
||
  nothing;
  loop
    beep;
    pause
  end
end;
loop
  ...
end
```

It can be obtained with sequential programs[1]:

```
signal I in
  emit I;
  loop
    signal S in
      emit S;
      present I then pause end;
      beep;
      present S then pause end
    end
  end
end
```

$\xrightarrow[I_0]{\emptyset, 1, 0} \quad \dots \quad \xrightarrow[I_1]{\emptyset, 1, 2} \quad \dots$

In summary, whereas in the first reaction of a program each `beep` instruction can be executed at most once, this in no longer true for subsequent reactions in general.

There are many reasons to dislike such program patterns:

- Synchronous circuit generation is tricky. In each instant, gates and wires of a synchronous circuit may only encode a Boolean piece of information such as "0 or 1 beep". Therefore, potentially instantly repeated `beep` instructions must be implemented using several wires.

- Debugging is difficult. As long as each instruction of the program is executed once at most per instant, reactions can be easily represented by marking (coloring) the traversed piece of code and the present signals. It becomes much more difficult, if repeated traversals may occur in an instant.

Because of nested loops and weak preemption, this may even be worse: arbitrary many beeps may be instantly generated by a single `beep` instruction. If $C[\ ]$ is the context:

```
loop
  trap T in
    beep;
    pause;
    exit T
  ||
    [ ]
  end
end
```

and $(p_n)_{n \in \mathbb{N}}$ the family of programs:

$$p_0 = \texttt{nothing}, \text{ and for all } n \in \mathbb{N},\ p_{n+1} = C[p_n]$$

then the number c_n of beeps produced in the second instant of execution of p_n is such that:

$$c_0 = 0, \text{ and for all } n \in \mathbb{N},\ c_{n+1} = n + c_n$$

By recursion,

$$c_n = \frac{n(n-1)}{2}$$

This number of beeps is quadratic in the number of `beep` instructions, and also quadratic in the size of the program. In the sequel, with the definition of schizophrenia, we shall capture this complexity, as `beep` instructions (existing or inserted at arbitrary locations) cannot be instantly repeated in non-schizophrenic programs.

[1] Here, the loop is not obviously non-instantaneous, in the sense of Chapter 4. As a result, the program is rejected by Esterel compilers. Nevertheless, it is correct (proper) and illustrates our point.

5.2 Instantly Reentered Subterms

In the examples of the previous section, not only are `beep` instructions enclosed in loops, but these loops always contain either:

- a parallel subterm, which terminates and is instantly restarted,
- a local signal declaration, which is left and instantly reentered,

in those instants of executions with repeated `beep` instructions. In this section, we shall formally define *instantly reentered (i.e. restarted) occurrences of subterms* and establish that this is always the case.

Example

In the following program, `O` is never emitted, as the test always consider a *fresh instance* or *incarnation* of `S`, which is not emitted at the time of the test:

```
loop
  signal S in
    present S then emit O end;
    pause;
    emit S
  end
end
```

$$\xrightarrow[I_0]{\emptyset, 1} \quad ... \quad \xrightarrow[I_1]{\emptyset, 1} \quad ...$$

Because `S` is declared within the loop, starting from the second instant of execution, each reaction involves two instances of `S`:

- an *old* instance, inherited from the previous instant, as `pause` is in the scope of `S`,
- a *new* instance, as the control leaves the scope of `S` and instantly reenters it.

We say that the scope of `S` is *instantly reentered*, which we formalize using tags.

Formal Characterization

As explained in Chapter 2, we can use tags to mark occurrences of subterms. For example, in the logical behavioral semantics with tags of Figure 2.5,

$$\texttt{loop}^1\ \texttt{pause}^2\ \texttt{end} \xrightarrow[\emptyset]{\emptyset, 1, \{1{,}2\}} \texttt{nothing}^2\texttt{;}^3\ \texttt{loop}^1\ \texttt{pause}^2\ \texttt{end} \xrightarrow[\emptyset]{\emptyset, 1, \{1{,}2{,}2{,}3\}} ...$$

Intuitively, if a tag of a program initially tagged with pairwise distinct tags is encountered twice in the same instant of execution, this means that corresponding subterm of the initial program is left and instantly reentered in this instant, because of some loop, which replicated the statement in a previous reaction. In the above example, the initial `pause` statement of tag 2 is left and instantly reentered in the second instant of execution, as both the residual $\texttt{nothing}^2$ and the statement $\texttt{pause}^2$ are reduced in this instant.

Therefore, if p is a well-tagged program, we say that a subterm q of label n of the program p is *instantly reentered* in some execution of p iff its tag n is repeated in the multiset M of one of the reaction of an execution of p:

$$\exists r, \exists I, \exists O, \exists k, \exists M, \exists r' : p \xrightarrow{*} r \wedge r \xrightarrow[I]{O, k, M} r' \wedge \{n, n\} \subset M$$

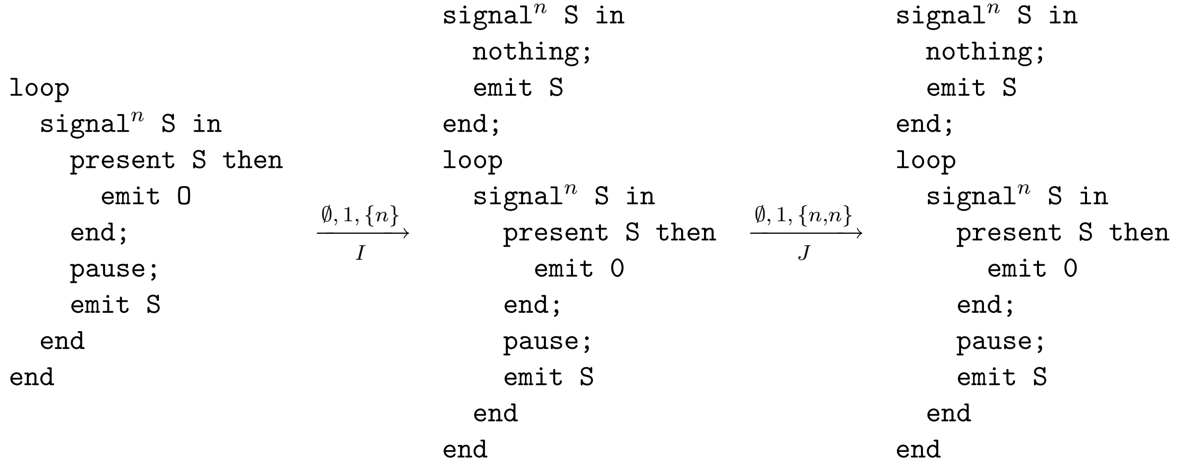


Figure 5.2: Instantly Reentered Signal Scope

Figure 5.2 illustrates this mechanism on our initial example. Initially, the program contains a unique signal declaration, which we label n. The proof tree of the first reaction involves this signal declaration, thus the multiset $\{n\}$. Because loop unfolding occurs, the residual now contains two declarations with tag n. In the second instant and thereafter, the tag n is encountered twice, leading to the multiset $\{n, n\}$. Thus, this signal declaration is instantly reentered.

While made using the logical behavioral semantics, the definition of instantly reentered subterms makes sense for the deterministic semantics as well. Thanks to Lemma 3.4, if a statement is instantly reentered in some execution of the deterministic semantics, then it is also instantly reentered in an execution of the logical behavioral semantics. Therefore, all the results we shall establish for the logical behavioral semantics are valid for the deterministic semantics as well.

Going back to beeps, we obtain:

Theorem 5.4. *If p is a well-tagged program and a* `beep` *instruction of p is instantly reentered then there exists a local signal declaration "*`signal`^{n} `in` ... `end`*" or parallel subterm "...* `||`^{n} *..." of p which is instantly reentered.*

Proof (Sketch). For a `beep` instruction to be instantly repeated, it has to be the case that the reaction starts before the `beep` instruction then executes the `beep` instruction, then reaches the end of a loop, then reaches the `beep` instruction again from the beginning of this loop, then stops. As a result, the reaction must visit twice the piece of code delimited by the `beep` instruction and the end of the loop, the first time traversing it, the second time stopping somewhere in between. It may stop for two reasons:

- because it reached some `pause` instruction. If so, this `pause` instruction must be conditioned in a way that changes between the first and the second iteration. As a result, the condition must depend on a signal local to the loop. Moreover, its scope is left and instantly reentered in this reaction.
- because of some non-terminating parallel branch. If so, the same parallel subterm has to terminated in the first iteration. Therefore, this parallel subterm is left and instantly reentered in this reaction.

□

In the sequel, we shall focus on instantly reentered signal declarations and parallel subterms. In fact, beeps are only one out of many ways to point at these patterns and reveal their complexity.

5.3 Schizophrenic Programs

Let us now forget the `beep` construct. We say that the program p is *schizophrenic* iff at least one signal declaration or parallel subterm of p is schizophrenic, where:

- *schizophrenic signal declaration:* a signal declaration of p is said to be schizophrenic iff it is instantly reentered in some execution of p.
- *schizophrenic parallel subterm:* a parallel subterm of p is said to be schizophrenic iff it is instantly reentered in some execution of p.

Signal Declarations

Local scoping is a typical feature of imperative languages. Nothing new here. What is specific to Esterel however, is that several (partial) traversals of the same local scope may take place simultaneously.

First, we remark that the number of such traversals can be large. In the following example, in the second instant of execution, three instances of `S` are computed. Using tags,

```
loop
  trap T in
    pause;
    exit T
  ||
    loop
      signal S^n in
        pause
      end
    end
  end
end
```

$$\xrightarrow[I_0]{\emptyset, 1, \{n\}} \quad \cdots \quad \xrightarrow[I_1]{\emptyset, 1, \{n,n,n\}} \quad \cdots$$

The number of instances of a signal is nevertheless statically bounded (syntactically bounded).

Theorem 5.5. *If the declaration of S is enclosed in n nested loops then up to $n+1$ instances of S have to be computed per instant.*

Proof. By structural induction on n, each loop adding at most one instance. □

Lemma 5.6. *The bound can be reached. The number of instances of signals computed per instant for a program of size n may be in the order of n^2.*

Proof. Using a construction similar to that of $(p_n)_{n\in\mathbb{N}}$ in Section 5.1. □

On the one hand, if the signal `S` is declared inside a loop, but is not schizophrenic, then its declaration can be moved outside the loop, as in:

Lemma 5.7. *Whatever p, the following statements are $\rightarrow$bisimilar:*

```
loop                        signal S in
  pause;                      loop
  signal S in       and         pause;
    p                           p
  end                         end
end                         end
```

Proof. Let $\sim$ be the least reflexive relation s.t. for all p and p':

- `loop pause; signal S in` p `end end`
  $\sim$ `signal S in loop pause;` p `end end`
- `nothing; signal S in` p `end; loop pause; signal S in` p `end end`
  $\sim$ `signal S in nothing;` p`; loop pause;` p `end end`
- `signal S in` p' `end; loop pause; signal S in` p `end end`
  $\sim$ `signal S in` p'`; loop pause;` p `end end`

This relation is a $\rightarrow$bisimulation. □

As a result, non-instantly reentered local signals can be interpreted as global signals with restricted visibility, which are very easy to deal with.

On the other hand, instantly reentered local signals, that is to say schizophrenic signals, are much more difficult to handle. To start with,

- As mentioned before, gates and wires in synchronous circuits may only assume a single Boolean value in each clock cycle. Therefore, if the scope of `S` is left and instantly reentered, then several wires must be used to implement the several statuses of `S` simultaneously computed. As a result, detecting schizophrenic signal scopes is required for the translation of Esterel programs into synchronous circuits. Moreover, the better the counting of simultaneous instances is, the smaller the circuit area will be.

- It is highly desirable for embedded code targets, either software or hardware, to rely on *statically allocated* memory spaces, and even further *Single Static Assignment* properties. As soon as every element of the description assumes only one value at each instant, important synthesis, optimization and mapping techniques become available. Again, to achieve this goal, simultaneous instances of a same signal must be identified and carefully disentangled.

In fact, whatever the targeted code representation, the computations of the simultaneous instances of a given signal may be interdependent. In general, they must be computed simultaneously rather than sequentially, as the sequence is a very special construct in Esterel. Let us consider the program:

```
signal S in
  present S else emit O end;
  nothing
end
```

$\xrightarrow[I_0]{\{\mathtt{O}\},0}$ `nothing`

In the first instant of execution, `S` is absent, so that `O` is emitted. This statement is of the form "`signal S in` p`;` q `end`". In a C-like imperative language, the behavior of p would be computed first, independently from that of q. But in Esterel, the computation of the behavior of this statement cannot be sequentialized as suggested by the "`;`" operator, by considering first its upper part only:

```
signal S in
  present S else emit O end
```

in particular deciding whether `O` is emitted or not, then the remaining lower part:

```
  nothing
end
```

as the program could well be the following[2], which does not emit O.

```
signal S in
  present S else emit O end        ∅,0
  emit S                          ────→     nothing
end                                 I0
```

Similarly, simultaneous iterations of the same loop cannot be sequentially computed[3]. For example, the following program emits O in its second reaction, because of the emission of S in iteration $n+1$ and the simultaneous test in iteration n:

```
signal S in
  loop
    emit S;
    pause;                            ∅,1             {O},1
    present S then emit O end        ────→    ...    ────→    ...
  end                                 I0              I1
end
```

In summary, if a reaction spans several (fragments of) iterations of a loop, then they have to be computed simultaneously rather than sequentially. In particular, if several incarnations of a given signal have to be considered in a single instant, then these incarnations have to be computed simultaneously. In hardware, this means several wires for the same signal, and in software, multiple memory cells. Therefore, such patterns have to be identified[4] first, and then require special care from the compiler.

Parallel Subterms

The parallel construct in Esterel is synchronous. In each reaction, a status must be computed for each "active" parallel statement. Indeed, the semantics of pure Esterel specifies that the completion code $m = \max(k, l)$ of a parallel subterm is obtained by combining the completion codes of its branches k and l. If the parallel scope is left and instantly reentered, then this computation potentially takes place several times in a single instant. For reasons similar to those we exposed for local signal declarations, these computations have to take place simultaneously, e.g. require again special care from the compiler.

[2] As explained in Chapter 1, the logical behavioral semantics of Esterel and the logical correctness criterion can be refined with the definition of a constructive semantics and of constructive programs. While the initial program is constructive, the revised one is not. Nevertheless, "forward dependencies" are possible for constructive programs, too. For example,

```
signal S in
  signal T in
    present S then pause else emit A end;
    p
  ||
    present T then pause else emit B end;
    q
  end
end
```

emits A but not B if $p =$ `emit T` and $q =$ `nothing`, emits B but not A if $p =$ `nothing` and $q =$ `emit S`. Both substitutions result in constructive programs.

In other words, there is no way to decide the behaviors of the two tests, without looking first at p and q, although they appear in sequence after these tests. Using such program patterns, one can show that instantly reentered signal and parallel scopes retain their complexity in the framework of the constructive semantics.

[3] Again, forward dependencies across iteration boundaries do occur in constructive programs.

[4] Of course, the alternative is to systematically assume the worse, as in FSM synthesis for instance. Here, we want to do better.

p	Ω_p
`nothing`	$\{0\}$
`pause`	$\{0,1\}$
`emit` S	$\{0\}$
`exit` T_d	$\{d+2\}$
p `||` q	$\{m \in \mathbb{N} s.t.\ \exists k \in \Omega_p, \exists l \in \Omega_q, m = \max(k,l)\}$ i.e. $\max(\Omega_p, \Omega_q)$
`loop` p `end`	$\Omega_p \backslash \{0\}$
`present` S `then` p `else` q `end`	$\Omega_p \cup \Omega_q$
`trap` T `in` p `end`	$\{l \in \mathbb{N}\ s.t.\ \exists k \in \Omega_p, l = \downarrow k\}$ i.e. $\downarrow\Omega_p$
p`;` q	if $0 \in \Omega_p$ then $(\Omega_p \backslash \{0\}) \cup \Omega_q$ else Ω_p
`signal` S `in` p `end`	Ω_p

Figure 5.3: Potential Completion Codes of Chains of Reactions

5.4 Static Analysis

In Chapter 7, we shall further discuss code generation for schizophrenic programs. In the rest of this chapter, we focus on identifying schizophrenic programs, more precisely on locating schizophrenic constructs within schizophrenic programs.

Deciding at compile time whether a program is schizophrenic or not a priori requires to explore all possible execution paths, which is unreasonable for large programs, as mentioned earlier. We need an effective decision procedure amenable to implementation. More precisely, we would like to ensure statically that statements are not schizophrenic. In this section, we shall build a conservative static analysis for this safety property.

The fact that p may terminate or exit and be instantly reentered in $C[p]$ depends on both $C[\ ]$ and p. For example, if p non-instantly terminates with code k:

p is instantly reentered?	$k=0$	$k=2$
`loop trap T in` $[p]$`; pause end end`	*no*	*yes*
`trap T in loop` $[p]$ `end end`	*yes*	*no*

Intuitively, we have to characterize both:

- the possible behaviors of p,
- the possible behaviors of $C[\ ]$,

so that, by combining these two pieces of information, we may decide whether p at position $C[\ ]$ can be instantly reentered in $C[p]$ or not.

Potential Completion Codes

In Chapter 4, in Figure 4.2, we defined function $\Gamma : p \mapsto \Gamma_p$ which provides an overapproximation of the set of possible completion codes of the first reaction of p. Similarly, in Figure 5.3, we define function $\Omega : p \mapsto \Omega_p$ which overapproximates the set of possible completion codes of chains of reactions starting from p. The computation of Ω_p matches that of Γ_p except for `pause` statements for which $\Gamma_{\texttt{pause}} = \{1\}$ whereas $\Omega_{\texttt{pause}} = \{0,1\}$.

Lemma 5.8. *For any statement p, $\Gamma_p \subset \Omega_p$.*

Proof. Structural induction. □

$$
\begin{array}{rcl}
\langle p \rangle & \stackrel{def}{=} & \emptyset \\
C[\texttt{present } S \texttt{ then } \langle p \rangle \texttt{ else } q \texttt{ end}] & \stackrel{def}{=} & C\langle \texttt{present } S \texttt{ then } p \texttt{ else } q \texttt{ end} \rangle \\
C[\texttt{present } S \texttt{ then } p \texttt{ else } \langle q \rangle \texttt{ end}] & \stackrel{def}{=} & C\langle \texttt{present } S \texttt{ then } p \texttt{ else } q \texttt{ end} \rangle \\
C[\texttt{loop } \langle p \rangle \texttt{ end}] & \stackrel{def}{=} & \{0\} \cup C\langle \texttt{loop } p \texttt{ end} \rangle \\
C[\langle p \rangle\texttt{; } q] & \stackrel{def}{=} & \text{if } \Gamma_q \cap C\langle p\texttt{;}q \rangle = \emptyset \text{ then } C\langle p\texttt{;}q \rangle \backslash \{0\} \text{ else } C\langle p\texttt{;}q \rangle \cup \{0\} \\
C[p\texttt{; } \langle q \rangle] & \stackrel{def}{=} & \text{if } 0 \in \Gamma_p \text{ then } C\langle p\texttt{; } q \rangle \text{ else } \emptyset \\
C[\texttt{trap } T \texttt{ in } \langle p \rangle \texttt{ end}] & \stackrel{def}{=} & \{k \in \mathbb{N}, \downarrow k \in C\langle \texttt{trap } T \texttt{ in } p \texttt{ end} \rangle\} \\
C[\texttt{signal } S \texttt{ in } \langle p \rangle \texttt{ end}] & \stackrel{def}{=} & \text{if } C\langle \texttt{signal } S \texttt{ in } p \texttt{ end} \rangle \cap \Omega_p = \emptyset \text{ then } \emptyset \text{ else STOP} \\
C[\langle p \rangle \texttt{ || } q] & \stackrel{def}{=} & \text{if } C\langle p \texttt{ || } q \rangle \cap \Omega_{p \texttt{ || } q} = \emptyset \text{ then } \emptyset \text{ else STOP} \\
C[p \texttt{ || } \langle q \rangle] & \stackrel{def}{=} & \text{if } C\langle p \texttt{ || } q \rangle \cap \Omega_{p \texttt{ || } q} = \emptyset \text{ then } \emptyset \text{ else STOP}
\end{array}
$$

Figure 5.4: Risk

Moreover, potential completion codes are preserved by the reduction:

Lemma 5.9. *For any statements p and p', if $p \to p'$ then $\Omega_{p'} \subset \Omega_p$.*

Proof. By structural induction on p. Let us for instance suppose $p =$ "`loop` q `end`". If $p \to p'$ then by rule (loop), there exists q' such that $q \to q'$ and $p' =$ "q'`; loop` q `end`". $\Omega_p = \Omega_{\texttt{loop } q \texttt{ end}}$.
$\Omega_p = \Omega_q \backslash \{0\}$, by definition of Ω for loops.
$\Omega_{p'} = \Omega_{q'\texttt{; loop } q \texttt{ end}}$.
$\Omega_{p'} \subset (\Omega_{q'} \backslash \{0\}) \cup \Omega_{\texttt{loop } q \texttt{ end}}$, by definition of Ω for sequences.
$\Omega_{p'} \subset (\Omega_{q'} \backslash \{0\}) \cup (\Omega_q \backslash \{0\})$, by definition of Ω for loops.
By induction hypothesis, $\Omega_{q'} \subset \Omega_q$, thus $\Omega_{p'} \subset \Omega_p$. □

As a result,

Theorem 5.10. *If $p \xrightarrow[I_0]{O_0, 1} \ldots \xrightarrow[I_n]{O_n, k} p_n$ for some $n \in \mathbb{N}$ then $k \in \Omega_p$.*

Proof. By recurrence on n, using the above lemmas. □

Risk

In Figure 5.4, we recursively define the partial function *risk* $: C[\ ], p \mapsto \textit{risk}(C[\ ], p)$ which we abbreviate into $C\langle p \rangle$. Intuitively, it tries to decide whether p at position $C[\ ]$ in $C[p]$ – thus the shortcut $C\langle p \rangle$ – may be instantly reentered or not, or stops early with the exception STOP if it thinks that p is contained in a schizophrenic subterm of $C[p]$.

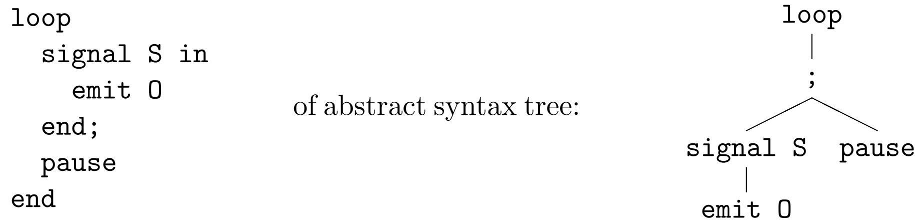
- First, the risk function is designed so that if p at position $C[\ ]$ in $C[p]$ may terminate with completion code k and be instantly reentered, then $k \in C\langle p \rangle$. We say that $C\langle p \rangle$ is the set of *risky* completion codes for p at position $C[\ ]$.

  The definition or the risk function specifies for instance that 0 is a risky completion code for any context having a loop as its innermost construct:

$$\forall C[\ ], \forall p : 0 \in C[\texttt{loop } \langle p \rangle \texttt{ end}]$$

- Second, the computation stops as soon as it encounters a potentially schizophrenic signal declaration or parallel construct.

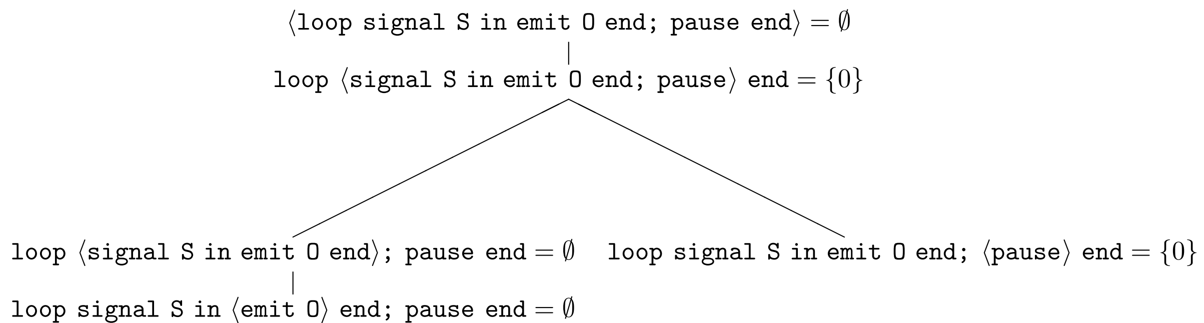
Let us now understand the way the computation of the risk function proceeds, by considering the program p_0:



Each node of the abstract syntax tree represents a possible decomposition "$p_0 = C[q]$" of the program p_0 into a subterm q and a context $C[\,]$. For instance, for the "`signal S`" node:

$$p_0 = \texttt{loop [signal S in emit O end]; pause end}$$

By definition, in order to compute the risk corresponding to a given node of the tree, one has to successively compute the risk of all nodes above it, starting from the root node. The top-down computation of the *risk* function for p_0 is thus the following:



Let us finally consider a few more examples:

- "`loop signal S in` $\langle p \rangle$ `end; pause end`" is empty, whatever p.

  The subterm p in this context cannot be instantly reentered.

- "`loop signal S in emit O end;` $\langle p \rangle$ `end`" is $\{0\}$, whatever p.

  The subterm p may be instantly reentered if terminates normally. It cannot be if it raises an exception.

- "`loop trap T in` $\langle p \rangle$`; pause end end`" is $\{2\}$, whatever p.

  As a result, p must raise exception `T` (completion code 2 means depth 0) to be potentially left and instantly reentered in context "`loop trap T in [ ]; pause end end`" .

- "`trap T in loop` $\langle p \rangle$ `end end`" is $\{0\}$, whatever p.

  In particular, p cannot be instantly reentered in context "`trap T in loop [ ] end end`" if it raises exception `T`. It may if it terminates normally ($k = 0$).

- "`loop signal S in` $\langle$`pause`$\rangle$ `end end`" aborts with exception STOP.

  The local signal declaration is potentially schizophrenic.

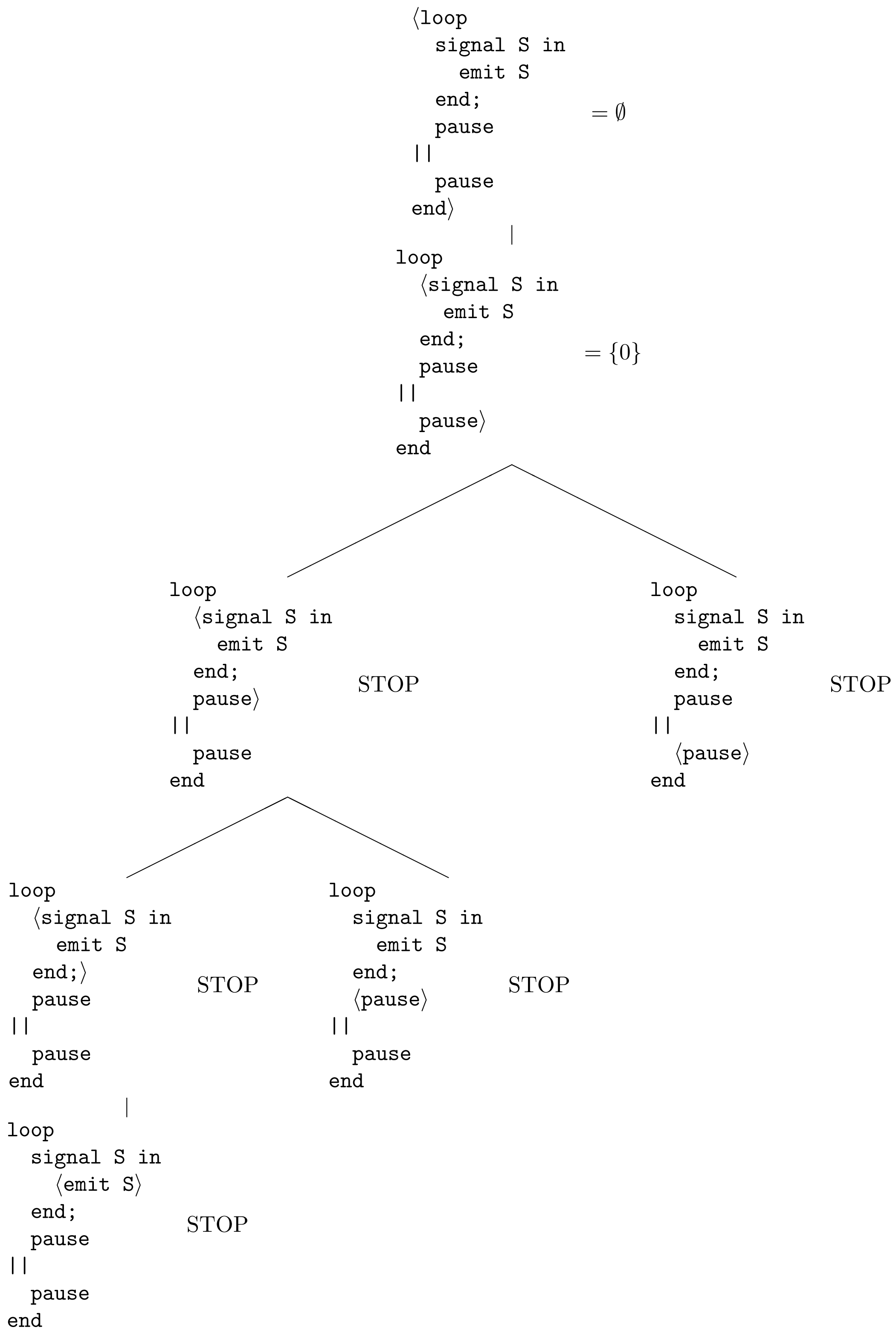


Figure 5.5: A Potentially Schizophrenic Program

- "`loop` $\langle$`emit O`$\rangle$ `|| pause end`" aborts with exception STOP.

  The parallel statement is potentially schizophrenic.

- aborts with exception STOP.

```
loop
  signal S in
    ⟨emit S⟩
  end;
  pause
||
  pause
end
```

  The details of this computation are given in Figure 5.5. Remark the computation does not stop because it concludes that the local signal declaration is potentially schizophrenic, but because it decides that the outer parallel construct is, in an early stage of the computation.

Formally, the following results can be established:

Theorem 5.11. *If the computation of $C\langle p\rangle$ completes normally and $C\langle p\rangle \cap \Omega_p = \emptyset$ then p at position $C[\,]$ cannot be left and instantly reentered in program $C[p]$.*

Corollary 5.12. *For any program p, if the computation of $C\langle q\rangle$ completes normally for every context $C[\,]$ and statement q such that $p = C[q]$ then p is not schizophrenic.*

Proof. In Chapter 7, we shall use this static analysis in order to efficiently rewrite schizophrenic programs into non-schizophrenic programs. Rather than proving now the correctness of the static analysis, we shall then prove the correctness of the optimized preprocessing. □

Decision Procedure

From these results, a decision procedure can be easily derived to ensure that a program p is not schizophrenic. First, the abstract syntax tree of the program is decorated with the values of functions Γ and Ω, simultaneously computed in one recursive traversal of the tree. Then, a second traversal computes the risk $C\langle q\rangle$ for every decomposition $p = C[q]$. This procedure either:

- returns ERROR (during phase 1): there is a potentially instantaneous loop;
- returns STOP (during phase 2): loop safe, but potentially schizophrenic;
- completes normally: loop safe, not schizophrenic.

It extends the decision procedure for the analysis of loop safety described in Chapter 4. It remains linear, under the same assumption of atomic set operations.

In addition, precise locations of potentially schizophrenic subterms can be reported. Formally, we update the definition of the *risk* function using tags as the following:

$$\begin{aligned} C[\texttt{signal}^n\ S\ \texttt{in}\ \langle p\rangle\ \texttt{end}] &\overset{def}{=} \text{if } C\langle \texttt{signal}\ S\ \texttt{in}\ p\ \texttt{end}\rangle \cap \Omega_p = \emptyset \text{ then } \emptyset \text{ else STOP}(n) \\ C[\langle p\rangle \ \texttt{||}^n\ q] &\overset{def}{=} \text{if } C\langle p\ \texttt{||}\ q\rangle \cap \Omega_{p\ \texttt{||}\ q} = \emptyset \text{ then } \emptyset \text{ else STOP}(n) \\ C[p\ \texttt{||}^n\ \langle q\rangle] &\overset{def}{=} \text{if } C\langle p\ \texttt{||}\ q\rangle \cap \Omega_{p\ \texttt{||}\ q} = \emptyset \text{ then } \emptyset \text{ else STOP}(n) \end{aligned}$$

In Chapter 7, we shall use this information to selectively rewrite schizophrenic subterms into non-schizophrenic equivalent subterms.

Examples

Let us consider again the example of Figure 5.2:

```
loop
  signal^n S in
    present S then emit O end;
    pause;
    emit S
  end
end
```

For this program, the decision procedure computes:

$$\begin{aligned}\Omega_{\texttt{signal S in present S then emit O end; pause; emit S end}} &= \{0, 1\} \\ \texttt{loop } \langle\texttt{signal S in present S then emit O end; pause; emit S end}\rangle\texttt{ end} &= \{0\}\end{aligned}$$

These sets have a non-empty intersection, so that the decision procedure returns STOP(n).

The decision procedure is conservative. It always detects schizophrenia, but may fail at establishing non-schizophrenia, as for instance in:

```
loop
  signal^n S in
    present S then emit O end;
    pause;
    emit S
  end;
  present I then pause end;
  present I else pause end;
  end;
end
```

As discussed in Chapter 4, our analysis cannot verify that the following statement cannot be traversed instantly:

```
present I then pause end;
present I else pause end
```

Therefore, it reports the signal declaration as potentially schizophrenic, whereas it is not.

In the experiments of Chapter 8, we shall observe that reported potential schizophrenia problems are rare in practice, thus false positives must be even less frequent.

SUMMARY

We have defined schizophrenic programs and schizophrenic constructs in schizophrenic programs. We have formalized an efficient, provably correct decision procedure that identifies all such programs, by pointing at potentially schizophrenic subterms within potentially schizophrenic programs. In the sequel, we shall say that p is *obviously not schizophrenic* if this decision procedure successfully establishes that p is not schizophrenic.

Chapter 6

Esterel*

We extend the pure Esterel language with a new `gotopause` primitive instruction, which acts as a non-instantaneous jump instruction compatible with Esterel synchronous concurrency. We first describe the syntax and intuitive semantics of the extended language, Esterel*, in Section 6.1. As jumps disregard the program structure, they cannot be easily specified within the logical behavioral semantics of Esterel. In Section 6.2, we formalize a state semantics for Esterel which we prove to be observationally equivalent to the logical behavioral semantics. In Section 6.3, we specify `gotopause` and the state semantics of Esterel*. In Section 6.4, we discuss loop safety and schizophrenia in Esterel*. In Chapter 7, we shall use `gotopause` to deal with schizophrenic programs.

6.1 A New Primitive Instruction: gotopause

The syntax of Esterel* is described in Figure 6.1. It is obtained from that of Esterel (Figure 2.1), by (*i*) labeling `pause` instructions, and (*ii*) introducing a new `gotopause` construct. Both instructions are labeled with integers. We want `gotopause` to behave as follows:

- When the control reaches a "`gotopause` *label*" instruction, it stops for the current instant, as if it had reached a regular `pause` instruction.
- When the execution is resumed in the following instant however, instead of restarting from the "`gotopause` *label*" location, it restart from the corresponding "*label*`:pause`" location.

For example, the execution of "`gotopause 1; emit S; 1:pause`" should not emit `S`.

p, q	::=	`nothing`	
		label`:pause`	`pause` instructions are now labeled.
		`gotopause` *label*	`gotopause` is a new construct.
		`p; q`	
		`p || q`	
		`[p]`	
		`loop` p `end`	
		`signal` S `in` p `end`	
		`emit` S	
		`present` S `then` p `else` q `end`	
		`trap` T `in` p `end`	
		`exit` T	

Figure 6.1: Primitive Pure Esterel* Constructs

6.2 Esterel State Semantics

In this section, we consider Esterel programs with labeled `pause` instructions, that is to say Esterel* programs without `gotopause` instructions. Function *unlabel* removes the labels of `pause` instructions, recovering a standard Esterel program from an Esterel* program. We call $\mathcal{L}(p)$ the set of labels of p. For example,

$$\begin{aligned} \mathit{unlabel}(\texttt{1:pause; emit S; 2:pause; 1:pause}) &= \texttt{pause; emit S; pause; pause} \\ \mathcal{L}(\texttt{1:pause; emit S; 2:pause; 1:pause}) &= \{1,2\} \end{aligned}$$

We do not require labels to be unique yet.

Logical Behavioral Semantics with Labels

In Figure 6.2, we formalize a *logical behavioral semantics with labels* for this subset of Esterel* by replacing the residual p' of the logical behavioral semantics with a set of labels L:

$$p \xmapsto[E]{E',k} L$$

We denote reactions of this semantics with the transition symbol "$\mapsto$".

The set L collects the labels of the *active* `pause` instructions of the statement, that is to say the `pause` instructions that *retain* the control at the end of the reaction. For example, if `S` is present, the reaction of "`present S then 1:pause else 2:pause end`" produces the set $\{1\}$. In particular, for the (parallel) rule, the computed set of labels is:

$$\gamma_1^m(L \cup L') = \begin{cases} L \cup L' & \text{if } m = 1 \\ \emptyset & \text{if } m \neq 1 \end{cases} \qquad \begin{array}{l} \text{where } m = \max(k,l),\ k \text{ and } l \text{ being the} \\ \text{completion codes of the parallel branches} \end{array}$$

Lemma 6.1. *If* $p \xmapsto[I]{O,k} L$ *then* $k \neq 1 \Leftrightarrow L = \emptyset$.

Proof. Structural induction. □

Corollary 6.2. *In rule* (sequence-0)*, the set of labels* L *corresponding to the reaction of* p *of completion code* 0 *is always empty.*

Hence, no two halves of a sequence may be simultaneously active.

Lemma 6.3. $\forall p, \forall I, \forall O, \forall k : \left[\exists p' : \mathit{unlabel}(p) \xrightarrow[I]{O,k} p' \Leftrightarrow \exists L : p \xmapsto[I]{O,k} L\right]$.

Proof. Structural induction. □

Intuitively, provided that the labeling is not ambiguous, it should be possible to reconstruct p' from L and p, and precisely relate these two semantics. We formalize this matching below, with the definition of states and the expansion of valid states.

States

We say that a statement is *well labeled* iff the labels of its `pause` instructions are pairwise distinct. From the combination of the well-labeled statement p and the set of labels $L \subset \mathcal{L}(p)$, we build the *state* $\lceil p|L \rfloor$. We say that a `pause` statement of label l is *active* in $\lceil p|L \rfloor$ iff $l \in L$.

$$\texttt{nothing} \xmapsto[I]{\emptyset,0} \emptyset \tag{nothing}$$

$$l\texttt{:pause} \xmapsto[I]{\emptyset,1} \{l\} \tag{pause}$$

$$\texttt{exit}\ T_d \xmapsto[I]{\emptyset,d+2} \emptyset \tag{exit}$$

$$\texttt{emit}\ S \xmapsto[I]{\{S\},0} \emptyset \tag{emit}$$

$$\frac{p \xmapsto[I]{O,k} L \qquad k \neq 0}{\texttt{loop}\ p\ \texttt{end} \xmapsto[I]{O,k} L} \tag{loop}$$

$$\frac{p \xmapsto[I]{O,k} L \qquad q \xmapsto[I]{O',l} L' \qquad m = \max(k,l)}{p\ \texttt{||}\ q \xmapsto[I]{O \cup O',m} \gamma_1^m(L \cup L')} \tag{parallel}$$

$$\frac{S \in I \qquad p \xmapsto[I]{O,k} L}{\texttt{present}\ S\ \texttt{then}\ p\ \texttt{else}\ q\ \texttt{end} \xmapsto[I]{O,k} L} \tag{present+}$$

$$\frac{S \notin I \qquad q \xmapsto[I]{O,k} L}{\texttt{present}\ S\ \texttt{then}\ p\ \texttt{else}\ q\ \texttt{end} \xmapsto[I]{O,k} L} \tag{present−}$$

$$\frac{p \xmapsto[I]{O,k} L}{\texttt{trap}\ T\ \texttt{in}\ p\ \texttt{end} \xmapsto[I]{O,\downarrow k} L} \tag{trap}$$

$$\frac{p \xmapsto[I]{O,0} L \qquad q \xmapsto[I]{O',k} L'}{p\texttt{;}\ q \xmapsto[I]{O \cup O',k} L \cup L'} \tag{sequence-0}$$

$$\frac{p \xmapsto[I]{O,k} L \qquad k \neq 0}{p\texttt{;}\ q \xmapsto[I]{O,k} L} \tag{sequence-k}$$

$$\frac{p \xmapsto[I \cup \{S\}]{O,k} L \qquad S \in O}{\texttt{signal}\ S\ \texttt{in}\ p\ \texttt{end} \xmapsto[I]{O \setminus \{S\},k} L} \tag{signal+}$$

$$\frac{p \xmapsto[I \setminus \{S\}]{O,k} L \qquad S \notin O}{\texttt{signal}\ S\ \texttt{in}\ p\ \texttt{end} \xmapsto[I]{O,k} L} \tag{signal−}$$

Figure 6.2: Logical Behavioral Semantics with Labels

We shall use states to represents possible points[1] in the execution of a program. For example, the state:

$$\lceil \texttt{present S then 1:pause else 2:pause end} | \{1\} \rfloor$$

specifies that the execution of:

```
present S then 1:pause else 2:pause end
```

has to be resumed from the `pause` instruction of label 1.

In particular, the state $\lceil p|\emptyset \rfloor$ means that the execution of p is over. So, we call it the *inactive state* of p. Reciprocally, a state with at least one active `pause` is an *active state*. We also define an extra state, called *initial state*, and noted $\lceil p|* \rfloor$, which tells that the execution of p has not started yet. States for Esterel where first introduced in [Mig94].

Intuitively, not all states are *reachable*. For example no execution of "`1:pause; 2:pause`" may reach the state $\lceil \texttt{1:pause; 2:pause} | \{1;2\} \rfloor$, since no reaction can activate two `pause` statements in a sequence. We formalize this idea with the definition of valid states.

Valid States

We say that the subterms q and r of respective tags m and n of the well-tagged statement p are *exclusive* in p, and write "$q\#r$", iff there exists three contexts $C[\,]$, $C_q[\,]$, and $C_r[\,]$ such that one of the following holds:

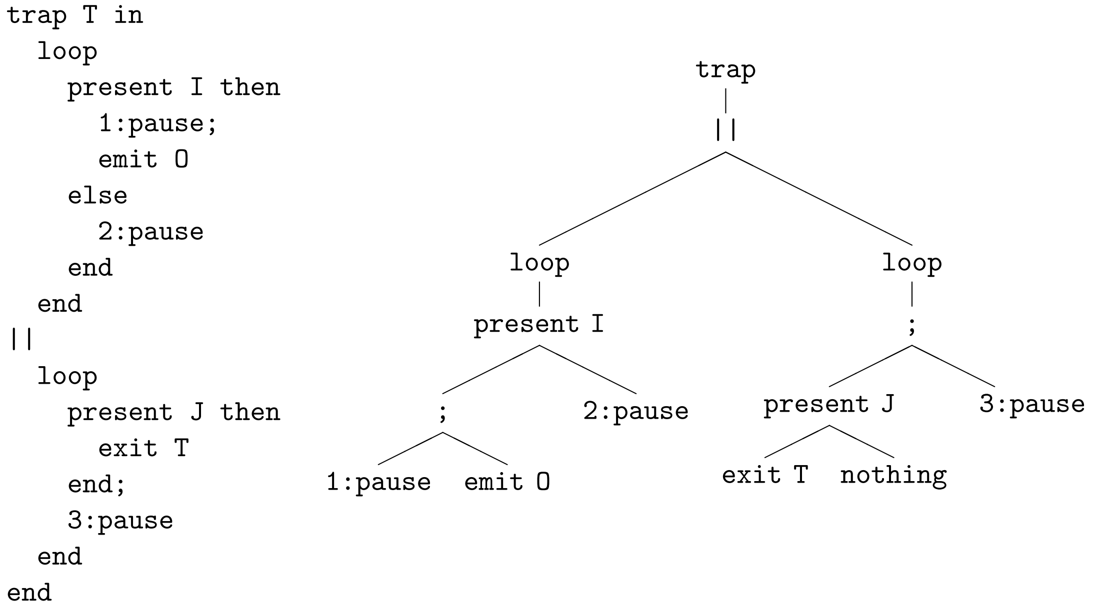
- $p = C[C_q[q^n];\ C_r[r^m]]$
- $p = C[C_r[r^n];\ C_q[q^m]]$

} q and r are composed in sequence in p

- $p = C[\texttt{present}\ S\ \texttt{then}\ C_q[q^m]\ \texttt{else}\ C_r[r^n]\ \texttt{end}]$
- $p = C[\texttt{present}\ S\ \texttt{then}\ C_r[r^n]\ \texttt{else}\ C_q[q^m]\ \texttt{end}]$

} q and r are connected by a `present` construct in p

For example, in "p`; [`q `||` r`]`", p and q are exclusive, p and r are exclusive, q and r are not exclusive. We say that q and r are *compatible*, which we note note "$q//r$". As another example, let us consider the well-labeled program:



[1] States represent starting and ending points of reactions, that is to say macro-steps. However, micro-steps within a reaction cannot be represented with such states.

initial state :	$\epsilon\lceil p\|*\rfloor$	$\stackrel{def}{=}$	p	
inactive state :	$\epsilon\lceil p\|\emptyset\rfloor$	$\stackrel{def}{=}$	`nothing`	
active states :	$\epsilon\lceil l$`:pause`$\|\{l\}\rfloor$	$\stackrel{def}{=}$	`nothing`	
$(L \neq \emptyset)$	$\epsilon\lceil p$`;` $q\|L\rfloor$	$\stackrel{def}{=}$	$\epsilon\lceil p\|L\rfloor$`;` q	if $L \subset \mathcal{L}(p)$
	$\epsilon\lceil p$`;` $q\|L\rfloor$	$\stackrel{def}{=}$	$\epsilon\lceil q\|L\rfloor$	if $L \subset \mathcal{L}(q)$
	$\epsilon\lceil$`present` S `then` p `else` q `end`$\|L\rfloor$	$\stackrel{def}{=}$	$\epsilon\lceil p\|L\rfloor$	if $L \subset \mathcal{L}(p)$
	$\epsilon\lceil$`present` S `then` p `else` q `end`$\|L\rfloor$	$\stackrel{def}{=}$	$\epsilon\lceil q\|L\rfloor$	if $L \subset \mathcal{L}(q)$
	$\epsilon\lceil$`trap` T `in` p `end`$\|L\rfloor$	$\stackrel{def}{=}$	`trap` T `in` $\epsilon\lceil p\|L\rfloor$ `end`	
	$\epsilon\lceil p$ `||` $q\|L\rfloor$	$\stackrel{def}{=}$	$\epsilon\lceil p\|L \cap \mathcal{L}(p)\rfloor$ `||` $\epsilon\lceil q\|L \cap \mathcal{L}(q)\rfloor$	
	$\epsilon\lceil$`loop` p `end`$\|L\rfloor$	$\stackrel{def}{=}$	$\epsilon\lceil p\|L\rfloor$`;` `loop` p `end`	
	$\epsilon\lceil$`signal` S `in` p `end`$\|L\rfloor$	$\stackrel{def}{=}$	`signal` S `in` $\epsilon\lceil p\|L\rfloor$ `end`	

Figure 6.3: State Expansion

In this example, the following relations hold for `pause` instructions:

- `1:pause` and `2:pause` are exclusive: they are connected by a "`present`" node,
- `1:pause` and `3:pause` are compatible: they are connected by a "`||`" node,
- `2:pause` and `3:pause` are compatible: they are connected by a "`||`" node.

We say that a state of p is *valid* iff it is the initial state $\lceil p|*\rfloor$ or some state $\lceil p|L\rfloor$ such that active `pause` statements are pairwise compatible. In other words, in a valid state, no two `pause` statements are active in both parts of a sequence or both branches of a `present` statement. If p_0 is the last program example we considered, then the state $\lceil p_0|\{1,3\}\rfloor$ is valid, whereas the state $\lceil p_0|\{1,2\}\rfloor$ is not.

Invalid states are states that cannot be reached in the execution of the program[2].

Lemma 6.4. *If p is well labeled and $p \xmapsto[I]{O,k} L$ then $\lceil p|L\rfloor$ is valid.*

Proof. In any reaction of p, a sequence "q`;` r" contained in p may be:

- not reduced. Then, no `pause` instruction of "q`;` r" is active.
- reduced by rule (sequence-0). Then, thanks to Corollary 6.2, no `pause` instruction of q is active.
- reduced by rule (sequence-k). Then, no `pause` instruction of r is active.

Similarly, `pause` statements cannot be simultaneously active in both the `then` and the `else` branch of a `present` statement. □

State Expansion

In Figure 6.3, we recursively define a *state expansion function* "ϵ". It derives a statement from a valid state. Remark that the rule for the empty set L has priority over the other rules. For example, $\epsilon\lceil$`trap` T `in` p `end`$|\emptyset\rfloor$ is `nothing` rather than "`trap` T `in nothing end`".

[2]The set of valid states contains the set of reachable states. But there are still unreachable valid states, for example: $\lceil$`1:pause || 2:pause`$|\{1\}\rfloor$ is both valid and unreachable in the execution of "`1:pause || 2:pause`". This is not an issue.

Basically, this ϵ function expands a state into a statement of "equivalent" semantics. For example, if `pause` instructions are active in the left component p of the sequence "p`;` q", then the execution of the sequence has to be continued by the end of this left component, which is exactly the expansion of the state of p, followed by the right component q of the sequence. Therefore, if $L \subset \mathcal{L}(p)$ then "$\epsilon\lceil p\texttt{;}\ q|L\rfloor$" is defined as "$\epsilon\lceil p|L\rfloor\texttt{;}\ q$".

The expansion retains labels. We observe that even if $\lceil p|L\rfloor$ is a valid state of the well-labeled statement p, the labeled statement $\epsilon\lceil p|L\rfloor$ is not necessarily well labeled, as loop unfolding may occur. For example,

$$\epsilon\lceil\texttt{loop 1:pause; 2:pause end}|\{1\}\rfloor = \texttt{nothing; 2:pause; loop 1:pause; 2:pause end}$$

This is not an issue, as we shall not build states out of such statements.

Lemma 6.5. *If $\lceil p|L\rfloor$ is valid, and k and l are the labels of two compatible* `pause` *statements of $\epsilon\lceil p|L\rfloor$, then k and l are the labels of two compatible* `pause` *statements of p.*

Proof. Structural induction. □

In other words, the expansion maintains exclusive `pause` statements exclusive (even if replicated). Validity is thus a stable property:

Theorem 6.6. *If $\lceil p|L\rfloor$ is valid and $\epsilon\lceil p|L\rfloor \overset{O,k}{\underset{I}{\longmapsto}} L'$ then $\lceil p|L'\rfloor$ is valid.*

Proof. Remark this lemma is not a corollary of Lemma 6.4, since $\epsilon\lceil p|L\rceil$ is not necessarily well labeled. Nevertheless, if $k \in L'$ and $l \in L'$ then there must be two compatible `pause` statements in $\epsilon\lceil p|L\rceil$ of respective labels k and l, similarly to the proof of Lemma 6.4. By Lemma 6.5, there must be two compatible `pause` statements in p of respective labels k and l. As a result, $\lceil p|L'\rfloor$ is valid. □

Thanks to this state expansion function, we can now express the fact that the logical behavioral semantics and the logical behavioral semantics with labels define the same reactions:

Theorem 6.7. *For every well-labeled statement p,*

- *If $p \overset{O,k}{\underset{I}{\longmapsto}} L$ then unlabel(p) $\overset{O,k}{\underset{I}{\longrightarrow}}$ unlabel($\epsilon\lceil p|L\rfloor$).*
- *If unlabel(p) $\overset{O,k}{\underset{I}{\longrightarrow}} p'$ then there exists L such that $p \overset{O,k}{\underset{I}{\longmapsto}} L$ and unlabel($\epsilon\lceil p|L\rfloor$) $= p'$.*

Proof. Structural induction. □

This proves that p' can be obtained from $\lceil p|L\rfloor$ and vice versa. The result of a reaction is equivalently characterized by either the residual p' or the set of active labels L we have just introduced. This is the key that enables the definition of a state semantics for Esterel.

Logical State Semantics

We define a logical state semantics ($\diamond\!\!\longrightarrow$) for well-labeled Esterel* programs without `gotopause` instructions as follows:

$$\lceil p|L\rfloor \overset{O,k}{\underset{I}{\diamond\!\!\longrightarrow}} \lceil p|L'\rfloor \quad \text{iff} \quad \epsilon\lceil p|L\rfloor \overset{O,k}{\underset{I}{\longmapsto}} L'$$

One reaction of the well-labeled statement p in the valid state $\lceil p|L\rfloor$ produces the valid state $\lceil p|L'\rfloor$ iff L' is the set of active labels computed by the logical behavioral semantics with labels for the statement $\epsilon\lceil p|L\rfloor$.

Theorem 6.8. *The logical behavioral and logical state semantics are observationally equivalent.*

Proof. Let "$\sim$" be the relation such that the unlabeled statement a is in relation with the valid state b, that is to say $a \sim b$, iff:

- either there exists p well labeled s.t. $a = \mathit{unlabel}(p)$ and $b = \lceil p|*\rfloor$,
- or there exists p and L s.t. $\lceil p|L\rfloor$ is valid, $a = \mathit{unlabel}(\epsilon\lceil p|L\rfloor)$, and $b = \lceil p|L\rfloor$.

This relation is total and surjective:

- if a is an unlabeled statement then it can be labeled with pairwise distinct labels, producing the well-labeled statement p, so that $a \sim \lceil p|*\rfloor$.
- if b is a valid state then $\mathit{unlabel}(\epsilon(b)) \sim b$.

Let a and b be such that $a \sim b$. Let us first consider the case $a = \mathit{unlabel}(p)$ and $b = \lceil p|*\rfloor$, where p is a well-labeled statement:

- If $a \xrightarrow[I]{O,k} a'$ then:
  - $\mathit{unlabel}(\epsilon\lceil p|*\rfloor) \xrightarrow[I]{O,k} a'$
  - there exists L such that $\begin{cases} \epsilon\lceil p|*\rfloor \xmapsto[I]{O,k} L \\ a' = \mathit{unlabel}(\epsilon\lceil p|L\rfloor) \end{cases}$ using Theorem 6.7
  - there exists L such that $\begin{cases} \lceil p|*\rfloor \diamond\!\!\xrightarrow[I]{O,k} \lceil p|L\rfloor \\ a' = \mathit{unlabel}(\epsilon\lceil p|L\rfloor) \end{cases}$ by definition of "$\diamond\!\!\rightarrow$"
  - there exists b' such that $b \diamond\!\!\xrightarrow[I]{O,k} b'$ and $a' \sim b'$.
- If $b \diamond\!\!\xrightarrow[I]{O,k} b'$ then:
  - there exists L such that $\begin{cases} \lceil p|*\rfloor \diamond\!\!\xrightarrow[I]{O,k} \lceil p|L\rfloor \\ b' = \lceil p|L\rfloor \end{cases}$
  - there exists L such that $\begin{cases} \epsilon\lceil p|*\rfloor \xmapsto[I]{O,k} L \\ b' = \lceil p|L\rfloor \end{cases}$ by definition of "$\diamond\!\!\rightarrow$"
  - $\mathit{unlabel}(\epsilon\lceil p|*\rfloor) \xrightarrow[I]{O,k} \mathit{unlabel}(\epsilon(b'))$ by Theorem 6.7,
  - there exists a' such that $a \xrightarrow[I]{O,k} a'$ and $a' \sim b'$.

Let us now consider the case $a = \mathit{unlabel}(\epsilon\lceil p|L\rfloor)$ and $b = \lceil p|L\rfloor$, where $\lceil p|L\rfloor$ is a valid state of the well-labeled statement p. As observed before, $\epsilon\lceil p|L\rfloor$ is not necessarily well labeled. Theorem 6.7 however requires well labeling. For this reason, we define $\overline{\epsilon\lceil p|L\rfloor}$ by replacing all occurrences of labels in $\epsilon\lceil p|L\rfloor$ by pairwise distinct labels. Reciprocally, we define $\underline{\mathit{underline}}$ as the inverse labeling transformation, that is to say the function that associates to a new label its old value. While it makes sense to apply $\overline{\mathit{overline}}$ only to a subterm of $\epsilon\lceil p|L\rfloor$, it is possible to apply $\underline{\mathit{underline}}$ to any term built from $\overline{\epsilon\lceil p|L\rfloor}$, as well as any subset of the labels of $\overline{\epsilon\lceil p|L\rfloor}$.

Lemma 6.9. *If* $\left\lceil \left(\overline{\epsilon\lceil p|L\rfloor}\right) |L' \right\rfloor$ *is valid then* $\underline{\epsilon \left\lceil \left(\overline{\epsilon\lceil p|L\rfloor}\right) |L' \right\rfloor} = \epsilon\lceil p|(\underline{L'})\rfloor$.

Proof. Structural induction. □

Thanks to this lemma, we now establish that:

- If $a \xrightarrow[I]{O,k} a'$ then:
  - $unlabel(\epsilon\lceil p|L\rfloor) \xrightarrow[I]{O,k} a'$
  - $unlabel\left(\overline{\epsilon\lceil p|L\rfloor}\right) \xrightarrow[I]{O,k} a'$ by relabeling
  - there exists L' s.t. $\begin{cases} \overline{\epsilon\lceil p|L\rfloor} \overset{O,k}{\underset{I}{\longmapsto}} L' \\ a' = unlabel\left(\epsilon\left\lceil\left(\overline{\epsilon\lceil p|L\rfloor}\right)|L'\right\rfloor\right) \end{cases}$ by Theorem 6.7
  - there exists L' s.t. $\begin{cases} \overline{\epsilon\lceil p|L\rfloor} \overset{O,k}{\underset{I}{\longmapsto}} L' \\ a' = unlabel(\epsilon\lceil p|(\underline{L'})\rfloor) \end{cases}$ by Lemma 6.9
  - there exists L' s.t. $\begin{cases} \epsilon\lceil p|L\rfloor \overset{O,k}{\underset{I}{\longmapsto}} L' \\ a' = unlabel(\epsilon\lceil p|L'\rfloor) \end{cases}$ by reverting to the old labeling
  - there exists b' such that $b \overset{O,k}{\underset{I}{\diamond\!\!\longrightarrow}} b'$ and $a' \sim b'$.
- If $b \overset{O,k}{\underset{I}{\diamond\!\!\longrightarrow}} b'$ then:
  - there exists L' s.t. $\begin{cases} \epsilon\lceil p|L\rfloor \overset{O,k}{\underset{I}{\longmapsto}} L' \\ b' = \lceil p|L'\rfloor \end{cases}$
  - there exists L' s.t. $\begin{cases} \overline{\epsilon\lceil p|L\rfloor} \overset{O,k}{\underset{I}{\longmapsto}} L' \\ b' = \lceil p|(\underline{L'})\rfloor \end{cases}$ by relabeling
  - there exists L' s.t. $\begin{cases} unlabel(\overline{\epsilon\lceil p|L\rfloor}) \xrightarrow[I]{O,k} unlabel\left(\epsilon\left\lceil\left(\overline{\epsilon\lceil p|L\rfloor}\right)|L'\right\rfloor\right) \\ b' = \lceil p|(\underline{L'})\rfloor \end{cases}$ by Theorem 6.7
  - $unlabel(\overline{\epsilon\lceil p|L\rfloor}) \xrightarrow[I]{O,k} unlabel(\epsilon(b'))$ by Lemma 6.9
  - $unlabel(\epsilon\lceil p|L\rfloor) \xrightarrow[I]{O,k} unlabel(\epsilon(b'))$ by reverting to the old labeling
  - there exists a' such that $a \xrightarrow[I]{O,k} a'$ and $a' \sim b'$.

As a result, the "$\sim$" relation is a bisimulation between "$\rightarrow$" and "$\diamond\!\!\rightarrow$". □

In particular, this confirms that pure Esterel programs are *finite state*.

Deterministic State Semantics

Similarly, we could derive a deterministic state semantics from the deterministic semantics of Chapter 3, using rules (signal++) and (signal−−) instead of (signal+) and (signal−), and establish observational equivalence as well.

6.3 Esterel* State Semantics

We now consider Esterel* programs with `gotopause` instructions.

Logical Behavioral Semantics with Labels

Without making any assumption about labels yet, we can extend the logical behavioral semantics with labels defined in previous section to cope with `gotopause` statements using rule:

$$\texttt{gotopause}\ l \xmapsto[I]{\emptyset, 1} \{l\} \qquad \text{(gotopause)}$$

It specifies, that "`gotopause` l" activates the `pause` instruction of label l. For example,

$$\texttt{gotopause 1; emit S; 1:pause} \xmapsto[I]{\emptyset, 1} \{1\}$$

Well-labeled Statements

Then, we say that an Esterel* program is *well labeled* iff:

- The `pause` instructions are labeled with pairwise distinct labels.
- The set of labels of the `gotopause` instructions is included in the set of labels of the `pause` instructions, that is to say every `gotopause` instruction has a target `pause` instruction.

However, we do not suppose that `gotopause` statements have pairwise distinct labels. Just as simultaneous emissions of the same signal are possible in Esterel, simultaneous jumps to the same target make sense in Esterel*.

Lemma 6.10. *If p is well labeled and $p \xmapsto[I]{O, k} L$ then $L \subset \mathcal{L}(p)$.*

Proof. Structural induction. □

Well-formed Statements

Unfortunately, arbitrary simultaneous jumps are not possible. The reaction of a well-labeled Esterel* program may produce an invalid state, as in:

$$\texttt{[gotopause 1 || 2:pause]; 1:pause} \xmapsto[\emptyset]{\emptyset, 1} \{1, 2\}$$

The state expansion of $\lceil$`[gotopause 1 || 2:pause]; 1:pause`$|\{1, 2\}\rfloor$ is undefined since two exclusive `pause` instructions are active here. Such a state does not make sense. Therefore, the program "`[gotopause 1 || 2:pause]; 1:pause`" cannot be considered to be correct, and should be rejected. This is dealt with through the definition (and compile time analysis) of well-formedness, which ensures that `gotopause` is compatible with Esterel concurrency.

Intuitively, if "k`:pause`" and "l`:pause`" are exclusive, then we shall ensure that they are never activated simultaneously, by requiring the corresponding "activators" to be exclusive. Formally, we say that a well-labeled program p is *well formed* iff:

$$\forall k, \forall l : k\!: \texttt{pause} \# l\!: \texttt{pause} \Rightarrow \begin{cases} \texttt{gotopause}\ k \# \texttt{gotopause}\ l \\ \texttt{gotopause}\ k \# l\!: \texttt{pause} \\ \quad k\!: \texttt{pause} \# \texttt{gotopause}\ l \end{cases}$$

In the above example, "`gotopause 1`" and "`2:pause`" are compatible (composed in parallel), while "`1:pause`" and "`2:pause`" are exclusive, so this program is not well formed.

Similarly, the following programs are not well formed:

- `[gotopause 1 || gotopause 2]; 1:pause; 2:pause`
- `[gotopause 1 || gotopause 2]; present S then 1:pause else 2:pause end`
- `gotopause 2 || 1:pause; 2:pause`
- `present S then 1:pause || gotopause 2 else 2:pause end`

On the other hand, any Esterel program with well-labeled `pause` instructions is a well-formed Esterel* program.

We can now recover the stability of validity as follows:

Theorem 6.11. *If p is well formed and $\lceil p|L \rfloor$ valid and $\epsilon\lceil p|L \rfloor \xmapsto[I]{O,k} L'$ then $\lceil p|L' \rfloor$ is valid.*

Proof. Similar to Theorem 6.6, thanks to well-formedness. □

Well-formedness is a syntactic condition. It can be checked easily while building the abstract syntax tree of a program. In fact, the required "compatible" and "exclusive" relations are already computed for optimization purposes. Moreover, in the sequel, we shall generate Esterel* programs that are well formed by construction and need no analysis.

The `gotopause` construct is compatible with concurrency in the sense that, for instance, well-formedness lets us compose any set of well-formed programs in parallel, provided that their respective sets of labels are disjoint.

Logical State Semantics of Esterel*

We define the logical state semantics ($\diamond\!\!\rightarrow$) of well-formed Esterel* programs just as we did in the absence of `gotopause` statements:

$$\lceil p|L \rfloor \xrightarrow[I]{O,k}_{\diamond} \lceil p|L' \rfloor \quad \text{iff} \quad \epsilon\lceil p|L \rfloor \xmapsto[I]{O,k} L'$$

One reaction of the well-formed statement p in the valid state $\lceil p|L \rfloor$ produces the valid state $\lceil p|L' \rfloor$ iff L' is the set of active labels computed for the statement $\epsilon\lceil p|L \rfloor$, by the logical behavioral semantics with labels including the (gotopause) rule.

The logical state semantics of Esterel* restricted to Esterel programs is exactly the logical state semantics of Section 6.2, which we have shown to be observationally equivalent to the logical behavioral semantics. Therefore, this semantics truly defines Esterel* as an extension of the original pure Esterel language.

Deterministic State Semantics of Esterel*

By extending the deterministic state semantics of Esterel rather than the logical state semantics of Esterel, we would define a deterministic state semantics for well-formed Esterel* programs.

6.4 Loop Safety and Schizophrenia

Because `gotopause` constructs enable non-instantaneous jumps only, they cannot by responsible for instantly diverging behaviors or instantly reentered statements. In other words, loop safety and schizophrenia remain in Esterel* exactly the issues they were in Esterel.

Safe Programs

In order to extend the decision procedure of Chapter 4 and characterize safe Esterel* programs, we define:

$$\Gamma_{\texttt{gotopause}\ l} = \Gamma_{l:\texttt{pause}} = \{1\}$$

We say that p is a *safe* Esterel* program iff the computation of Γ_q completes normally for every $C[\ ]$ and q such that $p = C[q]$.

Lemma 6.12. *If* $\lceil p|*\rfloor \diamond\!\xrightarrow[I]{O,k} \lceil p|L\rfloor$ *then* $k \in \Gamma_p$.

Proof. Similar to the proof of Chapter 4 for Esterel programs. □

Defining loop safe Esterel* programs and proving that safe Esterel* programs are loop safe is straightforward. We shall not go into details.

Schizophrenia

In order to define schizophrenic Esterel programs, we instrumented the logical behavioral semantics of Esterel with tags, and kept track of tags in the execution. We may do the same for Esterel* programs:

- We tag all constructs in Esterel* programs and states, as in: $\lceil \texttt{loop}^1\ 1 : \texttt{pause}^2\ \texttt{end}|\{1\}\rfloor$.
- We preserve tags in the state expansion function ϵ (introducing fresh tags in loop expansions):
$$\epsilon\lceil \texttt{loop}^1\ 1 : \texttt{pause}^2\ \texttt{end}|\{1\}\rfloor = \texttt{nothing}^2;^3\ \texttt{loop}^1\ 1 : \texttt{pause}^2\ \texttt{end}$$
- We compute the multiset of tags of a reaction for the logical behavioral semantics with labels of Esterel*:
$$\texttt{nothing}^2;^3\ \texttt{loop}^1\ 1 : \texttt{pause}^2\ \texttt{end} \xmapsto[I]{\emptyset,1,\{1,2,2,3\}} \{1\}$$
so that:
$$\lceil \texttt{loop}^1\ 1 : \texttt{pause}^2\ \texttt{end}|\{1\}\rfloor \diamond\!\xrightarrow[I]{\emptyset,1,\{1,2,2,3\}} \lceil \texttt{loop}^1\ 1 : \texttt{pause}^2\ \texttt{end}|\{1\}\rfloor$$
The precise values of fresh tags are irrelevant.
- We say that the Esterel* program p is *well tagged* iff all its constructs are tagged with pairwise distinct tags.
- We say that a well-tagged Esterel* program p is *schizophrenic* iff there exists a valid state $\lceil p|L\rfloor$ of p and a reaction $\lceil p|L\rfloor \diamond\!\xrightarrow[I]{O,k,M} \lceil p|L'\rfloor$ such that the tag of a parallel statement or signal declaration is repeated in the multiset M of the reaction.

For lack of time, we shall not precisely relate this definition of schizophrenia for Esterel* programs to the initial definition of schizophrenia for Esterel programs, but the connection is obvious.

Extending the decision procedure of Chapter 5 for the detection schizophrenic constructs in Esterel* programs, however, is not so easy. Nevertheless, the existing decision procedure will be enough to efficiently deal with schizophrenia in Esterel programs (but not in Esterel* programs), which is our primary concern, and the focus of this document.

SUMMARY

We have extended the Esterel language with a new `gotopause` primitive instruction, and formalized the semantics of the extended language, which we name Esterel*. More precisely, we have defined the class of well-formed Esterel* programs as a superset of the set of Esterel programs, and specified the semantics of well-formed programs so that it matches Esterel semantics for Esterel programs. Accordingly, we have have revised the definition of safe programs of Chapter 4, and schizophrenic programs of Chapter 5 to cope with Esterel* programs.

Chapter 7

Reincarnation

The programming style advocated by Esterel – local declarations and concurrency plus imperative loops – naturally leads to schizophrenic specifications. In [Mig94], Mignard proposed to automatically rewrite schizophrenic programs into semantically equivalent non-schizophrenic programs, thus making subsequent analysis, code generation, or optimization steps easier. In this chapter, we want to further investigate this approach: curing schizophrenia by a preprocessing. To start with, we analyze Mignard's rewriting technique in Section 7.1. Then, we describe how loops can be unfolded using the `gotopause` construct of Esterel* in Section 7.2. We make this unfolding efficient in Section 7.3. Following the choice of Chapter 5, we shall focus on the logical behavioral/state semantics. But the results for deterministic semantics would be exactly the same.

7.1 Exponential Reincarnation

Mignard's method consists in recursively duplicating loop bodies in a systematic fashion, that is to say recursively rewriting "`loop` p `end`" into "`loop` p`;` p `end`". Formally, this means defining function *dup* as the following:

$$\begin{aligned}
dup(\texttt{nothing}) &\stackrel{def}{=} \texttt{nothing} \\
dup(\texttt{pause}) &\stackrel{def}{=} \texttt{pause} \\
dup(p\texttt{;}\ q) &\stackrel{def}{=} dup(p)\texttt{;}\ dup(q) \\
dup(p\ \texttt{||}\ q) &\stackrel{def}{=} dup(p)\ \texttt{||}\ dup(q) \\
\boxed{dup(\texttt{loop}\ p\ \texttt{end})} &\stackrel{def}{=} \boxed{\texttt{loop}\ dup(p)\texttt{;}\ dup(p)\ \texttt{end}} \\
dup(\texttt{signal}\ S\ \texttt{in}\ p\ \texttt{end}) &\stackrel{def}{=} \texttt{signal}\ S\ \texttt{in}\ dup(p)\ \texttt{end} \\
dup(\texttt{emit}\ S) &\stackrel{def}{=} \texttt{emit}\ S \\
dup(\texttt{present}\ S\ \texttt{then}\ p\ \texttt{else}\ q\ \texttt{end}) &\stackrel{def}{=} \texttt{present}\ S\ \texttt{then}\ dup(p)\ \texttt{else}\ dup(q)\ \texttt{end} \\
dup(\texttt{trap}\ T\ \texttt{in}\ p\ \texttt{end}) &\stackrel{def}{=} \texttt{trap}\ T\ \texttt{in}\ dup(p)\ \texttt{end} \\
dup(\texttt{exit}\ T) &\stackrel{def}{=} \texttt{exit}\ T
\end{aligned}$$

For example,

$$dup(\texttt{loop pause; emit S end}) = \texttt{loop pause; emit S; pause; emit S end}$$

This program transformation is called *reincarnation* as it explicitly distributes the several simultaneous instances, that is to say *incarnations*, of each signal and parallel construct into several distinct "bodies". In other words, a subterm having several incarnations is replaced by several subterms, each of them having a unique incarnation.

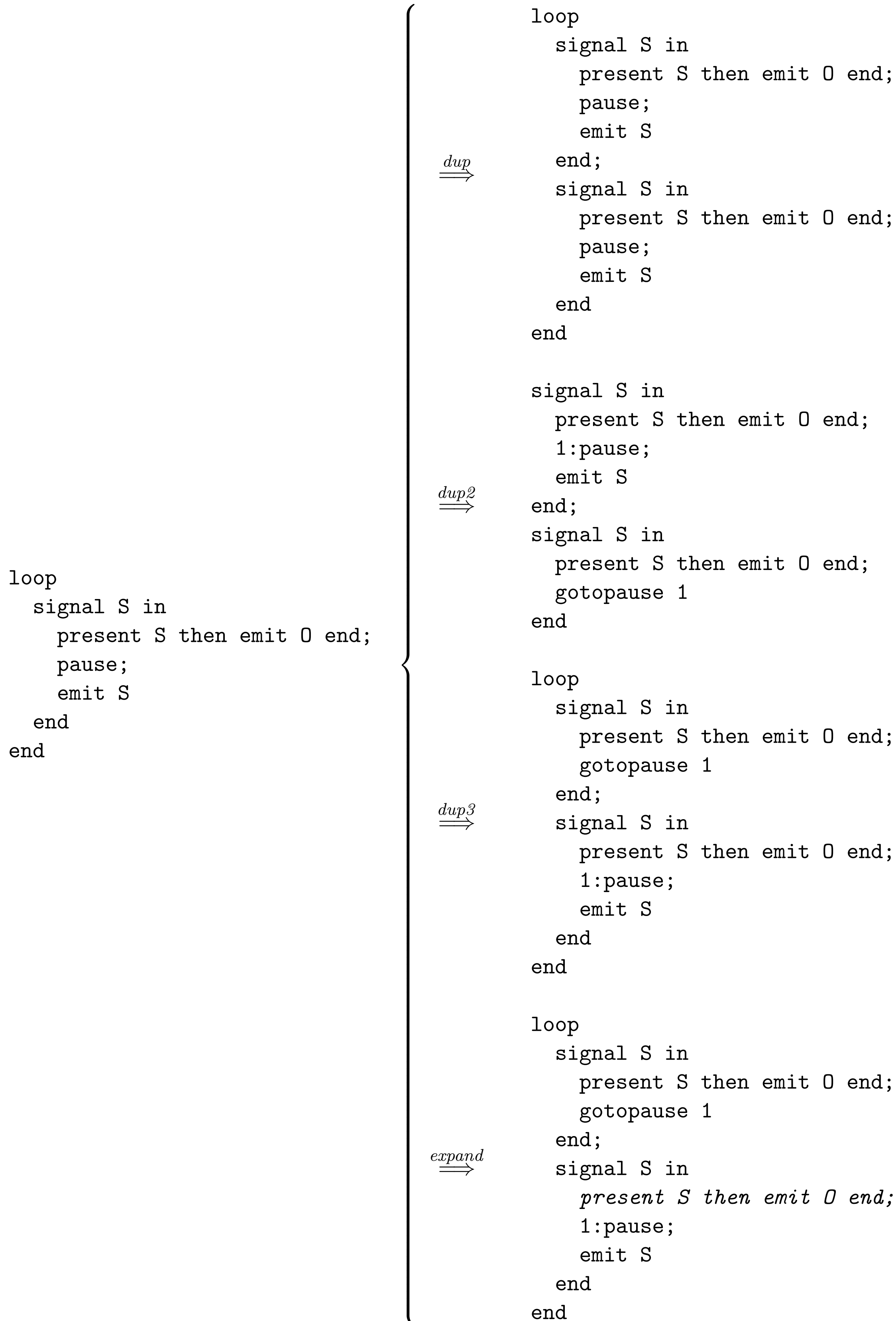


Figure 7.1: Reincarnation Techniques Compared

For instance, in the top rewriting of Figure 7.1, the initially schizophrenic signal S is duplicated, so that each resulting signal S is not schizophrenic. We shall describe and comment the other transformations listed in this figure in the rest of the chapter.

Let us prove the correctness of the transformation:

Lemma 7.1. *If p is non-instantaneous then "*`loop` *p* `end`*" and "*`loop` *p*; *p* `end`*" are $\rightarrow$bisimilar.*

Proof. Let "$\sim$" be the binary relation such that $a \sim b$ iff one of the following holds:

- $a = b$
- there exists q s.t. $a =$ "q; `loop` p `end`" and $b =$ "q; `loop` p; p `end`"
- there exists q s.t. $a =$ "q; `loop` p `end`" and $b =$ "q; p; `loop` p; p `end`"

This relation is total and surjective.

- If $a \xrightarrow[I]{O,k} a'$ with $a =$ "q; `loop` p `end`" and $b =$ "q; `loop` p; p `end`" then either:
  - $q \xrightarrow[I]{O,k} q'$, $k \neq 0$, $a' =$ "q'; `loop` p `end`", so that:
    $b \xrightarrow[I]{O,k}$ "q'; `loop` p; p `end`", with $a' \sim$"q'; `loop` p; p `end`".
  - $q \xrightarrow[I]{O_q,0} q'$, $p \xrightarrow[I]{O_p,k} p'$, $k \neq 0$, $a' =$ "p'; `loop` p `end`", $O = O_q \cup O_p$, so that:
    $b \xrightarrow[I]{O,k}$ "p'; p; `loop` p; p `end`", with $a' \sim$"p'; p; `loop` p; p `end`".
- etc.

Therefore, this relation is a $\rightarrow$bisimulation. □

Theorem 7.2. *If p is safe then p and $dup(p)$ are $\rightarrow$bisimilar.*

Proof. Let "$\sim$" be the least binary relation such that:

- `nothing` $\sim$ `nothing`, `pause` $\sim$ `pause`, `exit` $T \sim$ `exit` T, `emit` $S \sim$ `emit` S
- if $p \sim q$ then $\begin{cases} \texttt{loop } p \texttt{ end} \sim \texttt{loop } q \texttt{ end} \\ \texttt{trap } T \texttt{ in } p \texttt{ end} \sim \texttt{trap } T \texttt{ in } q \texttt{ end} \\ \texttt{signal } S \texttt{ in } p \texttt{ end} \sim \texttt{signal } S \texttt{ in } q \texttt{ end} \end{cases}$
- if $\begin{cases} p \sim q \\ u \sim v \end{cases}$ then $\begin{cases} p\texttt{; } u \sim q\texttt{; } v \\ p \texttt{ || } u \sim q \texttt{ || } v \\ \texttt{present } S \texttt{ then } p \texttt{ else } u \texttt{ end} \sim \texttt{present } S \texttt{ then } q \texttt{ else } v \texttt{ end} \end{cases}$
- if $p \sim q$ and p is non-instantaneous then $\begin{cases} \texttt{loop } p \texttt{ end} \sim \texttt{loop } q\texttt{; } q \texttt{ end} \\ \texttt{loop } p \texttt{ end} \sim q\texttt{; loop } q\texttt{; } q \texttt{ end} \end{cases}$

By induction, this relation is a $\rightarrow$bisimulation. If p is safe then $p \sim dup(p)$. □

Theorem 7.3. *If p is safe then $dup(p)$ is obviously not schizophrenic (in the sense of Chapter 5).*

Proof. Let us proceed by structural induction on p:

- dup(`nothing`), dup(`pause`), dup(`exit` T), dup(`emit` S) are obviously not schizophrenic.
- if $p =$ `loop` q `end` is safe and $dup(q)$ is obviously not schizophrenic:
  $dup(p) =$ `loop` $dup(q)$; $dup(q)$ `end`
  $0 \notin \Gamma_q$, thus `loop` $\langle dup(q) \rangle$; $dup(q)$ `end` $=$ `loop` $dup(q)$; $\langle dup(q) \rangle$ `end` $= \emptyset$

– if $dup(p) =$ `loop` $dup(q)$`;` $C[r]$ `end` then `loop` $dup(q)$`;` $C\langle r\rangle$ `end` $= C\langle r\rangle$
`loop` $dup(q)$`;` $C\langle r\rangle$ `end` $\cap\, \Omega_r = C\langle r\rangle \cap \Omega_r$

– if $dup(p) =$ `loop` $C[r]$`;` $dup(q)$ `end` then `loop` $C\langle r\rangle$`;` $dup(q)$ `end` $= C\langle r\rangle$
`loop` $C\langle r\rangle$`;` $dup(q)$ `end` $\cap\, \Omega_r = C\langle r\rangle \cap \Omega_r$

in both cases $C[r]$ is obviously not schizophrenic, thus p is obviously not schizophrenic.

- if $p =$ `trap` T `in` q `end` is safe and $dup(q)$ is obviously not schizophrenic:
$dup(p) =$ `trap` T `in` $dup(q)$ `end` and `trap` T `in` $\langle dup(q)\rangle$ `end` $= \emptyset$
if $dup(p) =$ `trap` T `in` $C[r]$ `end` then `trap` T `in` $C\langle r\rangle$ `end` $= C\langle r\rangle$
`trap` T `in` $C\langle r\rangle$ `end` $\cap\, \Omega_r = C\langle r\rangle \cap \Omega_r$, $dup(q) = C[r]$ is obviously not schizophrenic, thus $dup(p)$ is obviously not schizophrenic.

- etc.

In all cases, p is obviously not schizophrenic. □

Therefore, if p is safe then $dup(p)$ is observationally equivalent and obviously not schizophrenic.

```
loop
  trap T in
    signal A in
      pause;
      beep;
      exit T
    ||
      loop
        signal B in beep end;
        signal C in pause end
      end
    end
  end
end
```

$\overset{dup}{\Longrightarrow}$

```
loop
  trap T in
    signal A in
      pause;
      beep;
      exit T
    ||
      loop
        signal B in beep end;
        signal C in pause end;
        signal B in beep end;
        signal C in pause end
      end
    end
  end;
  trap T in
    signal A in
      pause;
      beep;
      exit T
    ||
      loop
        signal B in beep end;
        signal C in pause end;
        signal B in beep end;
        signal C in pause end
      end
    end
  end
end
```

Figure 7.2: Exponential Reincarnation with *dup*

Not only does this rewriting technique remove schizophrenia, but it makes non-schizophrenia obvious for the decision procedure we described in Chapter 5.

In fact, it would once and for all take care of schizophrenic programs if the transformation was efficient enough. It is not the case: $dup(p)$ can be exponentially larger than p because of nested unfoldings. The inner loop body "`signal B in beep end; signal C in pause end`" in the example of Figure 7.2, occurs four times in the rewritten program, that is to say 2^n times where n is the number of loops enclosing the statement.

Moreover, in this example, neither the duplication of the inner loop body alone, nor the duplication of the outer loop body would be enough to make the program non-schizophrenic.

7.2 Quadratic Reincarnation

From now on, we shall consider well-formed Esterel* programs, unless otherwise stated. Thanks to `gotopause`, we can completely eliminate loops in Esterel* programs, by *unfolding* every loop "`loop` p `end`" into the sequence "p`;` $\overline{p}$", where $\overline{p}$ is obtained from p by replacing all `pause` instructions with `gotopause` constructs, so that, when the control reaches $\overline{p}$, it jumps *back* to p, thus reproducing the behavior of "`loop` p `end`". For example, "`loop 1:pause end`" and "`1:pause; gotopause 1`" behave the same.

Moreover, $\overline{p}$ cannot be active, i.e. contain active `pause` instructions, as $\overline{p}$ contains no `pause` instruction at all. Therefore, most of $\overline{p}$ is never used, and it is enough to unfold the *surface* of p, that is to say the part of p that may be reduced in its first reaction. We define:

$$
\begin{aligned}
surf(\texttt{nothing}) &\stackrel{def}{=} \texttt{nothing} \\
\boxed{surf(label\texttt{:pause}) \stackrel{def}{=} \texttt{gotopause}\ label} & \\
surf(\texttt{gotopause}\ label) &\stackrel{def}{=} \texttt{gotopause}\ label \\
\boxed{surf(p\texttt{;}\ q) \stackrel{def}{=} \text{if } 0 \in \Gamma_p \text{ then } surf(p)\texttt{;}\ surf(q) \text{ else } surf(p)} & \\
surf(p\ \texttt{||}\ q) &\stackrel{def}{=} surf(p)\ \texttt{||}\ surf(q) \\
\boxed{surf(\texttt{loop}\ p\ \texttt{end}) \stackrel{def}{=} surf(p)} & \\
surf(\texttt{signal}\ S\ \texttt{in}\ p\ \texttt{end}) &\stackrel{def}{=} \texttt{signal}\ S\ \texttt{in}\ surf(p)\ \texttt{end} \\
surf(\texttt{emit}\ S) &\stackrel{def}{=} \texttt{emit}\ S \\
surf(\texttt{present}\ S\ \texttt{then}\ p\ \texttt{else}\ q\ \texttt{end}) &\stackrel{def}{=} \texttt{present}\ S\ \texttt{then}\ surf(p)\ \texttt{else}\ surf(q)\ \texttt{end} \\
surf(\texttt{trap}\ T\ \texttt{in}\ p\ \texttt{end}) &\stackrel{def}{=} \texttt{trap}\ T\ \texttt{in}\ surf(p)\ \texttt{end} \\
surf(\texttt{exit}\ T) &\stackrel{def}{=} \texttt{exit}\ T
\end{aligned}
$$

The three non-elementary rules are boxed:

- `pause` statements are changed into `gotopause` statements.
- If p is non-instantaneous then $surf(q)$ cannot be reached in "$surf(p)$`;` $surf(q)$", and is discarded.
- A loop cannot be taken instantly (cf. Lemma 5.2), so it may be removed from the surface.

Well-formedness is preserved:

Lemma 7.4. *If p is well formed then "$surf(p)$", "p`;` $surf(p)$", and "$surf(p)$`;` p" are well formed.*

Proof. Structural induction. □

The possible reactions of p and $surf(p)$ are the same:

Lemma 7.5. *If p is safe then $\forall I, \forall O, \forall k, \forall L : p \xrightarrow[I]{O,k} L \Leftrightarrow surf(p) \xrightarrow[I]{O,k} L$.*

Corollary 7.6. *If p is safe and non-instantaneous then $\texttt{loop}\ p\ \texttt{end} \xrightarrow[I]{O,k} L \Leftrightarrow surf(p) \xrightarrow[I]{O,k} L$.*

Corollary 7.7. *If p is safe then $\Gamma_{surf(p)} = \Gamma_{surf(p);\ p} = \Gamma_{p;\ surf(p)} = \Gamma_p$.*

Proof. Structural induction. □

We now establish that "$p;\ surf(p)$" and "$\texttt{loop}\ p\ \texttt{end}$" are equivalent if p is non-instantaneous:

Lemma 7.8. *If p is well formed then $\lceil p;\ surf(p)|L\rfloor$ is valid iff $\lceil \texttt{loop}\ p\ \texttt{end}|L\rfloor$ is valid.*

Proof. There is no `pause` instruction in $surf(p)$. □

Lemma 7.9. *If p is safe and non-instantaneous then:*

- *$\lceil \texttt{loop}\ p\ \texttt{end}|*\rfloor$ and $\lceil p;\ surf(p)|*\rfloor$ are $\diamond\!\!\rightarrow$bisimilar.*
- *for every L, $\lceil \texttt{loop}\ p\ \texttt{end}|L\rfloor$ and $\lceil p;\ surf(p)|L\rfloor$ are $\diamond\!\!\rightarrow$bisimilar if valid.*

Proof. Let "$\sim$" be the binary relation between valid states such that $a \sim b$ iff either:

- $a = b$
- $a = \lceil \texttt{loop}\ p\ \texttt{end}|*\rfloor$ and $b = \lceil p;\ surf(p)|*\rfloor$
- $a = \lceil \texttt{loop}\ p\ \texttt{end}|L\rfloor$ and $b = \lceil p;\ surf(p)|L\rfloor$ for some L.

This relation is total and surjective.

- If $\lceil \texttt{loop}\ p\ \texttt{end}|*\rfloor \overset{O,k}{\underset{I}{\diamond\!\!\longrightarrow}} \lceil \texttt{loop}\ p\ \texttt{end}|L\rfloor$ then $\texttt{loop}\ p\ \texttt{end} \xrightarrow[I]{O,k} L$, thus $p \xrightarrow[I]{O,k} L$ with $k \neq 0$, so that $p;\ surf(p) \xrightarrow[I]{O,k} L$, i.e. $\lceil p;\ surf(p)|*\rfloor \overset{O,k}{\underset{I}{\diamond\!\!\longrightarrow}} \lceil p;\ surf(p)|L\rfloor$.
- If $\lceil p;\ surf(p)|*\rfloor \overset{O,k}{\underset{I}{\diamond\!\!\longrightarrow}} \lceil p;\ surf(p)|L\rfloor$ then $p;\ surf(p) \xrightarrow[I]{O,k} L$. Since p is non-instantaneous, $p \xrightarrow[I]{O,k} L$ with $k \neq 0$, so that $\texttt{loop}\ p\ \texttt{end} \xrightarrow[I]{O,k} L$, i.e. $\lceil \texttt{loop}\ p\ \texttt{end}|*\rfloor \overset{O,k}{\underset{I}{\diamond\!\!\longrightarrow}} \lceil \texttt{loop}\ p\ \texttt{end}|L\rfloor$.
- If $\lceil \texttt{loop}\ p\ \texttt{end}|L\rfloor \overset{O,k}{\underset{I}{\diamond\!\!\longrightarrow}} \lceil \texttt{loop}\ p\ \texttt{end}|L'\rfloor$ then $\epsilon(\lceil \texttt{loop}\ p\ \texttt{end}|L\rfloor) \xrightarrow[I]{O,k} L'$ and $k \neq 0$, therefore $\epsilon(\lceil p|L\rfloor);\ \texttt{loop}\ p\ \texttt{end} \xrightarrow[I]{O,k} L'$, thus $\epsilon(\lceil p|L\rfloor);\ surf(p) \xrightarrow[I]{O,k} L'$, so that $\epsilon(\lceil p;\ surf(p)|L\rfloor) \xrightarrow[I]{O,k} L'$, i.e. $\lceil p;\ surf(p)|L\rfloor \overset{O,k}{\underset{I}{\diamond\!\!\longrightarrow}} \lceil p;\ surf(p)|L'\rfloor$.
- If $\lceil p;\ surf(p)|L\rfloor \overset{O,k}{\underset{I}{\diamond\!\!\longrightarrow}} \lceil p;\ surf(p)|L'\rfloor$ then $\epsilon(\lceil p;\ surf(p)|L\rfloor) \xrightarrow[I]{O,k} L'$ and $k \neq 0$, therefore $\epsilon(\lceil p|L\rfloor);\ surf(p) \xrightarrow[I]{O,k} L'$, thus $\epsilon(\lceil p|L\rfloor);\ \texttt{loop}\ p\ \texttt{end} \xrightarrow[I]{O,k} L'$, so that $\epsilon(\lceil \texttt{loop}\ p\ \texttt{end}|L\rfloor) \xrightarrow[I]{O,k} L'$, i.e. $\lceil \texttt{loop}\ p\ \texttt{end}|L\rfloor \overset{O,k}{\underset{I}{\diamond\!\!\longrightarrow}} \lceil \texttt{loop}\ p\ \texttt{end}|L'\rfloor$.

As a result, the "$\sim$" relation is a $\diamond\!\!\rightarrow$bisimulation. □

We then define the recursive rewriting of "loop p end" into "p; $surf(p)$" as follows:

$$
\begin{array}{rcl}
dup2(\texttt{nothing}) & \stackrel{def}{=} & \texttt{nothing} \\
dup2(label\texttt{:pause}) & \stackrel{def}{=} & label\texttt{:pause} \\
dup2(\texttt{gotopause}\ label) & \stackrel{def}{=} & \texttt{gotopause}\ label \\
dup2(p\texttt{;}\ q) & \stackrel{def}{=} & dup2(p)\texttt{;}\ dup2(q) \\
dup2(p\ \texttt{||}\ q) & \stackrel{def}{=} & dup2(p)\ \texttt{||}\ dup2(q) \\
\hline
\multicolumn{1}{|r}{dup2(\texttt{loop}\ p\ \texttt{end})} & \stackrel{def}{=} & \multicolumn{1}{l|}{dup2(p)\texttt{;}\ surf(p)} \\
\hline
dup2(\texttt{signal}\ S\ \texttt{in}\ p\ \texttt{end}) & \stackrel{def}{=} & \texttt{signal}\ S\ \texttt{in}\ dup2(p)\ \texttt{end} \\
dup2(\texttt{emit}\ S) & \stackrel{def}{=} & \texttt{emit}\ S \\
dup2(\texttt{present}\ S\ \texttt{then}\ p\ \texttt{else}\ q\ \texttt{end}) & \stackrel{def}{=} & \texttt{present}\ S\ \texttt{then}\ dup2(p)\ \texttt{else}\ dup2(q)\ \texttt{end} \\
dup2(\texttt{trap}\ T\ \texttt{in}\ p\ \texttt{end}) & \stackrel{def}{=} & \texttt{trap}\ T\ \texttt{in}\ dup2(p)\ \texttt{end} \\
dup2(\texttt{exit}\ T) & \stackrel{def}{=} & \texttt{exit}\ T
\end{array}
$$

For example,

$$
\begin{array}{rcl}
surf(\texttt{1:pause; emit S}) & = & \texttt{gotopause 1} \\
dup2(\texttt{loop 1:pause; emit S end}) & = & \texttt{1:pause; emit S; gotopause 1} \\
surf(\texttt{emit S; 1:pause}) & = & \texttt{emit S; gotopause 1} \\
dup2(\texttt{loop emit S; 1:pause end}) & = & \texttt{emit S; 1:pause; emit S; gotopause 1}
\end{array}
$$

Lemma 7.10. *If p is well formed then $\lceil dup2(p)|L\rfloor$ is valid iff $\lceil p|L\rfloor$ is valid.*

Proof. Structural induction based on Lemma 7.4. □

Theorem 7.11. *If p is safe then $\lceil p|*\rfloor$ and $\lceil dup2(p)|*\rfloor$ are $\diamond\!\!\rightarrow$bisimilar.*

Proof. Structural induction similar to Theorem 7.2 based on Lemma 7.9. □

Of course, unfolded programs no longer contain potentially instantaneous loops or potentially schizophrenic statements: there is no `loop` construct left, and `gotopause` only allows to *non-instantly* reenter previously traversed pieces of code (cf. Chapter 6).

Theorem 7.12. *If p is safe then $dup2(p)$ is not schizophrenic (in the sense of Chapter 6).*

Proof. If $dup2(p)$ is well tagged then, whatever L, the tags of $\lceil dup2(p)|L\rfloor$ are pairwise distinct, by structural induction on p. □

Algorithm

In order to perform this unfolding on an Esterel program, we first have to label its `pause` instructions. We rewrite a safe Esterel program p into a non-schizophrenic equivalent well-formed Esterel* program $dup2(\hat{p})$ as the following:

- We first label the `pause` statements of the program p with pairwise distinct labels, producing the well-formed Esterel* program $\hat{p}$.
- We then compute the image of $\hat{p}$ by function $dup2$.

```
                                              trap T in
                                                signal A in
                                                  1:pause;
loop                                              beep;
  trap T in                                       exit T
    signal A in                                 ||
      1:pause;                                    signal B in emit O end;
      beep;                                       signal C in 2:pause end;
      exit T                                      signal B in beep end;
    ||                                 dup2       signal C in gotopause 2 end
      loop                             ==>      end
        signal B in beep end;                 end;
        signal C in 2:pause end               trap T in
      end                                       signal A in
    end                                           gotopause 1;
  end                                           ||
end                                               signal B in beep end;
                                                  signal C in gotopause 2 end
                                                end
                                              end
```

Figure 7.3: Quadratic Reincarnation with *dup2*

In the example of Figure 7.1, the resulting program defines again two signals `S`. Remark that "`gotopause 1`" involves a jump from one signal scope to the other one. Because signal statuses are not preserved from one instant to the next, this is fine. In full Esterel however, variables, counters, registers, etc. retain their values. Therefore, this transformation in not correct for full Esterel. We shall discuss this issue in Chapter 8.

For the example of Figure 7.2, we obtain the program of Figure 7.3. The inner loop body only occurs three times in the resulting program instead of four with Mignard's rewriting technique. In particular, there are only three declarations of `B` and `C`. In general, by counting the number of primitive constructs within statements, we observe that:

Theorem 7.13. *surf*(p) *is smaller than* p*, and dup2*(p) *is at most quadratically larger than* p*.*

Proof. Structural induction. □

For example:

- `loop [`p` || loop `q` end] end`

  $\stackrel{dup}{\Longrightarrow}$
  ```
  loop
    [dup(p) || loop dup(q); dup(q) end];
    [dup(p) || loop dup(q); dup(q) end];
  end
  ```
  $\stackrel{dup2}{\Longrightarrow}$ `[`$dup2(\hat{p})$` || `$dup2(\hat{q})$`; `$surf(\hat{q})$`]; [`$surf(\hat{p})$` || `$surf(\hat{q})$`]`

- `loop [`p` || loop [`p` || loop `p` end] end] end`

  $\stackrel{dup}{\Longrightarrow}$ $2+4+8=14$ times p *(exponential growth)*

  $\stackrel{dup2}{\Longrightarrow}$ $2+3+4=9$ times p *(quadratic growth)*

Reciprocally, any such semantics-preserving transformation that removes `loop` constructs, must be of quadratic worst-case complexity in pure Esterel* extended with side effects. We have seen in Chapter 5 that a program of size n may instantly produce a quadratic number of side effects, which requires a quadratic number of instructions if no `loop` construct is allowed.

7.3 Quasi-Linear Reincarnation

In this section, we combine the unfolding of loops of Section 7.2, with the static analysis of schizophrenia of Chapter 5 in order to achieve a quasi-linear unfolding in practice.

Step 1

First, rather than having $surf(p)$ in sequence *after* p in the unfolding of "`loop` p `end`" into "p`;` $surf(p)$", we put it in sequence *before* p, and rewrite "`loop` p `end`" into "`loop` $surf(p)$`;` p `end`". Formally, we recursively define function *dup3* just as *dup2* except for loops:

$$dup3(\texttt{loop } p \texttt{ end}) \stackrel{def}{=} \texttt{loop } surf(p)\texttt{; } dup3(p) \texttt{ end}$$

For example,

$$dup3(\texttt{loop 1:pause; emit S end}) = \texttt{loop gotopause 1; 1:pause; emit S end}$$

Figure 7.1 provides another example. This transformation no longer removes `loop` constructs, but again preserves the semantics, still eliminates schizophrenia, and retains the quadratic worst-case complexity:

Lemma 7.14. *If p is well formed then $\lceil surf(p)\texttt{; } p|L\rfloor$ is valid iff $\lceil p|L\rfloor$ is valid.*

Lemma 7.15. *If p is safe and non-instantaneous then:*

- *$\lceil p|*\rfloor$ and $\lceil surf(p)\texttt{; } p|*\rfloor$ are $\diamond\!\!\to$bisimilar.*
- *for every L, $\lceil p|L\rfloor$ and $\lceil surf(p)\texttt{; } p|L\rfloor$ are $\diamond\!\!\to$bisimilar if valid.*

Theorem 7.16. *If p is safe then $\lceil p|*\rfloor$ and $\lceil dup3(p)|*\rfloor$ are $\diamond\!\!\to$bisimilar.*

Proof. Similar to the proofs of the previous section. □

Theorem 7.17. *If p is safe then $dup3(p)$ is not schizophrenic (in the sense of Chapter 6).*

Proof. If p is safe and non-instantaneous then whatever L:
$\epsilon\lceil \texttt{loop } surf(p)\texttt{; } dup3(p) \texttt{ end}|L\rfloor = \epsilon\lceil dup3(p)|L\rfloor\texttt{; loop } surf(p)\texttt{; } dup3(p) \texttt{ end}$.
$surf(p)$ is non-instantaneous, so the second copy of $dup3(p)$ cannot be reduced in any reaction of this term. Therefore, if $dup3(p)$ is not schizophrenic then `loop` $surf(p)$`;` $dup3(p)$ `end` is not schizophrenic, hence the result by induction. □

For the program of Figure 7.2, we obtain the transformation of Figure 7.4. While this new rewriting may seem pointless compared to the previous one, it provides a much better starting point for optimization, precisely because it retains the `loop` construct, thus making it possible to unfold pieces of loop bodies rather than whole loop bodies.

Step 2

Schizophrenia arises from the nesting of signal declarations or parallel statements within loops (cf. Chapter 5). Instead of *systematically* unfolding *whole* loop bodies, we could unfold problematic signal declarations and parallel statements only. This leads to the definition of function *dup4* in Figure 7.5, which combines unfolding and static analysis of schizophrenia.

```
loop                                              loop
  trap T in                                       trap T in
    signal A in                                       signal A in
      1:pause;                                          gotopause 1;
      beep;                                           ||
      exit T                                            signal B in beep end;
    ||                                                  signal C in gotopause 2 end
      loop                                            end
        signal B in beep end;                       end;
        signal C in 2:pause end                     trap T in
      end                                             signal A in
    end                                  =dup3=>        1:pause;
  end                                                   beep;
end                                                     exit T
                                                      ||
                                                        loop
                                                          signal B in beep end;
                                                          signal C in gotopause 2 end;
                                                          signal B in beep end;
                                                          signal C in 2:pause end
                                                        end
                                                      end
                                                    end
                                                  end
```

Figure 7.4: Quadratic Reincarnation with *dup3*

$$
\begin{aligned}
\mathit{dup4}(K, \texttt{nothing}) &\stackrel{def}{=} \texttt{nothing} \\
\mathit{dup4}(K, \mathit{label}\texttt{:pause}) &\stackrel{def}{=} \mathit{label}\texttt{:pause} \\
\mathit{dup4}(K, \texttt{exit } T) &\stackrel{def}{=} \texttt{exit } T \\
\mathit{dup4}(K, \texttt{emit } S) &\stackrel{def}{=} \texttt{emit } S \\
\mathit{dup4}(K, \texttt{present ... end}) &\stackrel{def}{=} \texttt{present } S \texttt{ then } \mathit{dup4}(K,p) \texttt{ else } \mathit{dup4}(K,q) \texttt{ end} \\
\mathit{dup4}(K, \texttt{loop } p \texttt{ end}) &\stackrel{def}{=} \texttt{loop } \mathit{dup4}(K \cup \{0\}, p) \texttt{ end} \\
\mathit{dup4}(K, \texttt{trap } T \texttt{ in } p \texttt{ end}) &\stackrel{def}{=} \texttt{trap } T \texttt{ in } \mathit{dup4}(\{k \in \mathbb{N}, \downarrow k \in K\}, p) \texttt{ end} \\
\mathit{dup4}(K, p\texttt{; } q) &\stackrel{def}{=} \begin{bmatrix} \mathit{dup4}((\text{if } K \cap \Gamma_q = \emptyset \text{ then } K \backslash \{0\} \text{ else } K \cup \{0\}), p)\texttt{;} \\ \mathit{dup4}((\text{if } 0 \in \Gamma_p \text{ then } K \text{ else } \emptyset), q) \end{bmatrix} \\
\mathit{dup4}(K, \texttt{signal } S \texttt{ in } p \texttt{ end}) &\stackrel{def}{=} \begin{bmatrix} \text{if} & K \cap \Omega_p = \emptyset \\ \text{then} & \texttt{signal } S \texttt{ in } \mathit{dup4}(\emptyset, p) \texttt{ end} \\ \text{else} & \begin{bmatrix} \texttt{signal } S \texttt{ in } \mathit{surf}(p) \texttt{ end;} \\ \mathit{skip}(\texttt{signal } S \texttt{ in } \mathit{dup4}(\emptyset, p) \texttt{ end}) \end{bmatrix} \end{bmatrix} \\
\mathit{dup4}(K, p \texttt{ || } q) &\stackrel{def}{=} \begin{bmatrix} \text{if} & K \cap \Omega_{p \texttt{ || } q} = \emptyset \\ \text{then} & \mathit{dup4}(\emptyset, p) \texttt{ || } \mathit{dup4}(\emptyset, q) \\ \text{else} & \texttt{[}\mathit{surf}(p) \texttt{ || } \mathit{surf}(q)\texttt{]}\,\texttt{;}\,\mathit{skip}(\texttt{[}\mathit{dup4}(\emptyset, p) \texttt{ || } \mathit{dup4}(\emptyset, q)\texttt{]}) \end{bmatrix}
\end{aligned}
$$

Figure 7.5: Reincarnation with Static Analysis

Since we have only defined "Ω" for safe Esterel programs, we only define function *dup4* for safe well-labeled Esterel programs, that is to say safe Esterel* programs without `gotopause` instructions. As discussed in Chapter 6, we shall concentrate in this document on the provably correct, efficient reincarnation of Esterel programs, leaving the handling of Esterel* for future work.

Function *dup4* now context-dependent. It takes two arguments: the statement p to rewrite and the (initially empty) risk K of the current context, in the sense of Chapter 5. The recursive computation of K exactly matches that of the *risk* function of Figure 5.4, except that no exception is ever raise; unfolding takes place instead.

As intended, `loop` constructs no longer replicate code on their own:

$$\begin{aligned} dup3(\texttt{loop 1:pause; emit S end}) &= \texttt{loop gotopause 1; 1:pause; emit S end} \\ dup4(K, \texttt{loop 1:pause; emit S end}) &= \texttt{loop 1:pause; emit S end} \end{aligned}$$

Let focus on the rules for parallel statements and signal declarations. These constructs are only expanded if potentially schizophrenic:

- if $K \cap \Omega_p = 0$ then $dup4(K, \texttt{signal } S \texttt{ in } p \texttt{ end})$ is simply $\texttt{signal } S \texttt{ in } dup4(\emptyset, p) \texttt{ end}$,
- if $K \cap \Omega_{p \,||\, q} = 0$ then $dup4(K, p \;||\; q)$ is simply $dup4(\emptyset, p) \;||\; dup4(\emptyset, q)$.

In contrast with loop bodies in safe programs, potentially schizophrenic signal declarations and parallel statements may be instantaneous. For instance, the signal `S` is both potentially schizophrenic and potentially instantaneous in:

```
loop
  trap T in
    signal S in
      present I then 1:pause; exit T end
    end;
    pause
  end
end
```

As a result, the surface of such a statement may instantly terminate. A more complex transformation than the recursive expansion of p into "$surf(p);\ p$" is now required for potentially schizophrenic constructs. In *dup4*, potentially schizophrenic constructs are expanded according to the rule "$p \mapsto surf(p);\ skip(p)$". Function[1] *skip* is defined as the following:

$$skip(p) = \begin{cases} \texttt{trap } T \texttt{ in exit } T;\ p \texttt{ end} & \text{if } 0 \in \Gamma_p \quad \text{(where } T \text{ is a fresh exception name)} \\ p & \text{if } 0 \notin \Gamma_p \quad \text{(to avoid unnecessary code)} \end{cases}$$

It makes it possible to skip over the second copy of p in "$surf(p);\ skip(p)$" if $surf(p)$ terminates instantly, hence its name. In contrast with Lemma 7.15, the following holds for potentially instantaneous programs in addition to non-instantaneous programs:

Lemma 7.18. *If p is safe then:*

- $\lceil p|*\rfloor$ *and* $\lceil surf(p);\ skip(p)|*\rfloor$ *are* $\diamond\!\!\rightarrow$*bisimilar.*
- *for every L,* $\lceil p|L\rfloor$ *and* $\lceil surf(p);\ skip(p)|L\rfloor$ *are* $\diamond\!\!\rightarrow$*bisimilar if valid.*

Proof. As usual. □

Corollary 7.19. *If p is safe then for all K :* $\lceil p|*\rfloor$ *and* $\lceil dup4(K, p)|*\rfloor$ *are* $\diamond\!\!\rightarrow$*bisimilar.*

[1] It makes sense to define a primitive construct as well: $p \texttt{ skip } q \stackrel{def}{=} p;\ skip(q)$, to help further optimization.

For the last example we obtain the non-schizophrenic equivalent program:

```
loop
  trap T in
    signal S in
      present I then gotopause 1 end
    end;
    trap U in
      exit U;
      signal S in
        present I then 1:pause; exit T end
      end
    end;
    pause
  end
end
```

For all safe well-labeled Esterel program p, we define: $expand(p) = dup4(\emptyset, p)$.

Corollary 7.20. *Whatever p, $\lceil p|*\rfloor$ and $\lceil expand(p)|*\rfloor$ are $\diamond\!\!\rightarrow$bisimilar.*

Theorem 7.21. *Whatever p, $expand(p)$ is not schizophrenic (in the sense of Chapter 6).*

Proof. This key result is established in Appendix A. □

Again, $expand(p)$ may be quadratically larger than p in the worst case. But this last algorithm is in practice quasi-linear, as we shall observe in Chapter 8.

Since, in the example of Figure 7.1, the signal `S` is schizophrenic, there is no difference between the programs computed by *dup3* and *expand*. For the more complex example of Figure 7.2, however, the computed program is shorter. In particular, there are only two declarations of `B` in Figure 7.6. Indeed, if we first unfold the declaration of `A` only, we obtain the program of Figure 7.7 where we renamed duplicated signals to improve readability. We observe that:

- Neither `A0`, nor `A1` is schizophrenic, thanks to the unfolding of the declaration of `A`.
- Neither `B0`, nor `C0` is schizophrenic (they belong to the surface of `A` (i.e. the scope of `A0`)).
- `B1` is (obviously) not schizophrenic.
- `C1` is schizophrenic.

As a result, the declaration of `C1` must still be unfolded, whereas the unfolding of declaration of `B1` should be avoided. This is exactly what our last reincarnation achieves: in the end, there are three declarations for `C`, but only two for `B` (cf. Figure 7.6).

Moreover, there remains only three `beep` instructions in this rewritten program, whereas there were four at least with the previous techniques, corresponding to the fact that at most tree beeps are generated in each reaction.

Algorithm

In summary, the reincarnation algorithm we propose for an Esterel program p consists of traversing p a first time to compute Γ and Ω and label the `pause` instructions of p with pairwise distinct labels, aborting if p is not safe, otherwise producing $\hat{p}$, then computing $expand(\hat{p})$.

```
loop
  trap T in
    signal A in
      1:pause;
      beep;
      exit T
    ||
      loop
        signal B in beep end;
        signal C in 2:pause end
      end
    end
  end
end
```

$\overset{expand}{\Longrightarrow}$

```
loop
  trap T in
    signal A in
      gotopause 1;
    ||
      signal B in beep end;
      signal C in gotopause 2 end
    end;
    signal A in
      1:pause;
      beep;
      exit T
    ||
      loop
        signal B in beep end;
        signal C in gotopause 2 end;
        signal C in 2:pause end
      end
    end
  end
end
```

Figure 7.6: Quasi-linear Reincarnation with *expand*

```
loop
  trap T in
    signal A in
      1:pause;
      beep;
      exit T
    ||
      loop
        signal B in beep end;
        signal C in 2:pause end
      end
    end
  end
end
```

$\Longrightarrow$

```
loop
  trap T in
    signal A0 in
      gotopause 1;
    ||
      signal B0 in beep end;
      signal C0 in gotopause 2 end
    end;
    signal A1 in
      1:pause;
      beep;
      exit T
    ||
      loop
        signal B1 in beep end;
        signal C1 in 2:pause end
      end
    end
  end
end
```

Figure 7.7: Partial Expansion (Outer Loop)

Theorem 7.22. *If p is a safe Esterel program then:*

- *$expand(\hat{p})$ is not schizophrenic.*
- *p and $\lceil expand(\hat{p})|*\rfloor$ are bisimilar w.r.t. to $\rightarrow$ and $\diamond\!\!\rightarrow$.*

Proof. Theorem 6.8, Corollary 7.20, and Theorem 7.21. □

Dead Code

Going back to the example of Figure 7.1, we observe that dead code is generated. In the output program, the second copy of "`present S then emit O end`" (in *italic*) is dead, and should not be made. The reincarnation algorithm we have just formalized, can be further refined to automatically avoid such dead code generation. We leave it for future work.

SUMMARY

We have specified a program transformation that rewrites any safe schizophrenic Esterel program into a non-schizophrenic and observationally equivalent well-formed Esterel* program. We have established it is correct, and observed that the output program is at most quadratically larger than the input program. Moreover, this algorithm embeds the static analysis of schizophrenia of Chapter 5, to achieve a low expansion ratio.

Chapter 8

Implementation

Applied to the pure Esterel* program p, the preprocessing we specified in the previous chapters:

- checks than all loops of p are non-instantaneous,
- rewrites pieces of p to produce a non-schizophrenic equivalent pure Esterel* program.

We now discuss implementation. In Section 8.1, we first consider the extension of our techniques to the full Esterel* language. Then, in Section 8.2, we describe a prototype compiler based on this preprocessing, in particular discussing `gotopause` implementation, and the benefits of such a separation of concerns. Finally, in Section 8.3, we report some early experiments.

8.1 Full Esterel* Support

Full Esterel adds to pure Esterel the ability to manipulate data of various kinds: private variables, shared values, counters, registers, etc. The good news is that data do not impact on loop safety and do not introduce more schizophrenia problems. Therefore, extending our static analyses to full Esterel/Esterel* is straightforward: we just abstract data away. The bad news is that data break our various reincarnation schemes using `gotopause`. Let us for instance consider a program where a variable is declared within a schizophrenic signal scope:

```
var X in
  loop
    signal S in
      var Y in
        Y := 1;
        1:pause;
        X := Y
      end
    end
  end
end
```

Variables are lexically scoped and declared with the "`var` V `in` ... `end`" construct. Their semantics has been formalized in [Pot02]. In this simple example, variables behave just as one would expect. In each reaction starting from the second one, `X` is loaded with the value 1, loaded in `Y` in the previous instant, because variables retain their value from one instant to the next.

In this (safe) program, the local signal `S` is schizophrenic. Thus, if we apply to this example the final reincarnation algorithm (*expand*) defined in Chapter 7, we obtain the following program, where we rename duplicated variables and signals for readability:

```
var X in
  loop
    signal S in
      var Y in
        Y := 1;
        1:pause;
        X := Y
      end
    end
  end
end
```

$\stackrel{expand}{\Longrightarrow}$

```
var X in
  loop
    signal S0 in
      var Y0 in
        Y0 := 1;
        gotopause 1
      end
    end;
    signal S1 in
      var Y1 in
        Y1 := 1;
        1:pause;
        X := Y1
      end
    end
  end
end
```

Being nested in the declaration of `S`, the declaration of `Y` is duplicated. As result, the `gotopause` statement involves a jump from the scope of the first copy (`Y0`) of the variable `Y` to the second one (`Y1`). This is not satisfactory:

- In the first reaction, `Y0` is set to value 1.
- In the second reaction, `X` receives the value of `Y1`.

But there is no connection between `Y0` and `Y1`. In particular, `Y1` is never initialized. As a result, the value affected to `X` must be wrong. We must update our program transformation. Intuitively, we have to do memory allocation for data *before* reincarnation.

We choose to introduce *static aliasing* in source Esterel* programs, expressing that the two distinct declarations of `Y` (`Y0` and `Y1`) in the expansed program in fact correspond to a unique "object" (i.e. wire, memory cell, whatever). By indexing variables on a global base before reincarnation, then preserving indexes during the reincarnation, we produce the program:

```
var X in
  loop
    signal S in
      var Y in
        Y := 1;
        1:pause;
        X := Y
      end
    end
  end
end
```

$\stackrel{index}{\Longrightarrow}$

```
var X index 1 in
  loop
    signal S in
      var Y index 2 in
        Y := 1;
        1:pause;
        X := Y
      end
    end
  end
end
```

$\stackrel{expand}{\Longrightarrow}$

```
var X index 1 in
  loop
    signal S0 in
      var Y0 index 2 in
        Y0 := 1;
        gotopause 1
      end
    end;
    signal S1 in
      var Y1 index 2 in
        Y1 := 1;
        1:pause;
        X := Y1
      end
    end
  end
end
```

To this end, we have introduced a new "`index`" keyword in Esterel* syntax.

Thanks to these indexes, we can easily alias variables `Y0` and `Y1` in subsequent code generation steps, leading to a correct implementation of the initial program. This technique can be applied to all kinds of data defined in full Esterel.

8.2 A Prototype Compiler

Using the extended preprocessing described in the previous section, we have implemented a prototype compiler for Esterel v5 [Ber00a] into Boolean equation systems, generating `sc6` files[1]. The compiler code consists of about 5000 lines of OCaml [Ler02], structured as follows:

1. parsing and macro expansion,
2. link (i.e. source-level inlining of submodules),
3. static analysis and reincarnation,
4. Boolean equation synthesis (of safe non-schizophrenic programs),
5. a bit of Boolean optimization (for `sc6` compliance).

Relevant to our discussion are Steps 3 and 4 and their relationship. Step 3 rewrites linked macro-expanded Esterel source code using the extended preprocessing. Step 4 essentially implements the naive (linear) circuit synthesis of Berry [Ber99], to which we add `gotopause` and data.

Compiling `gotopause` is straightforward: we allocate as usual one bit-register per `pause` statement. But in addition to the regular connection of one wire to the input pin of this register required by the `pause` statement itself, we connect (through an `or` gate) one extra wire per `gotopause` statement with corresponding label. In general, Esterel compilers are typically based on internal representations of programs as graphs, in which the `gotopause` instruction can be easily represented.

What makes our compiler architecture really attractive in our view is the combination of the following properties:

- Step 4 is completely independent from Step 3. In other words, the synthesis phase does not need to know anything about the static analysis/reincarnation phase. They can be implemented independently. But, of course, a shared parser is a good idea.
- Step 4 can assume programs to be loop safe and non-schizophrenic, leading to simplified code generation schemes [Mig94, Ber99], as, on the other hand, compiling `gotopause` constructs is straightforward.
- Programs can be further optimized (rewritten) between Step 3 and Step 4. On the one hand, there is virtually no loss of structural information in the preprocessing. On the other hand, `gotopause` makes it possible to define more powerful program transformations in Esterel* than in Esterel.
- The output of Step 3 is human readable: essentially no knowledge beyond that of the Esterel language is required to interpret correctly Esterel* programs. In particular, we believe that switching from the initial to the rewritten program, makes it easier to debug schizophrenic specifications in practice.

[1] The `sc6` file format defines a normalized circuit representation, which can in turn be converted into C programs by existing tools [Ber00b].

program	number of kernel statements			description
	source	algorithm (7.2)	algorithm (7.3)	
global	10286	566585	16867	avionics man-machine interface
cabine	7644	67680	8020	avionics cockpit interface
atds100	890	1372	990	video generator
ww	432	833	439	wristwatch
tcint	403	725	418	turbochannel bus

Table 8.1: Comparison

8.3 Experiments

We have conducted some early experiments, summarized in Table 8.1. We count the number of statements (after macro expansion) in programs of various kinds and sizes [Pot02], before and after reincarnation, using both the algorithms of Section 7.2 (quadratic rewriting scheme) and Section 7.3 (quasi-linear rewriting scheme). In the absence of static analysis, the expansion ratio is unacceptable. With static analysis however, it remains low in practice[2].

As a result, we claim that curing schizophrenia by program rewriting is very effective. First, quadratic worst-case complexity is unavoidable [Ber99, Ber00b, SBS04, Pot02]. Second, thanks to static analysis, our last reincarnation algorithm is indeed quasi-linear in practice. In particular, our compiler is just as effective[3] as the reference compiler for Esterel [Ber00b] which internally uses a static analysis of equal power, but much less formalized, and specific to circuits, being deeply entangled with low-level circuit synthesis algorithms.

SUMMARY

We have briefly sketched the extension of our preprocessing techniques to full Esterel, described our prototype compiler, and early experimental results.

[2] We expect to achieve an even tighter expansion with dead code elimination in “global” (not implemented yet, cf. Chapter 7), since constant-propagation in Step 5 is especially effective on this program.

[3] Both the circuits produced and the duration of the synthesis are essentially the same.

Chapter 9

Conclusion

In this work, we tried to clarify, formalize, and to a large extend solve the problems raised by loops, that is to say iterated executions of the same piece of code, in Esterel. We view this as an important step to achieve provably correct, efficient code generation, for both hardware and software target codes.

9.1 Results

Our research resulted in the following contributions:

- a revised logical semantics for Esterel [Tar04a]: the *deterministic semantics*, and a refined correctness criterion for Esterel programs: *properness* (cf. Chapter 3).
- an extended language [Tar04b]: *Esterel**, adding to Esterel a new primitive instruction: `gotopause` (cf. Chapter 6), and *static aliasing* capabilities (cf. Chapter 8).
- a preprocessing (cf. Chapter 7) to take care of all complex aspects of loops, by means of *static analysis techniques* [TdS03] and *program rewriting techniques* [TdS04], which we have then been able to prove correct, thanks to the formal definitions of *instantaneous loops* (cf. Chapter 4) and *schizophrenia* (cf. Chapter 5) provided here.
- an implementation of these preprocessing techniques and language extensions, with algorithms directly derived from their specifications (cf. Chapter 8).

We sum up the main features of interest in each of these topics below.

A Revised Semantics

The deterministic semantics introduced in this work, and its sibling reactive deterministic semantics, have a local and "pessimistic" approach to error detection in programs, whereas the approach of the logical behavioral semantics is global and "optimistic", in that our semantics reports errors as soon as pieces of code may be "wrong", while the reference semantics refrains from reporting errors as long as long pieces of code may be "correct". In our view, this stricter approach is more in line with what is required from a language for safety-critical application design.

Technically, our semantics eliminates the non-deterministic deduction rules of the logical behavioral semantics. As a result, formal reasoning about program behaviors becomes much easier, even for incorrect program behaviors, thanks to the reactive deterministic semantics. This was not the case with the constructive semantics [Ber99], which takes care of non-determinism as

```
1:pause;
action 1;
present A then gotopause 2 end;
present B then gotopause 3 end;
gotopause 1;

2:pause;
action 2;
gotopause 1;

3:pause;
action 3;
gotopause 2
```

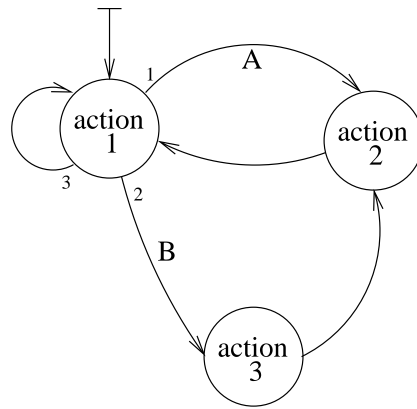


Figure 9.1: A Finite State Machine in Esterel*

well, but at the expense of a much more complex formalism, corresponding to a more ambitious goal: constructive causality analysis. It is in fact because we could separate concerns and achieve determinism while ignoring constructive causality issues, that we could provide our semantics. As a result, these semantics should not be seen as competitors but rather complementary formalisms, serving different purposes.

An Extended Language

It is broadly agreed that `goto` statements in programming languages are "harmful" [Dij68]. They are, however, at the heart of code generation processes (both software and hardware), whose job is (amongst other things) to flatten the structure of the program. By incorporating a jump instruction in the Esterel language itself, we aim at specifying a versatile intermediate format for the description of Esterel programs. We shall not, however, advocate for the usage of `gotopause` instructions in hand-written code.

Esterel* is currently the last of a long series of intermediate formats defined for the encoding of Esterel programs, (`oc` [Tan85], `ic` [Gon88, Par92], `sc` [For95], `GRC` [Pot02], etc.). What makes it very attractive in our view, is the following collection of properties:

- Esterel* extends Esterel. Thanks to `gotopause`, important structures can be now efficiently described using source code. For instance, finite state machines can be directly encoded in Esterel*, as illustrated in Figure 9.1. This is especially useful, since graphical design languages built on top of Esterel, such as SyncCharts [And95, And96], heavily rely on such representations of behaviors, and therefore can be more easily compiled to Esterel* than Esterel.

- Esterel* preserves high-level structural information about programs. In particular, it is possible to easily and efficiently compile Esterel* programs into `GRC` specifications. Since, as argued in [Pot02], `GRC` can be used for efficient software generation and hardware synthesis, then it must be so for Esterel*.

- In practice, the semantics of `gotopause` is very intuitive. Because jumps are required to be non-instantaneous (i.e. target `pause` instructions only), they cannot contribute to instantaneous loops, schizophrenia, or even causality cycles. This makes compilers for Esterel* very easy to implement. On the one hand, implementing `gotopause` is straightforward; on the other hand, compilers can assume that schizophrenia has been taken care of already. In particular, the unfolded programs we produce in Chapter 7, can be linearly translated into Boolean equation systems (i.e. `sc` code, cf. Chapter 8).

- Esterel* makes it possible to seamlessly evolve from high-level program descriptions down to lower-level representations, using efficient, provably correct program transformations, as illustrated in this work with reincarnation.

A Provably Correct Preprocessing

In our view, the variety of techniques currently in use to deal with instantaneous loops and schizophrenia reveals how difficult these issues are, from the formal point of view, as well as the implementation point of view. In this work, we formalized these two problems, and provided practical algorithms to deal with them. Using static analysis techniques we identified both potentially instantaneous loops and potentially schizophrenic constructs in programs. Then, using program rewriting techniques, we transformed schizophrenic Esterel programs into non-schizophrenic observationally equivalent Esterel* programs.

Thanks to our complete formalization of the various pieces of this puzzle (formal semantics, formal definitions of instantaneous loops and schizophrenia, formal specification of the static analyses and program transformations), we are able to complete a hand-written proof of the correctness of this preprocessor for the pure Esterel language. More precisely, we established:

- the correct rejection of loop unsafe programs,
- the correct preservation of the semantics of safe (i.e. accepted) programs,
- the correct transformation of safe programs into non-schizophrenic programs.

Implementation

We have implemented these algorithms as a standalone preprocessor. Because it outputs Esterel* programs, it can be used for software as well as hardware synthesis, even in existing compilers. As argued before, this should hopefully require only a minimal effort, consisting in implementing `gotopause` in existing compilers. In other words, these compilers could profit from an optimized expansion of schizophrenia, virtually for free. On the other hand, new Esterel compilers should no longer worry about schizophrenia.

To validate our approach, we have also implemented a prototype Esterel* compiler using Berry's circuit synthesis for non-schizophrenic programs [Ber99], which we adapted to Esterel*.

9.2 Future Work

There are many extensions to this work we would like to consider, concerning proofs, schizophrenia and reincarnation, as well as various program analyses and transformations in Esterel*.

Proofs

First, our proofs should be checked using a theorem prover/proof assistant, which obviously means an important effort to encode the models in a way that can be presented to the proof assistant. But this would bring us one (big) step closer to the formal verification of an optimizing compiler for Esterel.

Second, the definitions, results, and proofs concerning the reactive deterministic semantics and loop safety have to be extended from Esterel to Esterel*. This should be straightforward.

Finally, while we believe that extending compiler implementations with `gotopause` is easy, adapting the proof of a certified compiler for Esterel to Esterel* is not so simple. We would like to extend the work of Schneider et al. [Sch01a, Sch01b, SW01, SBS04] on certified Boolean equation synthesis for Esterel, to the Esterel* language. This again is a required step toward the embedding of our techniques in a certified compiler.

Schizophrenia

Our reincarnation algorithm could be further improved. In particular, as remarked in Chapter 7, it generates dead code. While this has no impact in the end, as dead code is eliminated by subsequent code generation phases, it would be nice to avoid generating dead code in the first place. More importantly, there are many practical situations where schizophrenic constructs do no harm. For example, the following code pattern is fairly common:

```
loop
  signal S in
    await I;
    ...
  end
end
```

which expands into:

```
loop
  signal S in
    trap T in
      loop
        pause;
        present I then exit T end;
      end
    end;
    ...
  end
end
```

Here, while S is schizophrenic in the sense of Chapter 5, two instances of S cannot be used simultaneously, because of the *pause* instruction inside the scope of S, occurring before any occurrence of S. Hence, unfolding could be avoided in this case. There are several such refinements, which we have already embedded in our implementation, but not yet formally specified and verified.

Our final rewriting scheme is limited to Esterel programs. We would like to extend it to handle Esterel* programs as well, that is to say `gotopause` instructions. While we have already experimented with such an extension (not presented here), we believe our current static analysis for Esterel* is not very good, and we are far from completing a correctness proof for the extended preprocessing. Further work is required.

Program Analyses and Transformations

In general, because in Esterel* we have taken care of schizophrenia by carefully setting apart the simultaneous incarnations of schizophrenic signals and parallel constructs, more accurate program analyses and transformation become possible. Computing distinct pieces of information for distinct incarnations of the same object is now easier. Moreover, different transformations can be applied to these incarnations.

There are good reasons to believe that this, for instance, makes it possible to specify, at the source level, a much better dead code elimination than before. We would like to formalize this analysis/transformation, in the provably correct kind of approach we adopted in this work, as well as investigate other program analyses and transformations.

In particular, the last and most speculative direction that we would like to mention here is constructive causality analysis. Constructive causality analysis is typically the most complex and computationally expensive piece of code in hardware compilers for Esterel. As discussed before, this led us to abstract away constructive causality in this work, by separating schizophrenia and constructive causality issues in a safe manner (i.e. considering schizophrenia for all logically correct programs). However, just as we transposed in this work the analysis and transformation algorithms existing at the Boolean equation level for schizophrenia to the source level with the definition of Esterel*, we would like to similarly investigate constructive causality analysis with source-level techniques, thus pursuing the work of Potop-Butucaru [Pot02] on constructive causality in the `GRC` intermediate format.

Appendix A

Proof of Theorem 7.21

In Chapter 7, we define a rewriting of safe Esterel programs – *expand* – prove it preserves the semantics of programs – p and $expand(p)$ are observationally equivalent (cf. Corollary 7.20) – and claim it cures schizophrenia – $expand(p)$ is not schizophrenic (cf. Theorem 7.21). We now formally establish this second result.

In Figure A.1, we define the function of contexts: $\langle\ \rangle : C[\ ] \mapsto \langle C[\ ]\rangle$, which mimics the definition of the risk function of Figure 5.4, except for the fact it does not raise exception STOP. As a result, it can be a function of $C[\,]$ only whereas the initial risk function had two parameters: $C[\,]$ and p.

In Figure A.2, we rewrite the definition of *dup4* of Figure 7.5 using the $\langle\ \rangle$ function we just introduced. This makes the connection of the computation of K in the initial definition of *dup4* and the risk function explicit, leading to easier proofs.

Moreover, we take tags into account in this new definition of *dup4*. For now on, we suppose that function *surf* tags statements with fresh tags, thus for instance:

$$\begin{aligned}&dup4(\langle[\,]\rangle, \texttt{loop}^1\ \texttt{signal}^2\ S\ \texttt{in pause}^3\ \texttt{end end}) = \\ &\quad \texttt{loop}^1\ \texttt{signal}^x\ S\ \texttt{in pause}^y\ \texttt{end;}^z\ \texttt{signal}^2\ S\ \texttt{in pause}^3\ \texttt{end end}\end{aligned}$$

where x, y, and z can be any integers such that $\{1, 2, 3, x, y, z\}$ are pairwise distinct. The precise values are irrelevant. In general, if p is well tagged (pairwise distinct tags) then $dup4(\langle[\,]\rangle, p)$ is well tagged as well. Thanks to these tags we shall easily identify subterms in $dup4(\langle[\,]\rangle, p)$.

We shall prove that:

Theorem A.1. *$dup4(\langle[\,]\rangle, p)$ is not schizophrenic, whatever p.*

Let us choose p safe and well tagged. For simplicity, we shall only consider signal declarations. Parallel statements can be dealt with using a similar approach. We shall thus prove that no signal declaration of $dup4(\langle[\,]\rangle, p)$ can be reduced twice in a single reaction:

Theorem A.2. *If* $\begin{cases} dup4(\langle[\,]\rangle, p) = C[\texttt{signal}^n\ S\ \texttt{in}\ q\ \texttt{end}] \\ \lceil dup4(\langle[\,]\rangle, p)|L\rfloor \xrightarrow[I]{O,k,M}_{\diamond} \lceil dup4(\langle[\,]\rangle, p)|L'\rfloor \end{cases}$ *then* $\{n, n\} \not\subset M$.

If $dup4(\langle[\,]\rangle, p) = C[\texttt{signal}^n\ S\ \texttt{in}\ q\ \texttt{end}]$ then one of the following holds:

- n is the tag of a signal declaration of p:
  there exists $C_0[\ ]$ and q_0 such that $p = C_0[\texttt{signal}^n\ S\ \texttt{in}\ q_0\ \texttt{end}]$.

- n is not a tag of p (i.e. n is a fresh tag):
  there exists $C_0[\,]$, q_0, and m such that $\begin{cases} p = C_0[\texttt{signal}^m\ S\ \texttt{in}\ q_0\ \texttt{end}] \\ \texttt{signal}^n\ S\ \texttt{in}\ q\ \texttt{end} = surf(\texttt{signal}^m\ S\ \texttt{in}\ q_0\ \texttt{end}) \end{cases}$

$$
\begin{array}{rcl}
\langle [\,] \rangle & \stackrel{def}{=} & \emptyset \\
\langle C[\mathtt{present}\ S\ \mathtt{then}\ [\,]\ \mathtt{else}\ q\ \mathtt{end}] \rangle & \stackrel{def}{=} & \langle C[\,] \rangle \\
\langle C[\mathtt{present}\ S\ \mathtt{then}\ p\ \mathtt{else}\ [\,]\ \mathtt{end}] \rangle & \stackrel{def}{=} & \langle C[\,] \rangle \\
\langle C[\mathtt{loop}\ [\,]\ \mathtt{end}] \rangle & \stackrel{def}{=} & \{0\} \cup \langle C[\,] \rangle \\
\langle C[[\,]\mathtt{;}\ q] \rangle & \stackrel{def}{=} & \text{if } \Gamma_q \cap \langle C[\,] \rangle = \emptyset \text{ then } \langle C[\,] \rangle \backslash \{0\} \text{ else } \langle C[\,] \rangle \cup \{0\} \\
\langle C[p\mathtt{;}\ [\,]] \rangle & \stackrel{def}{=} & \text{if } 0 \in \Gamma_p \text{ then } \langle C[\,] \rangle \text{ else } \emptyset \\
\langle C[\mathtt{trap}\ T\ \mathtt{in}\ [\,]\ \mathtt{end}] \rangle & \stackrel{def}{=} & \{k \in \mathbb{N}, \downarrow k \in \langle C[\,] \rangle\} \\
\langle C[\mathtt{signal}\ S\ \mathtt{in}\ [\,]\ \mathtt{end}] \rangle & \stackrel{def}{=} & \emptyset \\
\langle C[[\,]\ \mathtt{||}\ q] \rangle & \stackrel{def}{=} & \emptyset \\
\langle C[p\ \mathtt{||}\ [\,]] \rangle & \stackrel{def}{=} & \emptyset
\end{array}
$$

Figure A.1: Risk (Revised)

$$
\begin{array}{rl}
\mathit{dup4}(\langle C[\,] \rangle, \mathtt{nothing}^n) \stackrel{def}{=} & \mathtt{nothing}^n \\
\mathit{dup4}(\langle C[\,] \rangle, \mathit{label}\mathtt{:pause}^n) \stackrel{def}{=} & \mathit{label}\mathtt{:pause}^n \\
\mathit{dup4}(\langle C[\,] \rangle, \mathtt{exit}^n\ T) \stackrel{def}{=} & \mathtt{exit}^n\ T \\
\mathit{dup4}(\langle C[\,] \rangle, \mathtt{emit}^n\ S) \stackrel{def}{=} & \mathtt{emit}^n\ S \\
\mathit{dup4}(\langle C[\,] \rangle, \mathtt{present}^n\ ...\ \mathtt{end}) \stackrel{def}{=} & \left[\begin{array}{l}
\mathtt{present}^n\ S\ \mathtt{then} \\
\quad \mathit{dup4}(\langle C[\mathtt{present}\ S\ \mathtt{then}\ [\,]\ \mathtt{else}\ q\ \mathtt{end}] \rangle, p) \\
\mathtt{else} \\
\quad \mathit{dup4}(\langle C[\mathtt{present}\ S\ \mathtt{then}\ p\ \mathtt{else}\ [\,]\ \mathtt{end}] \rangle, q) \\
\mathtt{end}
\end{array}\right] \\
\mathit{dup4}(\langle C[\,] \rangle, \mathtt{loop}^n\ p\ \mathtt{end} \stackrel{def}{=} & \mathtt{loop}^n\ \mathit{dup4}(\langle C[\mathtt{loop}\ [\,]\ \mathtt{end}] \rangle, p)\ \mathtt{end} \\
\mathit{dup4}(\langle C[\,] \rangle, \mathtt{trap}^n\ T\ \mathtt{in}\ p\ \mathtt{end} \stackrel{def}{=} & \mathtt{trap}^n\ T\ \mathtt{in}\ \mathit{dup4}(\langle C[\mathtt{trap}\ T\ \mathtt{in}\ [\,]\ \mathtt{end}] \rangle, p)\ \mathtt{end} \\
\mathit{dup4}(\langle C[\,] \rangle, p\mathtt{;}^n\ q) \stackrel{def}{=} & \mathit{dup4}(\langle C[[\,]\mathtt{;}\ q] \rangle, p)\mathtt{;}^n\ \mathit{dup4}(\langle C[p\mathtt{;}\ [\,]] \rangle, q) \\
\mathit{dup4}(\langle C[\,] \rangle, \mathtt{signal}^n\ S\ \mathtt{in}\ p\ \mathtt{end}) \stackrel{def}{=} & \left[\begin{array}{ll}
\text{if} & K \cap \Omega_p = \emptyset \\
\text{then} & \mathtt{signal}^n\ S\ \mathtt{in}\ \mathit{dup4}(\langle C[\mathtt{signal}\ S\ \mathtt{in}\ [\,]\ \mathtt{end}] \rangle, p)\ \mathtt{end} \\
\text{else} & \left[\begin{array}{l}
\mathit{surf}(\mathtt{signal}\ S\ \mathtt{in}\ p\ \mathtt{end})\mathtt{;}^{\mathit{(new\ tag)}} \\
\mathit{skip}(\mathtt{signal}^n\ S\ \mathtt{in}\ \mathit{dup4}(\langle C[\mathtt{signal}\ S\ \mathtt{in}\ [\,]\ \mathtt{end}] \rangle, p)\ \mathtt{end})
\end{array}\right]
\end{array}\right] \\
\mathit{dup4}(\langle C[\,] \rangle, p\ \mathtt{||}^n\ q) \stackrel{def}{=} & \left[\begin{array}{ll}
\text{if} & K \cap \Omega_{p\ \mathtt{||}\ q} = \emptyset \\
\text{then} & \mathit{dup4}(\langle C[[\,]\ \mathtt{||}\ q] \rangle, p)\ \mathtt{||}^n\ \mathit{dup4}(\langle C[p\ \mathtt{||}\ [\,]] \rangle, q) \\
\text{else} & \left[\begin{array}{l}
\mathit{surf}(p\ \mathtt{||}\ q)\mathtt{;}^{\mathit{(new\ tag)}} \\
\mathit{skip}(\mathtt{[}\mathit{dup4}(\langle C[[\,]\ \mathtt{||}\ q] \rangle, p)\ \mathtt{||}^n\ \mathit{dup4}(\langle C[p\ \mathtt{||}\ [\,]] \rangle, q)\mathtt{]})
\end{array}\right]
\end{array}\right]
\end{array}
$$

Figure A.2: *dup4* (Revised)

Let us focus on the first case; the second one is similar:

Lemma A.3. *If* $C_0[\,] = X[Y[\,]]$ *then:*

- *"*$\texttt{signal}^n\ S\ \texttt{in}\ q\ \texttt{end}$*" is not schizophrenic in "*$dup4(\langle X[\,]\rangle, Y[\texttt{signal}^n\ S\ \texttt{in}\ q_0\ \texttt{end})]$*":*

  if $\lceil dup4(\langle X[\,]\rangle, Y[\texttt{signal}^n\ S\ \texttt{in}\ q_0\ \texttt{end})]|L\rfloor \overset{O,k,M}{\underset{I}{\diamond\!\!\longrightarrow}} s$ *then* $\{n,n\} \not\subset M$.

- *if "*$Y[\texttt{signal}^n\ S\ \texttt{in}\ q_0\ \texttt{end}]$*" may terminate with a risky completion code w.r.t. the context "*$X[\,]$*" then "*$\texttt{signal}^n\ S\ \texttt{in}\ q_0\ \texttt{end}$*" is not reducible in any reaction of the initial state* $\lceil dup4(\langle X[\,]\rangle, Y[\texttt{signal}^n\ S\ \texttt{in}\ q_0\ \texttt{end})])|*\rfloor$*, that is to say:*

  $$\text{if } \left\{\begin{matrix} \Omega_{Y[\texttt{signal}^n\ S\ \texttt{in}\ q_0\ \texttt{end}]} \cap \langle X[\,]\rangle \neq \emptyset \\ \lceil dup4(\langle X[\,]\rangle, Y[\texttt{signal}^n\ S\ \texttt{in}\ q_0\ \texttt{end})])|*\rfloor \overset{O,k,M}{\underset{I}{\diamond\!\!\longrightarrow}} L \end{matrix}\right\} \text{ then } n \notin M.$$

Proof. By structural induction on Y[ ]. The second item makes the induction possible. While finding the precise expression of this item is non-trivial, the induction is easy. □

With $X = [\,]$ and $Y = C_0[\,]$, we obtain: if $\lceil dup4(\langle[\,]\rangle, p|L\rfloor \overset{O,k,M}{\underset{I}{\diamond\!\!\longrightarrow}} s$ then $\{n,n\} \not\subset M$.
In other words, "$\texttt{signal}^n\ S\ \texttt{in}\ q\ \texttt{end}$" is not schizophrenic in $expand(p)$. □

Bibliography

[And95] Charles André. SyncCharts: A visual representation of reactive behaviors. RR 95-52, I3S, 1995.

[And96] Charles André. Representation and analysis of reactive behaviors: A synchronous approach. In *Proc. CESA'96*, pages 19–29. IEEE-SMC, 1996.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley, 1986.

[Bar81] Henk P. Barendregt. *The Lambda Calculus, Its Syntax and Semantics*, volume 103 of *Studies in Logics and the Foundations of Mathematics*. North-Holland, 1981.

[BB91] Albert Benveniste and Gérard Berry. The synchronous approach to reactive real-time systems. *Another Look at Real Time Programming, Proceedings of the IEEE, Special Issue*, 79(9):1270–1282, 1991.

[BC84] Gérard Berry and Laurent Cosserat. The synchronous programming language Esterel and its mathematical semantics. In *Seminar on Concurrency*, volume 197 of *Lecture Notes in Computer Science*, pages 389–448. Springer-Verlag, 1984.

[BCD+88] Patrick Borras, Dominique Clément, Thierry Despeyroux, Janet Incerpi, Gilles Kahn, Bernard Lang, and Valérie Pascual. Centaur: the system. In *Proc. SDE'88*, pages 14–24. ACM Press, 1988.

[BCE+03] Albert Benveniste, Paul Caspi, Stephen Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone. The synchronous languages twelve years later. *Proceedings of the IEEE, Special issue on Embedded Systems*, 91(1):64–83, 2003.

[BCPD99] Saddek Bensalem, Paul Caspi, Catherine Parent-Vigouroux, and Cécile Dumas Canovas. A methodology for proving control systems with Lustre and PVS. In *Proc. DCCA'99*, pages 89–107. IEEE Computer Society, 1999.

[BdS91] Frédéric Boussinot and Robert de Simone. The Esterel language. *Another Look at Real Time Programming, Proceedings of the IEEE, Special Issue*, 79(9):1293–1304, 1991.

[BdS95] Fredéric Boussinot and Robert de Simone. The SL synchronous language. RR 2510, INRIA, 1995.

[Ber89] Gérard Berry. Real-time programming: General purpose or special-purpose languages. In *Information Processing*, volume 89, pages 11–17. North-Holland, 1989.

[Ber90] Yves Bertot. Implementation of an interpreter for a parallel language in Centaur. In *Proc. ESOP'90*, volume 432 of *Lecture Notes in Computer Science*, pages 57–69, 1990.

[Ber92] Gérard Berry. Esterel on hardware. In *Mechanized reasoning and hardware design*, pages 87–104. Prentice-Hall, 1992.

[Ber93a] Gérard Berry. Preemption and concurrency. In *Proc. FSTTCS'93*, volume 761 of *Lecture Notes in Computer Science*, pages 72–93. Springer-Verlag, 1993.

[Ber93b] Gérard Berry. The semantics of pure Esterel. In *Program Design Calculi*, volume 118 of *Series F: Computer and System Sciences*, pages 361–409. NATO ASI Series, 1993.

[Ber99] Gérard Berry. *The Constructive Semantics of Pure Esterel*. INRIA, 1999. http://www-sop.inria.fr/esterel.org/.

[Ber00a] Gérard Berry. *The Esterel Language Primer v5_91*. INRIA, 2000. http://www-sop.inria.fr/esterel.org/.

[Ber00b] Gérard Berry and the Esterel Team. *The Esterel v5_92 Compiler*. INRIA, 2000. http://www-sop.inria.fr/esterel.org/.

[BG92] Gérard Berry and Georges Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Sci. of Comput. Program.*, 19(2):87–152, 1992.

[BH01] Sylvain Boulmé and Grégoire Hamon. Certifying synchrony for free. In *Proc. LPAR'01*, volume 2250 of *Lecture Notes in Computer Science*, pages 495–506. Springer-Verlag, 2001.

[Bou98] Amar Bouali. XEVE, an esterel verification environment. In *Proc. CAV'98*, volume 1427 of *Lecture Notes in Computer Science*, pages 500–504. Springer-Verlag, 1998.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. POPL'77*. ACM Press, 1977.

[CDT01] The Coq Development Team. *The Coq Proof Assistant Reference Manual Version 7.2*. INRIA, 2001. http://coq.inria.fr/doc-eng.html.

[Cha85] Daniel M. Chapiro. *Globally-asynchronous locally-synchronous systems (performance, reliability, digital)*. PhD thesis, Stanford University, 1985.

[CI89] Dominique Clément and Janet Incerpi. Programming the behavior of graphical objects using Esterel. In *Proc. TAPSOFT'89*, volume 352 of *Lecture Notes in Computer Science*. Springer-Verlag, 1989.

[CP99] Paul Caspi and Marc Pouzet. Lucid Synchrone: une extension fonctionnelle de Lustre. In *Proc. JFLA'99*. INRIA, 1999.

[CPP+02] Etienne Closse, Michel Poize, Jacques Pulou, Patrick Venier, and Daniel Weil. Saxo-RT: Interpreting Esterel semantics on a sequential execution structure. In *Proc. SLAP'02*, volume 65 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2002.

[Dav03] Maulik A. Dave. Compiler verification: a bibliography. *SIGSOFT Softw. Eng. Notes*, 28(6):2–2, 2003.

[dB72] Nicolaas G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation. *Indag. Math.*, 34:381–392, 1972.

[Dij68] Edsger W. Dijkstra. Go To statement considered harmful. *Comm. ACM*, 11(3):147–148, 1968. letter to the Editor.

[Dou97] Bruce P. Douglass. *Real-Time UML: Developing Efficient Objects for Embedded Systems*. Addison-Wesley, 1997.

[Edw00] Stephen A. Edwards. *Languages for Digital Embedded Systems*. Kluwer Academic Publishers, 2000.

[EKH04] Stephen A. Edwards, Vimal Kapadia, and Michael Halas. Compiling Esterel into static discrete-event code. In *Proc. SLAP'04*, *Electronic Notes in Theoretical Computer Science*. Elsevier, 2004.

[ET03] Esterel Technologies. *Esterel Studio 5.0*, 2003. http://www.esterel-technologies.com/.

[For95] Xavier Fornari. *Optimisation du contrôle et implantation en circuits de programmes Esterel*. PhD thesis, Ecole des Mines de Paris, 1995.

[GBGM91] Paul Le Guernic, Michel Le Borgne, Thierry Gauthier, and Claude Le Maire. Programming real time applications with Signal. *Another Look at Real Time Programming, Proceedings of the IEEE, Special Issue*, 79(9), 1991.

[Gla94] David S. Gladstein. *Compiler correctness for concurrent languages*. PhD thesis, Northeastern University, 1994.

[GLS99] Thierry Grandpierre, Christophe Lavarenne, and Yves Sorel. Optimized rapid prototyping for real-time embedded heterogeneous multiprocessors. In *Proc. CODES'99*, pages 74–78. ACM Press, 1999.

[GM93] Michael J. C. Gordon and Tom F. Melham. *Introduction to HOL: a theorem proving environment for higher order logic*. Cambridge University Press, 1993.

[Gon88] Georges Gonthier. *Sémantique et modèles d'exécution des langages réactifs synchrones: application à Esterel*. PhD thesis, Université d'Orsay, 1988.

[Gro90] Jan F. Groote. Transition system specifications with negative premises. In *Proc. CONCUR'90*, volume 458 of *Lecture Notes in Computer Science*, pages 332–341. Springer-Verlag, 1990.

[Hal93] Nicolas Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer Academic Publishers, 1993.

[Har87] David Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, 1987.

[HCRP91] Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous dataflow programming language Lustre. *Another Look at Real Time Programming, Proceedings of the IEEE, Special Issue*, 79(9):1305–1320, 1991.

[HLR92] Nicolas Halbwachs, Fabienne Lagnier, and Christophe Ratel. Programming and verifying real-time systems by means of the synchronous data-flow language Lustre. *IEEE Trans. Softw. Eng.*, 18(9):785–793, 1992.

[HP85] David Harel and Amir Pnueli. On the development of reactive systems. In *Logics and models of concurrent systems*, pages 477–498. Springer-Verlag, 1985.

[IEE94] IEEE Standard. *VHDL Language Reference Manual. IEEE Std. 1076-1993.*, 1994.

[Kah87] Gilles Kahn. Natural semantics. In *Proc. STACS'87*, volume 247 of *Lecture Notes in Computer Science*, pages 22–39. Springer-Verlag, 1987.

[Kle56] Stephen C. Kleene. Representation of events in nerve nets and finite automata. In *Automata Studies*, pages 3–41. Princeton University Press, 1956.

[KNT00] Mickaël Kerboeuf, David Nowak, and Jean-Pierre Talpin. Specification and verification of a steam-boiler with Signal-Coq. In *Proc. TPHOLs'00*, volume 1869 of *Lecture Notes in Computer Science*, pages 356–371. Springer-Verlag, 2000.

[Ler02] Xavier Leroy. *The Objective Caml system, release 3.06*, 2002. http://pauillac.inria.fr/ocaml/.

[LGT98] Agnès Lanusse, Sébastien Gérard, and François Terrier. Real-time modeling with UML: The ACCORD approach. In *Proc. UML'98*, volume 1618 of *Lecture Notes in Computer Science*, pages 319–335. Springer-Verlag, 1998.

[Mar91] Florence Maraninchi. The Argos language: Graphical representation of automata and description of reactive systems. In *Proc. IEEE Conf. on Visual Languages*, 1991.

[Mig94] Frédéric Mignard. *Compilation du langage Esterel en systèmes d'équations booléennes.* PhD thesis, Ecole des Mines de Paris, 1994.

[Mil89] Robert Milner. *Communication and Concurrency.* Series in Computer Science. Prentice-Hall, 1989.

[Mor02] Lionel Morel. Efficient compilation of array iterators for Lustre. In *Proc. SLAP'02*, volume 65 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2002.

[MP67] John L. McCarthy and James Painter. Correctness of a compiler for arithmetic expressions. In *Proceedings Symposium in Applied Mathematics, Vol. 19, Mathematical Aspects of Computer Science*, pages 33–41. American Mathematical Society, 1967.

[MRBS01] Hervé Marchand, Eric Rutten, Michel Le Borgne, and Mazen Samaan. Formal verification of programs specified with Signal: Application to a power transformer station controller. *Sci. Comput. Program.*, 41(1):85–104, 2001.

[NBT98] David Nowak, Jean-René Beauvais, and Jean-Pierre Talpin. Co-Inductive Axiomatization of a Synchronous Language. In *Proc. TPHOLs'98*, volume 1479 of *Lecture Notes in Computer Science*, pages 387–399. Springer-Verlag, 1998.

[Now99] David Nowak. *Spécification et preuve de systèmes réactifs.* PhD thesis, Université de Rennes, 1999.

[OMG03] Object Management Group. *Unified Modeling Language Specification, Version 1.5*, 2003.

[Pal92] Jens Palsberg. A provably correct compiler generator. In *Proc. ESOP'92*, volume 582 of *Lecture Notes in Computer Science*, pages 418–434. Springer-Verlag, 1992.

[Par92] Jean-Pierre Paris. *Exécution de tâches asynchrones depuis Esterel.* PhD thesis, Université de Nice, 1992.

[Plo81] Gordon Plotkin. A structural approach to operational semantics. Report DAIMI FN-19, Aarhus University, 1981.

[PMM+98] Axel Poigné, Matthew Morley, Olivier Maffeïs, Leszek Holenderski, and Reinhard Budde. The synchronous approach to designing reactive systems. *Form. Methods Syst. Des.*, 12(2):163–187, 1998.

[Pol81] Wolfgang Polak. *Compiler Specification and Verification.* Springer-Verlag, 1981.

[Pot02] Dumitru Potop-Butucaru. *Optimizations for Faster Execution of Esterel Programs.* PhD thesis, Ecole des Mines de Paris, 2002.

[PSS98a] Amir Pnueli, Michael Siegel, and Eli Singerman. Translation validation. In *Proc. TACAS'98*, volume 1384 of *Lecture Notes in Computer Science*, pages 151–166. Springer-Verlag, 1998.

[PSS98b] Amir Pnueli, Ofer Strichman, and Michael Siegel. The code validation tool CVT: Automatic verification of a compilation process. *International Journal on Software Tools for Technology Transfer*, 2(2):192–201, 1998.

[PSS99] Amir Pnueli, Ofer Strichman, and Michael Siegel. Translation validation: From Signal to C. In *Correct System Design, Recent Insight and Advances*, volume 1710 of *Lecture Notes in Computer Science*, pages 231–255. Springer-Verlag, 1999.

[SBS04] Klaus Schneider, Jens Brandt, and Tobias Schüle. A verified compiler for synchronous programs with local declarations. In *Proc. SLAP'04*, *Electronic Notes in Theoretical Computer Science*. Elsevier, 2004.

[Sch01a] Klaus Schneider. Embedding imperative synchronous languages in interactive theorem provers. In *Proc. ACSD'01*, page 143. IEEE Computer Society, 2001.

[Sch01b] Klaus Schneider. A verified hardware synthesis of Esterel programs. In *Proc. DIPES'00*, pages 205–214. Kluwer, B.V., 2001.

[SORS99] Natarajan Shankar, Sam Owre, John M. Rushby, and Dave W. J. Stringer-Calvert. *PVS Prover Guide.* Computer Science Laboratory, SRI International, 1999.

[SR98] Bran Selic and Jim Rumbaugh. Using UML for modeling complex real-time systems. Technical report, ObjecTime Limited, 1998.

[STB96] Ellen M. Sentovich, Horia Toma, and Gérard Berry. Latch optimization in circuits generated from high-level descriptions. In *Proc. ICCAD'96*, pages 428–435. IEEE Computer Society, 1996.

[Ste93] Susan Stepney. *High Integrity Compilation: A Case Study.* Prentice-Hall, 1993.

[Sto96] Neil R. Storey. *Safety-Critical Computer Systems.* Addison-Wesley, 1996.

[SW01] Klaus Schneider and Michael Wenz. A new method for compiling schizophrenic synchronous programs. In *Proc. CASES'01*, pages 49–58. ACM Press, 2001.

[Tan85] Jean-Marc Tanzi. *Traduction Structurelle des Programmes Esterel en Automates.* PhD thesis, Université de Nice, 1985.

[Tar04a] Olivier Tardieu. A deterministic logical semantics for Esterel. In *Proc. SOS Workshop'04*, *Electronic Notes in Theoretical Computer Science*. Elsevier, 2004.

[Tar04b] Olivier Tardieu. Goto and concurrency: Introducing safe jumps in Esterel. In *Proc. SLAP'04*, *Electronic Notes in Theoretical Computer Science*. Elsevier, 2004.

[TdS03] Olivier Tardieu and Robert de Simone. Instantaneous termination in pure Esterel. In *Proc. SAS'03*, volume 2694 of *Lecture Notes in Computer Science*, pages 91–108. Springer-Verlag, 2003.

[TdS04] Olivier Tardieu and Robert de Simone. Curing schizophrenia by program rewriting in Esterel. In *Proc. MEMOCODE'04*. IEEE Press, 2004.

[Ter00] Delphine Terrasse. A first step towards the proof of correctness of the v5 Esterel compiler in Coq. RR 4092, INRIA, 2000.

Index

bisimulation, 49

completion code, 37
  risky completion code, 88

context, 45

environment, 32

exception, 29
  depth, 33

execution, 37

function δ, 39

function ϵ, 97

function Γ, 70, 103

function Ω, 87

instant, 30

label, 93
  active label, 94

loop, 31
  potentially instantaneous loop, 70
  proper loop, 62

observational equivalence, 49

occurrence, 45
  compatible occurrences, 96
  dead occurrence, 49
  exclusive occurrences, 96
  instantly reentered occurrence, 82
  reduced occurrence, 48
  reducible occurrence, 48

position, 45

program, *see* statement

reaction, 30, 37

relevant deduction rule, 48

residual, 37

risk, 88

signal, 29
  input signal, 30, 32
  input+output signal, 30, 34
  inputoutput signal, 36
  local signal, 32
  output signal, 30, 32
  proper signal declaration, 62
  schizophrenic signal declaration, 84
  signal coherence law, 35
  signal status, 32

state, 94
  active state, 96
  initial state, 96
  reachable state, 96
  state expansion, 97
  valid state, 97

statement, 29
  bisimilar statements, 49
  correct statement, 44
  deterministic statement, 44
  instantaneous statement, 69
  logically correct statement, 44
  loop-safe statement, 66
  proper statement, 54
  reachable statement, 44
  reactive statement, 44
  safe statement, 75, 103
  schizophrenic statement, 84
  strongly correct statement, 44
  strongly deterministic statement, 44
  well-formed statement, 101
  well-labeled statement, 94, 101
  well-tagged statement, 46

subterm, *see* occurrence

tag, 46

Abstract

Esterel is an imperative concurrent design language for the specification of control-oriented reactive systems. Based on the synchronous paradigm, its semantics relies on a clear distinction of instants of computation. All primitive instructions of the language but one “pause” instruction execute in zero time. Execution is thus a sequence of instantaneous computations separated by explicit pauses. Arbitrary loops in this context are troublesome, potentially leading to a non-termination problem or a schizophrenia issue: first, instantaneous loops may prevent the instant to end; second, program blocks may be traversed several times within the same instant, thus having a “schizophrenic” behavior. Instantaneous loops are forbidden by the semantics. Such errors have to be anticipated, and programs rejected by compilers on this behalf. Moreover, efficient code generation for schizophrenic program patterns is complex. While many existing compilers already generate correct code for loops, the efficient implementations available today are neither generic (i.e. target-independent) nor formally specified or verified.

In this work, we thoroughly consider loops in Esterel, starting from the operational semantics of the language, all the way down to a provably correct implementation. We formally characterize the related issues and define efficient static analysis techniques to detect them in Esterel code. In order to get rid of schizophrenic behaviors by source-to-source rewriting – cure schizophrenia – we introduce in Esterel a new primitive instruction, which we call “gotopause”. It behaves as a non-instantaneous jump instruction compatible with concurrency. We describe a first program transformation that systematically replaces loops by the mean of gotopause statements, providing a loop-free equivalent program for any correct Esterel program. By combining static analysis and rewriting techniques, we obtain a preprocessor for Esterel that rejects incorrect loops and cure schizophrenia, which we have implemented. Due to our source-to-source transformation methodology, our preprocessor is highly generic; because of static analysis, it is very efficient; thanks to our fully formalized approach, we could formally establish its correctness.

Keywords: synchronous languages, structural operational semantics, concurrency, static analysis, program transformation, correct-by-construction algorithms.

Résumé

Esterel est un langage impératif concurrent pour la programmation des systèmes réactifs. A l’exception de l’instruction “pause”, les primitives du langage s’exécutent sans consommer de temps logique. L’exécution se décompose donc en une suite d’instants. Dans ce contexte, les boucles peuvent poser deux types de problèmes: d’une part une boucle instantanée peut bloquer l’écoulement du temps; d’autre part un bloc de code peut être traversé plusieurs fois au cours du même instant, conduisant à un comportement du programme dit “schizophrène”. Les boucles instantanées sont proscrites par la sémantique. Elles doivent donc être détectées par les compilateurs et les programmes correspondants doivent être rejetés. Par ailleurs, la compilation efficace des programmes schizophrènes est difficile. Ainsi, alors que plusieurs compilateurs pour Esterel sont disponibles, les algorithmes employés pour compiler les boucles ne sont ni portables, ni formellement spécifiés, et encore moins prouvés.

Dans ce document, nous étudions les boucles en Esterel, établissant une correspondance formelle entre la sémantique opérationnelle du langage et l’implémentation concrète d’un compilateur. Après avoir spécifié les problèmes posés par les boucles, nous développons des techniques d’analyse statique efficaces pour les détecter dans un code Esterel quelconque. Puis, de façon à guérir la schizophrénie, c’est à dire transformer efficacement les programmes schizophrènes en programmes non schizophrènes, nous introduisons dans le langage une nouvelle primitive appelée “gotopause”. Elle permet de transférer le contrôle d’un point du programme à un autre de façon non instantanée, mais sans contrainte de localité. Elle préserve le modèle de concurrence synchrone d’Esterel. Nous décrivons un premier algorithme qui, en dépliant les boucles à l’aide de cette nouvelle instruction, produit pour tout programme Esterel correct un programme non schizophrène équivalent. Enfin, en combinant analyse statique et réécriture, nous obtenons un préprocesseur qui rejette les boucles instantanées et guérit la schizophrénie, à la fois portable et très efficace. Nous l’avons implémenté. De plus, grâce à une approche formelle de bout en bout, nous avons pu prouver la correction de ce préprocesseur.

Mots-clés : langages synchrones, sémantique opérationnelle structurelle, concurrence, analyse statique, transformation de programmes, algorithmes corrects par construction.