Logica de sesiune şi aplicaţiile sale în industria transportului feroviar

Kiss Tibor

Conducător de doctorat: Prof. Dr. Bazil Pârv

Facultatea de Matematică şi Informatică
Universitatea Babeş-Bolyai Cluj-Napoca
Str. Mihail Kogălniceanu, nr. 1 RO-400084 Cluj-Napoca

Doctorat

Informatică November 2016

Cuprins

1 Introducere **1**
- 1.1 Ipoteze și obiective . . . 3
- 1.2 Contribuții . . . 4
- 1.3 Lucrări conexe . . . 6
- 1.4 Structura tezei . . . 9
- 1.5 Publicații . . . 9

2 Noțiuni introductive **10**
- 2.1 Calculul π . . . 10
- 2.2 Tipuri de sesiune . . . 10
- 2.3 Logica de separare . . . 10
  - 2.3.1 Logica Hoare . . . 10
  - 2.3.2 Logica de separare secvențială . . . 10
  - 2.3.3 Logica de separare concurentă . . . 10
- 2.4 Sumar . . . 10

3 Logica de sesiune **11**
- 3.1 Modelarea proceselor in logica de sesiune . . . 15
  - 3.1.1 Calculul π asincron cu constrângeri logice . . . 15
- 3.2 SESSION-HIP . . . 15
  - 3.2.1 Sintaxa SESSION-HIP . . . 15
  - 3.2.2 Semantica operațională . . . 18
- 3.3 Principiul de verificare . . . 20
  - 3.3.1 Limbajul de specificație . . . 22
  - 3.3.2 Reguli de verificare . . . 23

4 Dovedirea corectitudinii **25**
- 4.1 Semantica operațională concurentă . . . 25
- 4.2 Dovedirea respectării protocolului . . . 25
- 4.3 Dovedirea corectitudinii . . . 25
- 4.4 Sumar . . . 25

5 SESSION-HIP-SLEEK **26**
- 5.1 Demonstratorul SESSION-SLEEK . . . 26
  - 5.1.1 Demonstrarea corectitudinii trimiteri și primiri . . . 27
  - 5.1.2 Demonstrarea corectitudini alegerii interne și externe . . . 31

5.2 SESSION-HIP . . . 32
5.2.1 Expresivitatea totală a logicii de separare . . . 32
5.2.2 Logica de sesiune de ordin superior . . . 32

6 Compararea logicii de sesiune cu abordări similare 33
6.1 SESSION-HIP-SLEEK vs Heap-Hop . . . 33
6.2 SESSION-HIP-SLEEK vs Session C . . . 33
6.3 SESSION-HIP-SLEEK vs ParTypes . . . 33
6.4 SESSION-HIP-SLEEK vs instrumente cu stări de tip . . . 33
6.4.1 SESSION-HIP-SLEEK vs MOOSE . . . 33
6.4.2 SESSION-HIP-SLEEK vs Session Java . . . 33
6.4.3 SESSION-HIP-SLEEK vs BICA . . . 33
6.5 Sumar . . . 33

7 Aplicații in industria de transport feroviar 36
7.1 Modelarea și verificarea Interlocking-ului utilizând logica de sesiune . . . 36
7.1.1 Introducere . . . 36
7.1.2 Un exemplu ilustrativ . . . 36
7.1.3 Codificarea cerințelor in logica de sesiune . . . 36
7.1.4 Verificare protocol . . . 36
7.1.5 Rezultate experimentale . . . 36
7.1.6 Concluzii . . . 36
7.2 Programe de protecție automată a trenurilor . . . 36
7.2.1 Rezultate experimentale . . . 36
7.3 Sumar . . . 36

8 Concluzii şi direcţii de cercetare 37
8.1 Direcții viitoare . . . 40

Bibliografie 41

Capitolul 1
Introducere

În ultimul deceniu am putut observa o infiltrare nemaivăzută a tehnologiei în viața noastră de zi cu zi. Drept rezultat, suntem din ce în ce mai dependenți de echipamentele controlate de programe și se pare că în viitorul apropiat această tendință va continua și echipamentele controlate de programe vor deveni o parte indispensabilă a vieții noastre.

Prin urmare, este esențial ca aceste echipamente să fie de încredere, în special în infrastructurile critice, cum ar fi spitalele, aeronautică, industria automobilelor și industria feroviară etc. Ca o consecință, programele care controlează aceste echipamente sunt foarte răspândite și au un rol important în viața noastră de zi cu zi. Prin urmare calitatea slabă a acestor programe este inacceptabilă.

Astfel, un mare efort s-a depus și se depune în dezvoltarea unor algoritmi și unelte de verificare și de validare care pot îmbunătăți semnificativ calitatea acestor aplicații. Această idee de modelare și verificare nu este nouă, dimpotrivă este foarte răspândită și se întrebuințează cu mare suces în restul disciplinelor inginerești.

În ciuda eforturilor depuse, în momentul de față cea mai frecventă metodă de înbunătățire a calității programelor industriale este testarea. Din păcate, această metodă nu este adecvată pentru excluderea erorilor dintr-un program ("testarea programului poate fi o modalitate foarte eficientă de a arăta prezența unor erori, dar este inadecvată pentru a arăta absența lor" [37]). Prin urmare nu este de mirare că până și sistemele care au fost testate extensiv pot eșua, ceea ce poate duce la pierderi financiare drastice sau în cel mai rău caz, la pierderi de vieți omenești [129, 7].

Drept consecință, în ultimele decenii s-au propus o mulțime de instrumente noi pentru soluționarea acestei probleme. De obicei acești algoritmi se bazează pe metode formale pentru a sprijini verificarea automată sau semiautomată a programelor industriale [1, 108, 26, 2, 30]. De-a lungul ultimilor ani, unele dintre aceste instrumente au înregistrat o creștere semnificativă în calitate [18, 6], prin urmare, sunt folosite cu un oarecare succes în industrie. În ciuda maturității lor, aceste instrumente au o mulțime de defecte și neajunsuri. În primul rând, aceste instrumente sunt concepute pentru a dezvolta programe industriale de la zero și nu pentru a sprijini menținerea și îmbunătățirea programelor utilizate momentan în industrie, scrise în limbaje de programare imperative. Mai mult, ele sunt dezvoltate pentru a modela programe care sunt executate secvențial, fără a suporta proiectarea programelor care trebuie să ruleze într-un mediu distribuit și paralel.

Ca o urmare, aceste instrumente nu vizează verificarea programelor paralele utilizate pentru controlul sistemelor mari, care în general sunt încă scrise în limbaje de programare

tradiționale și constituie o parte destul de largă a sistemelor de control moderne. Mai mult decât atât, problemele legate de paralelism sunt de obicei modelate ca specificații informale cu suport minimal sau chiar fără nici un suport pentru verificarea formală. Problema devine și mai gravă dacă luăm în considerare faptul că o mulțime de programe care controlează infrastructurile critice de mari dimensiuni (de exemplu: sistemele de energie electrică și de distribuție a gazelor naturale, infrastructuri de telecomunicații și sisteme care controlează și monitorizează infrastructura feroviară) au fost și încă sunt dezvoltate în principal în limbaje de programare imperative, construite de obicei pentru arhitecturi secvențiale fără mecanisme clare, care să permită verificarea în mod exhaustiv a corectitudinii acestor programe.

Considerând toate problemele de mai sus, putem observa că dezvoltarea unor metode formale care satisfac nevoile partenerului nostru industrial - adică divizia de automatizare a căilor ferate Siemens - este foarte dificilă. Problema devine și mai complexă, dacă luăm în considerare că clientul nostru utilizează paradigma geografică pentru a dezvolta programe pentru sisteme de interlocking. Merită remarcat faptul că aceste sisteme, în general, trebuie să fie concurente, distribuite și să fie capabile să controleze infrastructuri de mari dimensiuni (de exemplu, infrastructura feroviară a unei țări). Așa cum reiese din cele de mai sus, aceste programe nu sunt vizate în mod direct de către instrumentele industriale menționate anterior. Verificarea formală a acestor programe ascunde unele provocări neașteptate care nu sunt imediat evidente: În primul rând, este foarte dificil să se specifice constrângeri globale în astfel de sisteme uriașe. În al doilea rând, presupunând că aceste constrângeri menționate anterior sunt specificate, ar fi foarte dificil să le verificăm din cauza caracterului abstract și incomplet al specificațiilor utilizate în paradigma geografică.

Analizând cele menționate mai sus, mai ales practicile industriale din momentul de față precum și instrumentele accesibile pe piață pentru dezvoltarea unor astfel de programe distribuite pentru centralizare feroviară, putem concluziona următoarele: În primul rând, fără echipamente software care să asigure dezvoltarea și verificarea automată a unor astfel de programe, nu se poate înregistra o creștere semnificativă a calității acestor programe. În plus, prețul acestor sisteme - având în vedere costurile de dezvoltare și de întreținere - va rămâne ridicat [47]. Pe de altă parte, dacă încercăm să urmărim principul de dezvoltare funcțională al programelor de centralizare [14, 5] și încercăm să explorăm spațiul stărilor unui astfel de program atunci putem observa că în general acestă explorare este imposibilă [5]. Verificarea conduce în general la o explozie de stări, prin urmare, o nouă metodologie de verificare ar fi binevenită.

Având în vedere problemele de verificare formală a programelor distribuite menționate mai sus, care cuprinde verificarea programelor feroviare [1], precum și succesul instrumentelor cum ar fi [1], [108] și [26] pentru dezvoltarea și verificarea programelor non-concurente, aduce o motivație profundă pentru a dezvolta instrumente similare pentru programele concurente.

Comunicația care este omniprezentă în sistemele software concurente, atât în schimbul de informații între entitățile software cât și în comunicația acestor entități cu mediile lor înconjurătoare, trebuie să fie modelată și verificată în mod corespunzător pentru a afirma corectitudinea sistemelor.

1.1 Ipoteze și obiective

Obiectivul general al acestei teze este de a contribui la bazele științifice ale verificării programelor concurente critice referitoare la securitatea vieții. Avem în vedere că siguranța unui sistem distribuit depinde de execuția corectă a programului său concurent.

Scopul nostru primar este de a oferi o tehnică de verificare automată, care se bazează pe o metodă formală bine fundamentată pentru sprijinirea verificării programelor concurente asincrone. Un obiectiv secundar al acestei teze este de a oferi un set de studii de caz, în care să se aplice metoda noastră, pentru a încuraja aplicarea acestei tehnici în industria feroviară, în special în verificarea sistemelor de centralizare.

Din cauza importanței acestei teorii, un număr mare de cercetători s-au ocupat de problema corectitudinii comunicării de-a lungul ultimelor decenii.

CSP (procese secvențiale care comunică) [21] și CSC (Calculul sistemelor de comunicare) [90] sunt printre primele teorii care abordează problemele de comunicare. Cele mai recente extensii ale acestor teorii se bazează pe tipuri de sesiune și contracte de comunicare [117]. În ultimul deceniu, tipuri de sesiune au fost adoptate la un număr mare de limbaje de programare și calculul proceselor care includ limbaje funcționale [105, 33], limbaje orientate obiect [52, 36], calculul proceselor mobile [51], și procesele de ordin superior [96]. Recent, tipuri de sesiune au fost, de asemenea, extinse cu logică [13] pentru a acționa ca un contract între entitățile de comunicare. Această extensie permite o verificare mai precisă a părților implicate, permițând o specificare concisă a mesajelor transmise pe care o parte trebuie să le asigure și pe care cealaltă parte poate să se bazeze. Există de asemenea o propunere de logică de sesiune multipart [12], dar această logică încearcă să rezume efectele proceselor implicate în protocol, fără a cuprinde verificarea proceselor implementate într-un limbaj imperativ.

[1]Considerăm verificarea programelor de centralizare feroviară ca fiind un caz particular al sistemelor distribuite anterior menționate

Cu toate că, aceste teorii sunt foarte promițătoare, rezultatele lor sunt în prezent impracticabile în industrie din mai multe motive: În primul rând, aceste verificări necesită o corespondență sintactică între primitivele limbajului de programare și primitivele limbajului de specificație protocol. În plus, majoritatea acestor mecanisme necesită un set de restricții privind utilizarea referințelor, pentru a permite o urmărire precisă a canalului.

Prin urmare, **ipoteza** noastră este că există o logică, care este o extensie a logicii de separare, care permite verificarea programelor scrise în limbajele de programare imperative și se concentrează în întregime pe verificarea modelelor de comunicare, în timp ce efectele proceselor sunt rezumate în pre- și post-condiție asociată fiecărui fir de execuție.

1.2 Contribuții

Spre deosebire de abordările anterioare, noi propunem o nouă logică numită logică de sesiune, care utilizează disjuncții pentru modelarea alegerilor interne și externe și poate să verifice aplicații scrise în limbaje populare (ca Java, C# sau C).

Logica propusă se bazează pe modelarer comunicației între două entități dar permite modelarea delegărilor, prin utilizarea unor canale de ordin superior. Spre deosebire de soluțiile anterioare [36], propunerea noastră folosește aceeași metode de trimitere / primire atât pentru canale cât și pentru valori. De exemplu, [36] necesită funcții separate de trimitere / primire pentru a permite trimiterea canalelor. Totuși beneficiul cel mai important este, că datorită utilizării disjuncțiilor pentru a modela alegerile interne și externe, nu avem nevoie de a utiliza primitive speciale pentru a suporta disjuncția și putem utiliza primitivele convenționale pentru a sprijini ambele tipuri de alegeri. În schimb, propunerile anterioare necesitau în mod clar extinderea limbajului de programare cu un set de primitive condiționale, specializat construite pentru a modela alegerile interne și externe. În plus, propunerea noastră se bazează pe o extensie a logicii de separare și astfel suportă verificarea limbajelor de programare care utilizează referințe și trimit mesajele prin referință. În afară de tipurile de sesiune, Villard [87] a propus o logică pentru mesajele transmise prin referință. Logica lor se bazează pe contracte globale bazate pe stări, în timp ce logica noastră mai generală este construită ca o extensie a logicii de separare cu disjuncții pentru a sprijini și verifica alegerea internă și externă fără primitive. Formulele lor pot fi de asemenea proiectate local pentru fiecare entitate de comunicare și pot fi transmise în mod liber între proceduri. Similar cu tipurile de sesiune, logica lor necesită o primitivă specială pentru alegerea externă și internă. În plus, nu are suport pentru verificarea protocoalelor optime, fără etichete.

Mai mult decât atât, canalele noastre pot suporta o varietate de mesaje, putem trata conținutul citit ca și cum ar fi dinamic tipizat. În plus, putem garanta că conversiile de

tip al datelor primite sunt corecte. Mai mult decât atât, noi putem garanta mult mai mult decât transmiterea corectă a datelor. Verificarea noastră poate asigura că memoria heap și proprietățile datelor transmise în canale sunt tratate corespunzător. În cele din urmă, prin utilizarea unei relații de subsumare, noi permitem ca specificațiile de canale să difere între fire și totuși să se asigure că acestea rămân compatibile, astfel încât să se prevină blocajul în comunicație. Ca să fim mai realiști, noi presupunem că comunicarea este asincronă și în modelul nostru transmiterea datelor nu se blochează, pe când citirea se blochează.

În cele ce urmează vom rezuma contribuția noastră:

- **Logica de sesiune:** În capitolul 3 în conformitate cu ipoteza noastră am prezentat o nouă logică de sesiune cu disjuncții pentru a specifica și a verifica protocoalele într-un limbaj de programare imperativ utilizat pe scară largă. Această lucrare a fost prezentată în [29, 70], iar teoria este o extensie a sistemului HIP-SLEEK [26].

- **Demonstrația corectitudinii:** În capitolul 4 dovedim matematic corectitudinea teoriei noastre.

- **Programul nostru de verificare automată:** În capitolul 5 prezentăm aplicația noastră *SESSION-HIP-SLEEK*, dezvoltată pe parcursul doctoratului. Aplicația este implementată în *Caml orientat obiect (OCaml)* și este formată din două sub-aplicații, un demonstrator *SESSION-SLEEK* și instrumentul de verificare *SESSION-HIP*. Uneltele sunt o extensie a aplicației *HIP-SLEEK* și facilitează o verificare automată a corectitudinii programelor care utilizează canale de comunicare pentru a implementa sisteme paralele și distribuite.

- **Compararea cu abordări similare:** În capitolul 6, prezentăm cele mai competitive șase instrumente pentru verificarea protocoalelor care folosesc tipuri de sesiune și contracte. Am comparat fiecare dintre aceste instrumente cu instrumentul nostru, dând o serie de exemple concrete pentru a evidenția diferențele cele mai importante.

- **Aplicarea teoriei noastre în industria feroviară:** În capitolul 7, prezentăm două posibile utilizări a logicii noastre de sesiune în dezvoltarea programelor pentru industria de transport feroviar. În primul rând, prezentăm o metodă completă de modelare și verificare care se poate aplica pentru a dezvolta programele de interlocking folosind metoda de dezvoltare geografică. Metoda de modelare a cerințelor din domeniul interlocking-ului și proiecția modelului pentru fiecare entitate a fost prezentată în [71, 69]. În plus, mai prezentăm și o aplicabilitate a teoriei noastre în dezvoltarea programelor pentru controlul automat al trenurilor. Pentru a demonstra puterea logicii

noastre, am codificat în logica de sesiune un set de cerințe din specificația *TBL1+* furnizate de compania Siemens, de asemenea, trei specificații din *OpenETCS*.

În această teză vom dovedi simplitatea, expresivitatea și aplicabilitatea logicii noastre printr-o mulțime de exemple.

1.3 Lucrări conexe

Teza noastră prezintă o metodă nouă de verificare a protocoalelor în limbajele de programare populare, care se încadrează în teoria de verificare a protocoalelor. În această direcție am identificat trei direcții principale de cercetare care vizează problema de verificare protocol: extragerea și verificarea modelelor de comunicație în mod automat, generarea automată de cod și verificarea codului sursă.

Prima metodă pornește de la codul sursă și extrage un model abstract al protocolului pe care se verifică proprietățile specificate inițial [77, 3]. În cazul unei erori, această abordarea oferă un model abstract dezvoltatorului în care trebuie să se identifice problema. Din păcate în general, acest model este prea complex, deci dificil de înțeles. Prin urmare aplicabilitatea acestei metode de verificare este limitată la niște probleme mici în care modelul poate fi ușor de înțeles.

Cea de a doua abordare sugerează o metodă de dezvoltare în care se pornește de la un model abstract de protocol și se generează o sursă de cod corectă, dar incompletă a protocolului [63, 99]. Pe de altă parte, și cei mai buni programatori fac greșeli și o extensie trivială care pare să fie corectă, poate conține greșeli, iar în astfel de cazuri corectitudinea codului final nu se mai poate asigura.

Teoria noastră face parte din verificarea automată a codului sursă. Din cauza numărului foarte mare de rezultate în această direcție, ne concentrăm asupra diferențelor esențiale dintre lucrarea noastră și verificarea statică a protocoalelor de ordin înalt.

În primul rând, vom compara teoria noastră cu tipurile de sesiune. Acesta este o disciplină de tip pentru verificarea corectitudinii comunicației unui program distribuit. Teoria a fost dezvoltată inițial în π -calculus [113, 24] și mai târziu extinsă să faciliteze verificarea limbajelor de programare funcționale și limbajelor de programare orientate pe obiect [60, 105]. Ideea principală a tipurilor de sesiune pornește de la ideea că aplicațiile sunt construite pornind de la un set de unități de proiectare numite modele de sesiune. Implementările actuale de tipuri de sesiune [60, 105] se axează pe verificarea statică a codului proceselor în raport cu specificația locală al fiecărui proces. Din păcate aplicabilitatea acestei teorii este foarte limitată. În primul rând, sistemele de tip existente permit verificarea unui set foarte limitat de limbaje de programare. Aceste limbaje trebuie să fie extinse cu blocuri

condiționale pentru alegerea internă și externă. Acestea trebuie să fie scrise într-un anumit fel, altfel verificarea nu este posibilă.

De exemplu, în lucrarea [36] propune o disciplină de tip pentru a verifica corectitudinea tipurilor de sesiune într-un limbaj de programare orientat pe obiecte, dar sistemul de tip necesită un limbaj de programare extins cu un set de primitive de comunicare: *send*, *receive*, *sendIf*, *receiveIf*. Dacă încercăm să eliminăm *sendIf*, *receiveIf* din limbajul lor de programare, atunci verificarea va deveni inconsistentă. Alte tipuri de sesiune, cum ar fi [52, 35, 100, 86] suferă de aceeași problemă. Problema principală a acestei teorii este aplicabilitatea limitată a teoriei în practică. Acest fapt a fost identificat în mai multe lucrări [63, 97]. O metodă de verificare care vizează limbaje de programare cu astfel de primtive nu este utilă în practică. Motivele principale pentru care aceste limbaje au fost extinse cu astfel de primitive sunt prezentate în [31, 60]. Aceste sisteme de tip se adresează verificării unui set limitat de limbaje de programare, care sunt construite direct cu scopul de a implementa sisteme distribuite. În contrast, abordarea noastră mai generală are ca scop asigurarea comportamentului corect al programelor scrise într-un limbaj de programare obișnuit, în raport cu o specificație logică mult mai expresivă, prin verificarea statică a codului sursă a programului.

Deoarece aceste sisteme de tip sunt destul de limitate, există o mulțime de lucrări [23, 25, 11, 88, 98, 59] care încearcă să verifice corectitudinea specificațiilor de tip prin verificare dinamică. Lucrarea [23] prezintă o verificare dinamică prin monitorizarea și analiza fluxului de informații într-o sesiune cu mai multe părți. O altă metodă de monitorizare și analiză a comunicației într-un sistem de comunicare cu mai multe părți și o însumare a metodelor de monitorizare este prezentată în [25]. Aceste lucrări adresează problema verificării dinamice a specificațiilor de protocol în limbajele de programare răspândite în indusrie, dar aceste verificări nu sunt exhaustive și nu se pot aplica numai pe coduri gata executabile.

Relația între π-calculus și logica de separare a fost studiată în [111] și [57], dar aceste lucrări reprezintă o tratare al calculului π, bazată pe teoria logicii de separare, fără a se concentra pe problema verificării protocoalelor. Această idee a fost studiată și în logica Hoare în lucrarea [89].

Din perspectiva altor limbaje de specificație de protocol, există o lucrare [74] care încearcă să codifice CSP (algebra de comunicare a proceselor) în logica Hoare. Din păcate această lucrare codifică doar operațiile de send și receive fără a codifica ramificarea și fără a permite manipularea referințelor și apelurilor de metode.

În plus, [12] a propus o logică pentru a extinde tipurilor de sesiune pentru a îmbunătăți capacitatea de expresivitate a limbajului de specificare studiat [13], în vederea unei specificări mai precise a comunicației și pentru a capta schimbarea stării entităților care sunt implicate în

această comunicație, dar din punct de vedere al verificării, propunerea are aceleași neajunsuri ca și cele anterior prezentate. Recent, Villard [87, 123] a dezvoltat o logică nouă bazându-se pe contracte globale. Comparând această logică cu logica noastră, putem observa că logica noastră este mult mai generală, deoarece este construită ca o extensie a logicii de separare cu disjuncții pentru a sprijini opțiuni de comunicare în limbaje imperative industriale.

Dacă considerăm instrumentele anterioare cum ar fi Mool [115], Moose [95], Bica [114], SessionJava [60] bazate pe starea tipului, și SessionC [100], ParTypes [116] bazate pe tipuri dependente indexate, putem observa că aceste instrumente pot verifica numai programe scrise într-un limbaj de programare care este extins în această direcție cu primitive de decizii de protocol, ceea ce limitează gradul de utilizare și nu permite verificarea programelor scrise în limbajele industriale de masă, cum ar fi limbajul C, Java sau ADA.

În concluzie, comparând rezultatul nostru cu rezultatele și lucrările menționate anterior, contribuția noastră se concentrează pe asigurarea corectitudinii aplicațiilor distribuite, prin verificarea corectitudinii codului sursă în raport cu o specificație de protocol și se diferențiază de celelalte teorii și instrumente prin aplicabilitatea sa pe limbaje imperative răspândite în industrie și prin expresivitatea mai bună a limbajului nostru de specificație.

Din perspectiva industriei feroviare, există mai multe lucrări care încearcă să aplice calculul proceselor CSP [94, 93, 66, 92, 109, 126] pentru modelarea și verificarea unor aspecte ale sistemelor de control feroviar. De exemplu, [66, 93] propune o tehnică nouă pentru a genera un model *CSP* || *B* dintr-un plan de interlocking cu instrumentul lor *Ontrack*, care poate fi verificată cu *ProB* model checker. [126] propune, de asemenea, o tehnică de modelare bazată pe CSP pentru a verifica dacă specificația funcțională care este dată într-o tabelă de control *respectă* toate *principiile de semnalizare*. În ciuda acestor abordări interesante, sistemele lor nu sunt adecvate pentru sistemele de cale ferată reale. În plus, nici una dintre metodele de verificare prezentate mai sus nu a vizat verificarea codului sursă și nu se scalează suficient pentru a putea verifica sisteme reale. Prin urmare, problema verificării sistemelor de control al căilor ferate este privită ca o problemă reală care trebuie rezolvată [124, 47].

Având în vedere cele expuse mai sus, al doilea scop al acestei teze este de a oferi o soluție pentru problemele menționate mai înainte, permițând verificarea statică a comunicării entităților într-un sistem de control al căilor ferate dezvoltate cu o abordare geografică utilizând logica de sesiune.

Scopul acestei teze este de a capta verificarea protocoalelor utilizate în sistemele de centralizare feroviare în limbajele imperative, utilizate pe scară largă în industrie, oferind un sprijin mai bun pentru dezvoltarea și verificarea sistemelor distribuite heterogene. Pentru a oferi acest lucru, echipamentul nostru asigură verificarea statică a fiecărei componente în mod

independent, pentru a asigura un set de proprietăți globale foarte importante pentru siguranța operării căilor ferate. Conceptul nostru se bazează pe ideea că, dacă fiecare componentă este verificată și este corectă în conformitate cu specificația ei de protocol, atunci sistemul compus din aceste componente este corect și respectă specificația globală a sistemului.

1.4 Structura tezei

1.5 Publicații

Capitolul 2
Noțiuni introductive

2.1 Calculul π

2.2 Tipuri de sesiune

2.3 Logica de separare

2.3.1 Logica Hoare

Demonstrarea corectitudinii logicii Hoare

2.3.2 Logica de separare secvențială

2.3.3 Logica de separare concurentă

2.4 Sumar

În acest capitol introducem teoriile de bază necesare pentru înțelegerea acestei teze. În primul rând, vom prezenta calculul π, care este un concept modern pentru modelarea matematică a proceselor concurente. Apoi, introducem noțiunea de tipuri de sesiune, care este o disciplină de tip pentru asigurarea comportamentului corect de comunicare a proceselor. În cele din urmă, vom prezenta logica de separare, care este o teorie nouă pentru verificarea programelor secvențiale și concurente care permit modificarea stărilor și referințelor.

Capitolul 3
Logica de sesiune

Vom introduce logica noastră de sesiune folosind un protocol simplu, care descrie o interacțiune dintre un cumpărător și vânzător. Această conversație începe de la cumpărător, care trimite denumirea produsului, care este de tipul unui `String`, către vânzător. Vânzătorul trebuie să răspundă la această cerere cu un preț care este de tipul `int`. Dacă cumpărătorul este mulțumit cu prețul primit, atunci acesta va trimite o adresă de tip `Addr` ca adresă de livrare și va primi de la vânzător data livrării ca tip `Date`. Dacă prețul nu satisface așteptările cumpărătorului, atunci acesta va închide comunicația. Acest exemplu este modelat ca o comunicație între două părți în Fig.3.1. De obicei, un canal este suficient pentru a permite comunicarea între două entități. Putem modela protocolul anterior din punctul de vedere al cumpărătorului folosind următoarea specificație de tip sesiune:

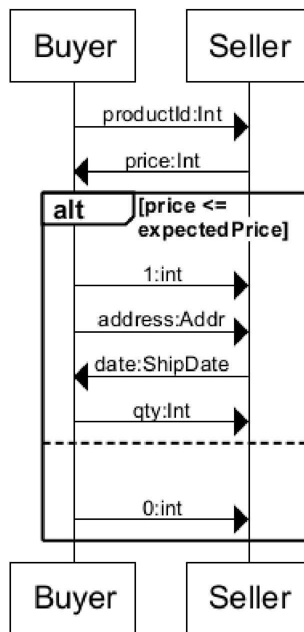

Figura 3.1 Diagrama de secvență pentru achiziționarea unui element

$$\begin{aligned}\texttt{buyer_ty} \;\equiv\;& \texttt{begin}; !\texttt{String}; ?\texttt{int};\\ & !\{\texttt{ok}: !\texttt{Addr}; ?\texttt{Date}; !\texttt{int}; \texttt{end},\ \texttt{quit}: \texttt{end}\}\end{aligned}$$

Modelul dual (complementar) al modelului anterior care descrie conversația din punctul de vedere al vânzătorului este următorul:

$$\begin{aligned}\texttt{seller_ty} \;\equiv\;& \sim\texttt{buyer_ty}\\ \equiv\;& \texttt{begin}; ?\texttt{String}; !\texttt{int};\\ & ?\{\texttt{ok}: ?\texttt{Addr}; !\texttt{Date}; !\texttt{int}; \texttt{end},\ \texttt{quit}: \texttt{end}\}\end{aligned}$$

În modelele anterioare, `!t` denotă trimiterea unei valori de tip `t` pe canalul curent, în timp ce `?t` se referă la o citire a unei valori de tip `t` pe canalul curent. Tipul de sesiune `!{ok : ..., quit : ...}` indică o alegere internă (decizia se face pe baza stării locale a procesului), în timp ce `?{ok : ..., quit : ...}` denotă o alegere externă (decizia se face pe baza etichetei primite). Opțiunile sunt reprezentate printr-un set de etichete care sunt transmise pe canal în timpul conversației. Cuvântul cheie `begin` marchează începutul unei conversații, pe când `end` marchează sfârșitul conversației, pentru un anumit canal. În general, un program care se poate verifica de către un sistem de tip sesiune este implementat într-un limbaj de programare care utilizează if -uri sau switch-uri speciale [60] ca, de exemplu, `outbranch` și `inbranch` pentru a modela alegerea internă și externă:

```
void buyer(buyer_ty c, String p)      | void seller(seller_ty c)
{ send(c, p);                         | { String p = receive(c);
 Double price = receive(c);           |  send(c, getPrice(p));
 Double budget = ...;                 |  inbranch(c) {
 if price <= budget then{             |   case ok : {
  outbranch(c, ok){                   |    Addr a = receive(c);
  Addr a = ...;                       |    ShipDate sd = ...;
  send(c, a);                         |    send(c, sd);
  ShipDate sd = receive(c);           |    int qty = receive(c);
  send(c, 3);                         |   }
  }} else outbranch(c, quit){}        |   case quit : { }
}                                     | } }
```

Folosind logica noastră de sesiune, comunicația prezentată anterior dintre cumpărător și vânzător se poate descrie în felul următor:

$$\begin{aligned}
\texttt{buyer_ch} &\equiv \texttt{!String};\texttt{?int};((\texttt{!1};\texttt{!Addr};\texttt{?Date};\texttt{!int})\vee\texttt{!0})\\
\texttt{seller_ch} &\equiv \sim\texttt{buyer_ch}\\
&\equiv \texttt{?String};\texttt{!int};((\texttt{?1};\texttt{?Addr};\texttt{!Date};\texttt{?int})\vee\texttt{?0})
\end{aligned}$$

Privind superficial, această logică pare a avea o mulțime de similitudini cu tipurile de sesiune, totuși dacă ne uităm cu mai multă atenție putem remarca o mulțime de diferențe. În primul rând, putem observa că specificația noastră în logica de sesiune nu conține declarațiile de `begin` și `end`. Acest lucru este posibil, deoarece verificarea noastră poate trata canalele fără nici un fel de restricție, deci nu trebuie să facem diferență între canalele deschise și cele închise. În al doilea rând, specificația noastră folosește disjuncția standard din logică [1], în

[1] Pentru a putea avea un protocol decidabil condiția de trimitere trebuie să descrie mulțimi disjuncte ca, de exemplu, protocolul `sender`

loc să utilizeze niște notații speciale pentru alegerea externă și internă. În al treilea rând, specificația noastră în logica de sesiune folosește valori (cum ar fi 1 sau 0) în loc de etichete (cum ar fi ok sau quit) sau chiar tipuri pentru a modela o alegere internă și externă, respectiv pentru a modela diferite scenarii. Acest lucru ne permite să utilizăm blocurile de decizii standard cum ar fi if-ul și switch-ul standard pentru decizii. Verificarea se poate face simplu utilizând disjuncția standard. Mai mult decât atât, noi putem să utilizăm formule complexe din logica de separare în loc de tipuri, care ne ajută să verificăm aplicațiile mult mai precis, putând capta erori operaționale sau erori funcționale care sunt mult peste corectitudinea verificării protocoalelor. Acestea captează și canalele de ordin superior unde canalul se poate trimite împreună cu specificația sa la o altă entitate ca un mesaj.

Pentru a ilustra cele prezentate mai sus, vom îmbunătăţi specificația logică prin extinderea formulei cu o constrângere, care să facă formula mai precisă și unde cerem ca prețul să fie un câmp numeric strict pozitiv în loc să fie numai numeric.

$$\begin{array}{lcl} \texttt{buyer_chan} & \equiv & !\texttt{String};?\texttt{r}{:}\texttt{int}\cdot\texttt{r}{>}0;((!1;!\texttt{Addr};?\texttt{Date};!\texttt{int})\vee!0) \\ \texttt{seller_chan} & \equiv & \sim\texttt{buyer_chan} \end{array}$$

Rețineți că specificația noastră utilizează mai multe abreviații. $?1$ este o prescurtare pentru $?\texttt{r}\cdot\texttt{r}{:}\texttt{int}\wedge\texttt{r}{=}1$, în timp ce $!\texttt{String}$ este o prescurtare pentru $!\texttt{r}\cdot\texttt{r}{:}\texttt{String}\wedge\texttt{true}$. Specificația seller_chan este dualul lui buyer_chan. O astfel de specificație se poate obține prin inversarea simbolurilor de citire și scriere.

În plus limbajul nostru suportă specificațiile în logica de separare pentru transmiterea referințelor, care sprijină verificarea aplicațiilor paralele, acestea folosind memorie partajată. Atunci când formula de separare este *emp* folosim următoarea abreviație $?\texttt{r}{:}\texttt{int}\cdot\texttt{r}{>}1$ pentru $?\texttt{r}\cdot\texttt{emp}\wedge\texttt{r}{:}\texttt{int}\wedge\texttt{r}{>}1$. Este demn de remarcat faptul că specificația firelor nu trebuie să fie perfect simetrică (să fie dualul celuilalt fir cu care comunică). De exemplu, putem utiliza o specificație de protocol mai puternică pentru procesul vânzătorului, care solicită de la acest proces să trimită cumpărătorului un preț care este de cel puțin 10 unități, după cum urmează:

$$\texttt{seller_sp} \equiv ?\texttt{String};!\texttt{r}{:}\texttt{int}\cdot\texttt{r}{>}10;((?1;?\texttt{Addr};!\texttt{Date};!\texttt{int})\vee?0)$$

După cele menționate mai sus, putem scrie un program care implementează protocolul de mai sus, așa cum se poate vedea în codul care urmează:

```
open(c) with buyer_chan;
(buyer(c,prod) || seller(c));
close(c);
```

```
void buyer(Chan c,String p)
  requires 𝒞(c,buyer_chan)
  ensures 𝒞(c,emp)
{ send(c,p);
 Double price = receive(c);
 Double budget = ...;
 if (price <= budget) then{
  send(c,1);
  Addr a = ...;
  send(c,a);
  ShipDate sd = receive(c);
  send(c,3);
 } else send(c,0);
}
```

```
void seller(Chan c)
  requires 𝒞(c,seller_sp)
  ensures 𝒞(c,emp)
{ String p = receive(c);
 send(c,getPrice(p));
 int usr_opt = receive(c);
 if (usr_opt==1){
  Addr a = receive(c);
  ShipDate sd = ...;
  send(c,sd);
  int qty = receive(c);
 } else
  assert usr_opt = 0;
}
```

Puteți observa că putem folosi direct condiționale standard în loc de construcții speciale pentru comunicare. Prima instrucțiune deschide un canal utilizând instrucțiunea `open` care acceptă ca argument o specificație de canal în logica de sesiune. Referința returnată este transmisă atât firului `buyer` cu specificația `buyer_chan` cât și firului `seller` cu specificația duală `seller_chan`. Cele două procese se execută în paralel. Fiecare proces poate avea propria sa specificație de protocol, dar aceste specificații trebuie să inducă semantic specificația originală a canalului și trebuie să asigure cel puțin proprietățile stabilite inițial. De exemplu specificația `seller_sp` al funcției `seller`, care cere ca prețul produsului să fie $\texttt{r}>10$, induce $\texttt{r}>0$ care corespunde cu specificația inițială din `seller_chan`.

Prin urmare când un canal este transmis unui fir, acesta trebuie să inducă specificația firului de execuție. În cazul firului `buyer` din exemplul nostru, acesta înseamnă că $\mathscr{C}(\texttt{c},\texttt{buyer_chan}) \vdash \mathscr{C}(\texttt{c},\texttt{buyer_chan})$. Pentru procesul `seller`, trebuie să demonstrăm că $\mathscr{C}(\texttt{c},\texttt{seller_chan}) \vdash \mathscr{C}(\texttt{c},\texttt{seller_sp})$. Această demonstrație este din nou foarte simplă, deoarece specificația pentru trimitere este o contra-variantă cum se poate observa mai jos.

$$\dfrac{\dfrac{\dfrac{\texttt{r}>10 \vdash \texttt{r}>0}{!\texttt{r}\cdot\texttt{r}>0 \vdash !\texttt{r}\cdot\texttt{r}>10}}{\texttt{seller_chan} \vdash \texttt{seller_sp}}}{\mathscr{C}(\texttt{c},\texttt{seller_chan}) \vdash \mathscr{C}(\texttt{c},\texttt{seller_sp})}$$

Deoarece post-condiția funcției `int getPrice(String)` ne asigură că valoarea returnată de funcție este mai mare decât 10, verificarea procesului este foarte simplă, deci îl vom omite (se poate găsi în teză).

3.1 Modelarea proceselor in logica de sesiune

3.1.1 Calculul π asincron cu constrângeri logice

3.2 SESSION-HIP

Pentru a demonstra teoria noastră, vă prezentăm limbajul nostru de programare imperativ: SESSION-HIP.

3.2.1 Sintaxa SESSION-HIP

O să formalizăm abordarea noastră pe un limbaj de programare concurentă și imperativă cu primitive de comunicare. Limbajul este o extensie a limbajului secvențial din [27]. Sintaxa limbajului este prezentat în Fig.3.2. Un program *Prog* scris în acest limbaj de programare conține un set de declarații *tdecl*, care poate să fie o declarație de structură de date *datat*, o declarație de predicate *spred* sau o declarație de metode *meth*. Definiția lui *spred* și *mspec* este prezentat în Fig. 3.5. Limbajul nostru se bazează pe expresii, așa că corpul metodei este o expresie (e) formată dintr-un set de instrucțiuni. Limbajul permite transmiterea parametrilor, atât prin referință cât și prin valoare. Acești parametrii permit codificarea unui ciclu într-un apel recursiv, unde schimbarea valorilor se realizează prin transmiterea parametrului ca referință. Această tehnică este standard pentru simplificarea limbajelor de programare. Limbajul permite crearea proceselor paralele cu ajutorul operatorului $||$. Procesele pot comunica între ele și cu mediul înconjurător prin intermediul canalelor de comunicație. Un canal se poate crea cu `new Chan()`, dar nu se poate utiliza până nu este deschis. Fiecare canal este păstrat ca o referință care se poate transmite. Noi utilizăm aceeași reguli de verificare ca și în HIP/SLEEK , dar pentru operațiunile pe canaluri asigurăm un set minim de instrucțiuni cu pre- și post-condiții. Un canal se poate deschide utilizând instrucțiunea `open` cu o specificație de protocol S. După deschidere avem două referințe la același canal, unul având specificația S și celălalt specificația complementară $\sim S$, după cum urmează:

$$\begin{array}{l}\texttt{void open(Chan c) with } S \\ \quad \texttt{requires emp} \\ \quad \texttt{ensures } \mathscr{C}(\texttt{c}, S) * \mathscr{C}(\texttt{c}, \sim S)\end{array}$$

Un canal poate fi închis, dacă ambele referinţe sunt accesibile şi ambele au consumat specificaţia, după cum urmează.

```
void close(Chan c)
    requires 𝒞(c,emp) * 𝒞(c,emp)
    ensures emp
```

În contrast cu tipurile de sesiune, limbajul nostru necesită un set minim de instrucțiuni: `send` și `receive`. Specificația operațiilor este definită mai jos. Menționăm că `res` este un cuvânt cheie rezervat pentru variabila returnată de funcția `receive`, iar `L(x)` este o formulă în logica de sesiune pentru variabila `x`.

```
t receive(Chan c)
    requires 𝒞(c,?r:t·L(r);rest)
    ensures L(res) * 𝒞(c,rest)
void send(Chan c, t x)
    requires 𝒞(c,!x:t·L(x);rest) * L(x)
    ensures 𝒞(c,rest)
```

Într-o comunicație între două entități, un canal este de obicei suficient pentru comunicație. Am notat cele două procese care comunică cu `P(c)` și `Q(c)`, unde `c` este canalul de comunicare. Cele două procese trebuie să aibă câte o specificație de protocol, adică `P_sp` și `Q_sp`. Având acești termeni, în general un proces se poate defini în felul următor:

```
t P(Chan c)
    requires 𝒞(c,P_sp) * Pre₁
    ensures 𝒞(c,R₁) * Post₁
t Q(Chan c)
    requires 𝒞(c,Q_sp) * Pre₂
    ensures 𝒞(c,R₂) * Post₂
```

Operația `close` trebuie să marcheze finalul conversației. La apelarea acestei metode, ambele capete ale canalului trebuie să aibă protocolul consumat. În următorul exemplu, operația `close` eșuează deoarece canalul nu este total consumat. Merită remarcat faptul, că specificația S_2 este recursivă.

```
S₂ ≡ !String;S₂
open(c) with S₂;
//𝒞(c,S₂) * 𝒞(c,~S₂)
//𝒞(c,S₂)          || //𝒞(c,~S₂)
for(i = 1 to 5)    || for(i = 1 to 10)
  send(c,i);       ||   int x = receive(c);
//𝒞(c,S₂) * 𝒞(c,~S₂)
close(c);//FAILS!
```

program definition	*Prog*	$::= tdecl^* \ meth^*$
type declaration	*tdecl*	$::= datat \mid spred$
data type	*datat*	$::= \texttt{data}\ c\ \{\ (t\ v)^*\ \}$
types	*t*	$::= c \mid prim \mid Chan \mid dyn$
primitive types	*prim*	$::= \texttt{int} \mid \texttt{bool} \mid \texttt{void}$
method definition	*meth*	$::= t\ mn\ (ref\ (t\ v)^*, (t\ x)^*)\ mspec\ \{e\}$
enpoint	*che*	$N \mid D$
expressions	*e*	$::= null \mid k^{prim} \mid v \mid v.f \mid v := e \mid v_1.f := v_2 \mid e_1; e_2$
		$\mid \texttt{if}\ (v)\texttt{then}\ e_1\ \texttt{else}\ e_2 \mid \texttt{t}\ v;\ e \mid mn(v^*; x^*)$
		$\mid \texttt{new}\ c(v^*) \mid \texttt{free}(v)$
		$\mid (v_l^*, (vc_l = red\ c_l\ che)^*)\{e_1\} \| (v_r^*, (vc_r = red\ c_r\ che)^*)\{e_2\}$
		$\mid \texttt{open}(c_1, c_2)\ \texttt{with}\ spred \mid \texttt{close}(c_1, c_2)$
		$\mid \texttt{send}(c, v) \mid \texttt{receive}(c)$

Figura 3.2 Un limbaj de programare concurent cu sesiuni.

Limbajul nostru permite tipurile dinamice `Dyn`. De exemplu, parametrii funcției `send` precum și variabila returnată de `receive` acceptă ca tip, tipul dinamic:

```
void send(Chan c, Dyn val){...}
Dyn receive(Chan c){...}

send(c, 3); send(c, "...");
int r = (int) receive(c);
String r = (String) receive(c);
```

Verificarea noastră asigură un mecanism simplu și sigur de conversie a tipurilor, după cum se poate observa mai jos:

```
Dyn t = receive(c);
switch t with {
  v1 : int → ...
  v2 : String → ...
}
```

Alternativa sintactică cu `if` arată în felul următor:

```
Dyn t = receive(c)
if (type(t) = int) {v1 = (int)t; ...}
else if (type(t) = String) {v2 = (String)t; ...}
else {assert false;}
```

Utilizând testarea dinamică a tipurilor, o comunicație care conține un ciclu se poate defini în felul următor:

$$\mathtt{S}_3 \equiv \mathtt{!Object;}(\mathtt{S}_3 \vee \mathtt{!0})$$

Desigur, în cazul în care ambele părți ale disjuncției așteaptă o valoare de același tip, specificația trebuie să arate în felul următor:

$$\mathtt{S}_4 \equiv \mathtt{!Object;}(\mathtt{!1;S}_4 \vee \mathtt{!0})$$

3.2.2 Semantica operațională

În această secțiune prezentăm un mediu de execuție în semantica operațională cu pași de calcul mici pentru limbajul nostru, după cum urmează:

$$State \triangleq Stack\ x\ Heap\ x\ CHeap \qquad Stack \triangleq Var \to Val \cup Cell$$

$$CHeap \triangleq Endpoint \xrightarrow{\text{fin}} MQueue\ x\ MQueue \qquad Heap \triangleq Cell \xrightarrow{\text{fin}} Val$$

$$MQueue \triangleq QueueId \xrightarrow{\text{fin}} Val^*$$

Starea actuală a mașinii este reprezentat printr-un tuplu $\langle e,s,h,c \rangle$ în care e este comanda, s este stiva, h reprezintă heap-ul și c denotă heap-ul canalelor de comunicație. Heap-ul canalelor este neconvențional în comparație cu modelele obișnuite ale logicii de separare, de aceea îl vom examina mai în profunzime.

În sistemul nostru, fiecare canal are două capete care sunt utilizate în general de două fire diferite. Comunicarea între cele două capete este modelat cu o pereche de cozi care joacă roluri diferite pentru cele două obiecte. Primul capăt vede prima coadă ca o intrare, în timp ce al doilea capăt consideră ca o coadă de ieșire și vice-versa pentru a doua coadă. Acest model oferă un mediu intuitiv pentru analizarea comunicației între procese. În consecință, rolul heap-ului este de a stoca referințele la aceste cozi.

În plus, dorim să remarcăm că separarea heap-ului standard, de la acest heap nu este necesară, dar ajută la prezentarea semanticii și simplifică demonstrarea corectitudinii verificării.

În continuare vom prezenta în mod informal semantica operațională al SESIUNII-HIP pentru comenzile standard.

Având tuplul $\langle e,s,h,c\rangle$ care reprezintă starea curentă al unei mașini, putem formaliza un pas de tranziție în felul următor: $\frac{valid(s_1,h_1,c_1)}{\langle e_1,s_1,h_1,c_1\rangle\hookrightarrow\ langlee_2,s_2,h_2,c_2\rangle}$. Formula de reducere poate fi interpretată în felul următor: dacă o mașină are o stare $\langle e_1,s_1,h_1,c_1\rangle$, iar starea $\langle s_1,h_1,c_1\rangle$ îndeplinește predicatul **valid** și vom executa expresia e_1, atunci mașina va schimba starea sa la $\langle e_2,s_2,h_2,c_2\rangle$. Pentru a înțelege regulile de mai jos, trebuie să prezentăm câteva notații standard, utilizate în regulile de reducere semantică.

Notați 3.2.1. Am introdus **skip** ca o expresie, care nu schimbă starea mașinii și $s[v\mapsto \upsilon]$ pentru a defini o variabilă **v**, care indică o valoare υ. În plus, $\perp$ pentru a denota o valoare necunoscută, **k** pentru a denota o constantă și $\mathbf{r}(\mathbf{v}^*,\mathbf{e})$ pentru a modela rezultatul invocării unei funcții, unde **e** reprezintă codul rămas după execuția apelului. Operațiunea de $s\cdot[v\mapsto\upsilon]$ adaugă variabila v cu valoare υ în stivă. Operațiunea de $s_1=s_2-s_3$ este o prescurtare pentru $s_2=s_1\cdot s_3$ și elimină s_3 din stiva s_2.

În continuare, vom da o listă de definiții formale pentru operatorii non-standard. Acești operatori joacă un rol important în demonstrarea corectitudinii regulilor noastre de verificare.

Definiție 3.2.1. (unirea a două funcții disjuncte) Unirea disjunctă a două funcții parțiale f și g cu același co-domeniu D este:

$$dom(f)\cdot dom(g) \xrightarrow{\text{fin}} D$$

$$(f\cdot g)(x) ::= \begin{cases} f(x) & \text{if } x\in dom(f) \\ g(x) & \text{if } x\in dom(g)\end{cases}$$

Definiție 3.2.2. (Sub-stări disjuncte) Două stări $\langle s_1,h_1,c_1\rangle$, $\langle s_2,h_2,c_2\rangle$ se pot numi sub-stări disjuncte al stării $\langle s,h,c\rangle$ și se vor nota $\langle s_1,h_1,c_1\rangle\#\langle s_2,h_2,c_2\rangle$, dacă condițiile următoare sunt îndeplinite:

1. $dom(s_1)\cup dom(s_2)\subseteq dom(s)\wedge dom(s_1)\cap dom(s_2)=\emptyset$
2. $dom(h_1)=\bigcup_{v_i\in dom(s_1)} part(v_i,h) \quad dom(h_2)=\bigcup_{v_j\in dom(s_2)} part(v_j,h)$
   $dom(h_1)\cup dom(h_2)\subseteq dom(h)\wedge dom(h_1)\cap dom(h_2)=\emptyset$
3. $dom(c_1)=\bigcup_{v_i\in dom(s_1)} part(v_i,c) \quad dom(c_2)=\bigcup_{v_j\in dom(s_2)} part(v_j,c)$
   $dom(c_1)\cup dom(c_2)\subseteq dom(c)\wedge dom(c_1)\cap dom(c_2)=\emptyset$

Acum, după ce am definit unirea a două funcții în definiția 3.2.1 și noțiunea de stări disjuncte în definiția 3.2.2, vom prezenta extensia noastră HIP pentru semantica operațională concurentă în Fig. 3.3.

Înainte de a oferi alte detalii despre limbaj, să trecem peste aceste reguli într-un mod informal:

- commanda **open(f)** creează un mediu de comunicare prin alocarea resurselor necesare și conectarea acestora în mod corespunzător. Mai precis, ea alocă două noduri terminale $\mathbf{l_1}, \mathbf{l_2}$ și două cozi goale $\mathbf{q_1}, \mathbf{q_2}$, și asociază $\mathbf{q_1}$ ca și coadă de intrare, $\mathbf{q_2}$ ca și coadă de ieșire la $\mathbf{l_1}$ și vice versa pentru $\mathbf{l_2}$. Executarea acestei comenzi este posibilă numai în cazul în care $\mathbf{l_1}, \mathbf{l_2}, \mathbf{q_1}, \mathbf{q_2}$ nu sunt alocate.
- **close(f)** elimină punctele finale și cozile de canal **f** din heap-ul de canal, în cazul în care parametrul **f** este un canal gol definit în mod corespunzător (aceasta înseamnă că punctele finale trebuie să fie duale și cozile trebuie să fie goale).
- **send(f,v)** mută toate resursele la care se face referire din **v** din heap-ul normal și heap-ului canalurilor în coada de mesaje de ieșire și elimină **v** din heap.
- **receive(f)** extrage prima variabilă din coadă și adaugă resursele corespunzătoare acestei variabile în heap-ul standard și heap-ul canalului.
- **red ch {L,D}** este o funcție de ajutor, care poate fi folosită pentru a extrage punctul final dintr-o variabilă a canalului. Această funcție ajută doar funcția fir pentru a extrage punctele finale și nu pot fi folosite în alte scopuri.
- $(\mathbf{v_l^*}, (\mathbf{ch_l} = \mathbf{red\ c\ L})^*)\{\mathbf{e_1}\} || (\mathbf{v_r^*}, (\mathbf{ch_r} = \mathbf{red\ c\ D})^*)\{\mathbf{e_2}\}$ este cea mai complexă regulă de reducere. Operația creează două medii de execuție pentru cele două fire, prin urmare firele se execută independent, iar verificarea se poate face simplu prin analiza parametrilor celor două funcții care se execută pe cele două fire. În cazul în care parametrii celor două funcții arată către memorii disjuncte, atunci firele sunt independente. În plus, canalele transmise la cele două fire trebuie să fie perechi și fiecare fir deține controlul unui capăt al canalului. În cazul în care sunt îndeplinite toate condițiile anterioare, cele două fire sunt reduse în mod independent.

3.3 Principiul de verificare

Mecanismul nostru de verificare este bazat pe interpretarea abstractă, si este o extensie a verificării cu logica Hoare, mai precis este o extensie a logicii de separare din [27].

O imagine schematică a mecanismului nostru de verificare este prezentat în Fig.3.4.

Sistemul necesită ca intrare un set de funcții cu pre- și post-condiții alături de predicatele, care sunt cerute de pre- și postcondițiile menționate anterior. Predicatele pot fi de două tipuri: predicate scrise în logica de sesiune și predicate definite în logica de separare.

În condiția în care cerințele menționate mai sus sunt îndeplinite, putem verifica dacă codul sursă al fiecărei funcții îndeplinește specificațiile sale. Verificarea se face în mod sistematic pentru fiecare expresie utilizând regulile de verificare corespunzătoare din subsecțiunea 3.3.2. O expresie este considerată a fi corectă dacă, și numai dacă, precondiția sa poate fi îndeplinită de către starea simbolică curentă. În acest caz, pre-condiția este îndepărtată și se adaugă post-condiția la starea actuală.

$$\frac{l_1, l_2, q_1, q_2 \notin dom(c)}{\langle open(f_c), s[f_c \mapsto \perp], h, c\rangle \hookrightarrow \langle skip, s[f_c \mapsto \upsilon(\mathscr{C}, ((\upsilon_1(\mathscr{C}, l_1), \upsilon_2(\mathscr{C}, l_2)))], h, c \cdot [l_1 \mapsto (q_1, q_2), l_2 \mapsto (q_2, q_1), q_1 \mapsto \emptyset, q_2 \mapsto \emptyset]\rangle}$$

$$\langle close(f_c), s[f_c \mapsto \upsilon(\mathscr{C}, ((\upsilon(\mathscr{C}, l_1), \upsilon(\mathscr{C}, l_2)))], h, c \cdot [l_1 \mapsto (q_1, q_2), l_2 \mapsto (q_2, q_1), q_1 \mapsto \emptyset, q_2 \mapsto \emptyset]\rangle \hookrightarrow \langle skip, s[f \mapsto \perp], h, c\rangle$$

$$\frac{v, f \in dom(s) \quad s(v) = \upsilon_t(\mathscr{H}, _) \vee s(v) = \upsilon_t(\mathscr{C}, _) \quad (h_t, c_t) = part(v, h) \quad h_1 = h - h_t \quad c_1 = c - c_t \quad s_1 = s - [v] \quad s(f) = \upsilon(\mathscr{H}, l) \quad l \notin dom(c_t)}{\langle send(f, v), s, h, c \cdot [l \mapsto (q_1, q_2), q_1 \mapsto m_1, q_2 \mapsto m_2]\rangle \hookrightarrow \langle skip, s_1, h_1, c_1 \cdot [l \mapsto (q_1, q_2), q_1 \mapsto m_1, q_2 \mapsto m_2 \oplus (\upsilon_t, h_t, c_t)]\rangle}$$

$$\frac{v, f \in dom(s) \quad s(v) = \upsilon_t(\mathscr{K}, _) \quad s_1 = s - [v] \quad s(f) = \upsilon(\mathscr{H}, l)}{\langle send(f, v), s, h, c \cdot [l \mapsto (q_1, q_2), q_1 \mapsto m_1, q_2 \mapsto m_2]\rangle \hookrightarrow \langle skip, s_1, h, c \cdot [l \mapsto (q_1, q_2), q_1 \mapsto m_1, q_2 \mapsto m_2 \oplus (\upsilon_t, \emptyset, \emptyset)]\rangle}$$

$$\frac{s(f) = \upsilon(\mathscr{H}, l) \quad c_1 = c \cdot c_t \quad h_1 = h \cdot h_t \quad \upsilon_t(\mathscr{H}, _) \vee \upsilon_t(\mathscr{C}, _)}{\langle receive(f), s, h, c \cdot [l \mapsto (q_1, q_2), q_1 \mapsto (\upsilon_t, h_t, c_t) \oplus m_1]\rangle \hookrightarrow \langle \upsilon_t, s, h_1, c_1 \oplus [l \mapsto (q_1, q_2), q_1 \mapsto m_1]\rangle}$$

$$\frac{s(f) = \upsilon(\mathscr{H}, l) \quad \upsilon_t(\mathscr{K}, _)}{\langle receive(f), s, h, c \cdot [l_1 \mapsto (q_1, q_2), q_1 \mapsto (\upsilon_t, h_t, c_t) \oplus m_1]\rangle \hookrightarrow \langle \upsilon_t, s, h_1, c_1 \cdot [l_1 \mapsto (q_1, q_2), q_1 \mapsto m_1]\rangle}$$

$$\frac{ch \in dom(s) \quad ch \mapsto \upsilon(\mathscr{C}, (\upsilon(\mathscr{C}, l_1), \upsilon(\mathscr{C}, l_2)) \quad l_1 \in dom(c) \quad l_1 = (q_1, q_2)}{\langle red\ ch\ N, s, h, c\rangle \hookrightarrow \langle \upsilon(\mathscr{C}, l_1), s, h, c\rangle}$$

$$\frac{ch \in dom(s) \quad ch \mapsto \upsilon(\mathscr{C}, (\upsilon(\mathscr{C}, l_1), \upsilon(\mathscr{C}, l_2)) \quad l_2 \in dom(c) \quad l_2 = (q_1, q_2)}{\langle red\ ch\ N, s, h, c\rangle \hookrightarrow \langle \upsilon(\mathscr{C}, l_2), s, h, c\rangle}$$

Figura 3.3 Semantica operațională a comenzilor de comunicare

Regulile logice generate de sistemul de verificare a programelor sunt verificate de demonstratorul de teoreme SESSION-SLEEK.

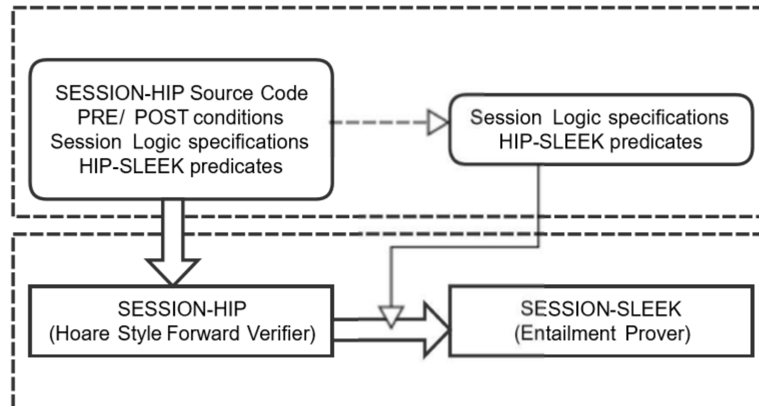

Figura 3.4 Principiul de verificare al SESSION-HIP-SLEEK

Acest demonstrator joacă un rol important în verificarea automată, dar poate fi omisă pentru moment. Vom reveni la această demonstrație în capitolul de implementare.

3.3.1 Limbajul de specificație

Am dezvoltat limbajul nostru de specificare ca o extensie a limbajului de specificații HIP (Fig. 3.5) din [27]. Limbajul permite utilizatorilor să definească predicate *spred* pentru a specifica proprietățile programului într-un domeniu combinat. Merită de reținut faptul că aceste predicate sunt construite ca niște constrângeri disjuncte Φ.

Shape predicate	*spred*	$::=$	$p(\mathtt{root}, v^*) \equiv \Phi$
Formula	Φ	$::=$	$\bigvee \sigma^*$
	σ	$::=$	$\exists\, v^* \cdot \kappa \wedge \pi$
Method specification	*mspec*	$::=$	*requires* Φ_{pr} *ensures* Φ_{po}
Session formula	S	$::=$	$\mathtt{emp} \mid ?r \cdot \Phi \mid !r \cdot \Phi \mid \sim S \mid S_1 ; S_2 \mid S_1 \vee S_2$
Heap formula	Δ	$::=$	$\Phi \mid \Delta_1 \vee \Delta_2 \mid \Delta \wedge \pi \mid \Delta_1 * \Delta_2 \mid \exists v \cdot \Delta$
	κ	$::=$	$\mathtt{emp} \mid v \mapsto c(v^*) \mid p(v^*) \mid \kappa_1 * \kappa_2 \mid \mathscr{C}(v, S)$
	π	$::=$	$\gamma \wedge \phi$
	γ	$::=$	$v_1 = v_2 \mid v = \mathtt{null} \mid v_1 \neq v_2 \mid v \neq \mathtt{null} \mid \gamma_1 \wedge \gamma_2$
Pureformula	ϕ	$::=$	$r : t \mid \varphi \mid b \mid a \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \neg \phi \mid \exists v \cdot \phi \mid \forall v \cdot \phi$
	b	$::=$	$\mathtt{true} \mid \mathtt{false} \mid v \mid b_1 = b_2$
	a	$::=$	$s_1 = s_2 \mid s_1 \leq s_2$
Presburger arithmetic	s	$::=$	$k^{\mathtt{int}} \mid v \mid k^{\mathtt{int}} \times s \mid s_1 + s_2$ $\mid -s \mid max(s_1, s_2) \mid min(s_1, s_2) \mid \lvert \mathtt{B} \rvert$
	φ	$::=$	$v \in \mathtt{B} \mid \mathtt{B}_1 = \mathtt{B}_2 \mid \mathtt{B}_1 \sqsubset \mathtt{B}_2 \mid \mathtt{B}_1 \sqsubseteq \mathtt{B}_2 \mid \forall v \in \mathtt{B} \cdot \phi \mid \exists v \in \mathtt{B} \cdot \phi$
Bag constraint	$\mathtt{B}$	$::=$	$\mathtt{B}_1 \sqcup \mathtt{B}_2 \mid \mathtt{B}_1 \sqcap \mathtt{B}_2 \mid \mathtt{B}_1 - \mathtt{B}_2 \mid \emptyset \mid \{v\}$

Figura 3.5 Limbajul de specificație.

O specificație de protocol pe un canal v este reprezentat prin $\mathscr{C}(\mathrm{v}, \mathrm{S})$ unde S reprezintă o specificație protocol și poate să fie o trimitere, o citire, o secvență de operațiuni de comunicare sau un set de secvențe de comunicare. S poate captura, de asemenea, proprietăți pure (de ex. de tip sau dată) sau proprietăți de heap a mesajelor transmise. O stare abstractă al unui program σ are în principal două părți: partea de heap (forma) κ în domeniul de separare și partea pură π în domeniul convex poliedric și domeniul de set (multi-set), în care π este format din partea de referințe γ, partea numerică ϕ și partea multi-set φ. $k^{\mathtt{int}}$ este o constantă întreagă. Simbolurile dreptunghiulare, ca $\sqsubset$, $\sqsubseteq$, $\sqcup$ și $\sqcap$ sunt operatori multi-set. Pe parcursul executării simbolice, starea abstractă a programului va fi o disjuncție de forma σ, notat prin Δ. O stare abstractă Δ poate fi normalizată la forma Φ [27].

Semantica acestui limbaj de specificare este dată în definiția 3.3.1, în care modelul relațional $(s, h, c) | = \Phi$ spune faptul că formula Φ se evaluează ca adevărat în (s,h,c). Pentru

a evita confuzia, trebuie să menționăm că $s_1\#s_2$ denotă un heap în care s_1 și s_2 sunt în domenii disjuncte. În plus $s_1 \cdot s_2$ indică unirea a două stive disjuncte s_1 și s_2. Operațiunile # și $\cdot$ pot fi aplicate pe heap-ul standard și heap-ul de canal cu același semnificație.

În continuare, vă prezentăm această definiție în mod informal. Regula $s,h,c \models \Phi_1 \vee \Phi_2$ spune că cel puțin una din formulele Φ_1 și Φ_2 trebuie să fie îndeplinită de către starea actuală. $s,h,c \models \exists v_{1...n} \cdot \kappa \wedge \pi$ indică faptul că stiva trebuie să îndeplinească π și stiva și heap-ul trebuie să îndeplinească κ. $s,h,c \models \kappa_1 * \kappa_2$ subliniază faptul că există două părți disjuncte ale heap-ului $h_1,\ c_1$ și $h_2,\ c_2$, în așa fel încât $h_1,\ c_1$ implică κ_1 și $h_2,\ c_2$ implică κ_2. $s,h,c \models \texttt{emp}$ presupune un heap gol. $s,h,c \models p \mapsto C(v_{1..n})$ spune că C trebuie să fie o structură de date, iar p trebuie să indice o locație heap, în care câmpurile lui C sunt stocate. În final, $s,h,c \models \mathscr{C}(v,S)$ spune că **v** indică în heap-ul canalelor o adresă $s(v) = l$ în care sunt stocate cozile de mesaje care corespund cu specificația de sesiune.

Definiție 3.3.1. (Modelul logicii de sesiune)

$$
\begin{array}{ll}
s,h,c \models \Phi_1 \vee \Phi_2 & \text{if } s,h,c \models \Phi_1 \quad \vee \quad s,h,c \models \Phi_2 \\
s,h,c \models \exists v_{1...n} \cdot \kappa \wedge \pi & \text{if } \exists v_{1...n}, s = [v_1 \mapsto \mathrm{v}_1, ..., v_n \mapsto \mathrm{v}_n] \wedge s \models \pi \wedge s,h,c \models \kappa \\
s,h,c \models \kappa_1 * \kappa_2 & \text{if } \exists h_1,h_2,c_1,c_2,\ h_1\#h_2 \wedge h_1 \cdot h_2 = h \wedge c_1\#c_2 \wedge c_1 \cdot c_2 = c \\
& s,h_1,c_1 \models \kappa_1 \wedge s,h_2,c_2 \models \kappa_2 \\
s,h,c \models \texttt{emp} & \text{if } dom(h) = \emptyset \wedge dom(c) = \emptyset \\
s,h,c \models p \mapsto C(v_{1..n}) & \text{if } \exists l, f_1, ..., f_n \quad s(p) = l \quad data\, C\{t_1\ f_1, ..., t_n\ f_n\} \in P \\
& \wedge h[l \mapsto C[f_1 \mapsto s(v_1), ..., f_n \mapsto s(v_n)]] \\
s,h,c \models \mathscr{C}(v,S) & \text{if } \exists l, q_i, q_o \quad s(v) = l \wedge c[l \mapsto (q_i, q_o)]
\end{array}
$$

3.3.2 Reguli de verificare

În continuare vom prezenta, regulile de verificare ale logicii de sesiune. Noțiunea fundamentală a mecanismului nostru de verificare este triplul Hoare. Un triplu $\{\Delta_1\}e\{\Delta_2\}$ din Fig.3.6 descrie modul în care executarea unei expresii e modifică o stare logică, care corespunde Δ_1 într-o stare logică, care corespunde Δ_2. Pentru a defini regulile noastre de verificare, avem nevoie de următoarele notații:

Notați 3.3.1. În formalismul nostru e reprezintă o expresie. În plus, vom folosi $\vdash \{\Delta_1\}e\{\Delta_2\}$ pentru a specifica un triplu Hoare, în care Δ_1 este pre-condiția și Δ_2 este post-condiția.

După ce avem toate ingredientele, vom oferi o descriere informală a regulilor noastre de verificare formală, pentru a ajuta cititorul să înțeleagă formulele fără dificultăți.

Să ne concentrăm pe regulile de comunicare din Fig. 3.6:

[SEND]

$$\frac{\begin{array}{l}\Phi_v = reach(v,\Delta_1)\\ \Delta_1 = \Delta * \mathscr{C}(f, \bigvee_{i\in I} !r\cdot\Phi_i; S_i) * \Phi_v\\ \exists j \in I \quad \Phi_v \vdash [v/r]\Phi_j \qquad \Delta_2 = \Delta * \mathscr{C}(f, S_j)\end{array}}{\vdash \{\Delta_1\} send(f,v) \{\Delta_2\}}$$

[RECEIVE]

$$\frac{\begin{array}{l}\Delta_1 = \Delta * \mathscr{C}(f, \bigvee_{i\in I} ?r\cdot\Phi_i; S_i)\\ \Delta_2 = \bigvee_{j\in I} \Delta * \mathscr{C}(f, S_j) * [res/r]\Phi_j\end{array}}{\vdash \{\Delta_1\} receive(f) \{\Delta_2\}}$$

[OPEN]

$$\frac{\Delta_1 = \Delta * \mathscr{C}(p_1, S) * \mathscr{C}(p_2, \sim S) \wedge f = (p_1, p_2)}{\vdash \{\Delta\} open(f)\ with\ S \{\Delta_1\}}$$

[CLOSE]

$$\frac{\Delta_1 = \Delta * \mathscr{C}(p_1, semp) * \mathscr{C}(p_2, semp) \wedge f = (p_1, p_2)}{\vdash \{\Delta_1\} close(f) \{\Delta\}}$$

Figura 3.6 Regulile de verificare ale primitivelor logicii de sesiune

- **[OPEN]** Conform acestei reguli, funcția *open* alocă cele două capete ale canalului și le asociază cu două specificații ale logicii de sesiune. Mai precis, specificația protocolului original, care decorează deschis va fi asociat cu primul capăt, iar dualul specificației anterioare va fi asociat cu al doilea capăt.
- **[CLOSE]** Regula verifică dacă variabila care este dată ca argument pentru funcția *close* indică un tuplu care are două referințe, cu cele două capete al unui canal. Specificația de sesiune a acestor puncte finale trebuie să indice un protocol gol, care este o cerință necesară în conformitate cu algebra procesului nostru. În cazul în care condițiile anterioare sunt satisfăcute, atunci canalul este dealocat, iar variabila este ștearsă din heap și stack.
- **[SEND]** Pre-condiția acestei reguli impune ca f și v să fie prezente în starea curentă. Mai mult decât atât, transmiterea acestui mesaj trebuie să fie pasul curent care trebuie executat, în conformitate cu specificația de protocol a acestui capăt al canalului. Acest lucru înseamnă, că mesajul trebuie să fie în conformitate cu una dintre specificațiile logice cerute de protocol. După cum se poate anticipa, accesul la datele transmise sunt pierdute după trimitere și specificația de protocol se modifică prin îndepărtarea în mod corespunzător a pasului de trimitere. Ca o consecință, un program care a fost verificat ca fiind corect, nu poate avea acces la resursele atașate la un mesaj după ce mesajul a fost trimis. Numai destinatarul mesajului îl va putea accesa, după ce va primit mesajul.
- **[RECEIVE]** În contrast cu *send*, regula *receive* este mult mai simplă și necesită doar o specificație de protocol, care începe cu o operațiune de *receive*. În cazul în care această condiție este îndeplinită de către capătul unui canal f, atunci operația de primire este validă, iar starea simbolică poate fi schimbată. Schimbarea în sine constă în consumarea tuturor specificațiilor *receive* menționate mai sus în protocol și adăugarea specificațiilor logice, ca disjuncție la starea actuală. Prin această adăugare, am acoperit toate acțiunile posibile de primire.

Capitolul 4
Dovedirea corectitudinii

4.1 Semantica operațională concurentă

4.2 Dovedirea respectării protocolului

4.3 Dovedirea corectitudinii

4.4 Sumar

În acest capitol vom extinde semantica operațională de la capitolul 2, în scopul de a oferi un cadru adecvat pentru demonstrarea corectitudinii mecanismului nostru de verificare. Apoi, vom furniza proprietățile necesare sub forma de teoreme pentru a demonstra corectitudinea verificării noastre. În cele din urmă, vom oferi o dovadă de soliditate pentru verificarea noastră.

Capitolul 5
SESSION-HIP-SLEEK

În acest capitol vom prezenta instrumentul nostru *SESSION-HIP-SLEEK*, dezvoltat pe parcursul tezei. Instrumentul este implementat în *Objective Caml (OCaml)* și este format din două subinstrumente: doveditorul *SESSION-SLEEK* și instrumentul de verificare *SESSION-HIP*. Uneltele sunt implementate ca o extensie a instrumentului *HIP-SLEEK* și facilitează verificarea automată a programelor care utilizează referințe putând comunica cu alte programe prin intermediul canalelor. Aceste instrumente acceptă ca intrare un nume de fișier și un set de opțiuni și produc ca ieșire un fișier textual. Mai precis instrumentul *SESSION-SLEEK* acceptă ca intrare un fișier cu extensia *slk* și un set de opțiuni. Fișierul poate conține un set de specificații de protocol, un set de predicate în logica de separare și un set de teoreme ce trebuie dovedite. Pentru acest fișier, *SESSION-SLEEK* poate produce o eroare de interpretare sau un set de rezultate pentru fiecare teoremă. În cazul în care dovedirea este validă ea produce un mesaj *OK*, în caz contrar, se arată teorema care nu poate fi demonstrată. Instrumentul permite să avem instrucțiuni speciale în fișierul de intrare pentru a putea afișa mai multe detalii despre procesul de dovedire.

Instrumentul *SESSION-HIP* necesită ca intrare un fișier cu extensia *ss* și un set de opțiuni. Fișierul ar trebui să conțină un set de predicate în logica de separare, un set de specificații de protocol și un set de funcții, cu pre- și post-condiții scrise folosind sintaxa lui *SESSION-HIP*. Dacă sintaxa este corectă, atunci instrumentul produce pentru fiecare funcție un rezultat. Rezultatul poate fi un mesaj *SUCCESS*, dacă funcția este corectă, în concordanță cu pre și post-condiția funcției sau un mesaj de eroare în cazul în care funcția are o eroare. În cazul în care există o eroare, mesajul de eroare oferă informațiile necesare, pentru a ajuta dezvoltatorul în depanarea programului.

5.1 Demonstratorul SESSION-SLEEK

Această secțiune este dedicată pentru a prezenta demonstratorul SESSION-SLEEK. După cum se poate observa în Fig.5.1, rolul acestui instrument este de a dovedi teoremele generate de SESSION-HIP. SESSION-HIP generează un set de teoreme, care sunt o combinație a logicii de sesiune, a logicii de separare și a logicii de ordinul întâi. Așa cum este de așteptat, nici unul dintre demostratoarele existente nu pot trata aceste predicate. Pentru a permite demonstrarea acestor formule, SESSION-SLEEK folosește un set de demonstatoare, dar are, de asemenea, un set de mecanisme proprii de demonstrație. O teoremă generată de SESSION-HIP arată în felul următor:

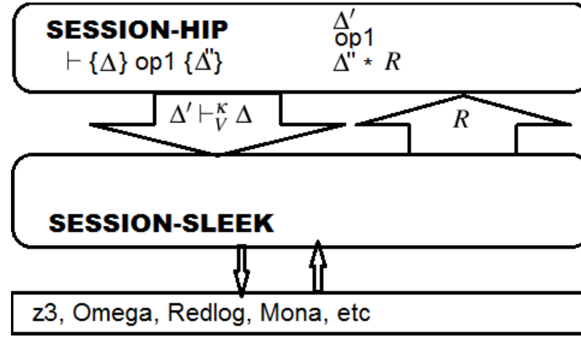

Figura 5.1 SESSION-SLEEK

$$\Delta' \vdash^{\kappa}_{V} \Delta * R \qquad (1)$$
$$\kappa * \Delta' \vdash \exists V \cdot (\kappa * \Delta) * R \qquad (2)$$

Conform regulilor anterioare [1], procesul de demonstrare constă în a verifica dacă formula antecedentă Δ' este suficient de precisă pentru a satisface consecința Δ și pentru a permite crearea unui rezidiu R. Merită de menționat, că din moment ce demonstratorul *SESSION-SLEEK* nu este completă, există cazuri în care o demonstrație este posibilă, dar nu poate fi demonstrată de către demonstrator.

Din moment ce există multe reguli de demonstație în *SLEEK* și pentru că prezentarea acestor reguli nu poate contribui în mod semnificativ la înțelegerea acestei teze, nu le vom prezenta aici în detaliu, acestea fiind prezentate foarte bine în mai multe articole ca [32, 27].

5.1.1 Demonstrarea corectitudinii trimiteri și primiri

În continuare vom prezenta regulile noastre de demonstrație. În instrumentul nostru am extins demonstratorul logicii de separare SLEEK [27] pentru a sprijini demonstrarea teoremelor din logica de sesiune (vezi Fig. 5.2). Rolul extensiei noastre este să verifice dacă substituirile în formulele din logica de sesiune sunt corecte. În cazul trimiterii, formula substituită trebuie să fie contravariantă, în timp ce în cazul operațiunii de primire, formula trebuie să satisfacă regula de covarianță, în raport cu formula inițială.

Instrumentul trebuie să utilizeze și regulile de compatibiliate pentru demonstrarea teoremelor. Regulile sunt prezentate în Fig. 5.3.

Putem observa, că formula de sesiune care corespunde trimiterii subsumează formula de sesiune corespunzătoare primirii. Ca un exemplu, în cazul disjuncțiilor, partea care trimite poate avea mai puține disjuncții, decât entitatea care primește mesajul.

[1] Notați că (1) este o reprezentare alternativă pentru (2)

$$\frac{\Delta_2 \vdash \Delta_1}{!r \cdot \Delta_1 \vdash !r \cdot \Delta_2}\,[\text{OUTPUT}] \qquad \frac{\Delta_1 \vdash \Delta_2}{?r \cdot \Delta_1 \vdash ?r \cdot \Delta_2}\,[\text{INPUT}]$$

$$\frac{e_1 \vdash e_2 \quad rest_1 \vdash rest_2}{e_1; rest_1 \vdash e_2; rest_2}\,[\text{SEQ-CHAN}] \qquad \frac{S_1 \vdash S_2}{\mathscr{C}(c,S_1) \vdash \mathscr{C}(c,S_2)}\,[\text{MATCH-CHAN}]$$

Figura 5.2 Reguli de demonstrare pentru logica de sesiune.

$$\frac{\Delta_2 \vdash \Delta_1 \quad \neg(\Delta_3 \wedge \Delta_1)}{!r_1 \cdot \Delta_1 \vee !r_2 \cdot \Delta_3 \vdash !r_1 \cdot \Delta_2}\,[\text{OUTPUT-OR}] \qquad \frac{\Delta_1 \vdash \Delta_2 \quad \neg(\Delta_3 \wedge \Delta_2)}{?r_1 \cdot \Delta_1 \vdash ?r_1 \cdot \Delta_2 \vee ?r_2 \cdot \Delta_3}\,[\text{INPUT-OR}]$$

Figura 5.3 Regulile de demonstație pentru disjuncți.

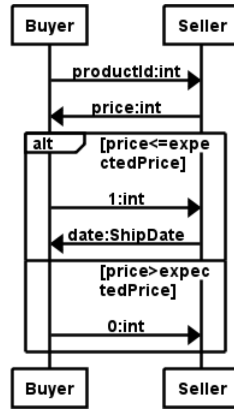

Figura 5.4 Protocol simplu de afaceri

Pentru simplificarea prezentației, vom prezenta regulile de demonstrație folosind un exemplu. În acest exemplu, conversația începe de la cumpărător, care trimite denumirea produsului, care este de tipul **string**, către vânzător. Vânzătorul trebuie să răspundă la această cerere cu un preț. Dacă cumpărătorul este mulțumit cu prețul primit, atunci acesta va acepta livrarea și va primi de la vânzător data livrării. Dacă prețul nu satisface așteptările cumpărătorului, atunci acesta va închide comunicația. Acest exemplu este modelat în Fig. 5.4. Putem rezuma acest protocol utilizând următoarele specificații în logica de sesiune:

$$\begin{aligned} \texttt{buyer_chan} &\equiv !\texttt{String}; ?\texttt{r:int} \cdot \texttt{r}{>}5; ((!1; ?\texttt{Date}) \vee !0) \\ \texttt{seller_chan} &\equiv \sim\texttt{buyer_chan} \\ \texttt{buyer_chan} &\equiv ?\texttt{String}; !\texttt{r:int} \cdot \texttt{r}{>}5; ((?1; !\texttt{Date}) \vee ?0) \end{aligned}$$

Ca un exemplu, trebuie să precizăm o specificație mai puternică pentru vânzător, în care prețul produsului vândut este cel puțin de 10 unități, după cum urmează:

$$\texttt{seller_sp} \equiv ?\texttt{String}; !\texttt{r:int} \cdot \texttt{r}{>}10; ((?1; !\texttt{Date}) \vee ?0)$$

Un cod care implementează protocolul anterior este prezentat în Fig.5.5.

```
open(c) with buyer_chan;
(buyer(c,prod) || seller(c));
close(c);
```

```
void buyer(Chan c,String p)
  requires 𝒞(c,buyer_sp)
  ensures 𝒞(c,emp)
{ send(c,p);
 Double price = receive(c);
 Double budget = getBudget();
 if (price <= budget) then{
  send(c,1);
  ShipDate sd = receive(c);
 } else send(c,0);
}
```

```
void seller(Chan c)
  requires 𝒞(c,seller_sp)
  ensures 𝒞(c,emp)
{ String p = receive(c);
 send(c,getPrice(p));
 int usr_opt = receive(c);
 if (usr_opt==1){
  ShipDate sd = getDate(p);
  send(c,sd);
 } else
  assert usr_opt = 0;
}
```

Figura 5.5 Implementarea unui protocol simplu de afaceri.

În această aplicație, canalul este deschis în procesul principal prin open, care ia ca argument specificația inițială al canalului buyer_chan. O referință la canalul deschis cu specificația buyer_chan este trimis la codul firului buyer în timp ce cealaltă referință cu specificația duală seller_chan este trimis la procesul seller. Cele două procese rulează în paralel. Fiecare proces poate avea propria sa specificație de protocol care trebuie să corespundă cu specificația de protocol primită ca parametru. Procesul seller, având ca specificația seller_sp care impune o constrângere mai puternică asupra prețului trimis, folosind $r>10$ în loc de $r>5$ definit în specificația canalului seller_chan. Atunci când un canal este trimis ca parametru la un fir, va trebui să ne asigurăm că specificația canalului însumează cele specificate în specificațiile firului. Pentru firul seller din exemplul nostru, acest lucru înseamnă că $\mathscr{C}(\texttt{c},\texttt{seller_chan}) \vdash \mathscr{C}(\texttt{c},\texttt{seller_sp})$. Această relație poate fi demonstrată prin SESSION-SLEEK folosind regula $(OUTPUT)$ după cum se poate vedea mai jos:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dots}{((?1;!\texttt{Date})\vee?0) \vdash ((?1;!\texttt{Date})\vee?0)}\;[\text{OUTPUT}] \quad \begin{array}{c}\texttt{r}>10 \vdash \texttt{r}>5 \\ !\texttt{r}\cdot\texttt{r}>5 \vdash !\texttt{r}\cdot\texttt{r}>10\end{array}}{!\texttt{r:int}\cdot\texttt{r}>5;((?1;!\texttt{Date})\vee?0) \vdash !\texttt{r:int}\cdot\texttt{r}>10;((?1;!\texttt{Date})\vee?0)}\;[\text{INPUT}]}{?\texttt{String};!\texttt{r:int}\cdot\texttt{r}>5;((?1;!\texttt{Date})\vee?0) \vdash ?\texttt{String};!\texttt{r:int}\cdot\texttt{r}>10;((?1;!\texttt{Date})\vee?0)}\;[\text{FOLD}]}{?\texttt{String};!\texttt{r:int}\cdot\texttt{r}>5;((?1;!\texttt{Date})\vee?0) \vdash \texttt{seller_sp}}\;[\text{UNFOLD}]}{\texttt{seller_chan} \vdash \texttt{seller_sp}}\;[\text{MATCH-CHAN]}}{\mathscr{C}(\texttt{c},\texttt{seller_chan}) \vdash \mathscr{C}(\texttt{c},\texttt{seller_sp})}$$

Procesul de demonstrare trebui să fie citit de jos în sus. Prima regulă $[MATCH-CHAN]$ din Fig.5.2 încearcă să potrivească în mod succesiv canalele, care pot fi dovedite a fi identice. În cazul nostru, prin aplicarea acestei reguli descoperim că $\mathscr{C}(c, seller_chan)$ și $\mathscr{C}(c, seller_sp)$ sunt identice, prin urmare, formula de dovedit poate fi redus la $seller_chan \vdash seller_sp$. În continuare, prin aplicarea regulii de $[UNFOLD]$ de la [27], vom înlocui predicatul *seller_chan* cu definiția sa. În același fel înlocuim *seller_ch* cu definiția sa. În continuare, prin aplicarea regulii de $[INPUT]$ de la Fig.5.2, putem reduce specificația protocolului consumând termenul $?String$ din protocol. Regula $[OUTPUT]$ poate fi de asemenea demonstrată Fig.5.2 pentru că însumarea pentru operația de trimitere este contravariantă. Restul dovedirii este trivială, pentru că partea din stânga și din dreapta a formulei este identică.

Pe de altă parte, dacă luăm în considerare următoarele specificații `buyer`:

$$\texttt{buyer_sp} \equiv \texttt{!String};\texttt{?r:int}\cdot\texttt{r}{>}10;((\texttt{!1};\texttt{?Date})\vee\texttt{!0})$$

atunci putem observa, că acesta impune o proprietate mai slabă asupra prețului, folosind $\texttt{r}{>}0$ în loc de $\texttt{r}{>}5$. Pentru firul `buyer` din exemplul nostru, acest lucru înseamnă că $\mathscr{C}(\texttt{c},\texttt{buyer_chan}) \vdash \mathscr{C}(\texttt{c},\texttt{buyer_sp})$. Această relație poate fi, de asemenea, dovedit de SESSION-SLEEK folosind regula noastră de demonstrație $(INPUT)$ după cum urmează:

...

$((\texttt{!1};\texttt{?Date})\vee\texttt{!0}) \vdash ((\texttt{!1};\texttt{?Date})\vee\texttt{!0})$

[INPUT]

$\texttt{r}{>}5 \vdash \texttt{r}{>}0$

$\texttt{?r}\cdot\texttt{r}{>}5 \vdash \texttt{?r}\cdot\texttt{r}{>}0$

$\texttt{?r:int}\cdot\texttt{r}{>}5;((\texttt{!1};\texttt{?Date})\vee\texttt{!0}) \vdash \texttt{?r:int}\cdot\texttt{r}{>}0;((\texttt{!1};\texttt{?Date})\vee\texttt{!0})$

[OUTPUT]

$\texttt{!String};\texttt{?r:int}\cdot\texttt{r}{>}5;((\texttt{!1};\texttt{?Date})\vee\texttt{!0}) \vdash \texttt{!String};\texttt{?r:int}\cdot\texttt{r}{>}0;((\texttt{!1};\texttt{?Date})\vee\texttt{!0})$

[FOLD]

$\texttt{!String};\texttt{?r:int}\cdot\texttt{r}{>}5;((\texttt{!1};\texttt{?Date})\vee\texttt{!0}) \vdash \texttt{seller_sp}$

[UNFOLD]

$\texttt{buyer_chan} \vdash \texttt{buyer_sp}$

[MATCH-CHAN]

$\mathscr{C}(\texttt{c},\texttt{buyer_chan}) \vdash \mathscr{C}(\texttt{c},\texttt{buyer_sp})$

Procesul de demonstrare este foarte similară cu cea anterioară, de aceea, vom explica doar $[INPUT]$, care impune ca restricția logică să fie covariantă, deci $?r\cdot r{>}5 \vdash ?r\cdot r{>}0$ poate fi demonstrată foarte simplu.

5.1.2 Demonstrarea corectitudini alegerii interne și externe

În afara acestor patru reguli de demonstrare, avem două reguli în Fig.5.3 care provin din regulile de compatibilitate. Pentru a ilustra aceste teoreme vom oferi o implementare în Fig.5.6, unde *buyer_spc* și *seller_spc* pot fi definite în felul următor:

$$\begin{array}{lcl} \texttt{buyer_spc} & \equiv & !\texttt{String};?\texttt{r:int}\cdot\texttt{r}>5;!1;?\texttt{Date} \\ \texttt{seller_spc} & \equiv & ?\texttt{String};!\texttt{r:int}\cdot\texttt{r}>5;((?1;!\texttt{Date})\vee(?2;!\texttt{int})\vee?0) \end{array}$$

Aceste două protocoale sunt compatibile, în conformitate cu normele de compatibilitate, prin urmare instrumentul $SESSION-SLEEK$ ne furnizează regulile necesare (Fig.5.3) pentru a dovedi că specificațiile canalelor însumează specificațiile firelor.

Specificația `buyer_spc` al procesului `buyer` impune un protocol mai puternic, folosind $!1;?\texttt{Date}$ în loc de $(!1;?\texttt{Date})\vee!0$. Acest lucru înseamnă că formula $\mathscr{C}(\texttt{c},\texttt{buyer_chan})\vdash\mathscr{C}(\texttt{c},\texttt{buyer_spc})$ trebuie să fie demonstrată, de SESSION-SLEEK folosind regula de demonstrație $(OUTPUT-OR)$ după cum se poate vedea mai jos:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{...}{!1;?\texttt{Date}\vdash !1;?\texttt{Date}}\ [\text{OUTPUT-OR}]}{((!1;?\texttt{Date})\vee!0)\vdash !1;?\texttt{Date}}\ [\text{INPUT}]}{!\texttt{r:int}\cdot\texttt{r}>5;((!1;?\texttt{Date})\vee!0)\vdash ?\texttt{r:int}\cdot\texttt{r}>0!1;?\texttt{Date}}\ [\text{OUTPUT}]}{!\texttt{String};?\texttt{r:int}\cdot\texttt{r}>5;((!1;?\texttt{Date})\vee!0)\vdash !\texttt{String};?\texttt{r:int}\cdot\texttt{r}>0!1;?\texttt{Date}}\ [\text{FOLD}]}{!\texttt{String};?\texttt{r:int}\cdot\texttt{r}>5;((!1;?\texttt{Date})\vee!0)\vdash \texttt{seller_spc}}\ [\text{UNFOLD}]}{\texttt{buyer_chan}\vdash\texttt{buyer_spc}}\ [\text{MATCH-CHAN]}}{\mathscr{C}(\texttt{c},\texttt{buyer_chan})\vdash\mathscr{C}(\texttt{c},\texttt{buyer_spc})}$$

Procesul de demonstrare este foarte similar cu cele anterioare, prin urmare, vom explica doar $[OUTPUT-OR]$, care permite ca alegerea internă să aibă mai puține opțiuni decât era specificat inițial, astfel încât corectitudinea formulei $((!1;?Date)\vee!0)\vdash !1;?Date$ poate fi demonstrată cu ușurință.

În final, dacă luăm în considerare specificațiile protocolului `seller` din Fig.5.6 atunci putem observa, că aceasta impune pentru alegerea externă mai multe opțiuni, față de cum a fost specificat inițial. În consecință demonstrația poate fi făcut cu ușurință, așa cum se poate

`void buyer(Chan c, String p)`
`  requires` $\mathscr{C}$`(c, buyer_spc)`
`  ensures` $\mathscr{C}$`(c, emp)`
`{ send(c, p);`
` Double price = receive(c);`
` send(c, 1);`
` ShipDate sd = receive(c);`
`}`
	`void seller(Chan c)`
	`  requires` $\mathscr{C}$`(c, seller_spc)`
	`  ensures` $\mathscr{C}$`(c, emp)`
	`{ String p = receive(c);`
	` send(c, getPrice(p));`
	` int usr_opt = receive(c);`
	` if (usr_opt==1){`
	`  ShipDate sd = getDate(p);`
	`  send(c, sd);`
	` } else if (usr_opt==2){`
	`  send(c, 5);`
	` } else`
	`  assert usr_opt = 0;`
	` }`

Figura 5.6 Procese compatibile în logica de sesiune.

vedea mai jos:

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\cdots}{?1;!\mathtt{Date} \vdash ?1;!\mathtt{Date}}\ [\text{INPUT-OR}]}{((?1;!\mathtt{Date})\vee ?0) \vdash ?1;!\mathtt{Date}}\ [\text{OUTPUT}]}{!\mathtt{r{:}int}\cdot \mathtt{r}>5;((?1;!\mathtt{Date})\vee ?0) \vdash !\mathtt{r{:}int}\cdot \mathtt{r}>5;?1;!\mathtt{Date}}\ [\text{INPUT}]}{?\mathtt{String};!\mathtt{r{:}int}\cdot \mathtt{r}>5;((?1;!\mathtt{Date})\vee ?0) \vdash ?\mathtt{String};!\mathtt{r{:}int}\cdot \mathtt{r}>5;?1;!\mathtt{Date}}\ [\text{FOLD}]}{?\mathtt{String};!\mathtt{r{:}int}\cdot \mathtt{r}>5;((?1;!\mathtt{Date})\vee ?0) \vdash \mathtt{seller_spc}}\ [\text{UNFOLD}]}{\mathtt{seller_chan} \vdash \mathtt{seller_spc}}\ [\text{MATCH-CHAN}]}{\mathscr{C}(\mathtt{c},\mathtt{seller_chan}) \vdash \mathscr{C}(\mathtt{c},\mathtt{seller_spc})}$$

5.2 SESSION-HIP

5.2.1 Expresivitatea totală a logicii de separare

5.2.2 Logica de sesiune de ordin superior

Capitolul 6
Compararea logicii de sesiune cu abordări similare

6.1 SESSION-HIP-SLEEK vs Heap-Hop

6.2 SESSION-HIP-SLEEK vs Session C

6.3 SESSION-HIP-SLEEK vs ParTypes

6.4 SESSION-HIP-SLEEK vs instrumente cu stări de tip

6.4.1 SESSION-HIP-SLEEK vs MOOSE

6.4.2 SESSION-HIP-SLEEK vs Session Java

6.4.3 SESSION-HIP-SLEEK vs BICA

6.5 Sumar

În acest capitol, am prezentat cele mai competitive șase instrumente pentru verificarea protocoalelor. Am comparat fiecare dintre aceste instrumente cu instrumentul nostru, dând o serie de exemple concrete și evidențiind cele mai importante diferențe. În scopul de a rezuma comparațiile, vom prezenta cele mai importante diferențe în Tabelele 6.1, 6.2.

Tabelul 6.1 poate fi interpretat în modul următor:

- Prima coloană conține denumirea instrumentelor.
- A doua coloană a tabelului conține o bifă (✓), în cazul în care instrumentul are suport pentru delegare sau cruce (✗), în cazul în care nu are nici un suport.
- A treia coloană conține o bifă (✓), în cazul în care instrumentul are suport pentru alegerea internă și externă sau o cruce (✗), în cazul în care nu are nici un suport.
- A patra coloană conține o bifă (✓), în cazul în care instrumentul are suport pentru buclă sau o cruce (✗), în cazul în care nu are nici un suport.

Tabela 6.1 Propietățiile uneltelor 1

Tool	Delegation	Internal External Choice	Loop	Require Special Primitives	Copyless Message Passing
SESSION-HIP-SLEEK	✓	✓	✓	✗	✓
Heap-Hop	✗	✓	✓	✓	✓
Session C	✗	✓	✓	✓	✗
ParType	✗	✗	✓	✓	✗
MOOSE	✓	✓	✓	✓	✗
Session Java	✓	✓	✓	✓	✗
Bica	✓	✓	✗	✓	✗

Tabela 6.2 Propietățiile uneltelor 2

Tool	Type Constraint	Logic Constraint on Data	Data Shape Predicate	Support Broadcast	Support Aliasing
SESSION-HIP-SLEEK	✓	✓	✓	✗	✓
Heap-Hop	✗	✓	✗	✗	✓
Session C	✓	✗	✗	✓	✗
ParType	✓	✗	✗	✓	✗
MOOSE	✓	✗	✗	✗	✗
Session Java	✓	✗	✗	✗	✗
Bica	✓	✗	✗	✗	✗

- Penultima coloană a tabelului conține o bifă (✓), în cazul în care instrumentul necesită primitive speciale pentru alegerile interne și externe sau cruce (✗), în cazul în care nu sunt necesare astfel de primitive. (De reținut, că în cazul în care instrumentul de verificare necesită aceste primitive, atunci limbajul trebuie să le aibă, în caz contrar, verificarea nu funcționează. Deci, este mai bine dacă instrumentul nu necesită prezența primitivelor.)
- Ultima coloană a tabelului conține o bifă (✓), în cazul în care instrumentul are suport pentru verificarea transmiterii mesajelor prin referință sau cruce (✗), în cazul în care nu are suport.

Tabelul 6.2 poate fi interpretat după cum urmează:

- Prima coloană conține denumirea instrumentelor.
- A doua coloană a tabelului conține o bifă (✓) în cazul în care instrumentul are suport pentru verificarea de tip sau cruce (✗) în cazul în care nu are suport

- A treia coloană conține o bifă (✓) în cazul în care instrumentul are suport pentru constrângerile logice a datelor transmise sau o cruce (✗) în cazul în care nu are suport.
- A patra coloană conține o bifă (✓) în cazul în care instrumentul are suport pentru logica de separare în constrângerea datelor transmise sau cruce (✗) în cazul în care nu are suport.
- Penultima coloană a tabelului conține o bifă (✓) în cazul în care instrumentul are suport pentru broadcasting sau cruce (✗) în cazul în care nu are suport
- Ultima coloană a tabelului conține o bifă (✓) în cazul în care instrumentul are suport pentru verificarea programelor care utilizează referință sau o cruce (✗) în cazul în care nu are suport

Spre deosebire de abordările anterioare, am dezvoltat un instrument bazat pe logica de sesiune, care utilizează disjuncțiile pentru specificarea și verificarea implementărilor protocoalelor de comunicație. Chiar dacă logica se bazează pe sesiuni de canal cu două capete, poate să trateze delegarea, prin utilizarea unor canale de ordin superior. Spre deosebire de instrumentele din subsecțiunile 6.4.1, 6.4.2, 6.4.3, propunerea noastră folosește aceeași metode de canal de trimitere/primire ca si pentru valori. Mai mult decât atât, datorită utilizării a disjuncțiilor pentru a modela deciziile interne și externe, limbajul poate utiliza doar instrucțiunile condiționate convenționale pentru a sprijini ambele tipuri de alegeri. În contrast, toate soluțiile prezentate anterior necesită o extindere a limbajului cu un set de instrucțiuni condiționate specializate, construite pentru a modela alegeri interne și externe. Ca urmare, toate abordările anterioare au fost limitate la astfel de limbaje cu instrucțiuni condiționate specializate, care au redus drastic aplicabilitatea lor.

În plus, limbajul nostru de specificitație se bazează pe o extensie a logicii de separare, astfel sprijină verificarea programelor care manipulează heap-ul și transmit mesajele prin referință. Comparativ cu instrumentul din subsecțiunea 6.1 putem observa, că instrumentul lor se bazează pe contracte globale bazate pe stări, în timp ce instrumentul nostru mai general, se bazează pe o logică de sesiune și este construit ca o extensie a logicii de separare cu disjuncții pentru a sprijini instrucțiunile condiționate standard. Mai mult decât atât, putem garanta, de asemenea, conversia corectă de tipuri prin verificarea comunicației. În plus, putem garanta că referințele și proprietățile valorilor transmise în canale sunt captate în mod corespunzător. În cele din urmă, prin utilizarea unei relații de însumare, putem preveni interblocarea firelor de execuție. Pentru a fi și mai aproape de realitate, în modelul nostru trimiterea este asincronă.

Capitolul 7
Aplicații in industria de transport feroviar

7.1 Modelarea și verificarea Interlocking-ului utilizând logica de sesiune

7.1.1 Introducere

7.1.2 Un exemplu ilustrativ

7.1.3 Codificarea cerințelor in logica de sesiune

7.1.4 Verificare protocol

7.1.5 Rezultate experimentale

7.1.6 Concluzii

7.2 Programe de protecție automată a trenurilor

7.2.1 Rezultate experimentale

7.3 Sumar

În acest capitol, am prezentat două posibilități de utilizare a logici noastre de sesiunii, în industria de dezvoltare a programelor pentru industria de transport feroviar. În primul subcapitol, am prezentat o metodă completă de dezvoltare și verificare pentru dezvoltarea de programe pentru interlocking utilizând abordarea de dezvoltare geografică. Metoda de codare a cerințelor de interlocking și proiecția entităților au fost prezentate în [71, 69]. Verificarea codului sursă și rezultatele experimentale sunt noi. Rezultatele experimentale au arătat că eficiența metodei noastre de verificare este adecvat pentru utilizarea în industria feroviară. În al doilea exemplu, prezentăm aplicabilitatea teoriei noastre în dezvoltarea de programe de control automat al trenurilor. Pentru a demonstra aplicabilitatea teoriei noastre, am codat în logica noastră de sesiune un set de cerințe din specificația TBL1 + furnizate de Siemens, și, de asemenea, trei specificațiile din OpenETCS. Pentru verificare am implementat și codul sursă corespunzător fiecărei cerințe, iar rezultatele au arătat că metoda noastră este adecvată pentru verificarea acestor coduri sursă.

Capitolul 8
Concluzii şi direcţii de cercetare

Odată cu creșterea explozivă a infrastructurilor de mari dimensiuni (cum ar fi internetul, GSM, etc.) controlate prin calculatoare, a crescut dramatic nevoia pentru dezvoltarea unor programe distribuite. Aceste programe sunt folosite în prezent în aproape toate domeniile: servicii financiare, servicii comerciale, divertisment, sistemul de sănătate, telecomunicații, apărare, automatizări industriale etc..

Pe de altă parte, proiectarea și dezvoltarea acestor programe este destul de dificilă. Dificultățile apar atât în asigurarea corectitudinii pentru garantarea siguranței, cât și în obținerea unor performanțe crescute de execuție. Din perspectiva siguranței, trebuie să se ia în considerare o serie de probleme noi, ca, de exemplu, utilizarea distribuită a resurselor comune sau sincronizarea acestor procese.

În ciuda acestor probleme de siguranță, datorită conceptului de separare aceste programe distribuite sunt utilizate pe scară largă în sistemele de siguranță critice în industrie, cum ar fi controlul zborului, controlul traficului aerian, automatizări industriale, industria auto și de cale ferată, ca să crească fiabilitatea și scalabilitatea acestor sisteme. Ca o consecință, o mașină modernă are mai mult de 70 de unități de control [22] conectate prin cel puțin 5 sisteme diferite de comunicare, o aeronavă modernă folosește cel puțin 7 calculatoare numai pentru sistemele fly-by-wire [19].

Prin urmare, în această teză am vizat problema verificării programelor distribuite în medii unde siguranța este critică. Ca o soluție noi propunem codificarea comportamentului de comunicare al acestor sisteme în limbajul nostru de specificație protocol (logică de sesiune) și verificarea în mod automat a corectitudinii codului sursă, în conformitate cu aceste specificații. În cele ce urmează, vom detalia principalele contribuții ale acestei teze. Contribuțiile sunt evidențiate în cele ce urmează:

- **Logica de sesiune:** În capitolul 3, în conformitate cu ipoteza noastră am prezentat o nouă logică de sesiune cu disjuncții pentru a specifica și a verifica implementarea corectă a protocoalelor de comunicare în programe. Logica noastră este definită momentan pentru canale cu două capete, totuși este capabilă să capteze în mod natural delegațiile, prin utilizarea unor canale de ordin superior. Din cauza utilizării disjuncțiilor pentru a modela alegerile interne și externe, putem folosi doar declarații condiționate pentru a sprijini astfel de decizii, spre deosebire de construcțiile de decizii specializate în propunerile anterioare. După cum propunerea noastră se bazează pe o extensie a logicii de separare, putem verifica programele care manipulează heap-ul și transmit mesajele prin referință. Logica de sesiune a fost prezentată în [29, 70] și este implementată ca o extensie a uneltei HIP/SLEEK [26].

- **Demonstrația corectitudinii:** După ce am definit logica noastră de sesiune, în capitolul 4 am dovedit corectitudinea teoriei noastre. În acest scop, definim o semantică operațională mai precisă pentru limbajul nostru de programare. Această semantică este în strânsă legătură cu semantica inițială, dar ne permite o explorare mai bună a legăturilor dintre limbajul de programare și specificația de protocol în procesul de demonstrație. Prin această demonstrație matematică garantăm pentru utilizatorii instrumentului nostru că un program care a fost verificat de către instrumentul nostru se conformează în mod sigur protocoalelor definite în logica de sesiune și asociate acestui program. Acesta este un pas important și ne permite să garantăm fiabilitatea instrumentului și a teoriei noastre.

- **Instrumentul nostru de verificare automată:** În capitolul 5 am prezentat instrumentul nostru SESSION-HIP-SLEEK, dezvoltat pe parcursul doctoratului. Instrumentul este implementat în *Obiectiv Caml (OCaml)* și este compus din două instrumente, și anume: un doveditor deductiv *SESSION-SLEEK* și un instrument de verificare *SESSION-HIP*. Uneltele sunt construite ca o extensie a instrumentelor oferite de *HIP-SLEEK* și facilitează verificarea automată a programelor care utilizează referințe și care pot comunica cu alte programe prin intermediul canalelor. Aceste instrumente acceptă ca intrare un nume de fișier și un set de opțiuni și produc o ieșire textuală. Mai precis, instrumentul *SESSION-SLEEK* acceptă ca intrare un fișier cu extensia *slk* și un set de opțiuni. Fișierul poate conține un set de specificații de protocol, un set de predicat în logica de separare și un set de cerințe de demonstrație. Pentru acest fișier, instrumentul *SESSION-SLEEK* poate produce o eroare de interpretare sau un set de rezultate pentru fiecare demonstrație. În cazul în care demonstrația este validată, ea produce un mesaj *OK*, în caz contrar, instrumentul indică formula care nu poate fi dovedită. Instrumentul permite să se amplaseze mai multe instrucțiuni speciale în fișierul de intrare pentru a permite afișarea mai multor detalii despre procesul de dovedire. Instrumentul *SESSION-HIP* necesită ca intrare un fișier cu extensia *ss* și, de asemenea, un set de opțiuni. Fișierul poate să conțină un set de predicate în logica de separare, un set de specificații de protocol și un set de funcții, cu pre- și post-condiții. Ca rezultat, în cazul în care sintaxa este corectă instrumentul produce pentru fiecare funcție un rezultat în formă de text. Rezultatul poate fi un mesaj *SUCCES* dacă funcția este corectă în raport cu pre- și post-condiții asociată funcției sau un mesaj de eroare în cazul în care funcția are o eroare. Mesajul de eroare oferă informațiile necesare, ca, de exemplu, linia de cod și predicatul logic care nu au putut fi dovedite, pentru a ajuta dezvoltatorul în depanarea programului.

- **Aplicații în industria transportului feroviar:** În capitolul 7, am prezentat două posibile utilizări a logicii noastre de sesiune în industria de dezvoltare de programe pentru industria de transport feroviar. În primul caz, am prezentat un proces complet și o metodă de verificare automată a programelor de interlocking care sunt dezvoltate utilizând procedeul de dezvoltare geografică. Metoda de codificare a cerințelor de interlocking și proiecția entităților au fost prezentate în [71, 69]. Verificarea codului sursă și rezultatele experimentale sunt noi. Rezul-

tatele au arătat că eficiența metodei noastre de verificare este adecvată pentru utilizarea în dezvoltarea programelor interlocking. În al doilea studiu de caz, am prezentat aplicabilitatea teoriei noastre în dezvoltarea de programe pentru controlul automat al sistemelor de protecție a trenurilor. Pentru a demonstra aplicabilitatea teoriei noastre, am codificat în logica de sesiune un set de cerințe din specificația *TBL1+* furnizate de Siemens și, de asemenea, trei specificații din *OpenETCS*. Am implementat codurile sursă corespunzătoare, iar rezultatele au arătat că această metodă de verificare este adecvată pentru verificarea acestor coduri sursă. Desigur, aplicabilitatea teoriei noastre nu se limitează la aceste cazuri de utilizare, și pot fi aplicate ori de câte ori corectitudinea comunicării trebuie să fie verificată. O aplicație potențială ar putea fi, de asemenea, verificarea implementării protocoalelor *CBTC*.

- **Comparația cu alte abordări:** În capitolul 6, am prezentat cele mai competitive șase instrumente pentru verificarea protocoalelor care folosesc tipuri de sesiune. Am comparat fiecare dintre aceste instrumente cu instrumentul nostru, oferind o serie de exemple concrete și evidențiind cele mai importante diferențe. Spre deosebire de abordările anterioare, am dezvoltat un instrument bazat pe logica noastră de sesiune care utilizează în mod natural disjuncțiile pentru a specifica și verifica implementarea protocoalelor de comunicație. Deși instrumentul nostru suportă o logică ce poate trata numai canale cu două capete, acesta poate trata delegarea prin utilizarea unor canale de ordin superior. Spre deosebire de instrumentele din subsecțiunile 6.4.1, 6.4.2, 6.4.3, soluția noastră folosește aceleași metode de trimitere și primire pentru transmiterea canalelor ca și pentru transmiterea valorilor. Mai mult decât atât, datorită utilizării disjuncților pentru a modela alegerile interne și externe, avem nevoie doar de utilizarea instrucțiunilor condiționate convenționale pentru a sprijini ambele tipuri de alegeri. În contrast, toate soluțiile prezentate anterior necesită ca limbajul de programare să fie extins cu un set de instrucțiuni condiționate noi pentru a modela alegerile interne și externe. Ca urmare, toate abordările anterioare au fost limitate la limbaje de programare cu astfel de instrucțiuni condiționate, ceea ce a redus drastic aplicabilitatea lor. În plus, limbajul nostru de specificație se bazează pe o extensie a logicii de separare şi, astfel, sprijină verificarea programelor care manipulează heap-ul și transmit mesajele prin referință. În comparație cu instrumentul din subsecțiunea 6.1, putem observa că instrumentul lor se bazează pe contracte globale specificate cu mașini de stări, în timp ce instrumentul nostru, mai general, se bazează pe o logică de sesiune și este construit ca o extensie a logicii de separare cu disjuncție pentru a sprijini instrucțiunile condiționale standard. În cazul nostru, formulele logice a protocoalelor pot fi, de asemenea, localizate pentru fiecare canal și pot fi transmise în mod liber între procedurii. Mai mult decât atât, putem garanta corectitudinea conversiilor de tip a datelor transmise prin verificarea comunicației. În plus, putem asigura faptul că memoria heap și proprietățile valorilor transmise în canale sunt captate în mod corespunzător. În final, utilizând relații de subsumare, permitem ca specificațiile capetelor unui canal să difere între fire, totuși putem asigura că firele nu se interblochează prin verificarea compatibilității specificației la fiecare punct de joncțiune. Mai

realistic ca abordările anterioare, noi presupunem că comunicația este asincronă și trimiterea nu blochează firul de execuție. Acest capitol a fost prezentat parțial în [68].

8.1 Direcții viitoare

Având aceste rezultate, ne putem gândi la mai multe direcții pentru continuarea cercetării în verificarea programelor paralele care utilizează transmiterea de mesaje pentru sincronizarea și schimbul de date. O direcție foarte interesantă de cercetare ar fi extinderea logicii noastre de sesiune la specificații cu mai multe entități și canale. După cum se poate observa, o simplă codificare a comunicației între mai multe entități în limbajul nostru de specificare logică de sesiune este banală și se poate face prin utilizarea unei structuri de date în care un câmp joacă rolul unui canal. Problema este că API-urile sistemelor de operare sunt diferite față de modelul precedent. De exemplu, conectarea între entități trebuie să fie realizată secvențial, si nu putem citi mesaje de pe diferite canale simultan folosind citirea standard ("receive"). Pentru acest lucru, API-urile oferă un set de funcții speciale. De exemplu, în Linux avem: "select", "pselect" [85], "poll", "ppoll" [84] și "epoll" [83]. Prin urmare, pentru a avea suport pentru aceste funcții, logica noastră de sesiune trebuie să fie extinsă în acest sens. Un subiect de cercetare și mai interesant ar fi investigarea posibilității combinării logicii noastre de sesiune cu [12]. Logica lor încearcă să rezume efectele proceselor implicate în protocol și să verifice corectitudinea unei aplicații scrise în π-calculus. Noi credem că ar fi interesant să investigăm verificarea acestor specificații într-un limbaj de programare mai realist, ca, de exemplu, "SESSION-HIP". Un alt subiect interesant de cercetare ar fi investigarea aplicabilității teoriei noastre în programarea orientată pe obiecte. De exemplu, putem investiga codificarea și verificarea ordinii de apelare a metodelor în clase, ca în [52]. Ca un subiect de cercetare tehnic, propunem dezvoltarea unui (mediu de dezvoltare) IDE pentru industria feroviară. Aceasta ar trebui să ofere o metodă de specificare mai ușoară pentru codificarea specificațiilor în logica de sesiune (de exemplu, în formă de diagrame de secvență cu adnotări) și, de asemenea, ar trebui să sprijine dezvoltarea și verificarea programelor interlocking utilizând metoda de dezvoltare geografică. Instrumentul anterior ar trebui să se bazeze pe teoria și instrumentele noastre. Pentru dezvoltarea programelor de interlocking, noi sugerăm utilizarea unui limbaj specific unui domeniu ca EURIS sau LARIS [50]. Verificarea trebuie efectuată în mod automat folosind instrumentul de verificare din capitolul 5, urmărind mecanismul nostru de verificare prezentat în secțiunea 7.1. Un alt subiect interesant ar fi verificarea unui set mare de drivere Linux, care controlează unele unități periferice folosind un set de mesaje transmise prin intermediul front side bus-ului (FSB) sau, mai recent, prin intermediul platformei de controler Hub în cazul unei arhitecturi Intel, sau prin intermediul SPI sau I2C sau alte busuri de comunicație periferice în alte arhitecturi RISC ([78–82]). Combinația a două teorii foarte des utilizate, și anume logica de separare și cea a tipurilor de sesiune, în cadrul logicii noastre coerente de sesiune a deschis prea multe posibilități pentru a putea fi enumerate în această lucrare, așa că, îl lăsăm pe cititor să descopere și să exploreze el însuşi restul posibilităților.

Bibliografie

[1] Abdulla, P. A., Deneux, J., Stålmarck, G., Ågren, H., and Åkerlund, O. (2004). Designing safe, reliable systems using scade. In *Leveraging Applications of Formal Methods*, pages 115–129. Springer. (page(s): [1], [3])

[2] Abrial, J.-R. (2010). *Modeling in Event-B: system and software engineering*. Cambridge University Press. (page(s): [1])

[3] Bai, G., Lei, J., Meng, G., Venkatraman, S. S., Saxena, P., Sun, J., Liu, Y., and Dong, J. S. (2013). Authscan: Automatic extraction of web authentication protocols from implementations. In *NDSS*. (page(s): [6])

[4] Baltazar, P., Mostrous, D., and Vasconcelos, V. T. (2012). Linearly refined session types. *arXiv preprint arXiv:1211.4099*. (page(s):)

[5] Banci, M. and Fantechi, A. (2005). Geographical versus functional modelling by statecharts of interlocking systems. *Electron. Notes Theor. Comput. Sci.*, 133:3–19. (page(s): [2])

[6] Behm, P., Desforges, P., and Meynadier, J.-M. (1998). Météor: An industrial success in formal development. In *B'98: Recent Advances in the Development and Use of the B Method*, pages 26–26. Springer. (page(s): [1])

[7] Ben-Ari, M. (2001). The bug that destroyed a rocket. *ACM SIGCSE Bulletin*, 33(2):58. (page(s): [1])

[8] Bernardi, G., Dardha, O., Gay, S. J., and Kouzapas, D. (2014). On duality relations for session types. In *Trustworthy Global Computing*, pages 51–66. Springer. (page(s):)

[9] Bernardi, G. and Hennessy, M. (2014). Using higher-order contracts to model session types. In *CONCUR 2014–Concurrency Theory*, pages 387–401. Springer. (page(s):)

[10] Bettini, L., Coppo, M., D'Antoni, L., De Luca, M., Dezani-Ciancaglini, M., and Yoshida, N. (2008). Global progress in dynamically interleaved multiparty sessions. In *CONCUR 2008-Concurrency Theory*, pages 418–433. Springer. (page(s):)

[11] Bocchi, L., Chen, T.-C., Demangeon, R., Honda, K., and Yoshida, N. (2013a). Monitoring networks through multiparty session types. In *FMOODS/FORTE 2013*, volume 7892 of *LNCS*, pages 50–65. Springer. (page(s): [7])

[12] Bocchi, L., Demangeon, R., and Yoshida, N. (2013b). A multiparty multi-session logic. In *Trustworthy Global Computing*, pages 97–111. Springer. (page(s): [3], [7], [40])

[13] Bocchi, L., Honda, K., Tuosto, E., and Yoshida, N. (2010). A theory of design-by-contract for distributed multiparty interactions. In *CONCUR 2010-Concurrency Theory*, pages 162–176. Springer. (page(s): [3], [7])

[14] Bonacchi, A. and Fantechi, A. (2014). On the validation of an interlocking system by model-checking. In Lang, F. and Flammini, F., editors, *Formal Methods for Industrial Critical Systems*, volume 8718 of *Lecture Notes in Computer Science*, pages 94–108. Springer International Publishing. (page(s): [2])

[15] Bonacchi, A., Fantechi, A., Bacherini, S., Tempestini, M., and Cipriani, L. (2013). Validation of railway interlocking systems by formal verification, a case study. In *Software Engineering and Formal Methods*, pages 237–252. Springer. (page(s):)

[16] Bono, V., Messa, C., and Padovani, L. (2011). Typing copyless message passing. In *Programming Languages and Systems*, pages 57–76. Springer. (page(s):)

[17] Bornat, R., Calcagno, C., O'Hearn, P., and Parkinson, M. (2005). Permission accounting in separation logic. In *ACM SIGPLAN Notices*, volume 40, pages 259–270. ACM. (page(s):)

[18] Boulanger, J.-L., Fornari, F.-X., Camus, J.-L., and Dion, B. (2015). Scade: Language and applications. (page(s): [1])

[19] Brière, D. and Traverse, P. (1993). Airbus a320/a330/a340 electrical flight controls-a family of fault-tolerant systems. In *Fault-Tolerant Computing, 1993. FTCS-23. Digest of Papers., The Twenty-Third International Symposium on*, pages 616–623. IEEE. (page(s): [37])

[20] Brookes, S. (2007). A semantics for concurrent separation logic. *Theoretical Computer Science*, 375(1):227–270. (page(s):)

[21] Brookes, S. D., Hoare, C. A., and Roscoe, A. W. (1984). A theory of communicating sequential processes. *Journal of the ACM (JACM)*, 31(3):560–599. (page(s): [3])

[22] Broy, M. (2006). Challenges in automotive software engineering. In *Proceedings of the 28th international conference on Software engineering*, pages 33–42. ACM. (page(s): [37])

[23] Capecchi, S., Castellani, I., and Dezani-Ciancaglini, M. (2011). Information flow safety in multiparty sessions. *Mathematical Structures in Computer Science*, pages 1–43. (page(s): [7])

[24] Castagna, G., Dezani-Ciancaglini, M., Giachino, E., and Padovani, L. (2009). Foundations of session types. In *Proceedings of the 11th ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, PPDP '09, pages 219–230, New York, NY, USA. ACM. (page(s): [6])

[25] Chen, T.-C., Bocchi, L., Deniélou, P.-M., Honda, K., and Yoshida, N. (2012). Asynchronous distributed monitoring for multiparty session enforcement. In *Trustworthy Global Computing*, pages 25–45. Springer. (page(s): [7])

[26] Chin, W.-N., David, C., and Gherghina, C. (2011a). A hip and sleek verification system. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pages 9–10. ACM. (page(s): [1], [3], [5], [37])

[27] Chin, W.-N., David, C., Nguyen, H. H., and Qin, S. (2012). Automated verification of shape, size and bag properties via user-defined predicates in separation logic. *Sci. Comput. Program.*, 77(9):1006–1036. (page(s): [15], [20], [22], [27], [30])

[28] Chin, W.-N., Gherghina, C., Voicu, R., Le, Q. L., Craciun, F., and Qin, S. (2011b). A specialization calculus for pruning disjunctive predicates to support verification. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, CAV'11, pages 293–309. (page(s):)

[29] Craciun, F., Kiss, T., and Costea, A. (2015). Towards a session logic for communication protocols. In *ICECCS*. IEEE. (page(s): [5], [37])

[30] Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., and Yakobowski, B. (2012). Frama-c. In *Software Engineering and Formal Methods*, pages 233–247. Springer. (page(s): [1])

[31] Dardha, O., Giachino, E., and Sangiorgi, D. (2012). Session types revisited. In *Proceedings of the 14th symposium on Principles and practice of declarative programming*, pages 139–150. ACM. (page(s): [7])

[32] David, C. and Chin, W.-N. (2011). Immutable specifications for more concise and precise verification. *SIGPLAN Not.*, 46(10):359–374. (page(s): [27])

[33] DeLine, R. and Fähndrich, M. (2001). Enforcing high-level protocols in low-level software. *ACM SIGPLAN Notices*, 36(5):59–69. (page(s): [3])

[34] Dezani-Ciancaglini, M. and De'Liguoro, U. (2009). Sessions and session types: An overview. In *Web Services and Formal Methods*, pages 1–28. Springer. (page(s):)

[35] Dezani-Ciancaglini, M., Giachino, E., Drossopoulou, S., and Yoshida, N. (2007). Bounded session types for object oriented languages. In *Formal Methods for Components and Objects*, volume 4709 of *Lecture Notes in Computer Science*, pages 207–245. Springer Berlin Heidelberg. (page(s): [7])

[36] Dezani-Ciancaglini, M., Mostrous, D., Yoshida, N., and Drossopoulou, S. (2006). Session types for object-oriented languages. In Thomas, D., editor, *ECOOP 2006 â€“ Object-Oriented Programming*, volume 4067 of *Lecture Notes in Computer Science*, pages 328–352. Springer Berlin Heidelberg. (page(s): [3], [4], [7])

[37] Dijkstra, E. W. (1972). The humble programmer. *Communications of the ACM*, 15(10):859–866. (page(s): [1])

[38] Dinsdale-Young, T., Dodds, M., Gardner, P., Parkinson, M. J., and Vafeiadis, V. (2010). Concurrent abstract predicates. In *ECOOP 2010–Object-Oriented Programming*, pages 504–528. Springer. (page(s):)

[39] Dodds, M., Feng, X., Parkinson, M., and Vafeiadis, V. (2009). Deny-guarantee reasoning. In *Programming languages and systems*, pages 363–377. Springer. (page(s):)

[40] ETCS, O. Open etcs. http://openetcs.org. [Online; accessed 21-Jan-2016]. (page(s):)

[41] ETCS, O. Open etcs: Model change. https://github.com/openETCS/modeling/blob/master/UserStories/DescriptionUserStory14.pdf. [Online; accessed 21-Jan-2016]. (page(s):)

[42] ETCS, O. Open etcs: Model change - sequence diagram. https://github.com/openETCS/modeling/blob/master/UserStories/UserStory14.pdf. [Online; accessed 21-Jan-2016]. (page(s):)

[43] ETCS, O. Open etcs: Revocation of movement authority. https://github.com/openETCS/modeling/blob/master/UserStories/DescriptionUserStory13.pdf. [Online; accessed 21-Jan-2016]. (page(s):)

[44] ETCS, O. Open etcs: Revocation of movement authority - sequence diagram. https://github.com/openETCS/modeling/blob/master/UserStories/UserStory13.pdf. [Online; accessed 21-Jan-2016]. (page(s):)

[45] ETCS, O. Open etcs: Route cancellation from end of route signal. https://github.com/openETCS/modeling/blob/master/UserStories/DescriptionUserStory15.pdf. [Online; accessed 21-Jan-2016]. (page(s):)

[46] ETCS, O. Open etcs: Route cancellation from end of route signal - sequence diagram. https://github.com/openETCS/modeling/blob/master/UserStories/UserStory15.pdf. [Online; accessed 21-Jan-2016]. (page(s):)

[47] Fantechi, A. (2013). Twenty-five years of formal methods and railways: What next? In *Software Engineering and Formal Methods*, pages 167–183. Springer. (page(s): [2], [8])

[48] Ferrari, A., Magnani, G., Grasso, D., and Fantechi, A. (2011). Model checking interlocking control tables. In *FORMS/FORMAT 2010*, pages 107–115. Springer. (page(s):)

[49] Floyd, R. (1967). Assigning meanings to programs. 14:65–81. (page(s):)

[50] Fokkink, W., Groote, J. F., Hollenberg, M., and van Vlijmen, B. (1999). *LARIS 1.0, LAnguage for Railway Interlocking Specifications*, volume 4204 of *Lecture Notes in Computer Science*. Springer Verlag, New York NY. (page(s): [40])

[51] Gay, S. and Hole, M. (2005). Subtyping for session types in the pi calculus. *Acta Informatica*, 42(2-3):191–225. (page(s): [3])

[52] Gay, S. J., Vasconcelos, V. T., Ravara, A., Gesbert, N., and Caldeira, A. Z. (2010). Modular session types for distributed object-oriented programming. *SIGPLAN Not.*, 45(1):299–312. (page(s): [3], [7], [40])

[53] Haxthausen, A., Peleska, J., and Pinger, R. (2013). *Applied Bounded Model Checking for Interlocking System Designs*, pages 21–26. Technical University of Denmark. (page(s):)

[54] Haxthausen, A. E. and Peleska, J. (2015). Model checking and model-based testing in the railway domain. In *Formal Modeling and Verification of Cyber-Physical Systems*, pages 82–121. Springer Fachmedien Wiesbaden. (page(s):)

[55] Haxthausen, A. E., Peleska, J., and Kinder, S. (2011). A formal approach for the construction and verification of railway control systems. *Formal aspects of computing*, 23(2):191–219. (page(s):)

[56] Hoare, C. (1969). An axiomatic basis for computer programming. pages 419–438. (page(s):)

[57] Hoare, T. and O'Hearn, P. (2008). Separation logic semantics for communicating processes. *Electronic Notes in Theoretical Computer Science*, 212(0):3 – 25. Proceedings of the First International Conference on Foundations of Informatics, Computing and Software (FICS 2008). (page(s): [7])

[58] Hon, Y. M. and Kollmann, M. (2006). Simulation and verification of uml-based railway interlocking designs. In *Automatic Verification of Critical Systems*, pages 168–172. (page(s):)

[59] Honda, K., Hu, R., Neykova, R., Chen, T.-C., Demangeon, R., Deniélou, P.-M., and Yoshida, N. (2014). Structuring communication with session types. In Agha, G., Igarashi, A., Kobayashi, N., Masuhara, H., Matsuoka, S., Shibayama, E., and Taura, K., editors, *Concurrent Objects and Beyond*, volume 8665 of *Lecture Notes in Computer Science*, pages 105–127. Springer Berlin Heidelberg. (page(s): [7])

[60] Honda, K., Vasconcelos, V. T., and Kubo, M. (1998). Language primitives and type discipline for structured communication-based programming. In *Programming Languages and Systems*, pages 122–138. Springer. (page(s): [6], [7], [8], [12])

[61] Honda, K. and Yoshida, N. (1995). On reduction-based process semantics. *Theoretical Computer Science*, 151(2):437–486. (page(s):)

[62] Hu, R., Kouzapas, D., Pernet, O., Yoshida, N., and Honda, K. (2010). Type-safe eventful sessions in java. In Dâ€™Hondt, T., editor, *ECOOP 2010 â€“ Object-Oriented Programming*, volume 6183 of *Lecture Notes in Computer Science*, pages 329–353. Springer Berlin Heidelberg. (page(s):)

[63] Hu, R. and Yoshida, N. (2016). Hybrid session verification through endpoint api generation. In *FASE 2016*, volume 9633 of *LNCS*, pages 401–418. Springer. (page(s): [6], [7])

[64] Hu, R., Yoshida, N., and Honda, K. (2008). Session-based distributed programming in java. In *ECOOP 2008–Object-Oriented Programming*, pages 516–541. Springer. (page(s):)

[Intel] Intel. Embedded intel486 processor hardware reference manual. http://www.pld.ttu.ee/~prj/486dev.pdf. [Online; accessed 21-Jan-2016]. (page(s):)

[66] James, P., Moller, F., Nguyen, H. N., Roggenbach, M., Schneider, S., Treharne, H., Trumble, M., and Williams, D. (2013). Verification of scheme plans using csp|| b. In *Software Engineering and Formal Methods*, pages 189–204. Springer. (page(s): [8])

[67] Kiss, T. (2015). Formal verification of laris programs using session types and first order logic. In *KEPT*. Babes-Bolyai University. (page(s):)

[68] Kiss, T. (2016). Comparison of session logic with session types. *Studia Informatica*, 61:54–66. (page(s): [40])

[69] Kiss, T., Craciun, F., and Parv, B. (2015a). Extending laris with session types: A case study in the railway interlocking. In *MIPRO*. IEEE. (page(s): [5], [36], [38])

[70] Kiss, T., Craciun, F., and Parv, B. (2015b). Verification of protocol specifications with separation logic. In *Intelligent Computer Communication and Processing (ICCP), 2015 IEEE International Conference on*, pages 109–116. IEEE. (page(s): [5], [37])

[71] Kiss, T. and Jánosi-Rancz, K. T. (2016). Developing railway interlocking systems with session types and event-b. In *IEEE International Symposium on Applied Computational Intelligence and Informatics*, page 163. IEEE. (page(s): [5], [36], [38])

[72] Kobayashi, N., Saito, S., and Sumii, E. (2000). An implicitly-typed deadlock-free process calculus. In *CONCUR 2000—Concurrency Theory*, pages 489–504. Springer. (page(s):)

[73] Kouzapas, D., Yoshida, N., and Honda, K. (2011). On asynchronous session semantics. In *Formal Techniques for Distributed Systems*, pages 228–243. Springer. (page(s):)

[74] Lamport, L. and Schneider, F. B. (1984). The “hoare logic” of csp, and all that. *ACM Trans. Program. Lang. Syst.*, 6(2):281–296. (page(s): [7])

[75] Le, D.-K., Chin, W.-N., and Teo, Y.-M. (2012). Variable permissions for concurrency verification. In *Formal Methods and Software Engineering*, pages 5–21. Springer. (page(s):)

[76] Le, D.-K., Chin, W.-N., and Teo, Y.-M. (2013). Verification of static and dynamic barrier synchronization using bounded permissions. In *Formal Methods and Software Engineering*, pages 231–248. Springer Berlin Heidelberg. (page(s):)

[77] Lie, D., Chou, A., Engler, D., and Dill, D. L. (2001). A simple method for extracting models from protocol code. In *2001. Proceedings. 28th Annual International Symposium on Computer Architecture*, pages 192–203. IEEE. (page(s): [6])

[78] Linux, C. Linux driver: aerdrv.c. https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/drivers/pci/pcie/aer/aerdrv.c?id=refs/tags/v4.7-rc7. [Online; accessed 21-Jan-2016]. (page(s): [40])

[79] Linux, C. Linux driver: atmel_usba_udc.c. https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/drivers/usb/gadget/udc/atmel_usba_udc.c?id=refs/tags/v4.7-rc7. [Online; accessed 21-Jan-2016]. (page(s):)

[80] Linux, C. Linux driver: pcie-designware.c. https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/drivers/pci/host/pcie-designware.c?id=refs/tags/v4.7-rc7. [Online; accessed 21-Jan-2016]. (page(s):)

[81] Linux, C. Linux driver: pcie-xilinx.c. https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/drivers/pci/host/pcie-xilinx.c?id=refs/tags/v4.7-rc7. [Online; accessed 21-Jan-2016]. (page(s):)

[82] Linux, C. Linux driver: spi-bcm2835.c. https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/drivers/spi/spi-bcm2835.c?id=refs/tags/v4.7-rc7. [Online; accessed 21-Jan-2016]. (page(s): [40])

[83] Linux, C. Linux manual page: epoll. http://man7.org/linux/man-pages/man7/epoll.7.html. [Online; accessed 21-Jan-2016]. (page(s): [40])

[84] Linux, C. Linux manual page: poll and ppoll. http://man7.org/linux/man-pages/man2/poll.2.html. [Online; accessed 21-Jan-2016]. (page(s): [40])

[85] Linux, C. Linux manual page: select and pselect. http://manpages.courier-mta.org/htmlman2/select.2.html. [Online; accessed 21-Jan-2016]. (page(s): [40])

[86] López, H. A., Marques, E. R., Martins, F., Ng, N., Santos, C., Vasconcelos, V. T., and Yoshida, N. (2015). Protocol-based verification of message-passing parallel programs. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 280–298. ACM. (page(s): [7])

[87] Lozes, É. and Villard, J. (2015). Shared contract-obedient channels. *Science of Computer Programming*, 100:28–60. (page(s): [4], [8])

[88] Marques, E. R. B., Martins, F., Vasconcelos, V. T., Ng, N., and Martins, N. D. (2013). Towards deductive verification of mpi programs against session types. In *PLACES'13*, volume 137 of *EPTCS*, pages 103–113. Open Publishing Association. (page(s): [7])

[89] Melham, T. F. (1992). A mechanized theory of the pi-calculus in hol. Technical report, NORDIC JOURNAL OF COMPUTING. (page(s): [7])

[90] Milner, R. (1980). A calculus of communication systems. lncs, vol. 92. (page(s): [3])

[91] Milner, R., Parrow, J., and Walker, D. (1992). A calculus of mobile processes, i. *Information and computation*, 100(1):1–40. (page(s):)

[92] Moler, F., Nguyen, H. N., Roggenbach, M., Schneider, S., and Treharne, H. (2012). Combining event-based and state-based modelling for railway verification. (page(s): [8])

[93] Moller, F., Nguyen, H. N., Roggenbach, M., Schneider, S., and Treharne, H. (2012a). Defining and model checking abstractions of complex railway models using csp|| b. In *Hardware and Software: Verification and Testing*, pages 193–208. Springer. (page(s): [8])

[94] Moller, F., Nguyen, H. N., Roggenbach, M., Schneider, S., Treharne, H., Lttgen, G., Merz, S., Margaria, T., Padberg, J., and Taentzer, G. (2012b). Railway modelling in csp|| b: the double junction case study. (page(s): [8])

[95] Mostrous, D. (2005). Moose: a minimal object oriented language with session types. *Master's thesis, Imperial College, London*. (page(s): [8])

[96] Mostrous, D. and Yoshida, N. (2015). Session typing and asynchronous subtyping for the higher-order π-calculus. *Information and Computation*, 241:227–263. (page(s): [3])

[97] Neykova, R. (2013). Session types go dynamic or how to verify your python conversations. In *PLACES'13*, volume 137 of *EPTCS*, pages 95–102. Open Publishing Association. (page(s): [7])

[98] Neykova, R., Yoshida, N., and Hu, R. (2013). Spy: Local verification of global protocols. In Legay, A. and Bensalem, S., editors, *Runtime Verification*, volume 8174 of *Lecture Notes in Computer Science*, pages 358–363. Springer Berlin Heidelberg. (page(s): [7])

[99] Ng, N., de Figueiredo Coutinho, J. G., and Yoshida, N. (2015). Protocols by default. In *Compiler Construction*, pages 212–232. Springer. (page(s): [6])

[100] Ng, N., Yoshida, N., and Honda, K. (2012). Multiparty session c: Safe parallel programming with message optimisation. In *Objects, Models, Components, Patterns*, pages 202–218. Springer. (page(s): [7], [8])

[101] O'hearn, P. W. (2004). Resources, concurrency and local reasoning. In *CONCUR 2004-Concurrency Theory*, pages 49–67. Springer. (page(s):)

[102] Palamidessi, C. and Herescu, O. M. (2005). A randomized encoding of the π-calculus with mixed choice. *Theoretical Computer Science*, 335(2–3):373 – 404. Process Algebra. (page(s):)

[103] Palumbo, M. (2014). The ertms/etcs signalling system. *Railway Signalling EU*, 1:1–60. (page(s):)

[104] Pierce, B. C. (1998). Programming in the pi-calculus. *An Experiment in Concurrent Language Design. Tutorial Notes for Pict Version*, 3. (page(s):)

[105] Pucella, R. and Tov, J. A. (2008). Haskell session types with (almost) no class. *SIGPLAN Not.*, 44(2):25–36. (page(s): [3], [6])

[106] Quaglia, P. and Walker, D. (1998). On encoding pπ in mπ. In *Foundations of Software Technology and Theoretical Computer Science*, pages 42–53. Springer. (page(s):)

[107] Sangiorgi, D. (1993). Expressing mobility in process algebras: first-order and higher-order paradigms. (page(s):)

[108] Schneider, S. (2001). *The B-method: An introduction*. Palgrave Oxford. (page(s): [1], [3])

[109] Simpson, A. et al. (1997). The mechanical verification of solid state interlocking geographic data. (page(s): [8])

[110] Takeuchi, K., Honda, K., and Kubo, M. (1994). An interaction-based language and its typing system. In *PARLE'94 Parallel Architectures and Languages Europe*, pages 398–413. Springer. (page(s):)

[111] Turon, A. and Wand, M. (2011). A resource analysis of the π-calculus. *Electronic Notes in Theoretical Computer Science*, 276(0):313 – 334. Twenty-seventh Conference on the Mathematical Foundations of Programming Semantics (MFPS XXVII). (page(s): [7])

[112] Vafeiadis, V. and Parkinson, M. (2007). A marriage of rely/guarantee and separation logic. In *CONCUR 2007–Concurrency Theory*, pages 256–271. Springer. (page(s):)

[113] Vallecillo, A., Vasconcelos, V. T., and Ravara, A. (2003). Typing the behavior of objects and components using session types. (page(s): [6])

[114] Vasconcelos, V. T. Bica. http://gloss.di.fc.ul.pt/bica. [Online; accessed 21-Jan-2016]. (page(s): [8])

[115] Vasconcelos, V. T. Mool. http://gloss.di.fc.ul.pt/mool. [Online; accessed 21-Jan-2016]. (page(s): [8])

[116] Vasconcelos, V. T. Partypes. http://gloss.di.fc.ul.pt/ParTypes. [Online; accessed 21-Jan-2016]. (page(s): [8])

[117] Villard, J. (2009a). Heap-Hop Dualized case study. http://www.lsv.ens-cachan.fr/Software/heap-hop/examples/send_list_dualized.hop. [Online; accessed 21-Jan-2016]. (page(s): [3])

[118] Villard, J. (2009b). Heap-Hop Home page. http://www.lsv.ens-cachan.fr/Software/heap-hop/index.html. [Online; accessed 21-Jan-2016]. (page(s):)

[119] Villard, J. (2009c). Heap-Hop List Bug Counterexample. http://www.lsv.ens-cachan.fr/Software/heap-hop/examples/send_list_bug.hop. [Online; accessed 21-Jan-2016]. (page(s):)

[120] Villard, J. (2009d). Heap-Hop Non Deterministic Counterexample. http://www.lsv.ens-cachan.fr/Software/heap-hop/examples/send_list_non_det.hop. [Online; accessed 21-Jan-2016]. (page(s):)

[121] Villard, J. (2009e). Heap-Hop Shared Read of Variable. http://www.lsv.ens-cachan.fr/Software/heap-hop/examples/shared_reads.hop. [Online; accessed 21-Jan-2016]. (page(s):)

[122] Villard, J. (2009f). Heap-Hop TCP Example. http://www.lsv.ens-cachan.fr/Software/heap-hop/examples/tcp.hop. [Online; accessed 21-Jan-2016]. (page(s):)

[123] Villard, J., Lozes, É., and Calcagno, C. (2009). Proving copyless message passing. In *Programming Languages and Systems*, pages 194–209. Springer. (page(s): [8])

[124] Vu, L. H. (2015). *Formal Development and Verification of Railway Control Systems*. PhD thesis, Technical University of Denmark, 2800 Kongens Lyngby, Denmark. In the context of ERTMS/ETCS Level 2. (page(s): [8])

[125] Vu, L. H., Haxthausen, A. E., and Peleska, J. (2014). Formal modeling and verification of interlocking systems featuring sequential release. In *Formal Techniques for Safety-Critical Systems*, pages 223–238. Springer. (page(s):)

[126] Winter, K. (2002). Model checking railway interlocking systems. In *Australian Computer Science Communications*, volume 24, pages 303–310. Australian Computer Society, Inc. (page(s): [8])

[127] Winter, K. (2012). Optimising ordering strategies for symbolic model checking of railway interlockings. In *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*, pages 246–260. Springer. (page(s):)

[128] Winter, K. and Robinson, N. J. (2003). Modelling large railway interlockings and model checking small ones. In *Proceedings of the 26th Australasian computer science conference-Volume 16*, pages 309–316. Australian Computer Society, Inc. (page(s):)

[129] Zhivich, M. and Cunningham, R. K. (2009). The real cost of software errors. (page(s): [1])