Univerza v Mariboru

Fakulteta za elektrotehniko, računalništvo in informatiko

Boštjan Vlaovič

Avtomatska tvorba modelov s sondami iz specifikacije sistema v jeziku SDL

Doktorska disertacija

Maribor, junij 2004

Univerza v Mariboru

Fakulteta za elektrotehniko, računalništvo in informatiko

Boštjan Vlaovič

Avtomatska tvorba modelov s sondami iz specifikacije sistema v jeziku SDL

Doktorska disertacija

Maribor, junij 2004

Avtor:	Boštjan Vlaovič
Naslov:	Avtomatska tvorba modelov s sondami iz specifikacije sistema v jeziku SDL
UDK:	621.39(043.3)
Ključne besede:	SDL, Promela, specifikacija sistemov, avtomatsko generiranje modelov, specifikacija zahtev, sonda, linearna temporalna logika, preverjanje modelov
Število izvodov:	9
Obdelava besedila:	Boštjan Vlaovič
Obdelava slik:	Boštjan Vlaovič
Razmnoževanje:	Laboratorij za mikroračunalniške sisteme, Fakulteta za elektrotehniko, računalništvo in informatiko
Kraj in datum:	Maribor, junij 2004

© Copyright 2004 by Boštjan Vlaovič
All Rights Reserved

Na Univerzi v Mariboru,

Fakulteti za elektrotehniko, računalništvo in informatiko je

Boštjan Vlaovič, univ. dipl. inž. el.

dne 8. 9. 2004 uspešno zagovarjal doktorsko disertacijo z naslovom

Avtomatska tvorba modelov s sondami iz specifikacije sistema v jeziku SDL

pred komisijo v sestavi:

1. red. prof. dr. Igor Tičar — predsednik
2. red. prof. dr. Zmago Brezočnik, mentor — član
3. izr. prof. dr. Tatjana Kapus — član
4. red. prof. dr. Ignac Lovrek — član

Zasnova sistema brez specifikacije zahtev ne more biti napačna.
Lahko je samo presenetljiva.

(povzeto po *Willem Louis van der Poel*)

Zahvale

Zahvaljujem se svojemu mentorju Zmagu Brezočniku za omogočen podiplomski študij in vso dodatno podporo, ki mi jo je nudil pri izdelavi doktorske disertacije. Mentorju in Tatjani Kapus se zahvaljujem za temeljit pregled disertacije in koristne predloge. Še posebna zahvala gre Aleksandru Vrežetu za ure plodnih debat, soočanja argumentov in za implementacijo rezultatov disertacije.

Zahvaljujem se Draganu Bošnačkemu z Univerze v Eindhovnu, ki je v izredno kratkem času na mojo prošnjo izvedel razširitev Spina verzije 4.1.1 za podporo diskretnemu času. Hvaležen sem tudi Gerardu J. Holzmannu, vodji Laboratorija za zanesljivo programsko opremo pri NASA in avtorju Spina, za hitre odgovore na moja elektronska sporočila in komentarje mojih predlogov in rešitev.

Nenazadnje se zahvaljujem tudi svoji družini in prijateljem, ki so mi v najtežjih trenutkih nudili oporo in razumevanje. Še posebna zahvala gre Katji Leskovšek, bodoči mamici, za njeno podporo, potrpežljivost in vsa odrekanja v času priprave disertacije in podiplomskega študija.

UDK: 621.39(043.3)

Ključne besede: SDL, Promela, specifikacija sistemov, avtomatsko generiranje modelov, specifikacija zahtev, sonda, linearna temporalna logika, preverjanje modelov

Avtomatska tvorba modelov s sondami iz specifikacije sistema v jeziku SDL

Povzetek

Doktorska disertacija predstavlja rezultate raziskav avtomatske tvorbe modelov v jeziku Promela iz specifikacije sistema v jeziku SDL. V splošnem se model sistema lahko pridobi ročno ali avtomatsko. Model sistema lahko ob ročnem pristopu tvori le strokovnjak z dobrim poznavanjem delovanja sistema in jezika za opis modelov. Kvaliteta modela je neposredno odvisna od strokovnjaka. Pri postopku ročne gradnje modela hitro pride do napak pri modeliranju lastnosti sistema. Disertacija formalno podaja algoritme za avtomatsko tvorbo modelov v jeziku Promela.

Tehnika preverjanja modelov nam lahko z matematično natančnostjo potrdi ali ovrže skladnost podanega modela sistema s specifikacijo zahtev. Za preverjanje lastnosti je potrebno v model sistema namestiti sonde. Te nam omogočajo opazovanje sprememb v modelu. Disertacija definira postopke za avtomatsko dopolnitev modela s sondami.

Teoretične raziskave smo podkrepili s praktično realizacijo algoritmov v orodju $sdl2pml$. Z njim smo uspešno verificirali specifikacijo protokola V.76. Vsi uporabljeni modeli protokola V.76 v disertaciji so pridobljeni avtomatično. Pregled doseženih rezultatov pokaže obetavnost pristopa in njegovo praktično vrednost.

UDK: 621.39(043.3)

Keywords: SDL, Promela, specification of systems, automatic generation of models, requirement specification, probe, linear temporal logic, model checking

Automatic Generation of Models with Probes from the SDL System Specification

Abstract

This dissertation presents research results in the domain of the mechanical extraction of models from the SDL specification of the system. A model can be generated manually or mechanically. If it is to be prepared manually, we will need an expert with the detailed knowledge of the system and the modelling language. The quality of the model is directly influenced by the expert. The process is prone to the incorrect modeling of the system's properties. In this dissertation algorithms for mechanical generation of Promela models are presented.

With the model checking technique the model's accordance with the correctness requirements can be established with the mathematical accuracy. Checking of system properties requires the use of probes in the model. Probes provide us with an insight to the model execution. This dissertation defines procedures for the automatic insertion of the probes in the model.

A theoretic research has been supported with a practical implementation of algorithms in the $sdl2pml$ tool. We have used it to successfully verify specification of the V.76 protocol. All models of the V.76 protocol used in the dissertation are obtained mechanically. Review of the results and their practical value show that our approach seems to be a promising one.

Kazalo

Zahvala **vii**

Povzetek **xi**

Kazalo **xv**

Slike **xix**

Tabele **xxiii**

Algoritmi **xxv**

1 Uvod v problematiko **1**
1.1 Specifikacija sistemov 2
1.2 Preverjanje pravilnosti sistemov 3
1.3 Kratek pregled obstoječih pristopov 4
1.4 Pregled vsebine 5

2 Promela — jezik za opis modelov sistemov **7**
2.1 Proces 8
2.2 Spremenljivke 14
2.3 Komunikacija 16

3 Tvorba modelov iz specifikacije v jeziku SDL **19**
3.1 Pojem podatkovnega tipa v jeziku SDL 20
3.1.1 Standardni podatkovni tipi 20
3.1.2 Definicija podatkovnega tipa 24
3.1.3 Sinonimi 36
3.1.4 Časovniki 38
3.1.5 Spremenljivke 40
3.2 Komunikacija 42
3.2.1 Signal 44
3.2.2 Signalna lista 46

3.2.3 Kanal 47
3.2.4 Signalna pot 49
3.2.5 Priključek 50
3.3 Struktura sistema 52
3.3.1 Blok 54
3.3.2 Proces 56
3.3.3 Procedura 71
3.4 Razširjen končni avtomat 76
3.4.1 Start in stop 76
3.4.2 Prehod 77
3.4.3 Stanje 77
3.4.4 Shrani 81
3.4.5 Vhod in vhod zvezdica 83
3.4.6 Prioritetni vhod 86
3.4.7 Implicitni prehod 88
3.4.8 Spontani prehod 91
3.4.9 Časovnik 92
3.4.10 Trajen signal 95
3.4.11 Izhod 97
3.4.12 Ustvari 103
3.4.13 Naslednje stanje 105
3.4.14 Oznaka in združi 106
3.4.15 Naloga 107
3.4.16 Odločitev 108
3.5 Primer modeliranja komunikacijskih konstruktov 111

4 Formalna verifikacija 117
4.1 Pravilnostne trditve 119
4.1.1 Osnovne pravilnostne trditve 119
4.1.2 Končna stanja 120
4.1.3 Stanja napredka 120
4.1.4 Dopustna stanja 121
4.1.5 Nikoli veljavne trditve 121
4.1.6 Temporalne formule 123
4.1.7 Zaključek 125
4.2 Dopolnitev modela s sondami 127
4.2.1 Iskanje zagatnih stanj 127
4.2.2 Kršitve semantičnih pravil jezika SDL 128
4.2.3 Iskanje potencialno nepravilnih izvajalnih poti 133
4.2.4 Iskanje cikličnih izvajalnih poti 133
4.2.5 Preverjanje temporalnih lastnosti modela 134

5 Primer verifikacije specifikacije protokola V.76 **137**
5.1 Popravki izvorne specifikacije 137
5.2 Specifikacija v jeziku SDL 137
5.3 Formalna verifikacija 142
5.3.1 Model okolja 143
5.3.2 Zagatna stanja 144
5.3.2.1 Prva dopolnitev sistema 145
5.3.2.2 Druga dopolnitev sistema 148
5.3.2.3 Tretja dopolnitev sistema 150
5.3.2.4 Četrta dopolnitev sistema 151
5.3.2.5 Peta dopolnitev sistema 152
5.3.2.6 Šesta dopolnitev sistema 153
5.3.2.7 Sedma dopolnitev sistema 155
5.3.2.8 Osma dopolnitev sistema 156
5.3.2.9 Deveta dopolnitev sistema 157
5.3.3 Kršitve semantičnih pravil jezika SDL 157
5.3.4 Iskanje potencialno nepravilnih izvajalnih poti 157
5.3.5 Iskanje cikličnih izvajalnih poti 160
5.3.6 Preverjanje temporalnih lastnosti modela 161
5.3.7 Zaključek 164

6 Zaključek **165**
6.1 Povzetek prispevkov disertacije 166
6.2 Zaključna razprava in usmeritev nadaljnjih raziskav 167

Literatura **169**

A Modeliranje operatorjev tipa charstring **173**

B Modeliranje vhodnih konstruktov **177**
B.1 Specifikacija sistema 177
B.2 Dodatne vhodne datoteke 178
B.2.1 sdl_run.ph 178
B.2.2 sdl_comm.ph 178
B.3 Model sistema 179

C Primeri avtomatske tvorbe modelov **185**
C.1 Dopolnitev modela s sondami 185
C.1.1 Model sistema `ustvari` 185
C.2 Protokol V.76 190
C.2.1 Specifikacija sistema v jeziku SDL 190
C.2.2 Model sistema v jeziku Promela 203
C.2.2.1 Osnovne definicije 203
C.2.2.2 Model uporabnika SUa 209
C.2.2.3 Model procesa `dispatch` v bloku `DLCa` 216

Biografija **235**

Bibliografija **237**

Slike

2.1 EBNF zapis sintakse specifikacije v jeziku Promela 8
2.2 EBNF zapis sintakse specifikacije procesnega tipa 9
2.3 EBNF zapis sintakse specifikacije stavka 11
2.4 EBNF zapis sintakse izraza 12
2.5 Primer uporabe konstrukta $unless$ 14
2.6 EBNF zapis sintaksa deklaracije spremenljivke 15
2.7 EBNF zapis sintakse definicije kanala 15
2.8 EBNF zapis sintakse definicije $mtype$ in uporabniškega podatkovnega tipa . 16

3.1 Modeliranje podatkovnih tipov $character$ in $charstring$ 22
3.2 EBNF zapis sintakse definicije podatkovnega tipa 25
3.3 Primer pretvorbe enostavnega sestavljenega tipa 26
3.4 Primer pretvorbe polja 28
3.5 EBNF zapis sintakse izraza 31
3.6 Primer pretvorbe naštevnega podatkovnega tipa 31
3.7 Primer pretvorbe podintervalnega tipa 36
3.8 EBNF zapis sintakse definicije sinonima 36
3.9 Primer pretvorbe sinonimov 38
3.10 EBNF zapis deklaracije časovnika 39
3.11 Dodatne definicije v modelu sistema za podporo časovnikom 39
3.12 EBNF zapis deklaracije spremenljivk 41
3.13 Komunikacija med procesi 43
3.14 Primer komunikacije med procesom in pridruženo vhodno vrsto 45
3.15 EBNF zapis sintakse definicije signala 45
3.16 EBNF zapis sintakse definicije signalne liste 47
3.17 EBNF zapis sintakse specifikacije kanala 48
3.18 Primer definicije kanala 49
3.19 Sintaksa specifikacije signalne poti 50
3.20 Sintaksa specifikacije priključka 51
3.21 Primer definicije signalne poti in priključka 52
3.22 Struktura sistema v SDL-ju 53

3.23 EBNF zapis sintakse specifikacije sistema 53
3.24 EBNF zapis sintakse specifikacije bloka 54
3.25 Primer uporabe podstrukture 55
3.26 EBNF zapis sintakse specifikacije podstrukture 55
3.27 Primer zapisa končnega avtomata v SDL 56
3.28 EBNF zapis sintakse specifikacije procesa 57
3.29 Primer dinamične izbire naslovnika 65
3.30 Model sistema na sliki 3.29 67
3.31 EBNF zapis sintakse specifikacije procedure 72
3.32 Primer vključevanja procedure s stanji v telo procesa 73
3.33 EBNF zapis sintakse specifikacije konstruktov start in stop 76
3.34 EBNF zapis sintakse specifikacije prehoda 77
3.35 EBNF zapis sintakse specifikacije konstrukta stanje 78
3.36 EBNF zapis sintakse specifikacije seznama stanj zvezdica 78
3.37 EBNF zapis sintakse specifikacije konstrukta shrani 81
3.38 Definicija polja za opis stanja kanala 82
3.39 Dodatni konstrukti za modeliranje sprejema 83
3.40 EBNF zapis sintakse specifikacije konstruktov vhod in omogočitveni pogoj 84
3.41 EBNF zapis sintakse specifikacije konstrukta prioritetni vhod 86
3.42 Poskus modeliranja sprejema prioritetnega signala 87
3.43 Rezultat formalne verifikacije modela na sliki 3.42 87
3.44 Modeliranje sprejema prioritetnega signala 89
3.45 Rezultat formalne verifikacije modela na sliki 3.44 90
3.46 EBNF zapis sintakse specifikacije konstrukta spontani prehod 92
3.47 Naključna izbira in dopolnitev procesnega tipa `__Timers` 92
3.48 EBNF zapis sintakse specifikacije konstrukta časovnik 93
3.49 Dodatne definicije v modelu sistema za podporo časovnikom 94
3.50 EBNF zapis sintakse specifikacije konstrukta trajen signal 96
3.51 EBNF zapis sintakse specifikacije konstrukta izhod 97
3.52 Primer povezav med procesi bloka 99
3.53 EBNF zapis sintakse specifikacije konstrukta ustvari 104
3.54 EBNF zapis sintakse specifikacije konstrukta naslednje stanje 106
3.55 EBNF zapis sintakse specifikacije konstruktov oznaka in združi 106
3.56 EBNF zapis sintakse specifikacije konstrukta naloga 107
3.57 EBNF zapis sintakse niza znakov 108
3.58 EBNF zapis sintakse specifikacije konstrukta odločitev 109
3.59 Primer sistema z dvema procesoma 112
3.60 Primer modeliranja vhodnih konstruktov 112
3.61 Primer formalne verifikacije modela sistema z orodjem `xdtspin` 114

3.62 Primer vodene simulacije sistema z orodjem `xdtspin` 116

4.1 Primer enostavnega procesa “never claim” 122
4.2 Proces “never claim” za trditev $\neg\Box(p \rightarrow (p \mathrm{U}\, q))$ 123
4.3 Pretvorba LTL formule $\Diamond\Box p$ v Büchijev avtomat 125
4.4 Pretvorba SI-EQLTL formule v Büchijev avtomat 126
4.5 Specifikacija sistema `ustvari` 128
4.6 Rezultati formalne verifikacije modela sistema `ustvari` 129
4.7 Nepravilna uporaba konstrukta odločitev v specifikaciji procesa `client` . 129
4.8 Izčrpna simulacija z orodjem OG ne zazna kršitve semantičnih pravil 130
4.9 Primer modeliranja konstrukta odločitev z vključitvijo sonde 130
4.10 Razširitev specifikacije procesa `client` 131
4.11 Primer sonde za preverjanje skladnosti s predpisano zalogo vrednosti 132
4.12 Specifikacija procesa `client` brez in z oznako stanja napredka 134

5.1 Grafični opis specifikacije sistema `V76test` 138
5.2 Grafični opis specifikacije bloka `DLCa` 139
5.3 Grafični opis specifikacije bloka `dataLink` 139
5.4 Delna specifikacija grafa procesa `AtoB` 139
5.5 Diagrami MSC 140
5.6 Delna specifikacija grafa procesa `DLC` 141
5.7 Prvotna specifikacija okolja 143
5.8 Delna specifikacija grafa procesa `dispatch` 144
5.9 Razširjen MSC diagram prvega protiprimera 145
5.10 Razširjen MSC diagram drugega protiprimera 146
5.11 Delna specifikacija procesa `dispatch` 147
5.12 MSC diagram protiprimera, ki ga izriše orodje $xdtspin$ 148
5.13 Razširjen MSC diagram tretjega protiprimera 149
5.14 Razširjen MSC diagram četrtega protiprimera 150
5.15 Razširjen MSC diagram petega protiprimera 151
5.16 Razširjen MSC diagram šestega protiprimera 151
5.17 Razširjen MSC diagram sedmega protiprimera 152
5.18 Dopolnjena specifikacija procesa `dispatch` 153
5.19 Spremenjena specifikacija procesa `SUa` 154
5.20 MSC diagram osmega protiprimera 155
5.21 Razširjen MSC diagram devetega protiprimera 156
5.22 Razširjen MSC diagram desetega protiprimera 157
5.23 Dopolnjena specifikacija procesa `DLC` s sprejemom ukaza `DISC` 158
5.24 Formalna verifikacija modela brez neveljavnih končnih stanj 158

5.25 Razširjen MSC diagram implicitnega sprejema signalov L_SetparmResp in L_EstabResp pri procesu dispatch v stanju ready 159
5.26 Razširjen MSC diagram implicitnega sprejema signala L_EstabResp pri procesu dispatch v stanju waitParmResp 160
5.27 Razširjen MSC diagram implicitnega sprejema signala L_SetparResp pri procesu dispatch v stanju waitEstabResp 160
5.28 Razširjen MSC diagram implicitnega sprejema signala L_ReleaseReq pri procesu DLC v stanju waitUAdisc 161
5.29 Razširjen MSC diagram sprejema ukaza DM pri procesu dispatch v stanju waitUA 161
5.30 Dopolnjena specifikacija procesa dispatch s sprejemom ukaza DM v stanju waitUA 162
5.31 Razširjen MSC diagram izvajalne poti, ki krši specifikacijo zahtev $\Box\neg$(environment_SUa_V_Data.val == 86) 163

Tabele

2.1 Operatorji in njihova prioriteta v Promeli 12
2.2 Standardni podatkovni tipi v Promeli 15

3.1 Definicije preslikav literalov SDL v Promelo 23
3.2 Rezultati formalne verifikacije modela 115

4.1 Pogosto uporabljene LTL formule 125

5.1 Primeri implicitnih sprejemov signalov 159
5.2 Zaporedje oddaje in sprejema signalov 163

Algoritmi

3.1 Algoritem analize in pretvorbe definicij novih podatkovnih tipov 27
3.2 Algoritem za pretvorbo izrazov 30
3.3 Algoritem analize in pretvorbe definicij podintervalnih tipov 32
3.4 Algoritem analize zaloge vrednosti 34
3.5 Algoritem za analizo in pretvorbo sinonimov 37
3.6 Algoritem analize definicij časovnikov 40
3.7 Algoritem za analizo spremenljivk 42
3.8 Algoritem analize signalov 46
3.9 Algoritem analize signalnih list 47
3.10 Algoritem analize kanalov 49
3.11 Algoritem analize signalnih poti 51
3.12 Algoritem analize priključkov 52
3.13 Algoritem za analizo definicije procesa, vhodnih konstruktov in pripravo pridruženega kanala 62
3.14 Algoritem za inicializacijo modela sistema 68
3.15 Algoritem za tvorbo procesnega tipa v modelu sistema 70
3.16 Algoritem analize procedur 74
3.17 Algoritem za pretvorbo telesa procesa ali procedure 75
3.18 Algoritem za pretvorbo konstruktov v telesu procesa ali procedure 75
3.19 Algoritem pretvorbe konstrukta stanje 79
3.20 Algoritem pretvorbe konstrukta vhod 84
3.21 Modeliranje sprejema signala 85
3.22 Algoritem pretvorbe konstrukta prioritetni vhod 90
3.23 Algoritem modeliranja implicitnega prehoda 91
3.24 Algoritem za pretvorbo spontanega prehoda 92
3.25 Algoritem za obravnavo izteka časovnika 94
3.26 Algoritem pretvorbe konstrukta postavi 95
3.27 Algoritem pretvorbe konstrukta resetiraj 95
3.28 Algoritem za pretvorbo trajnega signala 96
3.29 Algoritem analize vseh veljavnih naslovnikov za vse konstrukte izhod . . . 98
3.30 Algoritem analize vseh povezav 100

3.31 Algoritem pretvorbe konstrukta izhod 102
3.32 Algoritem pretvorbe konstrukta ustvari 104
3.33 Algoritem pretvorbe konstrukta naslednje stanje 105
3.34 Algoritem za pretvorbo konstrukta oznaka 106
3.35 Algoritem za pretvorbo konstrukta združi 107
3.36 Algoritem za pretvorbo konstrukta naloga 108
3.37 Algoritem za prirejanje vrednosti zapisom 109
3.38 Algoritem pretvorbe konstrukta odločitev 110
3.39 Algoritem pretvorbe odgovora v konstruktu odločitev 110
3.40 Algoritem pretvorbe zaključka konstrukta odločitev 111

1. poglavje

Uvod v problematiko

Sodobna družba se vedno bolj zanaša na pravilno delovanje digitalnih naprav v vseh svojih aktivnostih. Vedno pogosteje je od pravilnega izvajanja predvidenih operacij odvisno tudi človeško življenje. Še posebno hitro se razvijata telekomunikacijska in računalniška industrija. V zadnjih letih dosežki ene neposredno vplivajo na razvoj druge. Stičišče obeh industrij predstavlja komunikacija med napravami.

Tržni mehanizmi narekujejo kratke razvojne roke za nove izdelke in nenehne izboljšave obstoječih sistemov. Sočasno s krajšimi razvojnimi roki raste kompleksnost operacij. Zaradi povečevanja funkcionalne kompleksnosti telekomunikacijskih sistemov postaja njihova specifikacija vedno težja, možnost napak v specifikaciji in izvedbi pa vedno večja [23, 35]. Zato je potrebno izbrati ustrezne metode, orodja in postopke, da se temu problemu v čim večji meri izognemo. Tradicionalne metodologije razvoja programske opreme temeljijo na neformalnih tekstualnih opisih. Uporaba metodologij, ki ne temeljijo na formalnih metodah, omejuje možnost preverjanja pravilnosti delovanja specifikacij protokolov na testiranje. Iskanje napake je torej omejeno na preverjanje skladnosti s pripravljenimi testnimi zaporedji. Z večanjem kompleksnosti sistema se brez uporabe formalnih metod celovitost preverjanja sistema manjša [24, 10]. Večina sodobnih protokolov je tako kompleksnih, da bi preverjanje vseh mogočih izvajalnih poti trajalo nesprejemljivo dolgo, v sistemih s sočasnostjo pa to pogosto tudi ni mogoče. S tem dopuščamo možnost, da nepravilnosti v sistemu spregledamo. V primeru, ko se napaka odkrije v tržno uspešnem izdelku, so stroški pogosto zelo visoki [15]. Zato težimo k temu, da bi s sistematično uporabo formalih metod v čim večji meri zagotovili pravilno delovanje sistema [5]. Spekter uporabe formalnih metod zajema vse razvojne korake [2, 11]. Od natančnega zapisa specifikacije sistema pa vse do formalne verifikacije končne implementacije sistema [33, 38, 46]. V zadnjih treh desetletjih se je na področju formalnih tehnik opisovanja sistemov izdelala množica različnih formalizmov in pristopov k obravnavi problematike [9].

1.1 Specifikacija sistemov

V telekomunikacijski industriji se je za pripravo standardov in razvoj programske opreme uveljavil opisni jezik SDL (Specification and Description Language). Uporablja se tako za abstraktne opise sistemov, kot za natančne specifikacije izvedbe, ki se avtomatsko prevajajo v programske jezike za ciljne sisteme.

SDL je zapisan kot standard ITU (International Telecommunication Union) Z.100 [57]. Začetek razvoja jezika sega v leto 1972, prva verzija pa je bila izdana leta 1976 (oranžna knjiga). Sledile so verzije v letih 1980 (rumena knjiga), 1984 (rdeča knjiga) in 1988 (modra knjiga). Zadnji dve omenjeni verziji sta vsebovali mnoge razširitve in se smatrata kot prvi zreli verziji. Verziji iz leta 1988 so sledile še nove objektno orientirane verzije oz. dopolnitve v letih 1992, 1996 in 2000.

Stabilnost jezika je pri opisih velikih sistemov posebnega pomena. Zato so vsi sistemi, ki so opisani s SDL-88, praviloma veljavni tudi v novejših verzijah jezika. Za natančnejši pregled priporočamo [57, 48, 43, 4, 18, 49, 59]. V izdaji iz leta 2000 je SDL doživel prvo večjo spremembo pri opisovanju strukture sistema, saj se je koncept blokov in procesov združil v obliki agentov. Vsebuje tudi mnoge dodatne spremembe na področju obravnave podatkovnih tipov. Vsa komercialna orodja še ne zagotavljajo polne podpore zadnjih dveh verzij. Tudi pri našem delu imamo dostop samo do orodja ObjectGEODE (OG) podjetja Telelogic, ki podpira standarde iz let 1988 in 1992 ter nekatere razširitve iz leta 1996. Zaradi dolgoletnega razvoja je velika večina specifikacij napisana v verziji iz leta 1992, zato smo se odločili, da v disertaciji uporabimo to verzijo.

Jezik SDL je primeren za opisovanje sistemov s sočasnostjo, saj specifikacija temelji na med seboj komunicirajočih asinhronih procesih. Procesi so opisani s pomočjo razširjenih končnih avtomatov — EFSM (Extended Finite State Machine) — in se izvajajo sočasno. Pri opisu sistema navadno pričnemo z neformalnim opisom na zelo visokem nivoju abstrakcije. Postopoma se sistem opisuje vedno bolj podrobno. Končna oblika specifikacije je lahko zapisana formalno ali polformalno, saj SDL dopušča, da so deli specifikacije opisani neformalno. Večina komercialnih orodij omogoča, da se neformalni zapis izvede v obliki zunanjih programskih gradnikov.

V končni obliki so formalno ali polformalno opisani vsi deli sistema. Specifikacije lahko z dodatnimi orodji simuliramo, verificiramo ter avtomatsko generiramo izvorno kodo za različne programske jezike in ciljne sisteme. Našteto lahko nudi jezik, ki ima:

- nedvoumno, jasno, natančno in skladno specifikacijo,
- osnovo za določanje skladnosti implementacije s specifikacijo,
- osnovo za določanje skladnosti specifikacij med seboj in
- uporabna računalniška orodja za razvoj, vzdrževanje, analizo, simulacijo in formalno verifikacijo specifikacij.

1.2 Preverjanje pravilnosti sistemov

Eden izmed osnovnih problemov pri preverjanju pravilnosti obnašanja sistemov je, kako pokazati, da specifikacija zasnove ali implementacije sistema zadosti dani specifikaciji zahtev sistema. Problema se lahko lotimo tako, da sistem predstavimo z modelom [16, 38, 29], ki verno predstavlja obnašanje sistema. Želene lastnosti sistema preverjamo z različnimi trditvami, tiste najzahtevnejše pa zapišemo s pomočjo ene izmed temporalnih logik [55, 56, 42]. Pri temporalnih logikah sklepamo o tem, kako se resničnost izjave spreminja s časom. Tehnika preverjanja modelov [16] nam lahko z matematično natančnostjo potrdi ali ovrže skladnost podanega modela sistema s specifikacijo zahtev. Ta je lahko podana z zapisom želenih lastnosti s posebnimi oznakami stanj modela ali v obliki trditev in temporalnih formul [30].

V model sistema je za sledenje spremembam, ki so povezane s preverjanimi lastnostmi, potrebno namestiti t. i. sonde. Sonde so lahko trditve, ki vplivajo na potek formalne verifikacije, posebne oznake stanj ali spremenljivke, katerim se ob ključnih dogodkih spreminja vrednost, tako da odražajo trenutno stanje sistema. Na osnovi sprememb, ki jih zaznamo z uporabo sond, lahko sklepamo o skladnosti modela sistema s preverjanimi lastnostmi. Če model sistema ni skladen z zahtevami, je kot rezultat verifikacije na voljo natančna pot izvajanja, ki je privedla do napake.

Orodje za formalno verifikacijo predstavlja pomemben del razvojnega okolja, saj omogoča preverjanje pravilnosti delovanja specifikacije sistema z matematično natančnostjo. V telekomunikacijah se je na področju formalne verifikacije s tehniko preverjanja modelov uveljavilo orodje Spin [27, 28, 32, 31, 50, 26], ki je leta 2002 prejelo nagrado “ACM Software System Award”. Vhod v orodje je opis modela sistema z jezikom Promela.

Promela je jezik za opis modelov porazdeljenih sistemov s sočasnostjo, ki ga je razvil Gerard Holzmann. Poudarek je na preverjanju pravilnosti koordinacije in sinhronizacije dogodkov in ne na računski pravilnosti izoliranih sekvenčnih računskih postopkov. Zato je namenoma načrtovan tako, da vzpodbuja abstrakcijo v delih, ki opisujejo računske postopke. Jezik zato ne podpira določenih elementov, ki so prisotni v večini programskih jezikov (npr. realnih števila).

Model sistema, ki je predmet formalne verifikacije, se razlikuje od natančne specifikacije, ki predstavlja osnovo za avtomatsko generiranje izvorne kode v enem izmed programskih jezikov, vsaj v dveh pomembnih točkah.

- Model je praviloma abstraktna predstavitev dela preverjanega sistema, ki ga izberemo v odvisnosti od preverjanih lastnosti.
- Model vključuje dodatne elemente, ki nam omogočajo preverjanje lastnosti.

Za formalno verifikacijo specifikacije sistema z orodjem Spin je potrebno pridobiti model sistema v jeziku Promela. Spin ne podpira uporabe časovnikov, ki so v opisu protokolov v telekomunikacijah pogosto uporabljeni. Zato smo uporabili razširitev orodja s podporo

diskretnemu času [6, 7], ki je za verzijo orodja 3.3.10 nastala v okviru raziskav projekta Vires [3, 8, 7]. V disertaciji je uporabljena razširitev za verzijo orodja 4.1.1, ki je nastala na našo pobudo.

V splošnem se model sistema lahko pridobi ročno ali avtomatsko [37, 51, 34]. Model sistema lahko ob ročnem pristopu tvori le strokovnjak z dobrim poznavanjem delovanja sistema in jezika za opis modelov. Kvaliteta modela je neposredno odvisna od strokovnjaka. Pri postopku ročne gradnje modela hitro pride do napak pri modeliranju lastnosti sistema.

V nalogi se posvečamo avtomatski pridobitvi modela s pretvorbo opisa specifikacije sistema v jeziku SDL v opis modela v jeziku Promela. Tako je postopek pretvorbe popolnoma avtomatiziran. Ob preverjanju skladnosti modela s specifikacijo zahtev se lahko ob kršitvi pokaže samo troje:

1. specifikacija sistema ni skladna s specifikacijo zahtev,
2. model sistema ni verna predstavitev specifikacije ali
3. algoritmi za formalno verifikacijo vsebujejo napako.

V drugem primeru je potrebno dopolniti algoritme za avtomatsko tvorbo modela. Če so popravki uspešni, so vsi nadaljnji modeli pravilni. Brez večjih težav se lahko preverijo tudi starejše specifikacije, saj niso potrebni ročni popravki. V disertaciji opisujemo avtomatsko tvorbo modelov s sondami. S tem se odpravi vpliv človeškega dejavnika in zmanjša verjetnost vnosa napak v postopek formalne verifikacije.

1.3 Kratek pregled obstoječih pristopov

Na področju avtomatske tvorbe modelov dela več raziskovalnih skupin na univerzah in v industriji. Izpostaviti velja skupino raziskovalcev laboratorijev Bell Labs, ki so del podjetja Lucent Technologies. Njihovi dosežki avtomatske tvorbe modelov iz programskega jezika C in preverjanja telekomunikacijskega sistema so opisani v [37, 38]. V okviru te skupine je nastalo orodje Spin. Z raziskavami na področju avtomatske tvorbe modelov v jeziku Promela se ukvarjajo tudi v agenciji NASA. V [25] so predstavljeni rezultati avtomatske tvorbe modelov v jeziku Promela iz specifikacije sistemov v programskem jeziku Java.

Najvidnejše raziskave na področju avtomatske tvorbe modelov iz specifikacije sistema v jeziku SDL so bile opravljene v okviru projekta Vires. Problem je razdeljen na dva koraka. V prvem koraku se izvede pretvorba SDL opisa sistema v vmesni opisni format (Intermediate Format - IF) [13, 12, 14]. S tem se pridobi splošen opis sistema z uporabo orodja `sdl2if`, ki je na voljo le v binarni obliki. Izvedba je nedeljivo povezana z uporabo komercialnega razvojnega orodja ObjectGEODE in zato ni na voljo širši znanstveni skupnosti. Sledi pretvorba v zapis modela z jezikom Promela s programom `if2pml`, ki je prav tako nastal v okviru projekta Vires [8, 47]. Formalna verifikacija se izvaja z uporabo orodja DT Spin [7].

Praktičen primer uporabe pri verifikaciji protokola MASCARA je predstavljen v [51]. Avtomatska tvorba modelov s temi orodji je mogoča le za bistveno manjšo množico sistemov, kot jo omogoča orodje $sdl2pml$, ki je nastalo na osnovi algoritmov te disertacije. Razlike so podrobneje opisane v tretjem poglavju. V [47] je predstavljen poskus rešitve konstrukta shrani, ki ne vključuje sprejema prioritetnega signala in podvoji število pridruženih kanalov. S tem se posredno razpolovi število procesov v modelu sistema.

Obstajajo tudi raziskave na področju formalne verifikacije specifikacij v jeziku SDL z uporabo Petrijevih mrež [39, 40]. V [41] opisujejo orodje Emma, ki so ga uporabili za preverjanje pravilnosti specifikacije telekomunikacijskega sistema DX 200 v jeziku TeleNokia SDL (TNSDL). Poskus neformalnega zapisa pretvorbe nekaterih konstruktov jezika TNSDL v Promelo opisuje [52].

1.4 Pregled vsebine

Uvodnemu poglavju sledi poglavje 2, kjer je jedrnato predstavljen jezik za opis modelov sistemov. Promela predstavlja vhod v izbrano orodje za formalno verifikacijo s tehniko preverjanja modelov — Spin. V poglavju se predstavita sintaksa in semantika jezika, ki sta potrebni za razumevanje drugih poglavij. Sistematično obdelamo vse konstrukte in s praktičnimi primeri demonstriramo ključne lastnosti jezika. Poglavje ne vključuje sintakse programskega jezika C, ki ga lahko vključujejo določeni konstrukti v specifikaciji modela. Ti deli so označeni s <C-koda>.

V poglavju 3 formalno opišemo vse podprte konstrukte jezika SDL in definiramo množice, ki jih nedvoumno opisujejo. Jedro disertacije predstavljajo algoritmi za avtomatsko tvorbo modelov iz specifikacije v jeziku SDL. Poglavje vsebuje 40 algoritmov. V psevdo zapisu večinoma uporabljamo konstrukte jezika SDL. Zaradi večje preglednosti in lažjega sklicevanja v besedilu smo v algoritmih posamezne dele zapisali v simbolični obliki z vpeljavo novih funkcij, uporabo eksistenčnih kvantifikatorjev in osnovnih logičnih operatorjev. Sintaksi jezikov Promela in SDL sta zapisani z notacijo Extended Backus-Naur Form (EBNF). Izbire so tako ločene z navpično črto, opcijski deli so obdani z oglatimi oklepaji, združevanje se izvaja z uporabo zavitih oklepajev, zvezdica pa označuje nič ali več ponovitev predhodnega elementa. Rezervirane besede jezika so v besedilu poudarjene, imena dodatnih sintaktičnih pravil pa so navedena med koničastimi oklepaji. Spremenljivke so v algoritmih zapisane poševno. Celotno poglavje je naš prispevek k avtomatski tvorbi modelov in formalni verifikaciji sistemov.

V prvem delu 4. poglavja predstavimo pravilnostne trditve, ki se lahko uporabijo za zapis specifikacije zahtev pri preverjanju modelov z orodjem Spin. Za uspešno preverjanje specifikacij sistemov z metodo preverjanja modelov je potrebno pridobljeni model tudi primerno dopolniti s sondami. Jedro poglavja predstavlja definicija postopkov za avtomatsko

dopolnitev modela s sondami. Te vključujejo različne elemente jezika Promela. Od posebnih oznak stanj do med seboj odvisnih globalnih spremenljivk za preverjanje najzahtevnejših specifikacij zahtev, ki so zapisane s formulami LTL. Manjše število dodatnih ročnih posegov v zapis modela sistema zmanjša možnost vnosa neželenih sprememb v model sistema, ki bi lahko privedle do napačnih rezultatov formalne verifikacije.

Teoretične raziskave smo podkrepili s praktično realizacijo algoritmov v orodju $sdl2pml$. V poglavju 5 na praktičnem primeru specifikacije protokola V.76 demonstriramo uporabno vrednost rezultatov disertacije. Vsi uporabljeni modeli s sondami so pridobljeni avtomatično.

V zaključku povzamemo glavne prispevke k znanosti, ocenimo dosežene rezultate in nakažemo smernice za nadaljnje raziskave na področju avtomatske tvorbe modelov in formalne verifikacije sistemov.

Zaključku sledi seznam uporabljene literature.

V dodatku A je zaradi celovitosti disertacije predstavljeno modeliranje operatorjev tipa $CHARSTRING$.

V dodatku B so podani polni izpisi primerov specifikacije v jeziku SDL in ročno tvorjenih modelov, na katere se sklicujemo v tretjem poglavju.

V dodatku C so povzeti bistveni deli avtomatsko tvorjenih modelov s sondami, ki so uporabljeni na koncu tretjega in v petem poglavju. Celotnih izpisov nismo priložili zaradi njihove obsežnosti.

Dodatkom sledita biografija in bibliografija kandidata.

2. poglavje

Promela — jezik za opis modelov sistemov

Z orodjem Spin preverjamo modele, ki so opisani v jeziku Promela [36]. Ob razvoju orodja in jezika so predstavljali cilj obravnave porazdeljeni telekomunikacijski sistemi, ki se lahko opisujejo z neodvisnimi procesi. Jezik je zasnovan tako, da je graf vseh dosegljivih stanj sistema vedno končen. Zato je potrebno zagotoviti, da so tudi vsi konstrukti jezika končni. Tako je sistem lahko opisan samo s končnim številom procesov, vsebuje samo končno število spremenljivk in kanalov, zaloga vrednosti spremenljivk pa je vedno končna in definirana vnaprej.

Strukturo grafa določa semantika jezika Promela, do njegove predstavitve pa lahko pridemo s simulacijo izvajanja sistema. Vozlišča grafa predstavljajo stanja sistema, prehodi med stanji pa opisujejo nedeljivo izvajanje enega izmed procesov sistema. Nedeljivo izvajanje je lahko en ali več neprekinljivo združenih stavkov, ki spremenijo stanje sistema. Vsa stanja končnega grafa se lahko preverijo v končnem času in s končnimi pomnilniškimi viri. Pravilnost delovanja modela preverjamo s trditvami, ki opisujejo neželena stanja ali prehode sistema. Za preverjanje obstoja določenega podgrafa pa se morajo uporabiti močnejši opisi v obliki formul LTL.

Semantični stroj jezika operira s:

- procesi,
- spremenljivkami in
- komunikacijskimi kanali.

Procesom dodeljuje identifikacijska števila, določa vrstni red in pogostost izvajanja posameznega aktivnega primerka procesa, vrši nedeterministično izbiro med omogočenimi prehodi, procese obravnava kot končne avtomate ter z uporabo posebnih skritih spremenljivk zagotavlja pravilno izvajanje modela sistema.

Definicija 2.1 določa elemente, ki sestavljajo globalno stanje sistema.

Definicija 2.1 $Stanje$ sistema je osmerica $(gsprem, procesi, kanali, izključevalen, rokovanje, prekinitev, drugače, jecljanje)$, kjer je:

$gsprem$ končna množica globalnih spremenljivk;
$procesi$ končna množica procesov;
$kanali$ končna množica kanalov;
$izključevalen$ celo število, ki se uporablja pri uveljavljanju pravilne interpretacije konstruktov $atomic$ in d_step;
$rokovanje$ celo število, ki se uporablja pri uveljavljanju pravilne interpretacije sinhrone komunikacije;
$prekinitev$ boolova konstanta, ki se uporablja pri vrednotenju stavkov;
$drugače$ boolova konstanta, ki se uporabljata pri vrednotenju stavkov;
$jecljanje$ boolova konstanta, ki se uporablja pri razširitvi za podporo jecljanju.♦

Sintakso specifikacije v jeziku Promela prikazuje slika 2.1. Število aktivnih procesov vse dosedanje verzije Spina omejujejo na 256. Ta omejitev je namerna in želi vzpodbuditi razvijalca h generaciji čim manjšega modela, ki bi verno predstavljal del preverjanega sistema.

```
<specifikacija> ::= <modul> [<modul>]*

<modul> ::=  <definicija procesa>
           | <začetni proces>
           | <never>
           | <sled>
           | <uporabniški podatkovni tip>
           | <mtype>
           | <seznam deklaracij>
           | <c_code> '{' <C-koda> '}'
           | <c_decl> '{' <C-koda> '}'
           | <c_state> <niz znakov> <niz znakov> [<niz znakov>]
           | <c_track> <niz znakov> <niz znakov>

<never> ::= never <zaporedje>
<sled> ::= trace <zaporedje>
```

Slika 2.1: EBNF zapis sintakse specifikacije v jeziku Promela

Veljavni model mora imeti vsaj en aktiven proces. To je lahko katerikoli proces, ki se že ob definiciji določi kot aktiven. Če takšnega procesa ni, se mora v veljavni specifikaciji modela uporabiti <začetni proces> (slika 2.2). Elementa <never> in <sled> ne opisujeta model sistema, temveč v okviru lastnega procesa podajata pravilnostne zahteve. Uporabljata se pri formalni verifikaciji lastnosti obravnavanega modela.

2.1 Proces

Procesi definirajo delovanje modela in se opisujejo s procesnimi tipi. Vsak procesni tip lahko ima več primerkov aktivnih procesov, ki delujejo neodvisno, sočasno in asinhrono. Sinhronizacija med procesi se lahko izvaja samo z izrazi v telesu procesov in dodatnim pogojem,

ki se navede ob deklaraciji procesa. Vsak proces se lahko opiše s končnim avtomatom, ki ga določa definicija 2.2. Slika 2.2 prikazuje EBNF zapis sintakse specifikacije procesnega tipa.

Definicija 2.2 $Proces$ je šesterica $(pid, lsprem, lstanje, začetno, trenutno, prehod)$, kjer je:

pid pozitivno celo število, ki edinstveno določa aktivni proces;
$lsprem$ končna množica lokalnih spremenljivk, ki so vidne samo v območju procesa s pid;
$lstanje$ končna množica stanj;
$začetno$ element množice $lstanje$ in opisuje začetno stanje;
$trenutno$ element množice $lstanje$ in opisuje trenutno stanje;
$prehod$ končna množica prehodov med stanji iz $lstanje$.♦

```
<začetni proces> ::= [<prioriteta>] '{' <zaporedje> '}'

<definicija procesa> ::= [<aktiven>] proctype <ime procesnega tipa> ([<seznam deklaracij>])
                         [<prioriteta>] [<pogoj>] '{'<telo procesa> '}'

<aktiven> ::= active ['['<konstanta>']']
<konstanta> ::= true | false | skip | <število>[<število>]*
<število> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<ime procesnega tipa> ::= <ime>
<ime> ::= <črka>[<črka> | <število>]*
<seznam deklaracij> ::= <deklaracija>[;<deklaracija>]*
<prioriteta> ::= priority <konstanta>
<pogoj> ::= provided (<izraz>)
<telo procesa> ::= <zaporedje>
<zaporedje> ::= <korak> [; <korak>]*

<korak> ::=  <stavek> [ unless <stavek>] | <seznam deklaracij>
           | xr <spremenljivka>[, <spremenljivka>]*
           | xs <spremenljivka> [, <spremenljivka>]*

<črka> ::=  A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S
          | T | U | V | W | X | Y | Z | a | b | c | d | e | f | g | h | i | j | k | l | m
          | n | o | p | q | r | s | t | u | v | w | x | y | z
```

Slika 2.2: EBNF zapis sintakse specifikacije procesnega tipa

Proces se lahko aktivira ob deklaraciji z uporabo rezervirane besede **active** ali z operatorjem run v telesu kateregakoli aktivnega procesa. Procesi, ki so že ob deklaraciji definirani kot aktivni, se v nedeterminističnem vrstnem redu tvorijo ob zagonu sistema v številu, ki ga določa <konstanta> v <aktiven>. Definicija formalnih parametrov v <seznam deklaracij> predstavlja parametrizacijo procesnih tipov.

Po privzetem imajo vsi procesi enako prioriteto, ki jo lahko ovrednotimo z 1. Za namene simulacije pa se lahko definiciji aktivnega procesnega tipa ali ob uporabi operatorja run doda želena prioriteta. Ob naključni simulaciji se reža za izvajanje procesom z višjo prioriteto dodeli pogosteje. Proces, ki ima prioriteto 10, ima tako, statistično gledano, 10-krat

več priložnosti za izvajanje telesa procesa. Pri formalni verifikaciji prioriteta ni pomembna, saj se preverijo vsa možna izvajanja.

Dodatni pogoj ob deklaraciji procesnega tipa predstavlja posebno obliko sinhronizacije med procesi in lahko vsebuje v pridruženem izrazu poljubne globalne spremenljivke. Primerna uporaba omogoča definicijo posebnih zaporedij izvajanja aktivnih procesov. Uporaba se odsvetuje zaradi posledične neučinkovitosti nekaterih optimiziranih iskalnih algoritmov pri formalni verifikaciji.

Vsak proces vsebuje nič ali več deklaracij spremenljivk in enega ali več stavkov (slika 2.3). Izvaja se neodvisno od drugih procesov, dokler ne pride do konca telesa procesa. V tem trenutku zaključi izvajanje, vendar ostane aktiven. Proces lahko "umre" oz. postane neaktiven samo v trenutku, ko so vsi mlajši procesi neaktivni. Od posameznega procesa so mlajši vsi procesi, ki so se med izvajanjem sistema tvorili kasneje. Neaktivni proces se odstrani iz sistema, njegov pid pa se lahko ponovno uporabi ob naslednji aktivaciji poljubnega procesa.

Stavek se lahko izvede samo, kadar je izvršljiv. Če nobeden izmed razpoložljivih stavkov ni izvršljiv, se izvajanje procesa blokira, dokler vsaj en stavek ne postane izvršljiv. Kadar je izvajanje procesa blokirano, se v okviru tega procesa ne izvede noben prehod, ki bi spremenil stanje sistema. V definicijah uporabljamo ob sklicevanju na elemente druge definicije notacijo $ime_definicije.ime_elementa$.

Definicija 2.3 $Prehod$ je sedmerica $(p_id, izvor, cilj, pogoj, akcija, prioriteta, rv)$, kjer:

p_id edinstveno določa prehod in je nenegativno celo število;
$izvor$ in $cilj$ sta elementa množice $proces.lstanje$;
$pogoj$ je boolov izraz, ki se nanaša na globalno stanje sistema;
$akcija$ je funkcija, ki spremeni globalno stanje sistema;
$prioriteta$ je celo število, ki se uporablja pri uveljavljanju pravilne interpretacije konstrukta $unless$;
rv je celo število, ki se uporablja pri uveljavljanju pravilne interpretacije sinhrone komunikacije. ♦

Razumevanje pravil izvršljivosti je ključno za tvorbo modelov, ki verno predstavljajo lastnosti obravnavanega sistema. V določenem trenutku izvajanja modela je stavek lahko bodisi izvršljiv bodisi neizvršlj. V Promeli poznamo štiri osnovne tipe stavkov: izpis, prireditev, izraz in vhodno/izhodni stavek. Izpis in prireditev sta po definiciji vedno brezpogojno izvršljiva. V Promeli se lahko izraz (slika 2.4) obravnava kot stavek, ki je izvršljiv samo, kadar se ovrednoti na boolovo vrednost 1 oz. celo število, ki je različno od nič. Izraz mora biti v Promeli obvezno brez stranskih učinkov — ob vrednotenju izraza se ne sme spremeniti globalno stanje sistema. Izjemo predstavlja izraz, ki vsebuje operator run. Ta ne spremeni globalnega stanja sistema samo v primeru, ko je aktivno maksimalno podprto število procesov. V nasprotnem primeru se aktivira nov primerek procesa, izraz pa se ovrednoti na pozitivno število, ki je enako $proces.pid$. Za operator run so zato vpeljane dodatne omejitve

```
<stavek> ::=  if <opcije> fi                          /*izbira*/
            | do <opcije> od                          /*ponavljanje*/
            | atomic '{' <zaporedje> '}'
            | d_step '{' <zaporedje> '}'
            | '{' <zaporedje> '}'
            | <pošlji>
            | <sprejmi>
            | <priredi>
            | else
            | break
            | goto <ime>
            | <ime> ':' <stavek>                      /*označen stavek*/
            | print '(' <niz znakov> [ ',' <seznam argumentov> ] ')'
            | assert <izraz>
            | <izraz>

<pošlji> ::=  <spremenljivka> ! <argumenti pošlji>
            | <spremenljivka> !! <argumenti pošlji>

<argumenti pošlji> ::=  <seznam argumentov>
                      | <podizraz> (<seznam argumentov>)

<seznam argumentov> ::= <podizraz> [, <podizraz>]*

<sprejmi> ::=  <spremenljivka> ? <argumenti sprejmi>
             | <spremenljivka> ?? <argumenti sprejmi>
             | <spremenljivka> ? '<'<argumenti sprejmi>'>'
             | <spremenljivka> ?? '<'<argumenti sprejmi>'>'

<argumenti sprejmi> ::=  <argument sprejmi>[, <argument sprejmi>]*
                       | <argument sprejmi> (<argumenti sprejmi>)

<argument sprejmi> ::=  <spremenljivka> | eval (<spremenljivka>)
                      | [ - ] <konstanta>

<priredi> ::=  <spremenljivka> '=' <podizraz>
             | <spremenljivka> + + | <spremenljivka> - -

<niz znakov> ::= [<katerikoli viden znak ASCII razen znaka ">]*
<seznam argumentov> ::= <podizraz> [, <podizraz>]*
<opcije> ::= :: <zaporedje> [:: <zaporedje>]*
```

Slika 2.3: EBNF zapis sintakse specifikacije stavka

pri uporabi znotraj izraza [36]. Vse druge operatorje in njihovo prioriteto pri vrednotenju izrazov predstavlja tabela 2.1. Izvršljivost vhodno/izhodnih stavkov je definirana v odseku, ki opisuje komunikacijo.

Drugi elementi iz definicije izraza so namenjeni nadzoru izvajanja telesa sporočila in posebnim oblikam združevanja stavkov. Promela podpira pet tipov združevanja stavkov:

- atomarno zaporedje,
- deterministični koraki,
- izbira,
- ponavljanje in

```
<izraz promela> ::= <izraz>
<izraz> ::= <podizraz> | (<izraz>) | <izraz> <in-ali> <izraz>
          | <test kanala> (<spremenljivka>)

<podizraz> ::= (<podizraz>)
             | <podizraz> <binarni operator> <podizraz>
             | <unarni operator> <podizraz>
             | (<podizraz> - > <podizraz> : <podizraz>)
             | len(<spremenljivka>)
             | <preveri vrednost kanala>
             | <spremenljivka>
             | <konstanta>
             | timeout
             | np_
             | enabled(<podizraz>)
             | pc_value(<podizraz>)
             | name[<podizraz>] @ <ime>
             | run <ime> ([<seznam argumentov>])[<prioriteta>]

<binarni operator> ::= + | - | * | / | % | & | ^ | '|' | > | < | >= | <= | ==
                     | != | << | >> | <in-ali>

<unarni operator> ::= ~ | - | !
<in-ali> ::= && | ||
<test kanala> ::= full | empty | nfull | nempty

<preveri vrednost kanala> ::= <ime> ? '['<argumenti sprejmi>']'
                            | <ime> ?? '['<argumenti sprejmi>']'
```

Slika 2.4: EBNF zapis sintakse izraza

Tabela 2.1: Operatorji in njihova prioriteta v Promeli

operator	**asociativen**	**komentar**
(), [], .	od leve proti desni	oklepaj, oklepaj pri polju, ločilo elementov zapisa (najvišja prioriteta)
!, ~, ++, −−	od desne proti levi	negacija, komplement, inkrement, dekrement
*, /, %	od leve proti desni	množenje, deljenje, deljenje po modulu
+, −	od leve proti desni	seštevanje, odštevanje
<<, >>	od leve proti desni	levi in desni pomik
<, <=, >, >=	od leve proti desni	relacijski operatorji
==, !=	od leve proti desni	enako, ni enako
&	od leve proti desni	bitni operator in
^	od leve proti desni	bitni ekskluzivni ali
|	od leve proti desni	bitni ali
&&	od leve proti desni	logični in
||	od leve proti desni	logični ali
−>, :	od desne proti levi	operatorji pogojnega izraza
=	od desne proti levi	prirejanje (najnižja prioriteta)

- ubežne sekvence.

Dodatna združevanja omogočajo makroji in definicije $inline$, ki so posebna oblika makrojev s formalnimi parametri.

V atomarna zaporedja združujemo stavke, za katere želimo, da se izvedejo neprekinljivo. Torej v okviru iste izvajalne reže, ki je bila dodeljena procesu. Tako ne more priti do prepletanja izvajanja z drugimi procesi. To je, na primer, potrebno pri modeliranju operacije nad globalnimi spremenljivkami, ki je sestavljena iz več stavkov. S takšnim združevanjem lahko ob aktivaciji skupine procesov z operatorjem run zagotovimo, da se noben proces ne bo pričel izvajati pred aktivacijo zadnjega procesa. Nazoren primer predstavlja testiranje kanala, ki mu sledi pošiljanje sporočila v kanal (komunikacija je podrobneje obdelana v razdelku 2.3). Drugi stavek v zaporedju $nfull(kanal) -> kanal!sporočilo$ lahko postane neizvršljiv, če zaporedje ni definirano kot atomarno. Po testiranju kanala se lahko izvajalna reža dodeli procesu, ki pošlje sporočilo v isti kanal in s tem zapolni zadnjo prosto režo. Atomarno zaporedje lahko vključuje tudi nedeterministično izbiro, vendar se v primeru, ko ni izvršljiva nobena opcija, atomarnost zaporedju odvzame. Izvajalna reža se procesu vrne nedeterministično na osnovi odločitve semantičnega stroja.

Bolj strogo obliko združevanja omogoča stavek d_step. Vključeni stavki se vedno izvedejo v eni izvajalni reži. Da je takšno izvajanje vedno mogoče, mora zaporedje zadostiti naslednjim zahtevam:

- Izvajanje mora biti deterministično. Če se pojavijo nedeterministične opcije, se pri izvajanju vedno izvede ista — na primer, prva, ki je izvršljiva. Točen algoritem izbire ni definiran in je prepuščen implementaciji.
- Ne sme vsebovati skokov s stavkom $goto$.
- Noben stavek, razen testnega koraka, ne sme blokirati izvajanja zaporedja.

V primeru uporabe stavkov if in do, ki predstavljata izbiro in ponavljanje, je potrebno zagotoviti, da je presek vseh stražarjev prazna množica, unija pa univerzalna množica. Ker je od prvega koraka odvisna izvršljivost celotnega zaporedja, ga imenujemo stražar opcije (sliki 2.2 in 2.3). Izjavo d_step lahko vedno nadomestimo z $atomic$, saj zanjo ne veljajo zgornje omejitve. Iz zapisanega tudi sledi, da je d_step lahko del zaporedja, ki ga vključuje $atomic$, obratno pa ni dovoljeno.

Pri uporabi izbire se vedno izvede samo ena izmed opcij. Če je v določenem trenutku izvršljivih več opcij, se izbira izvede nedeterministično. Na mestu stražarja se lahko pojavi katerikoli stavek. Proces se blokira v primeru, ko ni možno izvesti nobene opcije. Enaka pravila veljajo pri uporabi stavka do, ki omogoča ponavljanje zaporedij do pojava rezervirane besede $break$. V obeh primerih se lahko na mestu stražarja uporabi rezervirana beseda $else$. Ta opcija postane izvršljiva v primeru, ko ni izvršljiva nobena druga opcija. Iz okvirov definiranega zaporedja lahko skočimo na poljubno mesto v telesu procesa z uporabo stavka $goto$. Cilj skoka je lahko katerikoli označen stavek (slika 2.3). V primeru, ko blokirajo vsi

procesi v sistemu, postane resničen izraz $timeout$. Uporabi se lahko kot stražar katerekoli opcije in je namenjen reševanju predvidenih zagatnih stanj.

Posebno obliko skoka predstavljajo ubežne sekvence z uporabo konstrukta $unless$. Primer 1 na sliki 2.5 prikazuje telo procesa, kjer se najprej izvede zaporedje `A`, nato se izvaja ponavljanje zaporedja, označenega z `B[1,2]`, do trenutka, ko postane izvršljiv stražar `c`. Sledi `C` in nato še `D`. Ključna razlika v primerjavi z drugim primerom na sliki je v tem, da se izvede `C` neposredno po zaključku kateregakoli stavka v okviru zaporedja `B[1,2]`. V drugem primeru se vedno izvede do konca celotno zaporedje `B[1,2]`.

```
A;
do
:: b1 -> B1
:: b2 -> B2
od unless {c -> C};
D
```

(1)

```
A;
do
:: b1 -> B1
:: b2 -> B2
:: c -> break
od;
C;
D
```

(2)

Slika 2.5: Primer uporabe konstrukta $unless$

2.2 Spremenljivke

Spremenljivke so lahko s stališča aktivnega procesa globalne ali lokalne. Definicije novih podatkovnih tipov pa so lahko samo na najvišjem nivoju modela in so torej vedno globalne.

Definicija 2.4 $Spremenljivka$ je peterica $(ime, območje, domena, privzeta, trenutna)$, kjer:

ime edinstveno določa spremenljivko na veljavnem območju;
$območje$ veljavnosti spremenljivke je lahko globalno ali lokalno;
$domena$ je končna množica celih števil, ki jo določa tip spremenljivke;
$privzeta$ je privzeta vrednost spremenljivke, ki je celo število iz množice $domena$;
$trenutna$ je trenutna vrednost spremenljivke, ki je prav tako celo število iz množice $domena$.♦

Lokalne spremenljivke so vidne v celotnem procesu in se ne morejo omejiti samo na posamezen izbor stavkov. Podobno velja za globalne spremenljivke, saj so vedno vidne v vseh procesih in ne v izbrani podmnožici procesov.

Vsaka spremenljivka mora biti pred uporabo deklarirana. Slika 2.6 prikazuje sintakso deklaracije spremenljivke. Promela definira devet standardnih podatkovnih tipov, ki jih prikazuje tabela 2.2.

<deklaracija> ::= [<vidnost>] <ime podatkovnega tipa> <deklaracija spremenljivke>
[, <deklaracija spremenljivke>]*

<ime podatkovnega tipa> ::= **bit** | **bool** | **byte** | **short** | **int** | **mtype** | **chan**
| <uporabniški podatkovni tip>

<deklaracija spremenljivke> ::= <ime spremenljivke>['[']<konstanta>']'] [=
<izraz> | '=' <definicija kanala>]

<vidnost> ::= **hidden** | **local** | **show**
<spremenljivka> ::= <ime spremenljivke> ['['<izraz>']'] [.<spremenljivka>]
<ime spremenljivke> ::= <ime>

Slika 2.6: EBNF zapis sintaksa deklaracije spremenljivke

Tabela 2.2: Standardni podatkovni tipi v Promeli

Ime tipa	**Zaloga vrednosti**
bit	$0, 1$
bool	$false, true$
byte	$0..255$
chan	$1..255$
mtype	$1..255$
pid	$0..255$
short	$-2^{15}..2^{15}-1$
int	$-2^{31}..2^{31}-1$
unsigned	$0..2^{n}-1$

Definicija 2.5 $Kanal$ je posebna oblika podatkovnega tipa in je trojica $(k_id, št_rež, vsebina)$, kjer je:

k_id pozitivno celo število, ki edinstveno določa kanal;
$št_rež$ celo število in določa število rež kanala;
$vsebina$ urejena množica $sporočil$ z maksimalno močjo $št_rež$.♦

Slika 2.7 prikazuje sintakso definicije kanala. Kanali so lahko definirani globalno ali lokalno. Ne glede na mesto definicije pa so vedno dostopni vsem procesom, ki poznajo k_id. Vsak kanal ima lahko končno veliko število rež, ki omogočajo hrambo prejetih signalov.

<definicija kanala> ::= '['<konstanta>']' **of** '{' <podatkovni tip> [, <podatkovni tip>]* '}'

Slika 2.7: EBNF zapis sintakse definicije kanala

Podatkovni tip $mtype$ je lahko v sistemu deklariran večkrat, vendar je število vseh elementov omejeno na 255. To je naštevni podatkovni tip, ki poljubnemu imenu priredi konstantno vrednost. Namenjen je predvsem poimenovanju signalov in je izjema v tem, da se lahko z uporabo operatorja $printn$ izpiše ime konstante na osnovi njene vrednosti. Konstantam se skladno z zaporednim mestom v deklaraciji priredijo števila med 1 in 255.

Podatkovni tip $unsigned$ se uporabi ob deklaraciji spremenljivk z zalogo vrednosti, ki jo lahko opišemo z manjšim številom bitov n, kjer je $1 <= n <= 32$. Zaradi optimalnejše

porabe pomnilnika in manjše dolžine vektorja stanj ob formalni verifikaciji se ta podatkovni tip ob avtomatski tvorbi modela pogosto uporablja.

Uporabniški podatkovni tip predstavlja v Promeli mehanizem za vpeljavo podatkovnih struktur (slika 2.8). Zapis je sestavljen iz končno velike množice elementov. Vsak element ima pridružen veljavni podatkovni tip. Pri definiciji kanala se lahko uporabijo tudi uporabniški podatkovni tipi. Ob komunikaciji se pri prenosu spremenljivke sestavljenega tipa (zapisov) obravnavajo vsi elementi v atomarni operaciji. Promela ne podpira neposrednega prirejanja spremenljivk sestavljenega tipa. Prirejajo se lahko samo posamezni končni elementi, ki ne predstavljajo zapisov.

<uporabniški podatkovni tip> ::= **typedef** <ime> '{'<seznam deklaracij>'}'
<mtype> ::= **mtype** [=]''<ime> [, <ime>]* ''

Slika 2.8: EBNF zapis sintakse definicije $mtype$ in uporabniškega podatkovnega tipa

Promela podpira samo enodimenzionalna polja. Velikost polja je določena ob deklaraciji spremenljivke z vrednostjo konstante v zapisu <deklaracija spremenljivke> na sliki 2.6. V kombinaciji z uporabniškimi podatkovnimi tipi je lahko vsak element polja tudi zapis.

Promela podpira tri dodatne oznake za optimizacijo algoritmov, ki se uporabljajo pri formalni verifikaciji. Kadar se v modelu uporabijo začasne globalne spremenljivke v okviru neprekinljivih zaporedij, ki ne vplivajo na delovanje sistema in torej ne sodijo v $sistem.gsprem$, se lahko označijo z oznako $hidden$. Tako označene spremenljivke niso sestavni del vektorja stanj sistema ob formalni verifikaciji. Drugo oznako lahko uporabimo pri sondah, ki jih namestimo v telo procesa. Sonde so globalne spremenljivke, ki jih uporabimo v specifikaciji zahtev v obliki formule LTL. Kadar ima do takšne spremenljivke izključen dostop samo en proces, ji ob deklaraciji lahko dodamo oznako $local$. Algoritmi pri formalni verifikaciji bodo obravnavali tako označeno spremenljivko enakovredno z drugimi lokalnimi spremenljivkami, kar zmanjša število dosegljivih stanj modela. Pri uporabi dodatnih oznak je potrebno biti izjemno pazljiv, saj lahko napačna uporaba vpliva na veljavnost modela. Ob verifikaciji se napačna uporaba oznak ne zazna. Tretja oznaka, $show$, nima vpliva na formalno verifikacijo in se uporablja pri globalnih ali lokalnih spremenljivkah, katerih vrednosti želimo prikazovati med simulacijo v diagramih MSC.

2.3 Komunikacija

Izmenjava podatkov med procesi se modelira z uporabo kanalov (definicija 2.5 in slika 2.7). Skupek prenašanih podatkov določa sporočilo.

Definicija 2.6 Sporočilo je urejena množica $spremenljivk$.♦

Ob definiciji kanala se izbere število rež, ki določajo kapaciteto kanala. Prejeti signali se hranijo v prostih režah. Ko so napolnjene vse reže, kanal ne more več sprejeti nobenega sporočila. Kanali, ki nimajo nobene reže, omogočajo samo sinhrono komunikacijo med dvema procesoma. Signal se ob uporabi sinhronih kanalov lahko pošlje samo v trenutku, ko se v ciljnem procesu ta lahko sprejme. Promela-stroj to interpretira kot sočasna dogodka.

Vsak signal oz. sporočilo lahko vsebuje končno mnogo elementov. Vrednosti prenašanih elementov morajo biti skladne s podatkovnimi tipi, ki so bili podani ob definiciji kanala. Posebno omejitev predstavlja definicija polja spremenljivk, saj ta ni možna. Zato je za vse takšne primere potrebno definirati nov zapis, ki ima samo en element in opisuje polje spremenljivk.

Pošiljanje signala se izvede s konstruktom pošlji (slika 2.3). Po privzetem je to mogoče samo v primeru, ko ima izbrani kanal prosto vsaj eno režo. V nasprotnem primeru stavek ni izvršljiv in proces se blokira. S posebnim parametrom ob zagonu programa Spin se lahko interpretacija spremeni tako, da je stavek izvršljiv, poslani signal pa se implicitno zavrže.

Poslani signal mora imeti enako število elementov kot ciljni kanal. Posamezen element je predstavljen z zapisom podizraza (slika 2.3). Promela definira dve obliki konstrukta pošlji. Redkeje je uporabljena druga oblika, ki jo označujeta dva znaka '!', in jo imenujemo urejeno pošiljanje. Za razliko od normalnega konstrukta pošlji, ki uvrsti signal v prvo prosto režo kanala po principu FIFO, se ob uporabi te oblike pošiljanja signal umesti v režo glede na vrednost vseh elementov sporočila. Signal se shraniti tik pred sporočilo, ki mu sledi po numerični vrednosti ob upoštevanju vseh elementov sporočila.

Ob sprejemu sporočil se pri osnovni obliki ukaza sprejme najstarejše sporočilo v kanalu. Pri sprejemu se lahko na mesta posameznih elementov zapiše spremenljivka, ovrednotena spremenljivka ali konstanta. V primeru uporabe zadnjih dveh konstruktov je izraz izvršljiv le takrat, ko je vrednost elementov sporočila enaka vrednosti spremenljivke oz. konstante. To omogoča modeliranje sprejema signalov z dodatnimi pogoji. V spremenljivke, ki so navedene na mestih posameznih elementov, se shranijo vrednosti elementov sprejetega signala. Pri naključnem sprejemu signala, ki ga označujeta dva znaka '?', je izraz izvršljiv, kadar katerokoli sporočilo v kanalu ustreza pogojem sprejema. Trenutna implementacija vrne najstarejše takšno sporočilo. Pri sprejemu se sporočilo navadno odstrani iz kanala. Kadar želimo modelirati obnašanje, ki sporočilo samo prebere in ga pusti v kanalu, je potrebno uporabiti zadnji dve obliki, ki ju definira <sprejmi> na sliki 2.3.

Trenutna vsebina kanala se lahko uporabi tudi kot stražar v stavku za nadzor poteka izvajanja. Kadar želimo samo preveriti, ali je sporočilo v kanalu skladno z določenimi zahtevami, je na voljo <preveri vrednost kanala>. Izjava je izvršljiva natanko takrat, ko bi bil izvršljiv tudi enakovreden zapis konstrukta sprejmi. Promela-stroj zagotavlja, da se vrednost uporabljenih spremenljivk ob preverjanju vrednosti kanala ne bo spremenila. Dodatno testiranje kanala se lahko izvaja z unarnimi operatorji:

- $full$ — vrne boolovo vrednost 1, če so vse reže kanala polne, sicer vrne boolovo vrednost 0,

- $nfull$ — vrne boolovo vrednost 1, če vse reže kanala še niso polne, sicer vrne boolovo vrednost 0,
- $empty$ — vrne boolovo vrednost 1, če so vse reže kanala prazne, sicer vrne boolovo vrednost 0 in
- $nempty$ — vrne boolovo vrednost 1, če niso vse reže kanala prazne, sicer vrne boolovo vrednost 0.

Dodatno optimizacijo specifikacije modela z vidika formalne verifikacije omogočata trditvi xr in xs. Prva dodeljuje procesu izključno pravico do sprejema signalov iz navedenega kanala, druga pa do pošiljanja v kanal (slika 2.2). V primeru nepravilne uporabe se neveljavne trditve med verifikacijo odkrijejo. Primerna uporaba pa lahko bistveno zmanjša število dosegljivih stanj modela. Za primer, ki je naveden v [36], se je doseglo zmanjšanje za 16 odstotkov.

S tem je zaključen zgoščen pregled jezika, ki ga bomo uporabili ob avtomatski tvorbi modelov iz specifikacij v jeziku SDL. V poglavjih, ki opisujejo modeliranje posameznih delov specifikacije, se sklicujemo na podane definicije in EBNF zapise sintakse jezika. Za podrobnejši pregled jezika priporočamo [36].

3. poglavje

Tvorba modelov iz specifikacije v jeziku SDL

V tem poglavju so predstavljeni algoritmi za avtomatsko pretvorbo specifikacije sistema v jeziku SDL v model sistema v jeziku Promela. Na področju računalništva predstavljajo jeziki notacijo, ki je uporabljena za opis računskih postopkov. Pravilom zapisa notacije pravimo sintaksa jezika, računskim postopkom, torej pomenu zapisa, pa semantika. Ob specifikaciji opisujemo delovanje sistema z uporabo konstruktov uporabljenega jezika, ki predstavljajo abstrakcijo dejanskih operacij. Osnovni cilj je torej natančna algoritmična preslikava opisa sistema iz abstraktnega zapisa v jeziku SDL v abstraktni zapis v jeziku Promela z ohranitvijo dejanskih operacij in s tem pomena specifikacije.

Pri opisu sistema nas prvenstveno zanimajo lastnosti, ki določajo njegovo delovanje. Sistemi v telekomunikacijah komunicirajo s svojim okoljem in so torej odprte narave. Vse, kar ni del sistema, se obravnava kot okolje, ki spoštuje omejitve sistema. V prejšnjem poglavju smo zapisali, da mora biti model sistema zaključena celota, zato je ob preverjanju pravilnosti delovanja potrebna tudi primerna predstavitev okolja.

Za opis specifikacij v SDL-ju obstajata dva sintaktično ekvivalentna zapisa: grafični in tekstovni. V besedilu in algoritmih se sklicujemo izključno na tekstovni zapis, ob definiciji gradnikov pa je predstavljen tudi grafični zapis.

Semantiko sistema določa abstraktni SDL-stroj s primerno interpretacijo specifikacije, ki jo sestavljajo strukturna pravila zapisa sistema, definicije podatkovnih tipov, grafi procesov in procedur ter komunikacijski podsistem.

Definicija 3.1 Specifikacija sistema $spec$ je peterica $(stru, def, kom, kon, unsup)$, kjer je:

$stru$ množica konstruktov, ki definirajo strukturo sistema;
def množica standardnih in delnih podatkovnih tipov ter drugih definicij;
kom množica konstruktov, ki definira komunikacijski podsistem;
kon množica konstruktov, ki se uporabljajo za opis grafa procesa ali procedure;
$unsup$ množica konstruktov, ki jih naloga ne obravnava. ♦

Definicija 3.2 Definicija def je peterica $(stip, ntip, ptip, sino, spre)$, kjer je:

$stip$ končna množica standardnih podatkovnih tipov;
$ntip$ končna množica definicij novih podatkovnih tipov;
$ptip$ končna množica definicij novih podintervalnih tipov;
$sino$ končna množica definicij sinonimov.♦

3.1 Pojem podatkovnega tipa v jeziku SDL

Podatkovni podsistem v opisnem jeziku SDL temelji na teoriji abstraktnih podatkovnih tipov. Na vsakem mestu v definiciji sistema je veljaven samo en abstraktni podatkovni tip. Njegove lastnosti določajo standardni podatkovni tipi in vsi strukturno višji elementi s svojimi delnimi definicijami podatkovnih tipov. Delni podatkovni tip zajema množico elementov s sorodnimi lastnostmi in pridružene operatorje (definicija 3.3).

Definicija 3.3 Definicija delnega podatkovnega tipa je trojica $(tip, operatorji, aksiomi)$, kjer je:

tip ime delnega podatkovnega tipa;
$operatorji$ končna množica operatorjev;
$aksiomi$ končna množica enačb.♦

Literali delnega podatkovnega tipa so operatorji brez argumentov. Operatorji definirajo operacije nad vsemi elementi abstraktnega podatkovnega tipa. Veljavne množice argumentov in rezultatov se navedejo v t. i. podpisu, ki operator podrobneje definira. Ob formalnem opisu so definirani z enačbami, ki opisujejo relacije med elementi. Tako so v [48] definirani vsi operatorji iz nabora standardnih podatkovnih tipov. Operator lahko definirajo enačbe ali neformalni opis. Razvojna okolja podpirajo neformalni opis z zunanjo implementacijo v obliki programskih gradnikov. V to skupino sodi večina specifikacij, kjer se jezik SDL uporablja tudi za implementacijo sistema. Naloga ne obravnava avtomatske tvorbe modela za te primere, čeprav vpeljava podpore za zunanje gradnike jezika C v Spinu nudi primerno osnovo tudi za ta izziv.

3.1.1 Standardni podatkovni tipi

Standardni podatkovni tipi predstavljajo množico vnaprej definiranih podatkovnih tipov. Podrobnejši opis in specifikacija se nahaja v [48].

Definicija 3.4 Standardni podatkovni tip $stip$, $stip \in spec.def$, je deveterica $(INTEGER, BOOLEAN, NATURAL, PId, REAL, CHARACTER, CHARSTRING, TIME,$

$DURATION$), kjer je:

$INTEGER$ množica desetiških celih števil;
$BOOLEAN$ množica logičnih vrednosti 1 in 0;
PId množica identifikacijskih številk procesov;
$NATURAL$ množica desetiških naravnih števil vključno z ničlo;
$CHARACTER$ množica znakov ASCII;
$CHARSTRING$ niz znakov;
$REAL$ množica desetiških realnih števil;
$TIME$ vrednost absolutnega časa sistema;
$DURATION$ vrednost relativnega časa sistema.♦

Standardni podatkovni tipi so veljavni v celotni specifikaciji, skladno z [57] pa se lahko nadomestijo tudi z lastnimi opisi. Takšnih primerov disertacija ne obravnava. Razvojno okolje OG sprememb standardnih tipov prav tako ne podpira. Pri tvorbi modela je podprta uporaba polj ($ARRAY$) in zapisov ($STRUCT$), ki omogočajo definicijo sestavljenih podatkovnih tipov. V nalogi se posvečamo izključno naštetim podatkovnim tipom in njihovim izpeljankam v okviru definicij delnih podatkovnih tipov.

Posebna obravnava je potrebna povsod, kjer nimamo enoumne preslikave v podatkovne tipe od Promele. Najenostavnejša je preslikava podatkovnih tipov $INTEGER$, $BOOLEAN$ in PId. Nekaj več težav je s primerno predstavitvijo tipa $NATURAL$, saj se predstavitev z uporabo tipa $unsigned$ ne sme uporabiti pri komunikaciji. Podatkovna tipa $CHARACTER$ in $CHARSTRING$ ter pridružene operatorje pa je potrebno v celoti modelirati v Promeli. Za vsak znak iz SDL-definicije podatkovnega tipa $CHARACTER$ je izvedena preslikava z uporabo makroja. Zaradi nedvoumne preslikave je bilo za terminalne simbole Promele potrebno uporabiti dodatno poimenovanje (tabela 3.1), ki sledi priporočilom v [44].

Slika 3.1 prikazuje predstavitev podatkovnih tipov $CHARACTER$ in $CHARSTRING$. Prvi se modelira z uporabo zapisa $pt_character$. Zapis ima samo en element tipa $unsigned$, katerega zaloga vrednosti se lahko zapiše s sedmimi biti in ima privzeto vrednost 0. Ob sklicevanju na spremenljivke tega tipa je tako potrebno imenu spremenljivke vedno dodati še pripono $.char$. Za takšno predstavitev smo se odločili zato, ker se ob definiciji kanala ne sme neposredno uporabiti podatkovni tip $unsigned$. Alternativni pristop, ki podpira tudi komunikacijo, je neposredna uporaba podatkovnega tipa $byte$ ob deklaraciji spremenljivk in definiciji kanalov. V tem primeru zapis ni pomnilniško optimalen, vektor stanj pa je po nepotrebnem daljši. Dodatna motivacija za predlagano rešitev izvira iz transparentne preslikave spremenljivk v model sistema in podobnosti s predstavitvijo podatkovnega tipa $CHARSTRING$, ki je polje elementov tipa $CHARACTER$. Ob uporabi orodij $sdl2if$ in $if2pml$ se podatkovni tip $CHARACTER$ preslika v int, predstavitev posameznih znakov pa se izvaja neposredno s številčno preslikavo. $CHARSTRING$ v omenjenih orodjih ni podprt.

V definiciji podatkovnega tipa $pt__charstring$ je edini element zapisa polje elementov podatkovnega tipa $byte$ (slika 3.1). Ker Promela ne podpira definicij polj z elementi podatkovnega tipa $unsigned$, smo za predstavitev vsakega znaka porabili dodaten bit v vektorju stanj. Model mora opisovati končen sistem, zato je največje podprto število elementov polja 32. S primerno definicijo konstante $pcv__charstring_max$ se lahko ta vrednost spremeni skladno s potrebami modela. Generator C-kode v OG to mejo postavlja na 510 znakov [53]. Pri definiciji smo upoštevali, da se v jeziku SDL prične indeks z 1, pri Promeli pa vedno z 0. To nam omogoča enostavnejšo in preglednejšo preslikavo specifikacije. Do posameznega elementa v spremenljivki dostopamo tako, da se imenu doda $.char[odmik]$, kjer $odmik$ predstavlja zaporedno število znaka v polju. Primer uporabe predstavlja model operatorja $strcpy()$, ki je prikazan na sliki 3.1.

```
typedef pt__character {unsigned char : 7 = 0}

#define pcv__charstring_max 33

typedef pt__charstring {byte char[pcv__charstring_max]}

inline strcpy(pfv__stringB,pfv__stringA){
   d_step{
      pfv__tmp=1;
      do
         :: ((pfv__tmp<=pcv__charstring_max) && pfv__stringA.char[pfv__tmp]!=NUL) ->
                  pfv__stringB.char[pfv__tmp]=pfv__stringA.char[pfv__tmp];pfv__tmp++;
         :: else -> break
      od
   }
}
```

Slika 3.1: Modeliranje podatkovnih tipov $character$ in $charstring$

Operatorji so modelirani z uporabo konstrukta $inline$ in so nedvoumni ter vedno izvršljivi. Atomarnost operacij se zagotovi z uporabo konstrukta d_step in primernim telesom operatorja, ki ne sme v nobenem primeru blokirati izvajanja procesa. Celotna specifikacija je predstavljena v prilogi A.

Promela ne podpira realnih števil. Pri pretvorbi se ponuja več potencialnih rešitev. Najkompleksnejša rešitev, ki bi omogočila podporo racionalnim številom, zahteva obravnavo realnih števil s predstavitvijo v obliki para celih števil ter primernim modeliranjem vseh operatorjev. Ta rešitev nudi celovito preslikavo, saj ne omejuje specifikacije sistema v nobenem elementu, vendar bistveno prispeva h kompleksnosti modela in verifikacijskega postopka. Alternativni pristop vključuje uporabo zunanjih gradnikov jezika C, vendar smo v tem primeru omejeni pri komunikaciji, saj ni mogoče definirati primernih kanalov za prenos realnih števil. Ob komunikaciji bi torej bilo potrebno izvajati zaokroževanje na cela števila. Z uporabo zunanjih gradnikov bi se odrekli simulaciji z orodjem Spin.

Tabela 3.1: Definicije preslikav literalov SDL v Promelo

```
#define NUL                 0
#define SOH                 1
#define STX                 2
#define ETX                 3
#define EOT                 4
#define ENQ                 5
#define ACK                 6
#define BEL                 7
#define BS                  8
#define HT                  9
#define LF                  10
#define VT                  11
#define FF                  12
#define CR                  13
#define SO                  14
#define SI                  15
#define DLE                 16
#define DC1                 17
#define DC2                 18
#define DC3                 19
#define DC4                 20
#define NAK                 21
#define SYN                 22
#define ETB                 23
#define CAN                 24
#define EM                  25
#define SUB                 26
#define ESC                 27
#define IS4                 28
#define IS3                 29
#define IS2                 30
#define IS1                 31
#define SPACE               32
#define EXCLAMATION_MARK    33
#define QUOTATION_MARK      34
#define HASH                35
#define CURRENCY            36
#define PERCENT_SIGN        37
#define AMPERSAND           38
#define ACUTE_ACCENT        39
#define LEFT_PARENTHESIS    40
#define RIGHT_PARENTHESIS   41
#define ASTERISK            42
#define PLUS                43
#define COMMA               44
#define MINUS               45
#define FULL_STOP           46
#define SOLIDUS             47
#define ZERO                48
#define ONE                 49
#define TWO                 50
#define THREE               51
#define FOUR                52
#define FIVE                53
#define SIX                 54
#define SEVEN               55
#define EIGHT               56
#define NINE                57
#define COLON               58
#define SEMICOLON           59
#define LESS_THAN           60
#define EQUALS_SIGN         61
#define GREATER_THAN        62
#define QUESTION_MARK       63
#define COMMERCIAL_AT       64
#define CHAR_A              65
#define CHAR_B              66
#define CHAR_C              67
#define CHAR_D              68
#define CHAR_E              69
#define CHAR_F              70
#define CHAR_G              71
#define CHAR_H              72
#define CHAR_I              73
#define CHAR_J              74
#define CHAR_K              75
#define CHAR_L              76
#define CHAR_M              77
#define CHAR_N              78
#define CHAR_O              79
#define CHAR_P              80
#define CHAR_Q              81
#define CHAR_R              82
#define CHAR_S              83
#define CHAR_T              84
#define CHAR_U              85
#define CHAR_V              86
#define CHAR_W              87
#define CHAR_X              88
#define CHAR_Y              89
#define CHAR_Z              90
#define LEFT_SQUARE_BRACKET 91
#define REVERSE_SOLIDUS     92
#define RIGHT_SQUARE_BRACKET 93
#define CIRCUMFLEX_ACCENT   94
#define LOW_LINE            95
#define GRAVE_ACCENT        96
#define CHAR_a              97
#define CHAR_b              98
#define CHAR_c              99
#define CHAR_d              100
#define CHAR_e              101
#define CHAR_f              102
#define CHAR_g              103
#define CHAR_h              104
#define CHAR_i              105
#define CHAR_j              106
#define CHAR_k              107
#define CHAR_l              108
#define CHAR_m              109
#define CHAR_n              110
#define CHAR_o              111
#define CHAR_p              112
#define CHAR_q              113
#define CHAR_r              114
#define CHAR_s              115
#define CHAR_t              116
#define CHAR_u              117
#define CHAR_v              118
#define CHAR_w              119
#define CHAR_x              120
#define CHAR_y              121
#define CHAR_z              122
#define LEFT_CURLY_BRACKET  123
#define VERTICAL_LINE       124
#define RIGHT_CURLY_BRACKET 125
#define TILDE               126
#define DEL                 127
```

Izločitev podpore za realna števila je namerna, saj je Promela namenjena modeliranju sistemov. Pri zapisu modela sistema se vzpodbuja abstrakcija, ki ohranja ključne lastnosti sistema in zanemarja računske podrobnosti. Ker je ob formalni verifikaciji sočasnih sistemov poudarek na preverjanju pravilnosti interakcije, koordinacije in sinhronizacije med procesi in ne na računskih operacijah, smo se odločili, da vsa realna števila predstavimo s celimi števili. Kot primer uporabe realnih števil lahko navedemo odločitveni stavek, ki se o nadaljevanju izvajanja odloča na osnovi izračuna korena števila. Model, ki bi preverjal pravilnost vseh možnih izvajanj z izračunom korena vseh potencialnih vrednosti spremenljivke, se lahko nadomesti z enostavno odločitvijo, ki se nedeterministično odloča med možnimi nadaljevanji izvajanja [36].

Podatkovna tipa $TIME$ in $DURATION$ sta uporabljena pri spremenljivkah, ki se nanašajo na absolutni oz. relativni čas izvajanja sistema in predstavljajo množico realnih števil. Časovno enoto v računalniških sistemih vedno določa konkretna implementacija sistema in se lahko predstavi s celimi števili. Promela podpira predstavitev časa z uporabo spremenljivk tipa $short$, zato smo takšno predstavitev prevzeli tudi pri tvorbi modela.

Ob statični analizi specifikacije sistema in tvorbi modela se podatkovni tipi, ki so v SDL-ju predstavljeni z realnimi števili, obravnavajo individualno. Med rezultati analize so tako na voljo vsi podatki o izvornem podatkovnem tipu in njegovi predstavitvi v modelu. Takšen pristop omogoča selektivne dopolnitve algoritmov povsod, kjer se je uporabila abstrakcija.

Poseben primer podatkovnega tipa SDL predstavljata $ARRAY$ in $STRUCT$. Prvi je predstavnik generatorjev [57, 48], drugi pa nudi osnovo za predstavitev sestavljenih tipov. Oba sta podrobneje predstavljena v razdelku 3.1.2.

3.1.2 Definicija podatkovnega tipa

Slika 3.2 prikazuje EBNF zapis sintakse delne definicije uporabniškega podatkovnega tipa. Celotno definicijo abstraktnega podatkovnega tipa v določenem mestu v strukturi sistema določa množica vseh delnih definicij podatkovnih tipov v območju veljavnosti. Imena podatkovnih tipov se lahko ponavljajo, ob deklaraciji spremenljivke pa je vedno veljavna tista definicija, ki je del abstraktnega podatkovnega tipa na mestu deklaracije.

V okviru konstrukta <novi tip> se definirajo zapisi, podatkovni generatorji in naštevni podatkovni tipi. Zapis omogoča združevanje poljubnih, lahko tudi že sestavljenih tipov v nov sestavljeni tip. Generatorji v jeziku SDL opisujejo parametrizirane delne definicije podatkovnih tipov. Primer podatkovnega generatorja je polje, kjer ob definiciji določimo dejanski podatkovni tip indeksa in elementov polja. Z naštevnim tipom vpeljemo nov enostaven nerazčlenjen podatkovni tip, tako da naštejemo množico vseh vrednosti, ki mu pripadajo. Takšna definicija ne vpelje samo novega podatkovnega tipa, temveč tudi množico imen za pripadajoče vrednosti, ki se lahko v specifikaciji uporabljajo kot konstante.

Sledi opis algoritma analize in pretvorbe definicij z uporabo konstrukta <novi tip>. Vse algoritme v nadaljevanju sestavljajo:

<uporabniški podatkovni tip> := <novi tip> | <podintervalni tip>

<novi tip> ::= **newtype** <ime podatkovnega tipa>
[<razširjene lastnosti>] [<privzeta vrednost>] [<seznam literalov>]
[**constants** <zaloga vrednosti>]
endnewtype [<ime podatkovnega tipa>];

<podintervalni tip> ::= **syntype** <ime podatkovnega tipa> = <ime nadtipa>
[<privzeta vrednost>] [**constants** <zaloga vrednosti>]
endsyntype [<ime podatkovnega tipa>];

<standardni tip> ::= $INTEGER$ | $BOOLEAN$ | $NATURAL$ | PId | $REAL$ | $CHARACTER$ |
$CHARSTRING$ | $TIME$ | $DURATION$ | $ARRAY$

<razširjene lastnosti> ::= <generator> | <definicija strukture>
<definicija strukture> ::= **struct** <seznam elementov>
<seznam elementov> ::= <ime elementa>{, <ime elementa>}* <podatkovni tip>;
<podatkovni tip> ::= <standardni tip> | <uporabniški podatkovni tip>
<seznam literalov> ::= **literals** <literal>{,<literal>}*
<literal> ::= <ime literala> | <razred imen>
<razred imen> ::= **nameclass** <regularni izraz>
<privzeta vrednost> ::= **default** <izraz>

Slika 3.2: EBNF zapis sintakse definicije podatkovnega tipa

- n-terice, ki predstavljajo rezultate analize,
- funkcije, ki omogočajo kompaktno predstavitev delov algoritma,
- konstrukti iz EBNF opisa sintakse specifikacije, ki so vedno zapisani v koničastih oklepajih in
- izpisi v navednicah, ki tvorijo model specifikacije v Promeli.

Zapis podatkov v primerni obliki za kasnejšo uporabo je predstavljen s funkcijo $shrani()$. V opisu algoritmov se bodo funkcije uporabljale na mestih, kjer podrobnejši opisi postopkov ne prispevajo k razumevanju algoritma.

Definicija 3.5 Analizo konstrukta <novi tip> opisujemo z $ntip$, $ntip \in spec.def$, in je trojica $(sdlime, pmlime, pmltip)$, kjer je:

$sdlime$ ime novega podatkovnega tipa v specifikaciji sistema;
$pmlime$ ime novega podatkovnega tipa v modelu sistema;
$pmltip\ zapis$, $polje$ ali naštevni podatkovni tip (npt).♦

Ob statični analizi specifikacije sistema shranimo vse potrebne informacije za primerno definicijo pasivnih elementov in modeliranje aktivnih elementov sistema (definicija 3.5). Primer pasivnega elementa je definicija novega uporabniškega tipa, aktivni elementi sistema pa so tisti, ki vplivajo na globalno stanje sistema (npr. spremenljivke). Ob analizi se hranijo morebitni komentarji iz specifikacije samo tam, kjer je to eksplicitno navedeno, drugače se v modelu ne predstavijo.

Promela določa samo globalno in lokalno območje veljavnosti, zato je za primerno podporo večkratnih definicij z istim imenom pri analizi potrebno shraniti mesto definicije v strukturi sistema. To urejamo s primernim poimenovanjem, ki odraža absolutno pot do mesta definicije. V algoritmih pridobimo trenutno mesto znotraj hierarhične strukture z uporabo funkcije $pot()$. Izvorno ime, ki je uporabljeno v specifikaciji sistema, se shrani v $ntip.sdlime$. Razširjeno ime, na katero se sklicujemo v modelu sistema, pa v algoritmu 3.1 sestavimo z ukazom $pot()//ntip.sdlime$. V [48] je dvojna poševnica definirana kot operator za spajanje elementov tipa $charstring$. Zadnji element zapisa, $pmltip$, hrani vse podatke za primerno modeliranje definiranega delnega podatkovnega tipa.

Algoritem 3.1 prikazuje analizo in pretvorbo definicij v okviru konstrukta <novi tip>. V prvem odločitvenem stavku se sprašujemo, ali v specifikaciji sistema še obstaja kakšen neobdelan konstrukt <novi tip>. V primeru, da smo vse konstrukte že obdelali, se izvajanje algoritma zaključi z uporabo rezervirane besede END. V nasprotnem primeru se preverja, ali definicija vsebuje katerega izmed konstruktov, ki ob pretvorbi niso podprti. Tudi v tem primeru se izvajanje algoritma zaključi, vendar z rezervirano besedo STOP, ki nakazuje neregularni zaključek. Orodje $sdl2pml$ v takšnih primerih vseeno poskuša tvoriti model sistema, uporabnika pa primerno opozori na morebitne pomanjkljivosti.

V nadaljevanju obravnavamo definicijo zapisa, polja ali naštevnega tipa. Zapis ima končno število elementov. Vsakemu elementu zapisa se pridruži ime in podatkovni tip, kar določa definicija 3.6.

Definicija 3.6 Sestavljen podatkovni tip $zapis$ je dvojica (ime, tip), kjer je:
ime ime elementa;
tip ime podatkovnega tipa elementa.♦

V prvem koraku pričnemo z definicijo tipa v modelu sistema. Sledi odločitveni stavek pri oznaki R2. V zanki obravnavamo preostale elemente zapisa in shanimo vse potrebne podatke. V zadnjem koraku te veje algoritma zaključimo definicijo podatkovnega tipa v modelu sistema in shranimo ime podatkovnega tipa. Sledi skok na oznako R1, kjer nadaljujemo z analizo specifikacije. Primer pretvorbe preprostega zapisa prikazuje slika 3.3.

```
NEWTYPE zapis
  STRUCT
    ime_elementa1   tip1;
    ime_elementa2   tip2;
ENDNEWTYPE zapis;

typedef pot()//zapis {
  tip1 ime_elementa1;
  tip2 ime_elementa2
};
```

Slika 3.3: Primer pretvorbe enostavnega sestavljenega tipa

Ne glede na mesto deklaracije spremenljivke sestavljenega tipa podatkovne tipe elementov določa mesto definicije zapisa. Tako lahko pride do primerov, ko se elementu zapisa

določi podatkovni tip, ki ni del abstraktnega podatkovnega tipa na mestu deklaracije spremenljivke, temveč je del abstraktnega podatkovnega tipa na mestu definicije sestavljenega podatkovnega tipa.

EBNF zapis dopušča dodatne omejitve zaloge vrednosti tudi pri definiciji zapisov, definicija konstrukta <zaloga vrednosti> pa ne predvideva konstruktov, ki bi omogočali obravnavo posameznih elementov sestavljenega tipa. Zato v primeru zapisov ne hranimo podatka o predvideni zalogi vrednosti.

```
START ;
R1:
DECISION ∃a: a ∈ spec.def.ntip ;
 ( TRUE ):
 ELSE:
  END ;
ENDDECISION;
DECISION ∃j : j ∈ {inherits, adding, map,
                   operators, axioms};
 ( TRUE ):
  STOP;
 ELSE:
ENDDECISION;
TASK a.pmlime := pot()//a.sdlime;
DECISION a.pmltip;
( zapis ):
 TASK "typedef //a.pmlime// {";
R2:
 DECISION ∃k : k ∈ a.zapis;
  ( TRUE ):
   TASK "//k.tip// //k.ime//[;]"
   TASK shrani(k); JOIN R2;
  ELSE:
 ENDDECISION;
 TASK "};"
 TASK shrani(a.sdlime, a.pmlime);
( polje ):
 TASK shrani(a.sdlime, a.pmlime);
 TASK shrani(a.polje.tip);
 TASK indeks(a.polje.ime);
 TASK shrani(a.polje.šte);
 TASK "typedef //a.polje.pmlime//
       {//a.polje.tip// val[//a.polje.šte//]};";
( npt ):
 DECISION ∃ <razred imen>;
  ( TRUE ):
   TASK a.npt.zal = tvori(<regularni izraz>);
  ELSE:
 ENDDECISION;
 TASK m:=0;
R3:
 DECISION m < št. literalov;
  ( TRUE ):
   TASK "#define //a.pmlime//_//<ime literala>// m"
   TASK m:=m+1; JOIN R3;
  ELSE:
 ENDDECISION;
 TASK a.npt.št := zaokroži(log(m)/log(2));
 TASK "typedef //a.pmlime// {unsigned
       val://a.npt.št//}";
 DECISION a.npt.št < 8;
  ( TRUE ):
   TASK "typedef comm_//a.pmlime// byte val";
  ELSE:
   DECISION a.npt.št < 16;
    ( TRUE ):
     TASK "typedef comm_//a.pmlime// short val";
    ELSE:
     TASK "typedef comm_//a.pmlime// int val";
   ENDDECISION;
 ENDDECISION;
 TASK a.npt.zal := m;
 TASK shrani(a.npt.zal);
 DECISION ∃ <privzeta vrednost>;
  ( TRUE ):
   CALL 3.2(a.npt.pri,<privzeta vrednost>); // (str. 30)
   TASK shrani(a.npt.pri);
  ELSE:
 ENDDECISION;
 TASK shrani(a.sdlime, a.pmlime);
ELSE:
 STOP
ENDDECISION;
TASK "hidden //a.pmlime// undefined_//a.pmlime//;";
JOIN R1;
END;
```

Algoritem 3.1: Algoritem analize in pretvorbe definicij novih podatkovnih tipov

Polje je homogena struktura, ki jo tvorijo elementi enega samega tipa (definicija 3.7). Posamezni element izberemo tako, da imenu polja dodamo indeks. Indeks pripada tipu, ki je bil definiran kot indeksni tip polja. SDL ne omejuje indeksnih tipov, kar predstavlja oviro pri avtomatski tvorbi izvorne kode in modeliranju specifikacije. Generator C-kode v OG

omejuje indeksne tipe na podintervalne tipe naravnih in celih števil ter naštevne tipe. Zaradi učinkovitejše tvorbe modela in prilagoditve Promeli smo tudi sami omejili množico veljavnih indeksnih tipov na tiste, katerih zalogo vrednosti določa podmnožica naravnih števil vključno z ničlo. Kljub omejitvi množica potencialnih indeksnih tipov nikakor ni majhna. S klicem funkcije $indeks()$ se določi dejanska zaloga vrednosti na osnovi predhodne analize podintervalnih in naštevnih tipov. Orodje $if2pml$ ne podpira polj.

Slika 3.4 prikazuje primer pretvorbe polja. Ker polja v Promeli niso podprta pri definiciji komunikacijskih kanalov, jih modeliramo z uporabo zapisa z enim samim elementom, ki ga označimo z val. Posledično je potrebno primerno obravnavati tudi vse operacije nad spremenljivkami teh tipov. Tako je potrebno v primeru prirejanja $ime_polja(indeks) := vrednost$, ki se pojavlja v specifikaciji SDL, v modelu stavek predstaviti z $ime_polja.val[indeks] = vrednost$. Pri komunikaciji se v modelu vedno sklicujemo samo na ime zapisa, ki je v tem primeru enak imenu polja. V primeru polja ne hranimo podatka o predvideni zalogi vrednosti, saj jo posredno določa $polje.tip$.

Definicija 3.7 Polje $polje$ je trojica $(ind, šte, tip)$, kjer je:
ind indeksni podatkovni tip polja;
$šte$ število elementov polja;
tip ime podatkovnega tipa elementov polja.♦

```
NEWTYPE polje
  ARRAY (ind, tip)
ENDNEWTYPE polje;

typedef pot()//polje {tip val[polje.ste]};
```

Slika 3.4: Primer pretvorbe polja

Pri razvojnem okolju OG smo opazili posebnost pri implementaciji dodeljevanja privzetih vrednosti elementom zapisov in polj. Neposredne definicije standard ne predvideva, orodje OG pa zanemari tudi posredne definicije, kadar so za $zapis.tip$ in $polje.tip$ uporabljeni podatkovni tipi s privzeto vrednostjo. To je še posebno občutljivo področje, saj Promela-stroj po privzetem vse spremenljivke inicializira na vrednost nič, medtem ko so v SDL-stroju do prve prireditve v množici nedefiniranih spremenljivk. Skladno z [57] postane obnašanje sistema v primeru, ko se v izrazu uporabi nedefinirana spremenljivka, nepredvidljivo in torej predstavlja napako v izvajanju. Algoritem podpira obe interpretaciji, orodje $sdl2pml$ pa je po privzetem skladno z omejeno različico. Orodji $sdl2if$ in $if2pml$ ob tvorbi modela ne upoštevata predpisanih privzetih vrednosti ob definiciji delnega podatkovnega tipa.

V mnogih specifikacijah se uporabljajo spremenljivke z veliko zalogo vrednosti (npr. naravna števila) tudi v primerih, ko obravnavamo izbiro med majhnim številom elementov. Za takšne primere je modro uporabiti naštevne tipe, ki jih definiramo tako, da opišemo množico vrednosti neposredno z naštevanjem ali uporabo regularnega izraza (definicija 3.8).

To omogoča tudi izredno učinkovito implementacijo in modeliranje sistema. V industrijskih projektih se naštevni tipi uporabljajo tudi kot ogrodje za definicijo neformalno zapisanih operatorjev, ki jih ta naloga ne obravnava. Algoritem 3.1 zato v primeru definicije operatorjev neregularno zaključi izvajanje.

V prvem koraku prehoda, ki obravnava naštevne tipe, preverimo, ali v definiciji obstaja konstrukt <razred imen> (slika 3.2), ki omogoča definicijo literalov z uporabo regularnega izraza. Pri zapisu vseh predvidenih vrednosti nam pomaga funkcija $tvori()$. Generator C-kode v OG ne podpira takšne definicije in pričakuje eksplicitno poimenovanje vseh literalov. Naštevni tip je po definiciji urejen. Literali predstavljajo sinonime za naravna števila. Vrednost narašča od leve proti desni. Zaradi želje po čim večji preglednosti tvorjenih modelov in s tem hitrejšega odkrivanja problematičnih delov specifikacije smo literale v modelu sistema predstavili z enakimi imeni, kot so jim bila dodeljena v specifikaciji. Alternativo predstavlja neposredna predstavitev z naravnimi števili, ki jih predstavljajo, vendar je sledenje izvajanju modela v tem primeru težavnejše. Z zanko, ki sledi oznaki R3, definiramo množico imen literalov ter pripadajoče vrednosti. Zaradi nedvoumne predstavitve posameznega literala se pred ime doda $pmlime$. S tem tudi zadostimo [57], ki omogoča, da se v imenih literalov pojavljajo tudi števila, ki drugače sodijo med rezervirane znake Promela-stroja.

Definicija 3.8 Naštevni tip npt je trojica $(št, pri, zal)$, kjer je:

$št$ število potrebnih bitov za predstavitev naštevnega tipa;
pri privzeta vrednost;
zal zaloga vrednosti podatkovnega tipa.♦

Zaradi omejene zaloge vrednosti naštevne tipe v modelu predstavljamo z $unsigned$ in predvidenim številom bitov za opis vseh vrednosti. Naštevni tipi omogočajo izredno učinkovito uporabo pomnilnika, zato je njihova uporaba priporočljiva, težava pa se pojavi pri komunikaciji, saj pri definiciji elementov kanala ni podprt podatkovni tip $unsigned$. To rešujemo z definicijo dodatnega podatkovnega tipa, ki se uporabi ob definiciji kanalov. Ker se neomejen $unsigned$ opisuje v Promeli z 32 biti, se v naslednjih korakih v algoritmu določi nadomestni standardni tip, ki omogoča najoptimalnejšo predstavitev pri komunikaciji. Funkcija $zaokroži()$ vrne navzgor zaokroženo število potrebnih bitov za predstavitev vseh literalov.

Pri neposredni uporabi omejenega tipa $unsigned$ ob deklaraciji spremenljivk bi v modelu sistema izgubili informacijo o izvornem tipu iz specifikacije sistema. Zato tudi v tem primeru, podobno kot pri poljih, uporabimo za ovojnico podatkovnega tipa zapis z enim elementom. Primer pretvorbe prikazuje slika 3.6. Pri prirejanju veljajo enake zahteve kot pri poljih. Definicija omogoča tudi določitev privzete vrednosti, kar preverja zadnji odločitveni stavek v algoritmu. Če obstaja v definiciji konstrukt <privzeta vrednost>, se skladno z algoritmom 3.2 pretvori v veljaven izraz Promele, ki predstavlja začetno vrednost ob deklaraciji spremenljivke. Prvi formalni parameter hrani rezultat pretvorbe, z drugim pa podamo izraz v jeziku SDL.

```
FPAR IN/OUT x ∈ <izraz promela>;
FPAR y ∈ <izraz>;
START ;
  DECISION ∃b : (b ∈ y) ∧
        (b ∉<literal> ∨ <sinonim>);
    ELSE:
      TASK x:= preimenuj(b);
    ( TRUE )
      R1:
      DECISION ∃i : i = (<literal> ∨ <sinonim>
                         ∨ <del "else">);
        ( TRUE ):
          TASK x := x//preimenuj(i); JOIN R1;
        ELSE:
      ENDDECISION;
      DECISION ∃j : j ∈<operator>;
        ( OR ):
          TASK x := x//||; JOIN R1;
        ( XOR ):
          TASK x := x//^; JOIN R1;
        ( AND ):
          TASK x := x//&&; JOIN R1;
        ( = ):
          TASK x := x//==; JOIN R1;
        ( / = ):
          TASK x := x//!=; JOIN R1;
        ( > ):
          TASK x := x//>; JOIN R1;
        ( >= ):
          TASK x := x//>=; JOIN R1;
        ( < ):
          TASK x := x//<; JOIN R1;
        ( <= ):
          TASK x := x//<=; JOIN R1;
        ( in ):
          TASK j ∈ unsup; STOP;
        ( + ):
          TASK x := x//+; JOIN R1;
        ( − ):
          TASK x := x//−; JOIN R1;
        ( ∗ ):
          TASK x := x//∗; JOIN R1;
        ( / ):
          TASK x := x///; JOIN R1;
        ( not ):
          TASK x := x//!; JOIN R1;
        ELSE:
          STOP;
      ENDDECISION;
  ENDDECISION;
END;
```

Algoritem 3.2: Algoritem za pretvorbo izrazov

V prvem odločitvenem stavku preverimo, ali je začetna vrednost podana v obliki literala podatkovnega tipa ali sinonima. V tem primeru funkcija $preimenuj()$ zagotovi primerno preslikavo imena glede na mesto v strukturi sistema. Pri uporabi sinonima je potrebno poiskati tistega, ki velja na mestu uporabe, saj se lahko v specifikaciji pojavlja več definicij z enakimi imeni. V nasprotnem primeru je bila privzeta vrednost podana v obliki izraza (slika 3.5).

Standard [57] eksplicitno predpisuje, da uporabljen izraz ne sme vsebovati aktivnih primitivov, ki predstavljajo dostop do spremenljivk, zato se lahko vsi uporabljeni izrazi ovrednotijo že med statično analizo. Zaradi pristnejše preslikave in lažjega sledenja izvajanja izraza ne ovrednotimo med analizo, temveč ga pretvorimo v Promelo. V prvem koraku preimenujemo vse literale in sinonime, v nadaljevanju pa sledi pretvorba operatorjev. Algoritem 3.2 opisuje splošno pretvorbo izraza, zato vsebuje tudi operatorje, ki se v definiciji <pridružena vrednost> ne morejo uporabiti (npr. $>$). Nekateri operatorji se lahko pretvorijo neposredno, druge pa je bilo potrebno primerno modelirati v Promeli z uporabo definicij $inline$.

Zaradi razlik pri obravnavi sprejema signalov z manjkajočimi elementi pri SDL- in Promela-stroju se za vsak delni podatkovni tip deklarira skrita globalna spremenljivka, ki nam omogoča primerno modeliranje izgube signala. Podrobnejši opis rešitve je podan v razdelku o komunikaciji (3.2). Orodji $sdl2if$ in $if2pml$ ne podpirata definicije privzete

<izraz> ::= <podizraz> | <klic procedure, ki vrne vrednost>
<podizraz> ::= <operand0> | <podizraz> => <operand0>
<operand0> ::= <operand1> | <operand0> { **or** | **xor** } <operand1>
<operand1> ::= <operand2> | <operand1> **and** <operand2>
<operand2> ::= <operand3> | <operand2> { = | /= | > | >= | < | <= | in } <operand3>
<operand3> ::= <operand4> | <operand3> { + | - | // } <operand4>
<operand4> ::= <operand5> | <operand4> { * | / | **mod** | **rem** } <operand5>
<operand5> ::= [- | **not**] <primitiv>
<primitiv> ::= <izhodiščni primitiv> | <aktivni primitiv> | <razširjen primitiv>

<izhodiščni primitiv> ::= <ime literala> | <identifikator operatorja> (<izhodiščni seznam izrazov>)
| (<izhodiščni izraz>) | <pogojni izhodiščni izraz>

<razširjen primitiv> ::= <sinonim> | <indeksiran primitiv> | <primitiv s polji> | <primitiv s strukturo>
<izhodiščni seznam izrazov> ::= <izhodiščni izraz> { , <izhodiščni izraz> }*
<identifikator operatorja> ::= <identifikator operatorja> | [<kvalifikator>] <citiran operator>
<operator> ::= OR | XOR | AND | = | / = | > | >= | < | <= | in | + | − | // | ∗ | / | mod | rem | !

Slika 3.5: EBNF zapis sintakse izraza

vrednosti, pri definiciji komunikacije pa se vedno uporabi int. S tem je analiza in pretvorba naštevnega podatkovnega tipa zaključena, izvajanje pa se nadaljuje pri oznaki R1.

```
NEWTYPE naštevni_tip
  LITERALS lit1, lit2, lit3, lit4
ENDNEWTYPE naštevni_tip;

#define pot()//naštevni_tip//__lit1 0
#define pot()//naštevni_tip//__lit2 1
#define pot()//naštevni_tip//__lit3 2
#define pot()//naštevni_tip//__lit4 3
typedef pot()//naštevni_tip {unsigned val :2}
typedef comm__pot()//naštevni_tip {byte val}
};
```

Slika 3.6: Primer pretvorbe naštevnega podatkovnega tipa

Posebno obliko definicije delnega podatkovnega tipa predstavlja konstrukt <podintervalni tip>, katerega zaloga vrednosti je podmnožica enega izmed obstoječih podatkovnih tipov. Obravnavamo ga v algoritmu 3.3, definicija 3.9 pa določa množico elementov, ki ga v celoti opisuje.

Definicija 3.9 Analizo konstrukta <podintervalni tip> opisujemo s $ptip$, $ptip \in spec.def$, in je sedmerica $(sdlime, pmlime, sdlnad, pmltip, št, pri, zal)$, kjer je:

$sdlime$ ime delnega podatkovnega tipa v specifikaciji sistema;
$pmlime$ ime delnega podatkovnega tipa v modelu sistema;
$sdlnad$ ime nadtipa v specifikaciji sistema;
$pmltip$ ime standardnega podatkovnega tipa v modelu sistema;
$št$ število potrebnih bitov za predstavitev podintervalnega tipa;
pri privzeta vrednost;
zal zaloga vrednosti.♦

```
START ;
R1:
 DECISION ∃a: a ∈ spec.def.ptip ;
  ( TRUE ):
  ELSE:
    END ;
 ENDDECISION;
 TASK a.pmlime := pot()//a.sdlime;
 TASK iskani_tip := <ime nadtipa>;
 TASK a.sdlnad := NONE;
 TASK a.št := NONE;
 DECISION ∃ <privzeta vrednost>;
  ( TRUE ):
    CALL 3.2(a.pri, <privzeta vrednost>); // (str. 30)
    ELSE:
      TASK a.pri := NONE;
  ENDDECISION;
  DECISION ∃ <zaloga vrednosti>;
   ( TRUE ):
     CALL 3.2(a.zal, <zaloga vrednosti>); // (str. 30)
     ELSE:
       TASK a.zal := NONE;
   ENDDECISION;
  R2:
   DECISION iskani_tip;
    ( INTEGER,NATURAL,REAL,TIME,DURATION ):
      TASK preveri(a.sdlnad, iskani_tip);
      CALL 3.4(a,iskani_tip); // (str. 34)
    ( BOOLEAN ):
      TASK preveri(a.sdlnad, iskani_tip);
      TASK a.pmltip := bool;
      TASK "#define a.pmlime bool";
    ( CHARACTER ):
      TASK preveri(a.sdlnad, iskani_tip);
      DECISION ∃i : i =<zaloga vrednosti>;
        ( TRUE ):
        ELSE: JOIN R3;
      ENDDECISION;
      DECISION maksimum(i)<64;
        ( TRUE ):
          TASK a.št := zaokroži(log(maksimum(i)+1)/log(2))
          TASK "typedef a.pmlime {unsigned val :a.št}";
          TASK a.pmltip := unsigned;
        ELSE:
        R3:
          TASK a.pmltip := pt_character;
          TASK "#define a.pmltip pt_character";
      ENDDECISION;
    ( CHARSTRING ):
      TASK preveri(a.sdlnad, iskani_tip);
      TASK "#define a.pmlime pt_charstring";
      DECISION ∃ <zaloga vrednosti>;
        ( TRUE ):
          STOP;
        ELSE:
      ENDDECISION;
      TASK a.pmltip := pt_charstring;
 ELSE:
  TASK iskani_tip := <ime nadtipa>;
  TASK a.nadtip := iskani_tip;
  TASK nadaljuj := TRUE;
 R4:
  DECISION ∃b : b = iskani_tip,
                b ∈ (spec.def.ptip);
   ELSE:
    TASK odloži();
   ( TRUE )
    DECISION a.pri = NONE;
     ( TRUE ):
      DECISION iskani_tip ∈ <standardni tip>;
       ( TRUE ): JOIN R2;
       ELSE:
        DECISION nadaljuj = TRUE;
         ( TRUE ):
          TASK nadaljuj := FALSE;
          DECISION ∃j : j = b.pri, j != NONE;
           ( TRUE ):
            CALL 3.2(k, j); // (str. 30)
            DECISION ∃k : k ⊂ a.zal;
             ( TRUE ):
              TASK a.pri := k;
             ELSE:
            ENDDECISION;
           ELSE:
          ENDDECISION;
          TASK iskani_tip := b.sdlnad;
          JOIN R4;
         ELSE:
          TASK iskani_tip := b.sdlnad;
          JOIN R4;
        ENDDECISION;
      ENDDECISION;
     ELSE:
      DECISION iskani_tip ∈ <standardni tip>;
       ( TRUE ): JOIN R2;
       ELSE:
        TASK iskani_tip := a.sdlnad;
        JOIN R4;
      ENDDECISION;
    ENDDECISION;
  ENDDECISION;
ENDDECISION;
TASK shrani(a.sdlime, a.pmlime, a.sdlnad);
TASK shrani(a.pmltip, a.št, a.pri, a.zal);
END ;
```

Algoritem 3.3: Algoritem analize in pretvorbe definicij podintervalnih tipov

Vse operacije nad spremenljivkami podintervalnih tipov se izvajajo enako, kot je to definirano za nadtip. Razlike se največkrat pojavijo ob predvideni zalogi vrednosti, ki jo

lahko nova definicija dodatno omeji. SDL izpostavlja razlike tudi pri prirejanju vrednosti, saj predpisuje, da lahko prirejamo samo spremenljivke istega podatkovnega tipa. Po definiciji posamezen literal pripada samo enemu podatkovnemu tipu. Podmnožica podintervalnih tipov je v modelu predstavljena z istim standardnim podatkovnim tipom v Promeli. Ob preverjanju modela in simulaciji se tako ne morejo zaznati prirejanja med spremenljivkami teh tipov. Kljub temu, da je detekcijo nepravilnih prirejanj mogoče izvesti s statično analizo, je ne izvajamo in jo prepuščamo razvojnemu orodju za SDL.

Če se spremenljivki priredi vrednost, ki je izven predpisanih mej, je nadaljnje izvajanje sistema nedefinirano. Ob analizi, ki je zapisana v algoritmu 3.3, za vsak podintervalni tip shranimo zalogo vrednosti. Orodje $sdl2pml$ omogoča, da se ob tvorbi modela vsaki atomarni operaciji doda posebna trditev, ki preverja, ali so vrednosti vseh uporabljenih spremenljivk v predvidenih mejah. S formalno verifikacijo modela lahko tako poiščemo vsa možna izvajanja, v katerih se v izrazih prekršijo omejitve, ki so bile postavljene z definicijo podatkovnega tipa. Strogo omejevanje zaloge vrednosti ob pisanju specifikacije je zaželeno zaradi gospodarnejše porabe pomnilnika in posrednega odkrivanja nepravilnih izvajanj. Pri tvorbi izvorne kode v jeziku C z orodjem OG so definirane zaloge vrednosti slabo podprte, med izvajanjem pa se ne vrši dinamično preverjanje skladnosti z definicijo. Zaloge vrednosti pri podintervalnih tipih prav tako ne podpirata orodji $sdl2if$ in $if2pml$.

V prvem odločitvenem stavku preverjamo, ali specifikacija vsebuje kakšno definicijo podintervalnega tipa, ki še ni bila analizirana. Izvajanje algoritma se zaključi, ko je izvršena analiza in pretvorba vseh definicij. V naslednjih prireditvenih stavkih se vpelje nova spremenljivka $iskani_tip$. V prvem prehodu se obravnava podatkovni tip, ki je zapisan v konstruktu <ime nadtipa> in predstavlja neposredni nadtip od podintervalnega tipa. Preostale spremenljivke se inicializirajo na $NONE$, ki predstavlja prazen zapis. Naslednji sklop preverja, ali obstaja neposredna definicija privzete vrednosti. Kadar takšna definicija obstaja, se z uporabo funkcije $pretvori_izraz()$ pretvori v primerno obliko in shrani v $ptip.pri$. Enak postopek se izvede za morebitno neposredno definicijo zaloge vrednosti.

V definicijah, kjer je nadtip eden izmed standardnih tipov SDL, se z uporabo funkcije $preveri()$ v primeru praznega zapisa v $ptip.sdlnad$ zapiše njegovo ime. Kadar zapis že obstaja, ga ne spreminjamo. Do tega pride v primerih, ko nadtip ni element $spec.def.stip$.

Pri podatkovnih tipih, katerih zaloga vrednosti so števila, sledi klic algoritma 3.4. Prvi formalni parameter je $spec.def.ptip$, ki ga trenutno obravnavamo. Vsak podintervalni tip temelji na enem izmed standardnih tipov, ki predstavlja izvorni tip in ga podajamo kot drugi formalni parameter funkcije. Kadar zaloga vrednosti ni definirana in je izvorni tip $NATURAL$, se po definiciji obeh podatkovnih tipov izvajanje algoritma zaključi. Prva definicija se uporabi ob deklaracijah spremenljivk, druga pa ob definiciji kanalov, ki prenašajo vrednosti tega tipa. V Promeli namreč ni podprta komunikacija z elementi tipa $unsigned$. Zato se definicije, podobno kot pri naštevnih tipih, izvedejo z uporabo zapisa z enim elementom. Posebna pozornost je potrebna ob uporabi spremenljivk tega tipa v modelu sistema.

Obravnavati jih je potrebno kot zapise in ne navadne spremenljivke. Za dvojne definicije podatkovnih tipov smo se tudi v tem primeru odločili zaradi želje po optimalni porabi pomnilnika. Največ se prihrani ob definicijah z zelo omejenimi zalogami vrednosti, kar je razvidno iz opisa zadnjega dela algoritma.

V primeru drugih izvornih tipov algoritem predpisuje preverjanje zaloge vrednosti nadtipa, saj jo podintervalni tip podeduje. Tu ponovno obstajata dve možnosti. Če nadtip nima definirane zaloge vrednosti, skočimo na oznako R2. V naslednjem koraku z uporabo makroja izenačimo obravnavani tip s standardnim tipom int, ki nudi največjo zalogo vrednosti. V tem primeru se ne izvede definicija novega tipa, saj je podintervalni tip enakovreden standardnemu tipu in se je v specifikaciji uporabil z namenom grupiranja spremenljivk.

Kadar med analizo ugotovimo, da je prišlo do takšne definicije v primeru izvornih podatkovnih tipov $DURATION$ ali $TIME$, se s klicem funkcije $označi()$ postavi vrednost semaforja $časovnik$ na int. V Promeli so spremenljivke časovnikov omejene na podatkovni tip $short$, zato je v tem primeru potrebno uporabnika opozoriti na prekoračitev privzetih mej ali pa primerno spremeniti modeliranje časovnikov.

```
FPAR x ∈ spec.def.ptip, y ∈ spec.def.stip
START ;
 DECISION x.zal = NONE;
  ( TRUE ):
   DECISION y = NATURAL;
    ( TRUE ):
     TASK "typedef x.pmlime {unsigned val :31}";
     TASK "typedef comm_//x.pmlime {int val}";
     END ;
    ELSE:
   ENDDECISION;
  ELSE:
   JOIN R1;
 ENDDECISION;
 DECISION x.sdlnad.zal = NONE;
  ( TRUE ):
   JOIN R2;
 ELSE:
  TASK x.zal := x.sdlnad.zal;
ENDDECISION;
R1:
DECISION x.zal >= 0;
( TRUE ):
 TASK x.st:= zaokroži(log(maksimum(x.zal))/log(2));
 TASK "typedef x.pmlime {unsigned val :x.st}";
 DECISION x.st < 8;
  ( TRUE ):
   TASK "typedef comm_//x.pmlime byte val";
ELSE:
 DECISION x.st < 16;
  ( TRUE ):
   TASK "typedef comm_//x.pmlime short val";
  ELSE:
   TASK "typedef comm_//x.pmlime int val";
  ENDDECISION;
 ENDDECISION;
 TASK x.pmltip := unsigned;
ELSE:
 DECISION |maksimum(x.zal)| <32768;
  ( TRUE ):
   TASK "#define x.pmlime short";
   TASK x.pmltip := short;
  ELSE:
  R2:
   TASK "#define x.pmlime int";
   TASK x.pmltip := int;
   DECISION y = (DURATION ∨ TIME);
    ( TRUE ):
     TASK označi(timer, int);
    ELSE:
   ENDDECISION;
 ENDDECISION;
 TASK x.st :=NONE";
ENDDECISION;
END ;
```

Algoritem 3.4: Algoritem analize zaloge vrednosti

Kadar zaloga vrednosti obstaja, se izvajanje nadaljuje na mestu z oznako R1. Prvi odločitveni stavek preverja, ali se predpisuje uporaba naravnih števil. V tem primeru sledi

analiza potrebnega števila bitov za opis maksimalne dovoljene vrednosti in primerna definicija podatkovnih tipov. Kadar so v zalogi vrednosti tudi negativna števila, se izvrši preverjanje absolutne maksimalne dovoljene vrednosti in se z uporabo makroja obravnavani podintervalni tip izenači s standardnim tipom $short$ ali int. S tem smo zaključili s pregledom algoritma 3.4. Nadaljujemo z opisom osnovnega algoritma 3.3 na mestu pretvorbe standardnega tipa $BOOLEAN$, ki se z uporabo makroja izenači s standardnim tipom $bool$.

Podpora urejenemu podatkovnemu tipu $CHARACTER$ je opisana v razdelku o standardnih podatkovnih tipih. V [48] so definirani relacijski operatorji, ki omogočajo definicijo zaloge vrednosti z zaprtimi ali odprtimi intervali. Za zapis spremenljivke tega tipa po privzetem potrebujemo 7 bitov. V prvem odločitvenem stavku zato preverjamo, ali lahko na osnovi morebitne deklaracije zaloge vrednosti to število še zmanjšamo. Dodatna optimizacija ob modeliranju praktičnih specifikacij bo mogoča samo v primeru, ko bo zaloga vrednosti omejena samo na števila, saj se vsi znaki, ki predstavljajo črke, opisujejo s števili, ki narekujejo 7-bitni zapis (tabela 3.1).

Za podatkovni tip $CHARSTRING$ ne podpiramo definicije zaloge vrednosti. Učinkovito uporabo tega konstrukta omejuje že sama definicija podatkovnega tipa, saj ne vsebuje relacijskih operatorjev, s katerimi bi se lahko izrazilo območje veljavnosti. Tako preostane samo eksplicitno naštevanje veljavnih literalov, za kar je bolje pri sprecifikaciji predvideti naštevni tip. V takšnem primeru se izvajanje algoritma ustavi. Orodje $sdl2if$ ne prevaja podintervalnih tipov, katerih nadtip je $CHARACTER$.

Sledi opis najpogosteje uporabljenega dela algoritma, ki opisuje pretvorbo podintervalnega tipa, katerega nadtip je prav tako podintervalni tip. V prvem koraku se v začasno spremenljivko $iskani_tip$ shrani vrednost nadtipa. Ta vrednost se bo v naslednjih korakih spreminjala tako dolgo, dokler ne pridemo do izvornega standardnega tipa. Zato se v naslednjem stavku nadtip zapiše med rezultate analize, saj ima vsak podintervalni tip samo enega roditelja. Na osnovi podatkov o starših se lahko z rezultati analize vedno poišče tudi izvorni standardni tip.

Zapis v oklepajih se sklicuje na množico obdelanih konstruktov, medtem ko se isti zapis brez oklepajev sklicuje na množico neobdelanih konstruktov. Odločitveni stavek, ki sledi oznaki R4, preverja, ali je roditelj v množici obdelanih podintervalnih tipov. V primeru, ko to ni tako, funkcija $odloži()$ poskrbi za odlog procesiranja do trenutka, ko so podatki o roditelju na voljo. Skladno z [57] se privzeta vrednost v primeru, če ob definiciji ni eksplicitno definirana, lahko deduje od roditelja. Zato v naslednjem odločitvenem stavku preverimo, ali ima roditelj definirano privzeto vrednost in jo s klicem funkcije $pretvori_izraz()$ shranimo v začasno spremenljivko k. Dedovanje je možno, samo kadar je vrednost v predpisanih mejah zaloge vrednosti. Čisto veljaven je namreč primer, ko je privzeta vrednost od roditelja izven meja, ki so eksplicitno zapisane v definiciji podintervalnega tipa. V tem primeru privzeta vrednost ni določena.

V naslednjih korakih se v zanki, ki jo označuje oznaka R4, išče izvorni standardni tip. Nato sledi skok na oznako R2, kjer se dopolnijo manjkajoči podatki na osnovi izvornega standardnega tipa po že opisanih postopkih. Izvajanje algoritma se zaključi tako, da se vsi pridobljeni podatki shranijo. Primer pretvorbe je prikazan na sliki 3.7. Ker sta obe definiciji omejeni na podmnožico naravnih števil, se lahko uporabi zapis z uporabo podatkovnega tipa $unsigned$. Spremenljivke bodo v obeh primerih imele privzeto vrednost 42 [1]. Podatek o privzeti vrednosti se bo v modelu predstavil ob deklaraciji spremenljivke. Tako smo se odločili zaradi enotne obravnave vseh podintervalnih tipov, saj v primeru uporabe makroja tega ni mogoče izvesti v okviru definicije.

Orodji $sdl2if$ in $if2pml$ podpirata samo okrnjeno različico podintervalnih tipov. Pretvorba se izvede samo za primere, ko je nadtip $INTEGER$. Za vse druge podatkovne tipe se definicija podintervalnega tipa ne prevaja. Ob pretvorbi se prav tako ne upošteva neposredna ali dedovana definicija privzete vrednosti in predpisana zaloga vrednosti.

```
SYNTYPE nadtip = INTEGER
  DEFAULT 42
  CONSTANTS 1:500
ENDSYNTYPE nadtip;

SYNTYPE podintervalni_tip = nadtip
  CONSTANTS 1:50
ENDSYNTYPE podintervalni_tip;

typedef pot()//nadtip {unsigned val :9}
typedef pot()//podintervalni_tip {unsigned val :6}
```

Slika 3.7: Primer pretvorbe podintervalnega tipa

3.1.3 Sinonimi

Sinonimi se v specifikacijah uporabljajo za poimenovanje osnovnih izrazov, ki se med izvajanjem sistema ne spreminjajo. Osnovni izraz je izraz, ki vsebuje samo konstantne vrednosti v obliki literalov ali sinonimov. Slika 3.8 prikazuje EBNF zapis sintakse definicije sinonima.

<sinonim> ::= **synonym** <definicija sinonima>{, <definicija sinonima>}*;
<definicija sinonima> ::= <ime sinonima> [<ime podatkovnega tipa>] = <izraz>

Slika 3.8: EBNF zapis sintakse definicije sinonima

Definicija sinonima se ne sme neposredno ali posredno sklicevati sama nase, njena vrednost pa je odvisna od mesta definicije sinonima. To zagotavlja, da bo vrednost sinonima v celotnem sistemu enaka tudi v primeru ponovne deklaracije katerega izmed operandov osnovnega izraza. Vrednost sinonima se lahko spremeni samo s ponovno deklaracijo, s katero se mu dodeli novo območje veljavnosti.

Definicija 3.10 Analizo sinonima opisujemo s $sino$, $sino \in spec.def$, in je četverica $(sdlime, pmlime, sdltip, izraz)$, kjer je:

$sdlime$ ime sinonima v specifikaciji sistema;
$pmlime$ ime sinonima v modelu sistema;
$sdltip$ ime podatkovnega tipa, ki določa zalogo vrednosti sinonima;
$izraz$ osnovni izraz, ki ga sinonim predstavlja. ♦

Algoritem 3.5 opisuje postopek analize in pretvorbe sinonimov. V prvem odločitvenem stavku se sprašujemo, ali v specifikaciji obstaja še kakšna neobdelana definicija sinonima. Nadaljnji koraki so odvisni od predpisanega podatkovnega tipa, ki določa zalogo vrednosti od podanega osnovnega izraza. Kadar je kot <ime podatkovnega tipa> predpisan standardni podatkovni tip $CHARACTER$, se s funkcijo $vrednost()$ pridobi numerična vrednost znaka na osnovi tabele 3.1.

Sinonime, katerih vrednost sodi v podatkovni tip $CHARSTRING$, je potrebno modelirati z uporabo globalne spremenljivke. Ker takšna spremenljivka med izvajanjem modela nikoli ne spremeni svoje vrednosti, ni potrebe, da bi jo hranili v vektorju stanj. To zagotovimo z uporabo predpone $hidden$. Zanka, ki jo označuje oznaka R2, poskrbi za primerno inicializacijo spremenljivke. Ta se izvede v osnovnem procesu modela v okviru prvega atomarnega prehoda (slika 3.9).

```
START ;
R1:
  DECISION ∃a: a ∈ spec.def.sino;
    ( TRUE ):
    ELSE:
      END ;
  ENDDECISION;
  TASK a.izraz := NONE;
  TASK a.sdlime := <ime sinonima>;
  TASK b := <ime podatkovnega tipa>;
  TASK a.pmlime := pot()//a.sdlime;
  DECISION b;
    ( CHARACTER ):
      TASK a.sdltip := b;
      TASK a.izraz := vrednost(<izraz>)
      TASK "#define a.pmlime a.izraz";
    ( CHARSTRING ):
      TASK a.sdltip := b;
      TASK a.izraz := <izraz>;
      TASK "hidden pt_charstring a.pmlime";
      TASK m:=0;
    R2:
      DECISION ∃i : i ∈<izraz>;
        ( TRUE ):
          TASK m := m+1;
          TASK "a.pmlime.char[m]:=vrednost(i)";
          JOIN R2;
        ELSE:
      ENDDECISION;
    ( ∈ (spec.def.ptip ∨ spec.def.ntip) ):
      TASK a.sdltip := b;
    ( BOOLEAN,NATURAL,INTEGER,REAL):
      TASK a.sdltip := b;
    ( TIME,DURATION ):
      TASK a.sdltip := b;
    ELSE:
      TASK a.sdltip := NONE;
  ENDDECISION;
  DECISION a.izraz = NONE;
    ( TRUE ):
      CALL 3.2(a.izraz, <izraz>); // (str. 30)
    TASK "#define a.pmlime a.izraz";
    ELSE:
  ENDDECISION;
  TASK "#define a.pmlime a.izraz";
  TASK shrani(a.sdlime, a.pmlime);
  TASK shrani(a.sdltip, a.izraz);
  JOIN R1;
END;
```

Algoritem 3.5: Algoritem za analizo in pretvorbo sinonimov

Vrednost izraza se lahko omeji tudi na uporabniške in preostale standardne podatkovne tipe. V teh primerih algoritem 3.2 zagotavlja primerno pretvorbo osnovnega izraza.

Ker se imena v specifikaciji SDL lahko ponavljajo, je še posebno pomembno, da se primerno preslikajo imena sinonimov in literalov. V algoritmu 3.1 se vsakemu literalu v naštevnem tipu z makrojem v modelu sistema priredi naravno število. Podobno velja za vse znake tipa $CHARACTER$. V modelu sistema tako vsi sinonimi, ki so definirani z makrojem, posredno predstavljajo cela števila. Slika 3.9 prikazuje primer pretvorbe sinonimov za vse opisane podskupine. Pri uporabniških podatkovnih tipih se sklicujemo na predhodne primere na slikah 3.6 in 3.7. Funkcija $pot*()$ določa tisto definicijo literala, ki je v območju veljavnosti in je najbližje trenutnem mestu v strukturi sistema. Takšna obravnava je potrebna zaradi morebitne večkratne uporabe imena `naštevni_tip` na različnih hierarhičnih nivojih. Orodje $sdl2if$ ob pretvorbi sinonima ne definira v modelu sistema, temveč <izraz> zapiše neposredno na mesto uporabe. Ob pretvorbi se ne preverja predpisana zaloga vrednosti. Zadnje velja tudi za OG, saj ob simulaciji ne javi napake ob definiciji ali uporabi sinonima, ki je izven predpisanih mej.

```
SYNONYM sinonimA INTEGER = 2;
SYNONYM sinonimB CHARACTER = B;
SYNONYM sinonimC naštevni_tip = lit1;
SYNONYM sinonimD podintervalni_tip = 42;
SYNONYM sinonimE CHARSTRING = "DIGIT";

#define sinonimA 2
#define sinonimB CHAR_B
#define sinonimC pot*()//naštevni_tip//__lit1
#define sinonimD 42

hidden pt__charstring pot()//sinonimE;
init{
 pot()//sinonimE[1].char = CHAR_D
 pot()//sinonimE[2].char = CHAR_I
 pot()//sinonimE[3].char = CHAR_G
 pot()//sinonimE[4].char = CHAR_I
 pot()//sinonimE[5].char = CHAR_T
}
```

Slika 3.9: Primer pretvorbe sinonimov

3.1.4 Časovniki

Ob specifikaciji lahko ločimo lastnosti sistema na kvalitativne in kvantitativne. Prve so odvisne samo od zaporedja dogodkov v sistemu, druge pa vključujejo element časa med dvema dogodkoma. Časovniki vnašajo v specifikacijo konstrukt za obravnavo časa. Pripadajo procesom in ob izteku v pridruženo vhodno vrsto pošljejo signal z lastnim imenom. Signali, ki jih sprožijo časovniki, se v ničemer ne razlikujejo od drugih signalov. Pred prvo uporabo je potrebno izvesti deklaracijo časovnika, ki jo prikazuje slika 3.10. V [57] lahko ima časovnik več aktivnih primerkov, med katerimi se loči na osnovi edinstvenih vrednosti pridruženih parametrov, ki jih določa <seznam podatkovnih tipov>. Generator C-kode v OG ne loči med

časovniki z različnimi parametri. Sprejem signala s parametri, ki ga sproži iztek časovnika, se obravnava enako kot sprejem drugih signalov. Vrednosti parametrov se shranijo v predvidene spremenljivke.

<časovnik> ::= **timer** <definicija časovnika>{, <definicija časovnika>}*
<definicija časovnika> ::= <ime časovnika> [<seznam podatkovnih tipov>]
[:= <osnovni izraz podatkovnega tipa duration>]

Slika 3.10: EBNF zapis deklaracije časovnika

Podpora modeliranju časovnikov v Promeli in razširitev orodja Spin je bila prvič predstavljena v [7]. Model sistema se razširi z dodatnimi definicijami, ki so prikazane na sliki 3.11. V obliki zapisa z enim elementom je definiran nov uporabniški podatkovni tip $timer$. Zalogo vrednosti elementov določa standardni podatkovni tip $short$. Po privzetem se spremenljivke tega tipa inicializirajo na vrednost, ki jo definira makro `OFF` in določa, da časovnik ni aktiven. Za razliko od predstavitve s signali, ki je uporabljena v SDL-ju, se v razširjenem modelu čas modelira z uporabo spremenljivk. Argumentacijo rešitve najdemo v [8]. Razširitev orodja Spin s podporo diskretnemu času ne podpira časovnikov s parametri. V razdelku 3.4.9 (str. 92) opisujemo rešitev te omejitve pri modeliranju časovnikov.

```
#define OFF (-1)
#define TOVAL (0)

typedef timer {short val=OFF}

active proctype __Timers() {
 end__Timers:
  do :: timeout
  od;
}
```

Slika 3.11: Dodatne definicije v modelu sistema za podporo časovnikom

Definicija 3.11 navaja vse elemente, ki so potrebni za primerno predstavitev časovnikov v modelu sistema. Ker je posamezen časovnik pridružen procesu, se rezultati analize hranijo v strukturah procesa, ki jih določa definicija 3.22.

Definicija 3.11 Časovnik $časo$, $časo \in spec.stru.proc$, je šesterica $(sdlime, pmlime, param, sdltip, pmltip, izraz)$, kjer je:

$sdlime$ ime časovnika v specifikaciji sistema;
$pmlime$ ime signala ob izteku časovnika v modelu sistema;
$param$ končna množica izrazov, ki predstavljajo parametre časovnika;
$sdltip$ končna množica podatkovnih tipov, ki določajo zalogo vrednosti parametrov časovnika;
$pmltip$ končna množica podatkovnih tipov za predstavitev v modelu sistema;
$izraz$ osnovni izraz, ki predpisuje privzeto vrednost časovnika.♦

V algoritmu 3.6 se analizirajo vsi deli definicije, ki jih predpisuje standard. V prvem koraku funkcija $razširitev()$ vnese definicije makrojev, ki so prikazani na slikah 3.11 in 3.49. Ti makroji dodajo Promeli podporo diskretni predstavitvi časa, ki je skladna z DT Spinom. Zanka z oznako R1 se izvaja tako dolgo, dokler ni izvedena analiza vseh procesov v specifikaciji. V okviru zanke z oznako R2 se izvede analiza vseh definicij časovnikov v obravnavanem procesu. Pri vsaki definiciji preverimo, ali predvideva uporabo dodatnih parametrov. Seznam uporabniških tipov določa njihovo zalogo vrednosti. Če se obravnavani podatkovni tip ne nahaja v množici poznanih podatkovnih tipov, se izvajanje algoritma neregularno zaključi z rezervirano besedo STOP. V nasprotnem primeru se tvori seznam vseh podatkovnih tipov in se ob zaključku zanke shrani. Analiza se zaključi s pretvorbo osnovnega izraza, ki določa privzeto vrednost časovnika.

START ;
 TASK $razširitev()$;
 R1:
  DECISION $\exists a$: $a \in spec.stru.proc$;
   (TRUE):
   ELSE:
    END;
  ENDDECISION
 R2:
  DECISION $\exists b$: $b \in a.časo$;
   (TRUE):
   ELSE:
    JOIN **R1**;
  ENDDECISION;
  TASK $b.sdlime$:= <ime časovnika>;
  TASK $b.pmlime$:= $pot()//b.sdlime$;
  DECISION $\exists$ <seznam uporabniških tipov>;
   (TRUE):
    DECISION $\exists i : i \in spec.def.\{stip \vee ntip \vee ptip\}$;
     (TRUE):
      TASK $dodaj(a.b.sdltip, i)$;
      TASK j := $pot * ()//i$;
      TASK $dodaj(a.b.pmltip, j)$;
     ELSE:
      STOP;
    ENDDECISION;
   ELSE:
    TASK $shrani(a.b.sdltip)$;
    TASK $shrani(a.b.pmltip)$;
  ENDDECISION;
  CALL 3.2($a.b.izraz$,<izraz>); // (str. 30)
  TASK $shrani(b.sdlime, b.pmlime, b.izraz)$;
  JOIN **R2**;
  END;

Algoritem 3.6: Algoritem analize definicij časovnikov

Konstrukti za delo s časovniki in podrobnejši opis modeliranja v Promeli in obravnave v DT Spinu so predstavljeni v razdelku 3.4.9.

3.1.5 Spremenljivke

Uporaba spremenljivk predstavlja razširitev končnih avtomatov. Število implicitnih stanj, ki jih predstavlja vpeljava spremenljivke, je odvisno od števila različnih vrednosti, ki so spremenljivki dodeljene med izvajanjem sistema. Globalno stanje procesa določa trenutno eksplicitno stabilno stanje in skupek vseh deklariranih spremenljivk. EBNF zapis deklaracije spremenljivk je prikazan na sliki 3.12.

Ob izrazih, ki spreminjajo vrednosti spremenljivk, avtomat prehaja med implicitnimi stanji. Takšnim izrazom pravimo, da so aktivni, saj je rezultat vrednotenja odvisen od trenutnega stanja sistema. Če ob deklaraciji spremenljivki ni dodeljena privzeta vrednost, jo

<definicija spremenljivk> ::= **dcl** <spremenljivke> {,<spremenljivke>}*

<spremenljivke> ::= <ime spremenljivke> {,<ime spremenljivke>}* <ime podatkovnega tipa>
[=<izraz>]

Slika 3.12: EBNF zapis deklaracije spremenljivk

SDL-stroj hrani v množici nedefiniranih spremenljivk. V to množico sodijo tudi spremenljivke, ki so jim bile prirejene vrednosti izven zaloge vrednosti, ki jo predpisuje $spre.sdltip$ (definicija 3.12). Uporaba spremenljivk iz te množice v aktivnem izrazu predstavlja napako, saj skladno z [57] izvajanje sistema od tega mesta naprej ni več predvidljivo.

Definicija 3.12 Spremenljivka $spre$, $spre \in spec.stru.\{proc \vee proced\}$, je četverica $(sdlime, sdltip, pmltip, vredno)$, kjer je:

$sdlime$ ime spremenljivke v specifikaciji;
$sdltip$ ime podatkovnega tipa v specifikaciji;
$pmltip$ ime podatkovnega tipa v modelu sistema;
$vredno$ trenutna vrednost spremenljivke v obliki izraza.♦

V specifikaciji SDL je veljavno območje spremenljivke načeloma vedno omejeno na telo procesa ali procedure. Izjemo predstavljajo odkrite spremenljivke. Vrednost spremenljivke, ki je deklarirana kot odkrita, je dostopna vsem procesom v okviru bloka SDL (razdelek 3.3.1). V vseh drugih primerih je potrebna izmenjava signalov. Posebno obliko izmenjave signalov predstavlja izvoz in uvoz spremenljivke. To SDL-stroj modelira z implicitno komunikacijo med procesoma in dodatnimi implicitnimi spremenljivkami. Trenutek sinhronizacije med dejansko vrednostjo in izvoženo vrednostjo določa proces, ki spremenljivko izvaža. To ne velja za spremenljivke na vpogled, ki vedno hranijo trenutno vrednost odkrite spremenljivke. Ob vsakem dostopu do izvožene vrednosti se izvede povpraševanje po trenutni vrednosti spremenljivke z implicitnim povpraševanjem in odgovorom. To prispeva h kompleksnosti modela in je v standardu prisotno izključno zaradi skladnosti s prejšnjimi verzijami. Uporaba se odsvetuje tudi zaradi možnega zagatnega stanja v posebnih primerih veljavnih specifikacij. Zaradi naštetega smo se odločili, da uvoza in izvoza ter spremenljivk na vpogled ne podpremo. Enako obnašanje sistema se lahko v specifikaciji opiše tudi z eksplicitnimi spremenljivkami in komunikacijo med procesi, zato s tem nismo zmanjšali moči jezika. Simulator v OG podpira izvoz spremenljivk samo delno, vendar funkcionalno. Orodje $sdl2if$ ne podpira odkritih in izvoženih spremenljivk.

V deklaracijah se spremenljivkam omeji zaloga vrednosti s pridruženim podatkovnim tipom. Zaradi strukturnih elementov jezika SDL (razdelek 3.3) je potrebno ob tvorbi modela poskrbeti za primerno preslikavo med imeni podatkovnih tipov. Ob prirejanju vrednosti z uporabo aktivnega izraza je zato potrebno zagotoviti, da je rezultat v predvidenih mejah. V nasprotnem je nadaljnje izvajanje sistema nepredvidljivo. Zato ob analizi deklaracij spremenljivk, ki jo opisuje algoritem 3.7, hranimo podatek o zalogi vrednosti in privzeti vrednosti

spremenljivke. Ob tvorbi modela se na konec atomarnega prehoda med eksplicitnimi stanji procesa doda trditev, ki je nepravilna vedno, ko se katerikoli izmed uporabljenih spremenljivk v okviru prehoda priredi vrednost, ki ni skladna s pridruženim podatkovnim tipom. S tem lahko ob formalni verifikaciji modela pridobimo vse poti izvajanja, ki kršijo specifikacijo modela. Izvršna koda, ki jo tvori OG generator C kode, ne izvaja dinamičnega preverjanja skladnosti s predpisano zalogo vrednosti [53]. Preverjanje skladnosti s predpisano zalogo vrednosti pri $sdl2if$ ni vključeno v tvorbo modela sistema.

START ;
R1:
DECISION $\exists a$: $a \in spec.stru.proc$;
(TRUE):
ELSE:
END ;
ENDDECISION;
R2:
DECISION $\exists b$: $b \in a.spre$;
(TRUE):
ELSE:
JOIN **R1**;
ENDDECISION;
TASK $a.b.pmltip$:= pot*()//$a.b.sdltip$
TASK $a.b.izraz$:= NONE;
DECISION $\exists j : j = (revealed \vee exported \vee viewed)$;
(TRUE):
STOP;
ELSE:
ENDDECISION;
DECISION $\exists k : k =$<privzeta vrednost>;
(TRUE):
CALL 3.2$(a.b.vredno, k)$; // (str. 30)
ELSE:
DECISION $\exists l : l \in a.b.pmltip.pri$
(TRUE):
CALL 3.2$(a.b.vredno, l)$; // (str. 30)
ELSE:
ENDDECISION;
ENDDECISION;
TASK $shrani(a.b.sdlime, a.b.sdltip,$
$a.b.pmltip, a.b.izraz)$;
JOIN **R2**;
END ;

Algoritem 3.7: Algoritem za analizo spremenljivk

Prvi odločitveni stavek v algoritmu 3.7 zagotavlja, da se analizirajo samo deklaracije spremenljivk v okviru procesov. Tiste, ki se pojavljajo v okviru procedur, zahtevajo ločeno obravnavo, ki je podana ob analizi procedur v razdelku 3.3.3. Algoritem se zaključi po analizi vseh deklaracij ali pri uporabi nepodprtih konstruktov. Če je pri deklaraciji spremenljivke eksplicitno podana privzeta vrednost, se izraz s klicem algoritma 3.2 ovrednoti in zapiše v $var.vredno$. Če privzeta vrednost ni podana ob deklaraciji spremenljivke, se preveri, ali obstaja definicija privzete vrednost v okviru definicije pridruženega podatkovnega tipa. S pomočjo funkcije pot*() poiščemo podatkovni tip, ki je veljaven na mestu deklaracije spremenljivke. To določa struktura sistema, ki jo opisuje razdelek 3.3. Orodje $sdl2if$ podpira privzeto vrednost samo ob deklaraciji spremenljivke. Posredna definicija v okviru podatkovnega tipa ni podprta.

3.2 Komunikacija

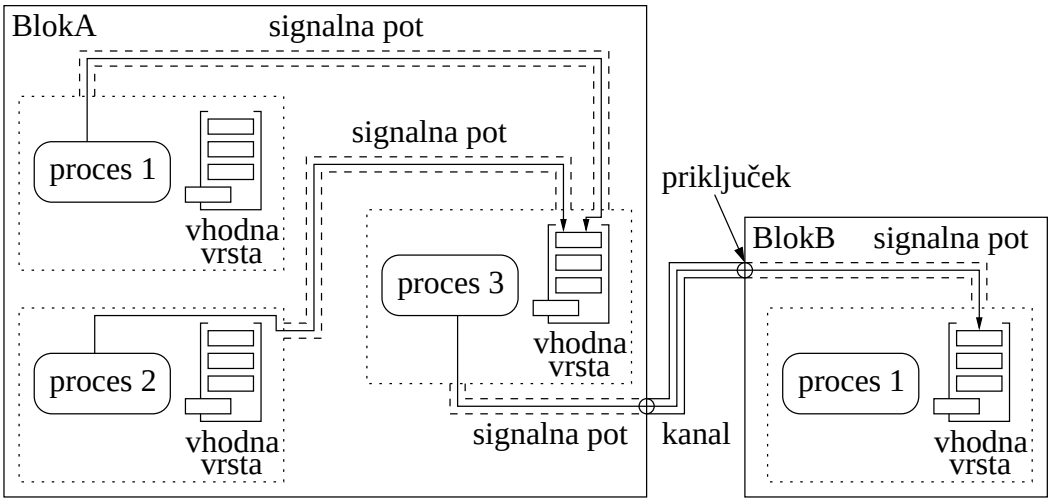
Zaradi uporabe strukturnih gradnikov in hierarhične delitve sistema neposredna komunikacija med dvema procesoma v specifikaciji pogosto ni mogoča. Zato je v jeziku SDL predvidenih več konstruktov, ki omogočajo povezovanje dveh poljubnih procesov. Povezave sodijo v statični del opisa in so poznane ob zagonu sistema. Definicija 3.13 navaja vse elemente

komunikacijskega podsistema, ki jih je potrebno analizirati za primerno modeliranje komunikacije.

Definicija 3.13 Komunikacijski podsistem kom, $kom \in spec$, je peterica $(sig, siglist, sigpot, kan, prik)$, kjer je:

sig končna množica signalov;
$siglist$ končna množica signalnih list;
$sigpot$ končna množica signalnih poti;
kan končna množica kanalov;
$prik$ končna množica priključkov.♦

Procesi komunicirajo z izmenjavo signalov. SDL-stroj vsakemu procesu implicitno pridruži vhodno vrsto, kjer se zbirajo prispeli signali, ki sodijo v množico veljavnih signalov procesa. Slika 3.13 prikazuje primer komunikacije med procesi. Neposredno komunikacijo med dvema procesoma v istem bloku omogoča signalna pot, ki ju povezuje. V primerih, ko je potrebno zagotoviti komunikacijo med procesi v različnih blokih, se signalne poti združujejo v kanale. Kadar se povezujejo komunikacijski gradniki na različnih hierarhičnih nivojih, se uporabi konstrukt <priključek>.



Slika 3.13: Komunikacija med procesi

V Promeli se komunikacija vrši neposredno med procesom in kanalom. Izpostaviti velja razliko med pomenom besede kanal v specifikaciji sistema in v modelu sistema. V specifikaciji je kanal povezovalni element, medtem ko v modelu sistema modelira pridruženo vhodno vrsto procesa. V modelu sistema je od elementov množice kom eksplicitno predstavljen samo sig (definicija 3.13). Analiza drugih konstruktov omogoča primerno modeliranje tistega dela SDL-stroja, ki določa komunikacijski podsistem.

Komunikacijske poti, ki vključujejo kanale, se delijo na tiste z zakasnitvijo in takšne, ki zakasnitve nimajo. SDL ne vsebuje konstrukta, ki bi vplival na čas trajanja zakasnitve. Zakasnitve so torej nedeterministične in so odvisne od implementacije v ciljnem sistemu.

Posebna obravnava nedeterminističnih zakasnitev pri tvorbi modela sistema je zato potrebna samo v primeru, ko se v okviru istega atomarnega prehoda naslavlja isti proces po poteh z različnimi zakasnitvami. Vsi drugi primeri so ob formalni verifikaciji obravnavani v okviru preverjanja vseh možnih izvajanj modela.

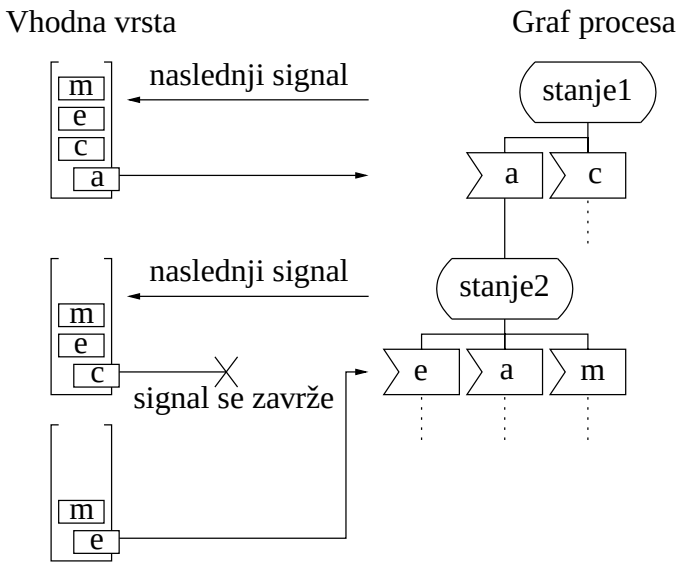
3.2.1 Signal

Signal je najnižji objekt pri komunikaciji in je definiran z imenom. Ob definiciji parametrov lahko signal prenaša podatke. Zalogo vrednosti in vrstni red določajo predpisani podatkovni tipi. Ob pošiljanju signala predstavljajo množico veljavnih dejanskih parametrov lokalne spremenljivke procesa, ki so skladne z definiranimi tipi. Definicija 3.14 določa vse elemente, ki nedvoumno opisujejo signal in omogočajo primerno predstavitev ob tvorbi modela.

Definicija 3.14 Signal sig, $sig \in spec.kom$, je trojica $(sdlime, pmlime, param)$, kjer je:
$sdlime$ ime signala v specifikaciji sistema;
$pmlime$ ime signala v modelu sistema;
$param$ je množica podatkovnih tipov, ki določajo zalogo vrednosti parametrov signala.♦

Vrstni red signalov se v komunikacijskih gradnikih po definiciji ohranja. Primerek signala se tvori ob uporabi konstrukta izhod (str. 97), ob sprejemu s konstruktom vhod (str. 83) pa preneha obstajati. Signal se vedno pošilja proti procesu ali okolju. Po privzetem se signali obravnavajo po metodi prvi noter, prvi ven. Takšen način dela se lahko zaobide z uporabo konstruktov shrani (str. 81) in prioritetni vhod (str. 83). Kadar je proces pripravljen sprejeti signal, to SDL-stroj sporoči implicitni pridruženi vhodni vrsti s signalom naslednji signal. V primeru, da vhodna vrsta vsebuje signal, ga posreduje procesu. Po sprejetju signala proces izvede izbran prehod v naslednje stanje. Če proces v trenutnem stanju ni pripravljen sprejeti posredovanega signala, se ta implicitno zavrže. Temu pravimo tudi implicitni prehod (str. 88). Kadar želimo takšen signal shraniti za kasnejšo uporabo v drugem eksplicitnem stanju procesa, se lahko uporabi konstrukt shrani (str. 81).

Vhodna vrsta je vedno, razen ob komunikaciji s procesom, pripravljena na sprejem signala. Izvajanje procesa in sprejem signalov sta neodvisna. Slika 3.14 prikazuje primer komunikacije na enostavnem primeru grafa procesa. Komunikacijo modeliramo s primerno predstavitvijo procesov in pridruženih kanalov. Ker se lahko procesi med izvajanjem sistema dinamično tvorijo in ubijajo, je potrebno zagotoviti tudi primerno modeliranje SDL-stroja. Novemu primerku procesa je potrebno ob tvorbi pridružiti primeren kanal. Do pomembne razlike med specifikacijo v jeziku SDL in modelom v jeziku Promela pride pri obravnavi kapacitete kanala, t. j. številu rež. SDL tega števila ne omejuje in predpostavlja neomejeno vhodno vrsto. Model sistema vedno opisuje končen sistem, zato je potrebno privzeto število rež omejiti. Po privzetem je to število omejeno na tri, saj bistveno vpliva na velikost grafa dosegljivih stanj. Skladno s potrebami, ki se pokažejo ob verifikaciji posameznega modela, pa se lahko poljubno določa s strani verifikatorja.



Slika 3.14: Primer komunikacije med procesom in pridruženo vhodno vrsto

<signal> ::= **signal** <ime signala>[(<tip prenašane vrednosti> {, <tip prenašane vrednosti>}∗)]
{, <ime signala>[(<tip prenašane vrednosti> {, <tip prenašane vrednosti>}∗)]}∗

<tip prenašane vrednosti> ::= <ime podatkovnega tipa> | <standardni podatkovni tip>

Slika 3.15: EBNF zapis sintakse definicije signala

Signal je lahko definiran na nivoju sistema, bloka ali procesa (slika 3.15). Signali, ki so definirani na določenem hierarhičnem nivoju, se lahko uporabljajo tudi v vseh nižjih nivojih. Tako se lahko signal, ki je definiran znotraj procesa, uporablja samo med različnimi primerki tega procesa. Signal, ki je definiran na nivoju sistema, pa se lahko uporabi znotraj celotnega sistema, torej povsod. V praksi je pametno upoštevati princip lokalnosti in signal definirati na najnižjem možnem nivoju. Definicije signalov na hierarhično različnih mestih lahko uporabijo enaka imena. Tako se lahko v sistemu pojavlja definicija večjega števila signalov z enakim imenom, a različnim številom parametrov in območjem veljavnosti.

V modelu sistema je ob tvorbi pridruženega kanala potrebno eksplicitno definirati vse signale, ki jih lahko proces sprejme. Zato se ob analizi specifikacije vsakemu procesu priredi množica veljavnih signalov. V primeru analize signalov je torej potrebno ločiti med množicami signalov, ki ju določata definiciji 3.13 in 3.22. Prva je množica vseh definicij signalov v specifikaciji sistema, druga pa je podmnožica prve in vsebuje signale, ki so pridruženi posameznemu procesu.

Algoritem 3.8 vrši analizo vseh definicij signalov v specifikaciji sistema. V prvem koraku z uporabo funkcije $pot()$ določimo mesto deklaracije v strukturi sistema. Sledi analiza morebitnih definicij parametrov. Z uporabo funkcije $poišči()$ se pridobi edinstveno ime veljavnega podatkovnega tipa iz rezultatov analize podatkovnih tipov. Pri podatkovnih tipih, ki se v modelu sistema predstavijo s podatkovnim tipom $unsigned$, je potrebno pri

```
START;
 TASK i:=0;
R1:
 DECISION ∃a: a ∈ spec.kom.sig;
  ( TRUE ):
  ELSE:
   JOIN R3;
 ENDDECISION;
 TASK a.pmlime := pot()//sdlime;
 R2:
  DECISION ∃b: b ∈ a.param;
   ( TRUE ):
    TASK a.b.sdlime := <tip prenašane vrednosti>;
    TASK a.b.pmlime := poišči((spec.def{stype
                          ∨ntype ∨ ptype}));
    DECISION a.b.pmltip = unsigned;
     ( TRUE ):
      TASK a.b.pmlime := "comm__//a.b.pmlime";
     ELSE:
    ENDDECISION;
    TASK shrani(a.b); JOIN R2;
  ELSE:
 ENDDECISION;
 TASK "#define a.pmlime i"
 TASK i := i+1; JOIN R1;
 R3:
  TASK j := round_up(log(i)/log(2));
  DECISION j;
   ( < 32768 ):
    DECISION j;
     ( < 256 ):
      TASK "#define pt__sig byte";
     ELSE:
      TASK "#define pt__sig short";
    ENDDECISION;
   ELSE:
    TASK "#define pt__sig int";
 ENDDECISION;
 END;
```

Algoritem 3.8: Algoritem analize signalov

komunikaciji uporabiti nadomestni podatkovni tip. Imenu podatkovnega tipa se, skladno z algoritmoma 3.1 in 3.4, doda predpona `comm__`. Pri definiciji signalov v Promeli se za oštevilčenje signalov navadno uporabi standardni podatkovni tip $mtype$, ki je omejen na 255 zapisov. Ker ne želimo omejiti števila podprtih signalov, se v modelu sistema za preslikavo imen uporabijo naravna števila. Število dejanskih definicij se beleži v začasni spremenljivki i. V zadnjem delu algoritma se na osnovi dejanskega števila definiranih signalov tudi izvede definicija podatkovnega tipa `pt__sig`. Ta se v modelu sistema uporabi za omejevanje vrednosti prvega parametra vseh kanalov, ki nosi ime prejetega signala.

Standard [57] dodatno predvideva razgradnjo signala na večje število signalov na nivoju bloka, kar je lahko v pomoč ob razgradnji specifikacije sistema na manjše in bolj podrobno opisane dele. Orodja OG, $sdl2if$ in $sdl2pml$ te razgradnje ne podpirajo. Podobno funkcionalnost nudi uporaba signalnih list, ki je s stališča berljivosti kode boljša izbira. S primerno metodologijo so dodatni posegi potrebni samo v primeru, ko se definirajo signali na novih poteh.

3.2.2 Signalna lista

Med procesi se pogosto prenaša večje število signalov. Zato jih lahko združimo v signalne liste, ki poenostavijo predstavitev komunikacijskega podsistema. Ob spremenjeni signalni listi se vsi popravki vnesejo samo na mestu definicije in ni potrebno popravljati definicij vseh signalnih poti in kanalov, ki prenašajo obravnavani signal. Element signalne liste je lahko signal, časovnik ali druga signalna lista, vendar ne sme, neposredno ali posredno, vključevati sama sebe (slika 3.16).

Grafična predstavitev	**Tekstovni zapis**
[element seznama {,element seznama}*]	<signalna lista> := **signallist** <ime signalne liste> = <seznam signalov>

<seznam signalov> ::= <element seznama>{, <element seznama>}
<element seznama> ::= <ime signala> | <ime časovnika> | (<ime signalne liste>)

Slika 3.16: EBNF zapis sintakse definicije signalne liste

Uporaba signalnih list je priporočljiva, saj zmanjša potrebne posege v specifikacijo sistema ob definiciji novega signala. Promela ne pozna enakovrednega konstrukta, saj se komunikacija vedno vrši neposredno med dvema procesoma. Analiza definicij signalnih list je potrebna zaradi pravilnega modeliranja delovanja SDL-stroja pri izbiri ciljnega procesa. Definicija 3.15 zajema vse elemente, ki v celoti opisujejo signalno listo in predstavlja osnovo za pripravo modela sistema.

Definicija 3.15 Signalna lista $siglist$, $siglist \in spec.kom$, je trojica $(sdlime, stru, signal)$, kjer je:

$sdlime$ ime signalne liste v specifikaciji sistema;
$stru$ mesto definicije v strukturi sistema;
$signal$ je končna množica signalov in je element $spec.kom.sig$.♦

Algoritem 3.9 v prvem odločitvenem stavku preveri, ali še obstaja kakšna neobdelana definicija signalne liste. Po pridobitvi imena signalne liste in mesta definicije v strukturi sistema se v okviru zanke z oznako R2 izvede analiza elementov seznama.

R1:
DECISION $\exists a$: $a \in spec.kom.siglist$;
(TRUE):
ELSE:
END ;
ENDDECISION;
TASK $a.sdlime$:= <ime signalne liste>;
TASK $a.stru := pot()$;
R2:
DECISION $\exists b$: $b \in a.signal$;
(TRUE):
TASK $a.b.sdlime$:= <element seznama>
TASK $i := a.b.sdlime$
TASK $a.b.pmlime := poišči(i, (spec.kom\{sig \vee siglist\} \vee spec.def.časo))$;
TASK $shrani(a.b)$;
JOIN **R2**;
ELSE:
ENDDECISION;
TASK $shrani(a.sdlime, a.stru)$;
JOIN **R1**;

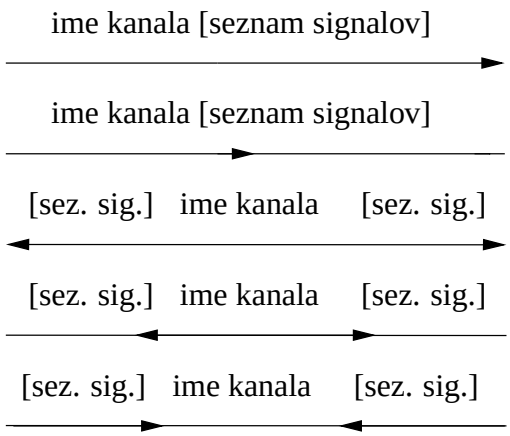
Algoritem 3.9: Algoritem analize signalnih list

3.2.3 Kanal

V opisu specifikacije sistema bloke povezujemo s kanali, ki predstavljajo transportno progo za signale. Kanal povezuje dva bloka ali blok in njegovo okolje in je lahko eno- ali dvosmeren. Kanal mora vsebovati seznam veljavnih signalov za vse definirane smeri. Slika 3.17

prikazuje EBNF zapis sintakse specifikacije kanala. Kadar kanal definira dve progi, morata biti usmerjeni v nasprotno smer. Takšen opis določa dvosmerni kanal. Signali med prenosom po kanalu ohranijo vrstni red.

Grafična predstavitev



Tekstovni zapis

<kanal> ::=
channel <ime kanala> [**nodelay**]
<proga kanala> [<proga kanala>]
endchannel <ime kanala>;

<proga kanala> ::=
from <izvorna točka kanala> **to**
<ponorna točka kanala> **with**
<seznam signalov>;

<izvorna točka kanala> ::=
{<ime bloka> | <**env**>}

<ponorna točka kanala> ::=
{<ime bloka> | <**env**>}

Slika 3.17: EBNF zapis sintakse specifikacije kanala

Po privzetem kanali vnašajo zakasnitve, ki niso eksplicitno definirane. Z rezervirano besedo **nodelay** se definira kanal brez zakasnitev. Kanal brez zakasnitev opisujejo grafični simboli s puščicami na koncu povezave.

Kanal se v modelu sistema ne predstavlja eksplicitno. Analiza kanalov je potrebna zaradi primernega modeliranja SDL-stroja, ki določa semantična pravila komunikacije med procesi v primeru posrednega naslavljanja skupine procesov. Definicija 3.16 določa elemente, ki kanal opisujejo v celoti.

Definicija 3.16 Kanal je šesterica kan, $kan \in spec.kom$, $(sdlime, ime, izvor, ponor, zaka, signal)$, kjer je:

$sdlime$ ime kanala v specifikaciji sistema;
ime edinstveno ime kanala v specifikaciji sistema;
$izvor$ ime izvornega elementa v strukturi sistema;
$ponor$ ime ponornega elementa v strukturi sistema;
$zaka$ boolova konstanta, ki zavzame vrednost 1, če kanal vnaša zakasnitev;
$signal$ množica prenašanih signalov.♦

Algoritem 3.10 prikazuje analizo kanalov. Dvosmerni kanal se obravnava kot dva enosmerna kanala. Iz predhodnih analiz se za vsak kanal sestavi seznam vseh prenašanih signalov. To omogoča izdelavo analize veljavnih poti za vsak signal v primeru pošiljanja signalov z uporabo rezervirane besede **via**. Dodatno se zabeleži tudi, ali kanal vnaša zakasnitev v komunikacijo ali ne.

START ;
R1:
DECISION $\exists a$: $a \in spec.kom.kan$;
(TRUE):
ELSE:
END ;
ENDDECISION;
TASK $a.sdlime$:= <ime kanala>;
TASK $a.ime$:= $pot()//a.sdlime$;
TASK i := <izvorna točka kanala>;
TASK $a.izvor$:= $poišči(i, (spec.stru.blok))$;
TASK j := <ponorna točka kanala>;
TASK $a.ponor$:= $poišči(j, (spec.stru.blok))$;
R2:
DECISION $\exists b$: $b \in$<seznam signalov>;
(TRUE):
TASK k := <element seznama>;
TASK m := $poišči(k, (spec.kom\{sig \vee siglist\} \vee spec.def.časo))$;
TASK $dodaj(a.signal, a.b)$; JOIN **R2**;
ELSE:
ENDDECISION;
DECISION $\exists$ **nodelay**
(TRUE):
TASK $a.zaka$:= 1;
ELSE:
TASK $a.zaka$:= 0;
ENDDECISION;
TASK $shrani(a.sdlime, a.ime, a.izvor, a.ponor)$;
TASK $shrani(a.zaka, a.signal)$;
JOIN **R1**;

Algoritem 3.10: Algoritem analize kanalov

Primer na sliki 3.18 prikazuje definicijo kanala, ki povezuje podbloka $B11$ in $B12$. Ker kanal definira komunikacijo za obe smeri, je rezultat analize zapisan v dveh samostojnih zapisih. V seznamu signalov je v obeh primerih uporabljena tudi signalna lista, ki se od imena signala loči s tem, da je zapisana med oklepaji.

```
CHANNEL kan
FROM B12 TO B11 WITH (SL_sig_listA),sig1;
FROM B11 TO B12 WITH (SL_sig_listB),sig2;
ENDCHANNEL kan;
```

Slika 3.18: Primer definicije kanala

3.2.4 Signalna pot

Signalna pot povezuje proces z drugim procesom v bloku in okoljem bloka (slika 3.13) in predstavlja prenosno pot za signale. Lahko je eno- ali dvosmerna. Dvosmerna signalna pot je enakovredna dvema enosmernima. To se odraža tudi pri analizi, saj imajo dvosmerne signalne poti ločen zapis za vsako smer. Zaporedje in vsebina signalov se med prenosom ne spremeni in se dostavi z enega konca na drugega brez zakasnitev. Kadar končni točki povezujeta dve signalni poti, morata biti definirani za nasprotni strani. Pri povezovanju z višjeležnimi strukturnimi elementi se ena končna točka poimenuje z rezervirano besedo **env**. Dodatni opis pa je podan s konstruktom priključek (str. 50). Signalna pot se lahko priključi na proces ali kanal (str. 47). Definicija 3.17 določa elemente, ki v celoti opisujejo signalno pot. Signalne poti v modelu sistema niso eksplicitno predstavljene, zato je analiza, ki jo prikazuje algoritem 3.11, potrebna za primerno modeliranje SDL-stroja pri izbiri veljavnih ciljnih procesov pri nepopolnem naslavljanju v okviru konstrukta izhod (str. 97). Primer definicije signalne poti je na sliki 3.21.

Definicija 3.17 Analiza signalne poti $sigpot$, $sigpot \in spec.kom$, je peterica $(sdlime, ime, izvor, ponor, signal)$, kjer je:

$sdlime$ ime signalne liste v specifikaciji sistema;
ime edinstveno ime signalne liste v modelu sistema;
$izvor$ ime izvornega elementa v strukturi sistema;
$ponor$ ime ponornega elementa v strukturi sistema;
$signal$ je končna množica signalov.♦

Grafična predstavitev

ime signalne poti [seznam signalov]

[sez. sig.] ime signalne poti [sez. sig.]

Tekstovni zapis

<signalna pot> :=
signalroute <ime signalne poti>=
<proga signalne poti> [<proga signalne poti>]

<proga signalne poti> ::= **from** <izvorna točka poti> **to** <ponorna točka poti> **with** <seznam signalov>
<izvorna točka poti> ::= {<ime procesa> | **env**} <ponorna točka poti> ::= {<ime procesa> | **env**}

Slika 3.19: Sintaksa specifikacije signalne poti

Seznam veljavnih signalov je presek signalov, ki so definirani v obeh končnih točkah kot izhodni in vhodni signali za izbrano smer komunikacije. Enosmerna signalna pot se lahko zaključuje tudi v izvornem procesu. S tem se omogoči komunikacija med različnimi primerki procesa.

Algoritem 3.11 prikazuje analizo signalnih poti v specifikaciji sistema. V prvem koraku se pridobi ime signalne poti. Sledi definicija edinstvenega imena v strukturi sistema. Nadaljujemo z iskanjem edinstvenega imena izvornega in ponornega elementa povezave, ki je skladno z mestom definicije signalne poti v strukturi sistema. Pri tem nam pomaga funkcija $poišči()$. V primeru povezovanja z višjeležnim elementom se kot ime shrani kar rezervirana beseda **env**. V zanki z oznako R2 se izvede analiza vseh signalov, ki jih lahko signalna pot prenaša. Edinstveno ime v modelu sistema se išče med množico signalov, signalnih list in časovnikov.

3.2.5 Priključek

Kadar povezujemo konstrukte na istem hierarhičnem nivoju, sta obe končni točki nedvoumno določeni. Ob povezovanju z višjeležnimi elementi pa je potrebno vpeljati nov konstrukt, ki omogoča povezovanje signalnih poti in podkanalov na kanale, ki se zaključujejo na višjeležnem elementu. Slika 3.20 prikazuje sintakso specifikacije konstrukta priključek. Višjeležni kanal lahko prenaša več signalnih poti in podkanalov.

Priključek lahko določa povezavo med:

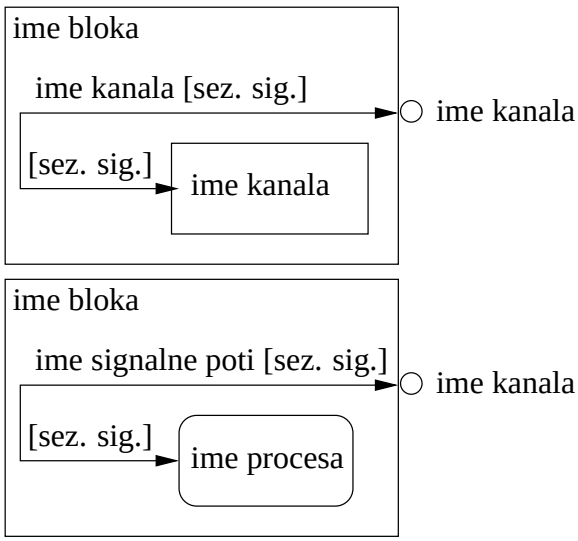
- kanalom in eno ali več signalnimi potmi in
- kanalom in enim ali več kanalov.

START ;
R1:
DECISION $\exists a$: $a \in spec.kom.sigpot$;
(TRUE):
ELSE:
END ;
ENDDECISION;
TASK $a.sdlime$:= <ime signalne poti>;
TASK $a.ime$:= $pot()$//$a.sdlime$;
TASK i := <izvorna točka poti>;
TASK $a.izvor$:= $poišči(i, (spec.stru.proc))$;
TASK j := <ponorna točka poti>;
TASK $a.ponor$:= $poišči(j, (spec.stru.proc))$;

R2:
DECISION $\exists b$: $b \in a.signal$;
(TRUE):
TASK k := <element seznama>;
TASK m := $poišči(k, (spec.kom\{sig \vee siglist\} \vee spec.def.časo))$;
TASK $dodaj(a.signal, m)$; JOIN **R2**;
ELSE:
ENDDECISION;
TASK $shrani(a.sdlime, a.ime, a.izvor)$;
TASK $shrani(a.ponor, a.signal)$;
JOIN **R1**;

Algoritem 3.11: Algoritem analize signalnih poti

Grafična predstavitev



Tekstovni zapis

<priključek> ::=
connect {<seznam kanalov> **and** <seznam kanalov>}
| {<seznam kanalov> **and** <seznam signalnih poti>}

<seznam kanalov> ::=
<ime kanala> {, <ime kanala>}∗

<seznam signalnih poti> ::=
<ime signalne poti> {,
<ime signalne poti>}∗

Slika 3.20: Sintaksa specifikacije priključka

Definicija 3.18 zajema vse elemente definicije priključka, ki so potrebni za primerno analizo komunikacijskega podsistema specifikacije. Priključek v modelu sistema ni predstavljen eksplicitno, analiza pa je potrebna zaradi modeliranja oddaje signala.

Definicija 3.18 Priključek $prik$, $prik \in spec.kom$, je dvojica $(sez1, sez2)$, kjer je:

$sez1$ množica kanalov, ki so na levi strani rezervirane besede **and**;
$sez2$ množica kanalov in signalnih poti, ki so na desni strani rezervirane besede **and**. ♦

Jedro algoritma 3.12 je sestavljeno iz dveh zank. Prva izvaja analizo levega dela priključka, druga pa desnega. Z uporabo funkcije $poišči()$ se določi edinstveno ime kanala oz. signalne poti v specifikaciji sistema. Zanašamo se na predhodne analize signalnih poti in kanalov ter trenutno mesto v strukturi sistema. Imena dodamo na seznam elementov priključka

z uporabo funkcije $dodaj()$. Slika 3.21 prikazuje primer definicije priključka, ki povezuje kanal kan in signalno pot sig_pot.

START ;
R1:
DECISION $\exists a$: $a \in spec.kom.prik$;
(TRUE):
ELSE:
END ;
ENDDECISION;
R2:
DECISION $\exists b$: $b \in$ <seznam kanalov>;
(TRUE):
TASK $i := poišči(b, (spec.kom.kan))$;
TASK $dodaj(a.sez1, i)$; JOIN **R2**;
ELSE:
ENDDECISION;
R3:
DECISION $\exists c$: $c \in$ (<seznam kanalov> $\vee$ <seznam signalnih poti>);
(TRUE):
TASK $j := poišči(c, (spec.kom.\{kan \vee sigpot\}))$;
TASK $dodaj(a.sez2, j)$; JOIN **R3**;
ELSE:
ENDDECISION;
TASK $shrani(a)$;
JOIN **R1**;

Algoritem 3.12: Algoritem analize priključkov

```
CONNECT kan AND sig_pot

SIGNALROUTE sig_pot
FROM P1 TO ENV WITH (SL_signalna_lista),Sig1;
FROM ENV TO P1 WITH (SL_signalna_lista);
```

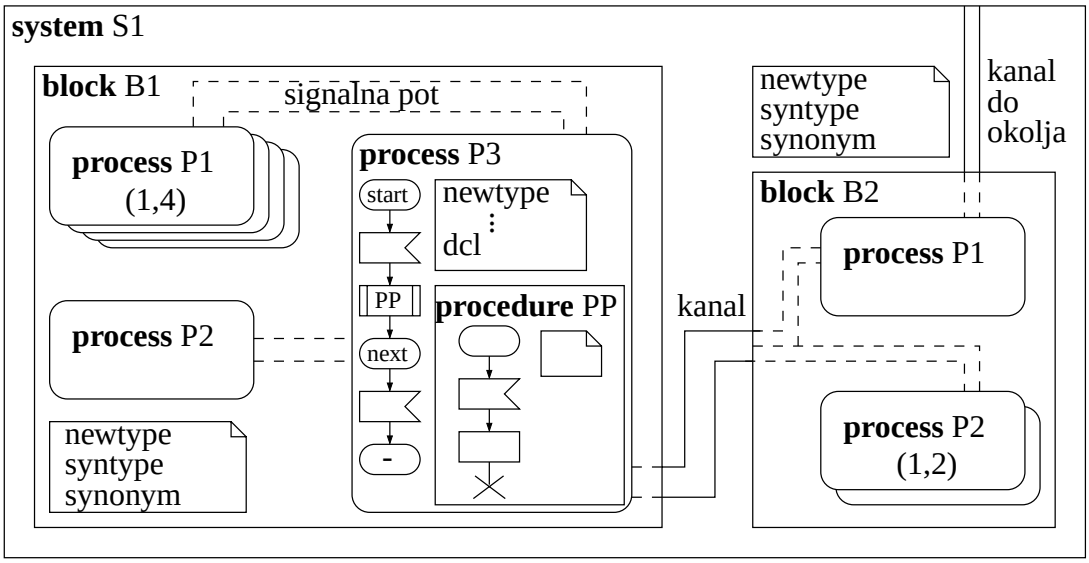
Slika 3.21: Primer definicije signalne poti in priključka

3.3 Struktura sistema

Strukturni elementi omogočajo preglednejši in lažje obvladljiv opis specifikacije sistema. Najvišji nivo arhitekture predstavlja objekt <sistem>. Sistem je grafično predstavljen kot pravokotnik. Vse, kar je zunaj sistema, je okolje (slika 3.22). Sistem razgradimo na bloke, ki združujejo opise večjih podsklopov sistema. Definicija sistema mora vsebovati vsaj en blok. Za potrebe komunikacije so bloki povezani s kanali, ki so prenosni medij za signale. V primeru komunikacije z okoljem kanal povezuje blok z okoljem, kar se grafično prikaže kot povezava z zunanjo mejo sistema.

Bloki se lahko nadalje delijo na podbloke. Kadar se z delitvijo na manjše bloke zaključi, se delovanje posameznih blokov opiše s procesi. Procesi so predstavljeni z razširjenimi komunicirajočimi končnimi avtomati. Povezujejo jih signalne poti, ki se na višjem hierarhičnem nivoju združujejo s kanali. Dodatno delitev v okviru jezika SDL nudijo procedure, ki združujejo dele kode, ki se uporabijo na več mestih.

Definicije podatkovnih tipov, časovnikov in spremenljivk se v okviru specifikacije lahko izvedejo v okviru konstrukta, ki je v grafičnem zapisu prikazan kot pravokotnik z zavihkom. Mesto definicije v strukturi sistema določa območje veljavnosti definicije.



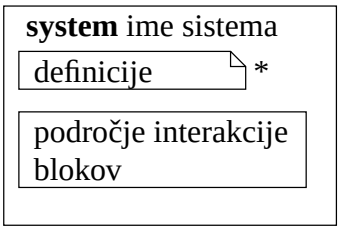
Slika 3.22: Struktura sistema v SDL-ju

Definicija 3.19 Strukturne elemente opisujemo s $stru$, $stru \in spec$, ki je trojica $(blok, proc, proced)$, kjer je:

$blok$ končna množica blokov;
$proc$ končna množica procesov;
$proced$ končna množica procedur.♦

Sledi formalni in neformalni opis konstruktov jezika SDL, ki skladno z definicijo 3.1 sodijo v množico $spec.stru$. Formalni opis sintakse konstruktov je zapisan z uporabo opisnega jezika EBNF. Pri opisu grafičnega dela so na desni strani simbolov uporabljeni znaki "*", "+" in "]", ki pomenijo, "nič ali več primerkov", "eden ali več primerkov" in "nič ali en primerek". V primeru, ko se na določenem mestu lahko uporabi samo eden izmed razpoložljivih simbolov, sta za združevanje uporabljena zavita oklepaja ({,}), za znak izbire pa navpična črta (|). To notacijo smo uporabili zaradi kompaktnejšega zapisa definicije grafičnih konstruktov. Dodatna razlaga pomena posameznega konstrukta je podana v spremnem besedilu. Slika 3.23 prikazuje grafično in tekstovno predstavitev sistema.

Grafična predstavitev



Tekstovni zapis

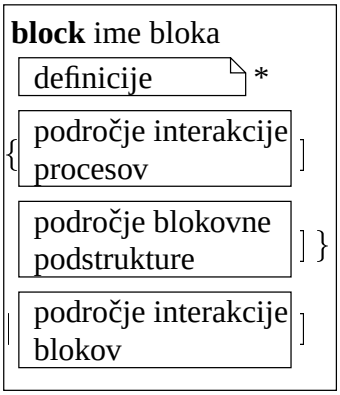
```
<sistem> ::=
    system <ime sistema>;
      {<definicija blokov>
      | <definicija kanalov>
      | <definicija signalov>
      | <definicija signalnih list>
      | <definicija podatkov>
      | <definicija procedur>}+
    endsystem [<ime sistema>];
```

Slika 3.23: EBNF zapis sintakse specifikacije sistema

3.3.1 Blok

Blok predstavlja svojevrsten podsistem in se grafično predstavi kot pravokotnik. S strukturiranim pristopom si lahko olajšamo opis velikih sistemov. Bloki so med seboj neodvisni. Vsak blok je lahko nadalje razdeljen na podbloke ali pa vsebuje enega ali več procesov. Če blok ne vsebuje področja interakcije procesov, mora vsebovati področje blokovnih podstruktur. Blok predstavlja primeren način za združevanje procesov v logične skupine.

Grafična predstavitev



Tekstovni zapis

```
<blok> ::=
    block <ime bloka>;
      {<povezava kanalov in signalnih poti>
      | <definicija signalov>
      | <definicija signalnih list>
      | <definicija podatkov>
      | <definicija procedur>
      | <definicija procesov>
      | <definicija signalnih poti>}*
      [<blokovna podstruktura>]
    endblock [<ime bloka>];
```

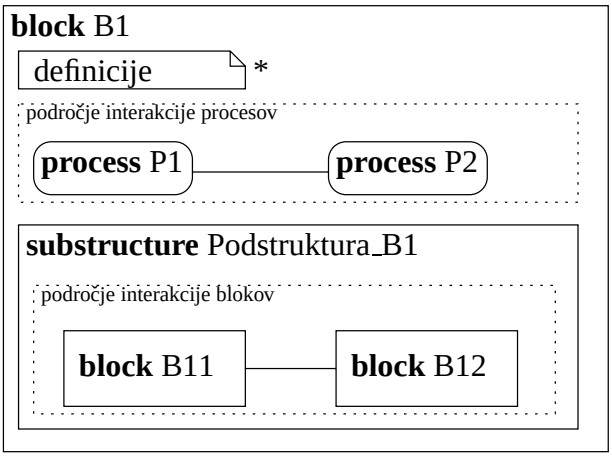
Slika 3.24: EBNF zapis sintakse specifikacije bloka

Definicije, ki jih prikazuje grafična predstavitev bloka, so podrobneje navedene v tekstovnem zapisu sintakse na sliki 3.24. Podatkovni tipi, ki se definirajo na nivoju bloka, so vidni v vseh podstrukturah. Blok lahko ima pridružene kanale, na katere se priključujejo signalne poti procesov. Definicija kanalov je del specifikacije enega nivoja višje — sistema oz. nadbloka.

SDL omogoča različno podrobne opise sistema: od abstraktnega nivoja, kjer je samo en proces, do izredno podrobnega opisa, ki predstavlja osnovo za avtomatsko generiranje izvorne kode v izbranem programskem jeziku in ima več sto procesov. Blok zato lahko hkrati vsebuje področje interakcije procesov in podstrukturo bloka. Podstruktura v celoti nadomesti opis bloka s procesi s podrobnejšo razgradnjo na podbloke ter nadalje na procese. Primer uporabe podstruktur prikazuje slika 3.25. Iz zapisanega sledi, da SDL definira štiri različne predstavitve blokov:

1. končni blok — vsebuje samo področje interakcije procesov,
2. razdeljen blok — sestavljajo ga podbloki, vsebuje področje interakcije blokov,
3. sestavljen blok — vsebuje področje interakcije procesov in podstrukturo,
4. podstruktura — predstavlja posebno različico bloka.

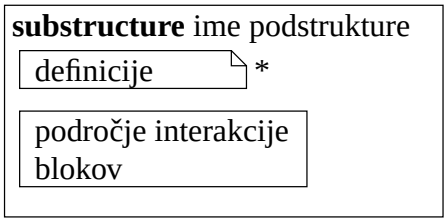
Opis sistema se z uporabo te tehnike lahko prične na visokem abstraktnem nivoju in se postopoma dopolnjuje z natančnejšimi opisi z uporabo podstruktur. SDL ne omejuje števila podstruktur. Vmesniki za komunikacijo se ohranjajo, zato se lahko različni nivoji abstrakcije



Slika 3.25: Primer uporabe podstrukture

s pridom uporabijo tudi pri avtomatski gradnji modela sistema za namene formalne verifikacije. Naloga obravnava samo prvi dve obliki bloka. Podpora preostalima dvema oblikama ne predstavlja posebnega izziva, saj sta samo posebna primera prvih dveh oblik. Podpora se bo izvedla v okviru orodja $sdl2pml$ s primernim preoblikovanjem vhodne specifikacije. Slika 3.26 prikazuje grafični simbol in tekstovni opis konstrukta.

Grafična predstavitev



Tekstovni zapis

<podstruktura> ::=
substructure <ime podstrukture>;
{<povezava kanalov in signalnih poti>
| <definicija signalov>
| <definicija signalnih list>
| <definicija procesov>
| <definicija signalnih poti>
| <definicija podatkov>
| <definicija procedur>}
endsubstructure [<ime podstrukture>];

Slika 3.26: EBNF zapis sintakse specifikacije podstrukture

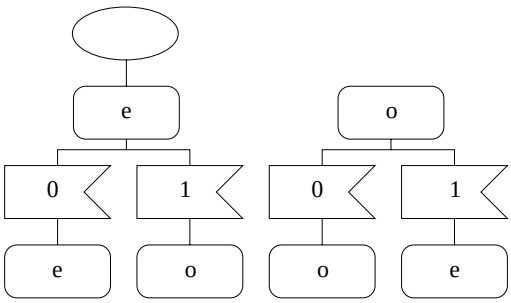
Ker Promela nima gradnikov, s katerimi bi se lahko modelirali bloki, je rezultat pretvorbe t. i. plosek opis sistema, saj je sestavljen samo iz procesov. S tem se izgubijo vsi strukturni elementi specifikacije. Ob tvorbi modela je potrebno ohraniti vse omejitve, ki jih določa struktura sistema. Zato je potrebno izvesti analizo strukture sistema in pridobiti vse elemente, ki jih določa definicija 3.20. Še posebna pozornost je potrebna pri zagotavljanju enakih območij veljavnosti za posamezne definicije podatkovnih tipov, procedur in primerno preslikavo komunikacijskih povezav.

Definicija 3.20 Blok opisujemo z $blok$, $blok \in spec.stru$, in je dvojica $(sdlime, ime)$, kjer je:

$sdlime$ ime bloka v specifikaciji sistema;
ime edinstveno ime bloka v strukturi sistema.♦

3.3.2 Proces

Proces opisujemo s telesom, ki predstavlja razširjeni končni avtomat stanj. Končni avtomat sestavljajo množica stanj, začetno stanje, vhodna abeceda in prehajalna funkcija. Slednja določa naslednje stanje sistema na osnovi trenutnega stanja in vhodnega simbola. Za zapis končnega avtomata v SDL se lahko uporabita samo konstrukta <vhod> in <stanje> (slike 3.27, 3.40, 3.35). Ob sprejetem signalu na vhodu se avtomat postavi v novo stanje. Naslednja stanja določa prehajalna funkcija.



$Q = \{e, o\}$
$q_0 = e$
$A = \{0, 1\}$
$f(e, 0) \to e, f(e, 1) \to o,$
$f(o, 0) \to o, f(o, 1) \to e.$

Slika 3.27: Primer zapisa končnega avtomata v SDL

Pri opisu večjih sistemov postanejo opisi končnih avtomatov zelo veliki in nepregledni. Hitri eksploziji števila stanj se pri opisu sistemov v jeziku SDL izognemo z definicijo razširjenega končnega avtomata stanj, ki vpelje lokalne spremenljivke. S tem opisi postanejo preglednejši, saj se večji del množice stanj predstavlja posredno z uporabo spremenljivk. Spremenljivka, ki v specifikaciji lahko zavzame 2^8 različnih vrednosti, predstavlja končni avtomat z 256 stanji. Katera stanja bodo predstavljena eksplicitno in katera implicitno z uporabo spremenljivk, ni določeno in je odvisno od pisca specifikacije. Za medsebojno sinhronizacijo med procesi, ki se izvajajo sočasno, je potrebno vpeljati dodatno razširitev — komunikacijo. Procesi komunicirajo s signali, ki predstavljajo množico zunanjih dogodkov. Komunikacija je podrobneje predstavljena v razdelku 3.2.

Definicija 3.21 Razširjen komunicirajoči končni avtomat je šesterica (Q, q_0, A, f, Z, X), kjer je:

Q množica stanj;
q_0 začetno stanje, $q_0 \in Q$;
A vhodna abeceda;
f prehajalna funkcija;
Z množica spremenljivk;
X izhodna abeceda.♦

Jezik SDL definira še dodatne konstrukte, ki vpeljejo naslednje razširitve:

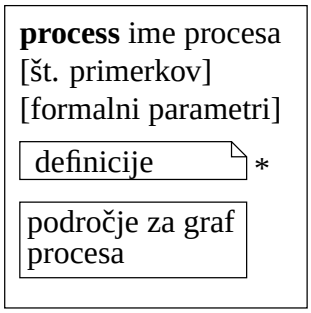
- vejitve — med prehodom se lahko na osnovi vrednosti vidnih spremenljivk izvajajo vejitve, ki se zaključujejo v različnih novih stanjih;

- spontani prehod — aktivira prehod tudi brez vhodnega dogodka, kar omogoča opis nedeterminističnega izvajanja;
- konstrukt <časovnik> — omogočeno je modeliranje časa;
- konstrukt <shrani> — izbrani vhodni signali se lahko shranijo za kasnejšo obdelavo.

Procesi predstavljajo jedro specifikacije sistema, saj definirajo njegovo delovanje. Med izvajanjem sistema se lahko vsak proces pojavi v več samostojnih in sočasno delujočih primerkih. Ob definiciji procesa se izbere privzeto število primerkov procesa ob zagonu sistema ter predvideno maksimalno število primerkov ob izvajanju sistema. V primeru, ko se proces ob izvajanju sistema pojavi v več primerkih, je obnašanje celotne skupine procesov definirano z enotnim opisom.

Proces je povezan z drugimi procesi v bloku ali okolici z eno ali več signalnimi potmi (slika 3.22), ki si jih delijo vsi primerki procesa. Definicije signalnih poti določajo signale, ki se lahko prenašajo med povezanimi procesi. Če se proces povezuje s procesi v okolici lastnega bloka, se signalne poti priključijo na kanale, ki povezujejo vpletena bloka. Signalne poti se lahko zaključujejo tudi v procesu samem [18, 58]. S tem je omogočena komunikacija med različnimi primerki procesa. Slika 3.28 prikazuje grafično in tekstovno sintakso specifikacije procesa.

Grafična predstavitev



Tekstovni zapis

```
<proces> ::=
    process <ime procesa>;
      [<število primerkov procesa>]
      [<seznam formalnih parametrov>]
      [<seznam veljavnih signalov>]
      {<definicija signalov>
      | <definicija signalnih list>
      | <definicija procedur>
      | <definicija podatkov>
      | <definicija spremenljivk>
      | <definicija signalnih poti>
      | <definicija časovnikov>}*
      <telo procesa>
    endprocess [<ime procesa>];
```

```
<formalni parametri procesa> ::= fpar <tip parametrov>{,<tip parametrov>}*
<tip parametrov> ::= <ime spremenljivke>{,<ime spremenljivke>}* <ime podatkovnega tipa>
<seznam veljavnih signalov> ::= signalset <seznam signalov>;
<število primerkov procesa> ::= ([<začetno število>] [, [<maksimalno število>]])
```

Slika 3.28: EBNF zapis sintakse specifikacije procesa

Procesi se na logičnem nivoju izvajajo sočasno in neodvisno. Sinhronizacija med procesi se izvaja izključno z medsebojno komunikacijo. Graf procesa je sestavljen iz stanj in prehodov med stanji. Prehod se izvede ob vhodnem dogodku, ki ga predstavlja sprejem signala. Ob zaključku prehoda se proces nahaja v naslednjem stanju in pričakuje naslednji

vhodni dogodek. Naslednje stanje je lahko enako izvornemu stanju. Definicija 3.22 opisuje vse elemente, s katerimi nedvoumno opišemo proces in predstavlja osnovo za primerno modeliranje v Promeli.

Definicija 3.22 Proces $proc$, $proc \in spec.stru$, je dvanajsterica $(sdlime, pmlime, fpar, kanal, sigset, stanje, sigin, sigout, časo, sprem, zač, maks)$, kjer je:

$sdlime$ ime procesa v specifikaciji;
$pmlime$ ime procesa v modelu sistema;
$fpar$ množica formalnih parametrov;
$kanal$ ime pridruženega kanala v modelu sistema;
$sigset$ množica eksplicitno in implicitno definiranih veljavnih signalov;
$stanje$ množica eksplicitno definiranih stanj;
$sigin$ množica signalov, ki jih proces ob sprejemu obravnava;
$sigout$ množica signalov, ki jih proces pošilja;
$časo$ množica časovnikov (definicija 3.11, str. 39);
$sprem$ množica spremenljivk;
$zač$ število primerkov ob zagonu sistema;
$maks$ maksimalno število dovoljenih primerkov med izvajanjem sistema.♦

Definicija 3.23 Formalni parameter $fpar$, $fpar \in spec.stru.proc$, je trojica $(sdlime, sdltip, pmltip)$, kjer je:

$sdlime$ ime formalnega parametra v specifikaciji;
$sdltip$ ime podatkovnega tipa, ki omejuje zalogo vrednosti parametra;
$pmltip$ ime podatkovnega tipa v modelu sistema.♦

Definicija 3.24 Eksplicitno definirano stanje $stanje$, $stanje \in spec.stru.proc$, je sedmerica $(pmlime, sigsta, trajen, odloč, konč, zvez, spon)$, kjer je:

$pmlime$ ime stanja;
$sigsta$ množica signalov, ki se lahko sprejmejo v stanju;
$trajen$ množica trajnih signalov, ki so pridruženi stanju;
$odloč$ množica konstruktov odločitev, ki so pridruženi stanju;
$konč$ boolova konstanta, ki zavzame vrednost 1, če sodi stanje v množico veljavnih končnih stanj procesa;
$zvez$ boolova konstanta, ki zavzame vrednost 1, če stanje vsebuje vhod zvezdica;
$spon$ boolova konstanta, ki zavzame vrednost 1, če stanje vsebuje spontani prehod.
♦

Definicija 3.25 Signal $sigsta$, $sigsta \in spec.stru.proc.stanje$, ki se lahko sprejme v obravnavanjem stanju, je četverica $(pmlime, shrani, prio, pogoj)$, kjer je:

$pmlime$ ime obravnavanega signala $\in spec.stru.proc.sigin$;
$shrani$ boolova konstanta, ki zavzame vrednost 1, če se signal v obravnavanem stanju nahaja v konstruktu shrani;
$prio$ boolova konstanta, ki zavzame vrednost 1, če se signal v stanju obravnava prioritetno;
$pogoj$ boolov izraz, ki določa omogočitveni pogoj, za sprejem signala. ♦

Definicija 3.26 Trajni signal $trajen$, $trajen \in spec.stru.proc.stanje$, je dvojica $(pogoj, prio)$, kjer je:

$pogoj$ boolov izraz, ki omogoči sprejem trajnega signala;
$prio$ celo število, ki določa prioriteto trajnega signala.♦

Definicija 3.27 Odločitev $odloč$, $odloč \in spec.stru.proc.stanje$, je trojica $(ime, vpraš, odgov)$, kjer je:

ime edinstveno ime konstrukta odločitev;
$vpraš$ vprašanje v konstruktu odločitev;
$odgov$ množica odgovorov.♦

Definicija 3.28 Preslikava parametrov obravnavanih signalov $sigin$, $sigin \in spec.stru.proc$, je dvojica $(pmlime, param)$, kjer je:

$pmlime$ ime obravnavanega signala;
$param$ zaporedno mesto parametrov signala v pridruženem kanalu. ♦

Definicija 3.29 Element množice signalov $sigout$, $sigout \in spec.stru.proc$, je dvojica $(pmlsig, proc)$, kjer je:

$pmlsig$ ime obravnavanega signala;
$proc$ množica procesov, ki lahko signal sprejmejo in do njih obstaja veljavna pot.♦

Definicija 3.30 Element množice procesov $proc$, $proc \in spec.stru.proc.sigout$, je dvojica $(pmlime, pot)$, kjer je:

$pmlime$ edinstveno ime procesa v modelu sistema;
pot veljavna pot do procesa v specifikaciji sistema.♦

Definicija 3.31 Spremenljivka $sprem$, $sprem \in spec.stru.proc$, je dvojica $(ime, pmltip)$, kjer je:

ime ime spremenljivke;
$pmltip$ ime podatkovnega tipa v modelu sistema.♦

Vsak proces se v Promeli modelira s procesnim tipom. Pridružena vhodna vrsta se predstavi s kanalom. Pravilno modeliranje SDL-stroja, ki omogoča izredno sproščen opis komunikacije, je kompleksna naloga. Izbiro naslovnika v [57] določa neposredni naslov procesa (PId, ime procesa) ali pa ga določajo komunikacijski konstrukti, ki izvorni proces povezujejo z okoljem — preostalimi procesi v okviru lastnega bloka, višjeležnimi bloki in oddaljenimi procesi. V množico oddaljenih procesov sodijo vsi procesi, ki se ne nahajajo v istem bloku kot obravnavani proces. Pri neposrednem naslavljanju se v konstruktu <izhod> (str. 97) v polju <naslov> zapiše edinstvena identifikacijska številka procesa (PId). V primerih, ko imamo v sistemu samo en primerek ciljnega procesa, se lahko uporabi tudi njegovo polno ime. Polno ime vsebuje tudi mesto procesa v strukturi sistema. Ko se naslavlja proces na istem hierarhičnem nivoju, torej v okviru istega bloka, se lahko uporabi tudi samo ime procesa. Kadar se proces pojavi v več primerkih, naslavljanje z imenom ne predstavlja neposrednega naslavljanja, saj SDL-stroj signal dostavi naključno izbranemu primerku. Podobno velja za primere naslavljanja z omejevanjem poti. Ta lahko nedvoumo določa proces

samo v primeru, ko ne obstaja več primerkov procesa. Pogoj za uporabo izbrane poti je, da se signal nahaja v seznamu prenašanih signalov ter da obstaja pot od izvornega procesa do izbrane prenosne poti. Če se ne navede niti ciljni proces niti predvidena pot, se signal dostavi naključno izbranemu procesu po naključno izbrani poti.

OG omejuje podano definicijo s prepovedjo uporabe posrednih poti. Razvijalec je tako ob uporabi rezervirane besede **via** omejen samo na signalne poti v okviru lastnega bloka. V primeru, ki je prikazan na sliki 3.22, bi se v okviru procesa P3 lahko v opisu poti uporabile vse signalne poti v bloku B1, prepovedana pa bi bila uporaba kanala, ki povezuje bloka B1 in B2. To prispeva k natančnejšemu opisu specifikacije sistema, kar posredno zmanjša verjetnost vnosa neželenega nedeterminizma ob komunikaciji [18], hkrati pa se omeji izraznost jezika SDL.

SDL dodatno definira rezervirano besedo **all**, ki se lahko zapiše na mesto naslovnika. S tem mehanizmom se lahko opisuje pošiljanje več naslovnikom hkrati. Zaporedje in izbira poti se določi z omejevanjem navedenih poti. Posamezna pot se lahko v seznamu navede večkrat. Signal se pošlje enkrat za vsako navedbo. Proces, ki signal dejansko sprejme, se določi nedeterministično iz množice vseh veljavnih procesov. Množico veljavnih procesov sestavljajo procesi, ki lahko obravnavani signal sprejmejo in do njih obstaja veljavna pot s pričetkom v navedeni poti. Poslan signal torej sprejme en in samo en proces. Pri tradicionalnem pojmovanju pošiljanja več naslovnikom hkrati bi poslan signal sprejeli vsi veljavni procesi. OG in $sdl2if$ ne podpirata pošiljanja več naslovnikom hkrati.

Pogoj za uspešno oddajo signala je, da obstaja neposredna ali posredna pot do ciljnega procesa ali skupine procesov, ki ima obravnavani signal v seznamu veljavnih signalov. Če to ni mogoče, se signal zavrže. Kadar lahko signal sprejme več procesov, se ta dostavi naključno izbranemu procesu. Celovita interpretacija specifikacije je v OG onemogočena z deterministično izbiro ciljnega procesa iz množice veljavnih prejemnikov [54]. Nedeterministične izbire prav tako ne podpirata orodji $sdl2if$ in $if2pml$. Prvo prevaja konstrukt <izhod> samo za primere, ko je ciljni proces edinstveno določen. Ker dinamična tvorba procesov ni podprta, je temu zadoščeno samo z uporabo imena procesa ali primernim omejevanjem poti, saj PId ni na voljo. Zapisano velja samo za primere, ko je v sistemu vedno aktiven samo en primerek procesa. V nasprotnem primeru, se v komentarju zapišejo rezultati statične analize, konstrukt pa se ne prevaja. Rezultati analize predstavljajo osnovo za tvorbo modela z orodjem $if2pml$. V modelu je uporabljen samo prvi iz množice veljavnih naslovnikov. Rezultat takšne pretvorbe je okrnjen model sistema, ki obravnava samo podmnožico vseh možnih izvajanj sistema. V primeru, ko množico veljavnih naslovnikov tvorijo primerki istega procesa, pa komunikacija ni podprta. S tem se bistveno poenostavi tvorba modela sistema, saj se posamezen signal vedno pošilja enemu in samo enemu procesu.

Za celovito modeliranje komunikacijskega dela SDL-stroja je potrebno obravnavati vse trenutno aktivne primerke procesov. Da je komunikacija med procesi mogoča, moramo poznati edinstveno številko pridruženega kanala in njegovo strukturo za vse veljavne procese.

V primeru, ko ciljni proces ni nedvoumno določen, je potrebno izvršiti nedeterministično izbiro iz množice veljavnih procesov.

Za polno podporo komunikacije med procesi je potrebno zagotoviti:

- dinamično tvorbo procesov,
- modeliranje vhodne vrste,
- analizo potencialnih prejemnikov posameznih signalov.

Modeliranje dinamične tvorbe procesov z operatorjem run je predstavljeno v razdelku 3.4.12 (str. 103). Vsak proces v sistemu lahko sprejme edinstven nabor signalov. Zato v okviru algoritma 3.13 za vsak proces pripravimo pridružen kanal in definiramo preslikavo vseh parametrov signalov. Ob pošiljanju istega signala različnim naslovnikom je tako potrebno izvesti primerno preslikavo vseh parametrov signala glede na pridružen kanal izbranega naslovnika. Analiza vseh potencialnih naslovnikov se izvede za vsak signal, ki ga lahko posamezen proces pošlje v okviru svojega telesa (algoritem 3.29, str. 98).

Signale, ki jih lahko proces prejme, določajo priključene signalne poti in eksplicitno definiran seznam veljavnih signalov. Zaradi optimalnega modeliranja pridruženega kanala je potrebno izvesti natančno analizo vseh signalov, ki jih lahko posamezen proces obdela. Na osnovi pridobljenih podatkov se definira kanal z minimalnim številom potrebnih elementov, ki lahko zadostijo prenosnim potrebam vseh signalov. Dodatno se izvede analiza lastnosti posameznih signalov v eksplicitno definiranih stanjih sistema. Signal se v posameznem stanju lahko obravnava prioritetno, se shrani za kasnejšo obravnavo, pri pogojnem sprejemu pa se mu pridruži boolov izraz, ki opisuje pogoj za sprejem signala.

Inicializacija vseh začetnih procesov se izvede v okviru procesa $init()$ (algoritem 3.14). Začetni proces je proces, ki ima začetno število primerkov procesa večje od nič. V začasni spremenljivki i se hrani maksimalno število možnih primerkov procesov v sistemu. Ob inicializaciji se upošteva osnovni proces $init()$, ki je ob uporabi v modelu sistema vedno aktiven že ob zagonu, in dodatni prikriti proces, ki se uporabi ob formalni verifikaciji.

Algoritem 3.13 opisuje analizo procesov, obravnavanih signalov in tvorbo pridruženih kanalov. Prva zanka z oznako R1 zagotavlja, da se izvede analiza vseh procesov v specifikaciji sistema. V naslednjih korakih se iz definicije procesa pridobi ime procesa in seznam veljavnih signalov. Sledi analiza definicije začetnega in maksimalnega števila procesov. Kadar nista definirani, se za obe uporabi vrednost ena. Začetno število določa število aktivnih procesov ob zagonu sistema. Pri modeliranju je pomembnejše največje število dovoljenih aktivnih primerkov procesa, saj določa velikost pomožnih elementov v modelu sistema. Zapiše se z makrojem v naslednjem koraku algoritma. Vsakemu procesu se pridruži tudi skrita spremenljivka, ki hrani podatek o trenutnem številu aktivnih procesov. V konstruktu ustvari mora biti vrednost te spremenljivke vedno manjša od največjega dovoljenega števila primerkov, sicer model sistema ni skladen s semantiko jezika SDL. V takšnem primeru se nov primerek procesa ne sme ustvariti. Skrite spremenljivke se ne hranijo v vektorju stanj, zato ne prispevajo h kompleksnosti formalne verifikacije.

```
START ;
 TASK i := 3
 R1:
  DECISION ∃a: a ∈ spec.stru.proc;
   ( TRUE ):
   ELSE:
    JOIN R5;
  ENDDECISION;
  TASK a.sdlime := <ime procesa>;
  TASK a.pmlime := pot()//a.sdlime;
  TASK dodaj( a.sigset,
                <seznam veljavnih signalov>);
  DECISION ∃b : b ∈<število primerkov procesa>,
                b = <začetno število>;
   ( TRUE ):
    TASK a.zač := b;
   ELSE:
    TASK a.zač := 1;
  ENDDECISION;
  DECISION ∃c : c ∈<število primerkov procesa>,
                c =<maksimalno število>;
   ( TRUE ):
    TASK a.maks := c;
    TASK "#define table_//a.pmlime//_max //c";
    TASK "hidden byte table_//a.pmlime//_free;";
   ELSE:
    TASK a.maks := 1;
    TASK "#define table_//a.pmlime//_max 1";
    ENDDECISION;
    DECISION ∃d: d ∈ spec.kon : d =<vhod> ∨
                 ∨ <shrani> ∨ <prioritetni vhod>;
     ( TRUE ):
      TASK a.kanal = chan_//a.pmlime//[//a.maks//];
      R2:
       DECISION ∃d: d ∈ spec.kon : d =<vhod> ∨ <shrani>
                    ∨ <trajen signal> ∨ <prioritetni vhod>;
        ( TRUE ):
         TASK g ∈ spec.stru.proc.stanje;
         TASK g = stanje(pot());
         TASK g.konč = končno(g);
         DECISION d ∈<trajen signal>;
          ( TRUE ):
           TASK d.pogoj= <boolov izraz>;
           TASK d.prio = <celoštevilčni literal>;
           TASK dodaj(g.trajen, d);
           JOIN R2;
          ELSE:
         ENDDECISION;
        R3:
         DECISION ∃e : ( (e ∈ (d ∧ (spec.kom.sig)) ∨
                         ((e ∈ d) ∧ e.sdlime = *) ∧
                         (e.sdlime =<ime signala>))
         ∨ ((e ∈ a.časo) ∧ (e.sdlime =<ime časovnika>));
          ( TRUE ):
           DECISION e.sdlime = NONE;
            ( TRUE ):
             TASK g.spon = TRUE;
            ELSE:
           ENDDECISION;
           DECISION e.sdlime = *;
            ( TRUE ):
             TASK g.zvez = TRUE;
            ELSE:
           ENDDECISION;
           TASK e.pmlime:=poišči(e.sdlime, (spec.kom.sig));
DECISION d ∈<prioritetni vhod>;
 ( TRUE ):
  TASK h ∈ spec.stru.proc.stanje.sigsta;
  TASK h.pmlime := e.pmlime ;
  TASK h.prio := TRUE ;
  ELSE:
   DECISION d ∈<shrani>;
    ( TRUE ):
     TASK h.pmlime := e.pmlime ;
     TASK h.shrani := TRUE;
     ELSE:
      TASK h.pmlime := e.pmlime ;
      DECISION ∃j : j ∈<omogočitveni pogoj>
       ( TRUE ):
        TASK h.pogoj := j;
       ELSE:
        TASK h.pogoj := TRUE;
      ENDDECISION;
     ENDDECISION;
    ENDDECISION;
    TASK dodaj(a.g, h);
   JOIN R3;
  ELSE:
   JOIN R2;
  ENDDECISION;
 ELSE:
  TASK k := podmožica(a.sigin);
 R4:
  DECISION ∃f : f ∈ a.sigin;
   ( TRUE ):
    TASK preslikaj(f.param, k);
    TASK dodaj(a.sigin, f);
    JOIN R4;
   ELSE:
  ENDDECISION;
  TASK "#define chan_//a.kanal//_select //i";
  TASK "pt_chan a.kanal = [cv_buff] of
                {pt_sig,pt_pid,//k//};";
 TASK i := i + a.maks;
 ENDDECISION;
ELSE:
ENDDECISION;
JOIN R1;
R5:
 TASK "#define pt_pid pid";
 TASK "#define pt_chan chan";
 TASK "#define pt_select byte";
 TASK "#define pt_sig byte";
 TASK "#define cv_procnum //i";
 TASK okolje();
 TASK "typedef pt_chandat { pt_sig name;
                bool save; bool prio};"
 TASK "typedef pt_chandata { pt_chandat
                data[cv_buff]};"
 TASK "pt_chan table_channame_channum[cv_procnum];";
 TASK "pt_chan table_pid_channum[cv_procnum];";
 TASK "pt_sig table_pid_channame[cv_procnum];";
 TASK "byte table_channum_ptr[cv_procnum];";
 TASK "pt_chandata
          table_channum_nsp[cv_procnum];";
 TASK "bool table_channum_prio[cv_procnum];";
 TASK "pt_chan table_channame_channum[cv_procnum];";
 TASK "bit error=false;";
 END ;
```

Algoritem 3.13: Algoritem za analizo definicije procesa, vhodnih konstruktov in pripravo pridruženega kanala

Če največje dovoljeno število ni navedeno v definiciji procesa, algoritem odstopa od standarda. Skladno z [57] v takšnem primeru število aktivnih primerkov procesa ni omejeno. Zaradi zahteve po končnem modelu je potrebno to število omejiti. V mnogih specifikacijah je smiselno v takšnem primeru število dovoljenih primerkov procesov omejiti na ena. Podana interpretacija tvori minimalni model sistema in s tem učinkovitejšo verifikacijo. Zaradi odstopanja od standarda se v tem primeru ob tvorbi modela z orodjem $sdl2pml$ izpiše opozorilo.

Dodatno omejitev predstavlja orodje Spin, saj omejuje število aktivnih procesov na 256. Pri tvorbi modela se pri prekoračitvi tega števila izpiše obvestilo, omejitev pa se, zaradi morebitne podpore večjega števila procesov v prihodnosti, ne upošteva.

V naslednji odločitvi se preveri, ali telo procesa vsebuje kakšen vhodni konstrukt. Če ga ne vsebuje, se procesu ne pridruži kanal, izvajanje algoritma pa se nadaljuje pri oznaki R1. Takšni procesi zahtevajo ločeno obravnavo tudi ob tvorbi s konstruktom ustvari. V nasprotnem primeru, se kanalu dodeli ime s predpono `chan__`. V okviru zanke z oznako R2 sledi analiza vseh konstruktov vhod, spontani vhod, shrani in prioritetni vhod. V začasno spremenljivko se shrani kazalec na stanje $\in spec.stru.proc.stanje$, ki se pridobi na osnovi trenutnega mesta v specifikaciji ($pot()$). Orodje $sdl2pml$ podpira razširitev jezika SDL z dodatno oznako stanja v okviru komentarja. S pomočjo funkcije $končno()$ se preveri, ali je obravnavano stanje v množici veljavnih končnih stanj procesa. Pomen in obravnava veljavnih končnih stanj sta opisana v poglavju 4. V naslednjem odločitvenem stavku se preveri, ali ime signala predstavlja rezervirana beseda NONE, ki opisuje spontani prehod. Stanje, ki vsebuje spontane prehode, se dodatno označi zaradi posebne obravnave v algoritmu 3.19. Algoritem enakovredno obravnava signale in časovnike, ki jih modeliramo s signali (razdelek 3.4.9).

V primeru skupne definicije množice signalov v okviru enega vhodnega konstrukta se izvede analiza za vse signale. Analiza signalov je opisana v zanki z oznako R3. Vhod zvezdica (razdelek 3.4.5, str. 83) se tu obravnava enakovredno z drugimi signali. Dejanska množica signalov, ki jih predstavlja, se določi ob modeliranju sprejema v okviru algoritma 3.20.

Med analizo je potrebno ločiti med signali in časovniki, ki jih proces dejansko obravnava, in signali, ki so procesu pridruženi eksplicitno v okviru definicije <veljavni signali> ali implicitno z definicijo v okviru signalnih poti. Pridružen kanal se tvori samo za tiste signale in časovnike, ki jih proces lahko sprejme. Ostali veljavni signali se v modelu sistema zavržejo že na strani pošiljatelja (algoritem 3.29).

V okviru analize se signal ali časovnik pridruži procesu ($\in spec.stru.proc.sigin$) in vsem stanjem ($\in spec.stru.proc.stanje.sigsta$), v katerih se lahko obravnava. Ločena obravnava signala v posameznih stanjih sistema je potrebna zato, ker se lahko signal v različnih stanjih pojavlja v okviru različnih vhodnih konstruktov. V vsakem stanju se zabeleži, ali signal zahteva posebno obravnavo. Ta je potrebna v primeru uporabe prioritetnega

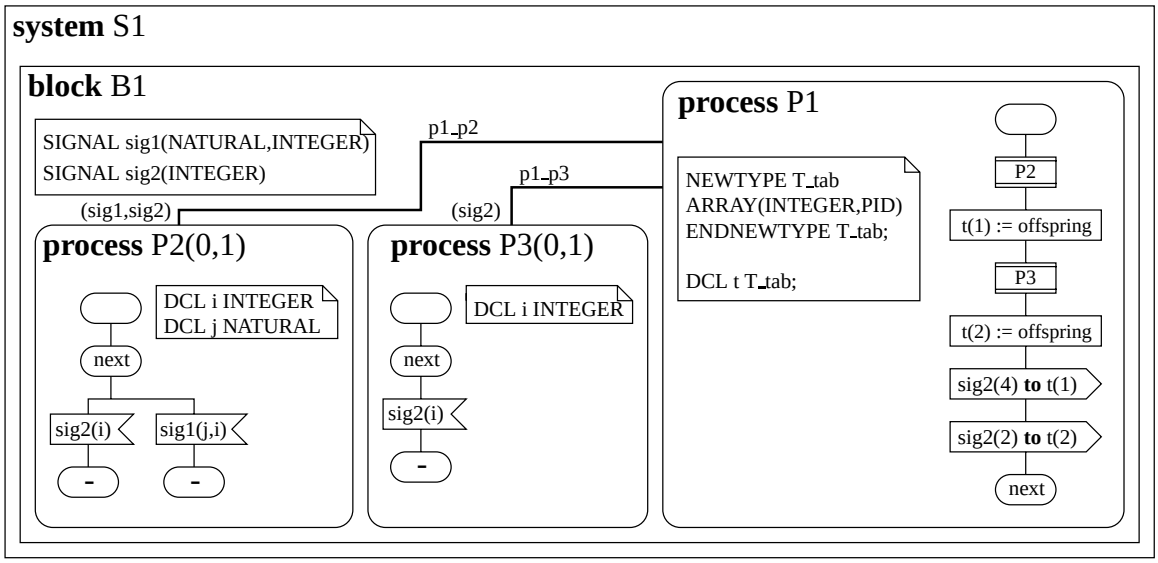
vhoda, konstrukta shrani ali omogočitvenega pogoja. Analiza se izvede v okviru naslednjih treh odločitev.

Po zaključku analize vseh signalov, ki jih proces lahko sprejme, se s pomočjo funkcije $podmnožica()$ pridobi najmanjša skupna podmnožica vseh elementov, ki so definirani v okviru obravnavanih signalov. Ta množica določa definicijo pridruženega kanala. Ker ima lahko vsak signal drugačno zaporednje formalnih parametrov, se v zanki z oznako R4 s pomočjo funkcije $preslikaj()$ dodeli vsakemu parametru obravnavanega signala zaporedna številka v pridruženem kanalu. Rezultat analize je množica preslikav med zaporednim mestom parametra v signalu in definiranim pridruženim kanalom. Ob zaključku analize procesa se definira tudi pridružen kanal s privzetim številom rež in pridobljenimi parametri. Prvi in drugi parameter pridruženega kanala sta vedno ime signala in PId pošiljatelja.

Zadnja veja algoritma z oznako R5 se izvede ob zaključku analize procesov. Definicija podatkovnih tipov `pt_pid` in `pt_chan` bi lahko bila odvisna od dejanskega števila procesov v modelu sistema, vendar je za primere, ko število procesov in kanalov preseže 256, potrebno izvesti razširitev Spina. Zato sta omenjena podatkovna tipa definirana kot sinonima podatkovnih tipov `pid` in `chan`. V naslednjem koraku se definira konstanta `cv_procnum`, ki predstavlja največje predvideno število aktivnih procesov, ki ga dopušča specifikacija sistema.

Praviloma se pričakuje, da je specifikacija, ki jo želimo formalno verificirati, zaprte narave — da opisuje tudi okolje sistema. Za posebne primere, ko želimo okolje sistema predstaviti z lastnim modelom v Promeli, se za komunikacijo z okoljem definira tudi kanal za te potrebe. To je v algoritmu predstavljeno s funkcijo $okolje()$. Potrebni podatki se pridobijo na osnovi analize kanalov, ki povezujejo sistem z okoljem sistema.

Zaradi razlik med SDL- in Promela-strojem je potrebno zagotoviti tudi primerno preslikavo med PId-i in številkami in imeni pridruženih kanalov. Dodatno je potrebno izvajati dinamično analizo vseh signalov v pridruženih kanalih. V specifikaciji se procesi neposredno naslavljajo z uporabo PId-a, v modelu sistema pa se naslavlja pridružen kanal. Primerno preslikavo med PId-i in številko pridruženega kanala zagotavlja polje `table_pid_channum[]`, ki je dostopno vsem procesom in se osveži ob vsaki tvorbi procesa. Velikost polja določa maksimalno število procesov, kot indeks pa se uporabi PId procesa. Podobno preslikavo nudi polje `table_pid_channame[]`, ki beleži preslikavo med PId-om procesa in konstanto, ki predstavlja njegovo ime. Tabeli sta potrebni pri modeliranju pošiljanja signala na naslov, ki ga določa izraz tipa PId (primer na sliki 3.30, str. 67). Preostali elementi se uporabijo pri modeliranju prioritetnega vhoda in konstrukta shrani. Polje `table_channum_ptr[]` zagotavlja preslikavo med številko kanala v modelu sistema in kazalcem na prvo prosto mesto kanala. Definirana podatkovna tipa sta potrebna za deklaracijo dvodimenzionalnega polja `table_channum_nsp`, ki hrani dodatne oznake za vse signale v pridruženih kanalih. Oznake se določajo dinamično med izvajanjem modela ob



Slika 3.29: Primer dinamične izbire naslovnika

analizi vhodne vrste glede na trenutno eksplicitno stanje posameznega procesa. Podrobnejši opis je podan v razdelku 3.4.3 (str. 77).

S skupino procesov se sklicujemo na proces SDL, ki se lahko med izvajanjem sistema pojavi v več primerkih. V modelu se skupina procesov predstavi s procesnim tipom in poljem pridruženih kanalov. Procesni tip ima natančno en pridružen kanal tudi v primeru, ko lahko proces sprejme več deset različnih signalov z različnimi parametri. Če se podatkovni tipi v posameznih signalih ponavljajo, se v pridruženem kanalu predstavijo z najmanjšo potrebno množico. Prednost takšnega pristopa je optimalna poraba pomnilnika in varčevanje s kanali, katerih število je v različici 4.1.1 orodja Spin pri formalni verifikaciji sistema omejeno na 256. Zaradi opisanega postopka imajo pridruženi kanali praviloma različno strukturo — zaporedje podatkovnih tipov parametrov signalov. Pri neposrednem naslavljanju s PId-om je tako potrebno, poleg številke kanala, poznati tudi njegovo strukturo.

Takšen sistem opisuje specifikacija na sliki 3.29, saj se signal `sig2` pošlje na osnovi dinamično pridobljenega PId-a. S pomočjo primera in njegovega modela bomo dodatno predstavili algoritem 3.13 in opisali dinamično naslavljanje z uporabo izraza tipa PId. Slika 3.30 prikazuje poenostavljen primer modela sistema. Ker specifikacija vsebuje samo osnovni obliki konstruktov izhod in vhod, model ne vsebuje dodatnih elementov za podporo prioritetnim signalom, pogojnemu sprejemu ter konstruktu shrani. Kadar se opis nanaša na posamezno vrstico v modelu sistema, je ta navedena v oklepaju.

Primer opisuje sistem z enim blokom in tremi procesi. Ob analizi procesa se najprej določi edinstveno ime v modelu sistema in največje predvideno število primerkov procesa ($maks$). Proces P1 nima definiranega začetnega in maksimalnega števila primerkov, zato se mu po privzetem dodeli za obe vrednosti ena (37). Ker v njegovem telesu ne obstaja konstrukt vhod, se mu ne dodeli pridružen kanal. Procesoma P2 in P3 se dodeli po en pridružen kanal, saj definicija predpisuje, da bo v sistemu aktiven največ en primerek posameznega

procesa (39, 41). Dodatno se tvorijo kazalci na prosto mesto v polju kanalov (38, 40, 42), ki hkrati hranijo število trenutno aktivnih primerkov procesa. Ime kanala je enako imenu procesa s predpono `chan_` (45, 49). Zaradi poenotenega klica makroja `run_proc` tudi za procese brez pridruženega kanala se definira kanal `chan_none` (44), ki ima samo en element tipa bit. V obravnavanem primeru ima polje kanalov v obeh primerih samo en element. Imenu kanala se dodeli edinstvena številka v modelu sistema (48, 51). Potrebno je ločiti med številko kanala, ki jo kanalu ob inicializaciji dodeli Promela-stroj in se hrani v tabeli `table_pid_channum[]`, ter številko imena kanala, ki se definira ob analizi procesa in se hrani v tabeli `table_pid_channame[]`.

Z uporabo funkcije $poišči()$ pridobimo edinstveno ime signala glede na območje veljavnosti. Analiza signalov se je izvedla predhodno v okviru algoritma 3.8 (str. 46). Edinstveno ime obeh signalov se doda v množico obravnavanih signalov. V naslednjem koraku funkcija $podmnožica()$ opravi analizo parametrov za vse signale, ki so v množici obravnavanih signalov. Podatki o parametrih se nahajajo v množici $spec.kom.sig.param$ in se pridobijo v okviru algoritma 3.8.

Ob tvorbi modela se procesoma P2 in P3 pridružita kanala z različnimi elementi. Kanal, ki je pridružen procesu P2, ima na tretjem mestu `comm_natural` in na četrtem `int` (45-47). Pridružen kanal procesa P3 ima na tretjem mestu `int` (49-50). Definicije makrojev, ki predstavljajo preslikavo med podatkovnimi tipi, niso del algoritma 3.13. V primeru so dodane na osnovi analize, ki je bila predstavljena v algoritmih 3.1 in 3.4 (str. 27 in 34). Najmanjšo množico vseh parametrov tako sestavljata `comm_natural` in `int`. V nadaljevanju algoritma se s funkcijo $preslikaj()$ shranijo podatki o preslikavi med dejanskimi parametri signala in pridruženega kanala. To potrebujemo pri modeliranju konstrukta <izhod>, kjer na osnovi podatkov o ciljnem procesu in imenu signala zapišemo dejanske parametre signala na primerna mesta v pridruženem kanalu ciljnega procesa. Za signal `sig1` se zabeleži preslikava z zaporednimi števili 2 in 3, za signal `sig2` pa s 3. Prvi dve mesti sta vedno rezervirani za ime signala in PId od pošiljatelja. Skupno število elementov pridruženega kanala je torej štiri. V okviru zadnjega dela algoritma se definirata še podatkovna tipa `pt_pid` (4) in `pt_chan` (6) ter pomožni polji `table_pid_channum[]` (53-54) in `table_pid_channame[]` (55-56).

Inicializacija sistema se izvaja v okviru procesa $init$ (58-63), ki je, skladno z definicijo Promela-stroja, aktiven ob pričetku izvajanja modela. V okviru atomarne operacije se izvede tvorba primernega števila primerkov posameznih procesnih tipov in pridružitev kanalov (60-62). Algoritem 3.14 opisuje tvorbo začetnih procesov, ki je natančneje predstavljena v razdelku 3.4.12 (str. 103), v primeru pa se modelira z makrojem `run_proc`(13-35). S klicem makroja (60-62) se tvori edini začetni proces v sistemu — P1. Kot parametri makroja se zapišejo število trenutno aktivnih primerkov procesa, največje dovoljeno število primerkov procesa, ime procesnega tipa, ime pridruženega kanala in njegova edinstvena številka.

```
1 #define null 0
2 #define cv__buff 3
3 #define cv__procnum 5
4 #define pt__pid pid
5 #define pt__sig byte
6 #define pt__chan chan
7 #define pt__select byte
8 #define comm__natural byte
9 #define sig1 1
10 #define sig2 2
11 bit error = false;
12
13 #define run__proc(free,max,name,\
14                  channel,select)\
15  atomic{\
16  if\
17  :: free < max ->\
18     if\
19     :: channel == chan__none -> \
20        offspring = run name(_pid)\
21     :: else ->\
22       offspring = run\
23         name(channel,_pid);\
24       table_pid_channum[offspring] =\
25         channel;\
26       table_pid_channame[offspring] =\
27         select\
28     fi;\
29     if\
30     ::(offspring==0) -> error = true\
31     ::(offspring!=0) -> free++\
32     fi\
33  :: else -> error = true\
34  fi\
35  }
36
37 #define table__P1__max 1
38 byte table__P1__free
39 #define table__P2__max 1
40 byte table__P2__free
41 #define table__P3__max 1
42 byte table__P3__free
43
44 pt__chan chan__none = [0] of {bit}
45 pt__chan chan_P2[1] = [cv__buff]
46  of {pt__sig,pt__pid,
47      comm__natural,int};
48 #define chan__P2__select 3
49 pt__chan chan_P3[1] = [cv__buff]
50  of {pt__sig,pt__pid,int};
51 #define chan__P3__select 4
52
53 pt__chan
54  table_pid_channum[cv__procnum];
55 pt__sig
56  table_pid_channame[cv__procnum];
57
58 init{
59  pt__pid offspring;
60  run__proc(table__P1__free,
61            table__P1__max,P1,
62            chan__none,null);
63 }
64
65 proctype P1(pt__pid parent){
66 pt__pid offspring;
67 pid t[3];
68
69 atomic{
70 run__proc(table__P2__free,table__P2__max,P2,
71  chan_P2[table__P2__free],chan__P2__select);
72 t[1] = offspring;
73 run__proc(table__P3__free,table__P3__max,P3,
74  chan_P3[table__P3__free],chan__P3__select);
75 t[2] = offspring;
76 if
77 :: table_pid_channame[t[1]] ==
78       chan__P2__select ->
79         table_pid_channum[t[1]]!
80         sig2(_pid,null,1)
81 :: table_pid_channame[t[1]] ==
82      chan__P3__select ->
83         table_pid_channum[t[1]]!
84         sig2(_pid,1);
85 fi;
86
87 if
88 :: table_pid_channame[t[2]] ==
89      chan__P2__select ->
90        table_pid_channum[t[2]]!
91        sig2(_pid,null,2)
92 :: table_pid_channame[t[2]] ==
93      chan__P3__select ->
94        table_pid_channum[t[2]]!
95        sig2(_pid,2);
96 fi;
97 }
98 }
99
100 proctype P2(pt__chan input; pt__pid parent){
101  pt__pid offspring;
102  pt__pid sender;
103  xr input;
104  int i;
105  byte j;
106
107 next: end_next:
108 do
109 :: input?sig1(sender,j,i)->atomic{goto next;}
110 :: input?sig2(sender,_,i)->atomic{goto next;}
111 od
112 }
113
114 proctype P3(pt__chan input;pt__pid parent){
115 pt__pid offspring;
116 pt__pid sender;
117 xr input;
118 int i;
119 next: end_next:
120 do
121 :: input?sig2(sender,i)->atomic{goto next;}
122 od
123 }
124
125 active proctype monitor(){
126 end: atomic{  ! ((error == false)
127         || (_nr_pr<cv__procnum))->
128            assert(false)
129 }
130 }
```

Slika 3.30: Model sistema na sliki 3.29

```
START ;
 TASK "init{";
 TASK "pt_pid offspring;";
 TASK "atomic{";
R1:
 DECISION ∃a : a ∈ spec.stru.proc
  ( TRUE ):
   TASK i := a.zač;
  R2:
   DECISION i >= 1;
    ( TRUE ):
     TASK free := table_//a.pmlime//__free;
     TASK max := table_//a.pmlime//__max;
     TASK select := table_//a.pmlime//__select;
     TASK chan := chan_//a.pmlime//[//free//];
     TASK "if";
     TASK "::  free < max − >";
     TASK "   if";
     TASK "   ::  channel == chan_none −>";
     TASK "       offspring = run";
     TASK "       a.pmlime//(_pid)"
     TASK "   ::  else −>";
     TASK "       offspring = run ";
     TASK "     a.pmlime//(//chan//,_pid,//a.fpar//);";
     TASK "     table_pid_channum[offspring] ="
     TASK "     chan;";
     TASK "     table_pid_channame[offspring] ="
     TASK "     select";
     TASK "   fi;";
     TASK "   if;";
     TASK "   ::  (offspring==0) −> error = true";
     TASK "   ::  (offspring!=0) −> free++";
     TASK "   fi";
     TASK "::  else −> error = true";
     TASK "fi";
     TASK i = i − 1;
     JOIN R2;
    ELSE:
   JOIN R1;
  ENDDECISION;
 ELSE:
  TASK "}}";
  END:
ENDDECISION;
```

Algoritem 3.14: Algoritem za inicializacijo modela sistema

V Promeli število primerkov posameznega procesa ni omejeno ob definiciji. Poskus tvorbe novega primerka procesa je neuspešen le v primeru, ko je doseženo maksimalno dovoljeno število aktivnih procesov v modelu sistema. Spin to število omejuje na 256. Zato se v algoritmu 3.13 (str. 62) za vsak proces definira sinonim s pripono `_max`, ki predstavlja največje dovoljeno število primerkov procesa (37, 39, 41).

Proces P1 v prvem prehodu tvori procesa P2 (70-71) in P3 (73-74). PId-a od novih procesov se zapišeta v definirano polje (72, 75) in se kasneje uporabita v naslovu konstrukta <izhod> (razdelek 3.4.11). Ob analizi konstruktov <izhod> v telesu procesa P1 ni mogoče ugotoviti, kateremu procesu se bo poslal signal `sig2`. Zato je potrebno modelirati pošiljanje za vse veljavne naslovnike in pripraviti pravilno preslikavo med parametri signala `sig2` in pridruženega kanala procesov P2 in P3.

Ob statični analizi specifikacije in tvorbi modela lahko z analizo konstrukta <izhod> in komunikacijskih konstruktov pridobimo množico vseh potencilnih naslovnikov (algoritem 3.29). Kadar se ob izvajanju sistema naslovnik določi na osnovi izraza tipa PId, je potrebno zagotoviti, da se signal pošlje v pravilen pridružen kanal. Rešitev se modelira tako, da se pripravi pošiljanje signala za vse potencialne naslovnike (76-85, 87-96). Stražarji vršijo primerjavo med zapisom v tabeli `table_pid_channame[]` in številko imena kanala, za katerega je pripravljen posamezen izhod. Tako se zagotovi, da se signal pošlje v pridružen kanal procesa s PId-om iz naslova.

Alternativni pristop predstavlja enoten kanal za celoten model sistema. Tako bi vsi procesi imeli enak pridružen kanal in težav s poznavanjem strukture kanala ne bi bilo. V tem primeru bi večina inicializiranih kanalov neoptimalno zasedala pomnilnik, kar bi bistveno

povečalo potreben spominski prostor za analizo istega sistema. To velja še posebno v primerih, ko se ob doslednemu omejevanju zaloge vrednosti vseh spremenljivk v sistemu definira mnogo uporabniških podatkovnih tipov.

Delovanje opisanega sistema smo tudi simulirali in formalno verificirali. V okviru procesa `monitor` (125-130) preverjamo, ali lahko pride med tvorbo procesov do kršitve semantičnih pravil SDL-stroja. V definiciji makroja `run_proc` se v takšnem primeru postavi globalna spremenljivka `pv_run_time_error` na boolovo vrednost 1. Dodatno preverjamo, ali dejansko število procesov v sistemu lahko preseže število, ki ga predpisuje specifikacija. Pri izčrpni verifikaciji z orodjem Spin smo dodatno preverili, kateri stavki se v sistemu nikoli ne izvršijo in ali se izvajanje katerega procesa zaključi v prepovedanem stanju. Skladno s pričakovanji so rezultati pokazali, da se pri tvorbi procesa P1 nikoli ne izvede pridružitev kanala. Prav tako se ne izvedeta vrstici 83 in 90, kar potrjuje, da se nikoli ne izbere napačen pridružen kanal.

S tem smo zaključili z osnovno analizo procesa. Nadaljujemo s predstavitvijo algoritma 3.15, ki podaja pravila za tvorbo procesnega tipa v modelu sistema. V prvem odločitvenem stavku preverimo, ali še obstaja kakšna neobdelana definicija procesa. Sledi določitev edinstvenega imena v modelu sistema. V okviru zanke z oznako R2 se izvede analiza formalnih parametrov procesa. Za vsak formalni parameter se shrani ime lokalne spremenljivke in pridružen podatkovni tip. Ker lahko v strukturi sistema pride do večkratne definicije podatkovnega tipa, se poleg imena iz specifikacije sistema shrani tudi $pmlime$, ki nedvoumno določa podatkovni tip v modelu sistema. Pridobimo ga z uporabo funkcije $poišči()$ na osnovi rezultatov preteklih analiz. Naslednji odločitveni stavek preveri, ali je morebiti kateri izmed formalnih parametrov zapis. Tega Promela ne podpira, zato se v tem primeru izvajanje algoritma predčasno zaključi z rezervirano besedo STOP.

Sledi tvorba modela z definicijo procesnega tipa in lokalnih spremenljivk. Te v modelu hranijo vrednosti, ki jih SDL-stroj implicitno pridruži vsakemu procesu:

- *self* — lastni naslov, torej številka PId,
- *sender* — naslov procesa, ki je poslal nazadnje sprejeti signal,
- *offspring* — naslov zadnjega procesa, ki ga je proces tvoril in
- *parent* — naslov procesa, ki je ta proces tvoril.

V modelu sistema so eksplicitno definirane spremenljivke, ki hranijo podatke o roditelju procesa ter zadnjem potomcu in pošiljatelju. Spremenljivka $self$ je v modelu sistema predstavljena implicitno s $_pid$, $parent$ pa je deklariran kot formalni parameter procesnega tipa. Pridruženemu kanalu se določi ime $input$ in je prav tako zapisan kot formalni parameter. Dejanski kanal izbere model SDL-stroja ob tvorbi procesa. Ker iz tega kanala bere podatke samo en proces, se lahko označi z xr, kar omogoča dodatno optimizacijo algoritmov za formalno verifikacijo sistema.

```
START ;
R1:
 DECISION ∃a: a ∈ spec.stru.proc;
  ( TRUE ):
  ELSE:
    END;
 ENDDECISION;
 TASK a.sdlime := <ime procesa>;
 TASK a.pmlime := pot()//sdlime;
R2:
 DECISION ∃b : b ∈ proc.fpar;
  ( TRUE ):
   TASK a.b.sdlime := <ime spremenljivke>;
   TASK a.b.sdltip := <ime podatkovnega tipa>;
   TASK a.b.pmltip := poišči((spec.def{stype
                        ∨ ntype ∨ ptype}));
   TASK shrani(a.b);
   JOIN R2;
  ELSE:
 ENDDECISION;
 DECISION ∃k : k ∈ (a.fpar),
               k.pmltip ∈ spec.def.ntip,
               k.pmltip = zapis;
  ( TRUE ):
   STOP;
  ELSE:
 ENDDECISION;
 TASK "proctype a.pmlime (pt_chan input;
               pt_pid parent; a.fpar){";
 TASK "pt_pid offspring,sender;
 DECISION a.sigin ≠ ∅
  ( TRUE ):
   TASK "pt_pid sender;
   TASK "xr input;
   TASK "byte pv_ptr, pv_cur;"
  ELSE:
 ENDDECISION;
R3:
 TASK i := i + 1;
 DECISION ∃m : m ∈ (a.časo)
  ( TRUE ):
   TASK "timer tmr__//m.sdlime";
   TASK j := indeks();
   TASK "#define m.sdlime j";
  R4:
   DECISION ∃n : n ∈ (m.param)
    ( TRUE ):
     TASK "m.pmltip m.sdlime//__//i";
     TASK i := i + 1;
     JOIN R4;
    ELSE:
   ENDDECISION;
   JOIN R3;
  ELSE:
 ENDDECISION;
R5:
 DECISION ∃o : o ∈ (a.spre);
  ( TRUE ):
   DECISION o.vredno = NONE;
    ( TRUE ):
     TASK "o.pmltip  o.sdlime;";
     JOIN R5;
    ELSE:
     CALL 3.2(p, o.vredno); // (str. 30)
     TASK "o.pmltip  o.sdlime = p;";
     JOIN R5;
   ENDDECISION;
  ELSE:
 ENDDECISION;
 CALL 3.17(a); // (str. 75)
JOIN R1;
```

Algoritem 3.15: Algoritem za tvorbo procesnega tipa v modelu sistema

Sledi definicija vseh časovnikov in pomožnih spremenljivk za obravnavo parametrov. Imenu časovnikov je dodana predpona `tmr__`. S pomočjo funkcije $indeks()$ pridobimo naslednje edinstveno zaporedno število za signal, ki se tvori ob poteku časovnika. Pomen edinstvenih števil je podrobneje predstavljen v razdelku 3.2.1 (str. 44). V zanki R4 se za vsak časovnik tvori po ena spremenljivka za vsak parameter. Ime spremenljivke je sestavljeno iz imena časovnika in zaporednega števila parametra v definiciji časovnika. Podatkovni tip spremenljivke določa predhodna analiza v algoritmu 3.6 (str. 40).

Naslednji sklop algoritma, ki ga označuje oznaka R5, tvori deklaracije za vse lokalne spremenljivke procesa in jim priredi morebitno privzeto vrednost. Sledi pretvorba telesa procesa in zaključek procesnega tipa (razdelek 3.4.2).

Ob zagonu sistema vrstni red tvorjenja procesov ni določen. Komunikacija med procesi se omogoči, ko se tvori zadnji začetni proces. Začetni proces je proces, ki ima začetno

število primerkov procesa večje od nič. Vrednosti uporabljenih formalnih parametrov ob tvorbi procesa ob inicializaciji sistema niso definirane.

Proces je aktiven do končanja s konstruktom stop. Med delovanjem sistema se lahko procesi zaključujejo in tvorijo. Nov proces lahko tvori katerikoli izmed živih procesov znotraj istega bloka. Vse našteto definira SDL-stroj in je v modelu sistema potrebno primerno modelirati v jeziku Promela. Orodje $sdl2if$ ne podpira dinamične tvorbe procesov. V modelu sistema se vedno tvori maksimalno število procesov. Komunikacija s takšnimi procesi ni podprta (str. 60). Orodje $if2pml$ prav tako ne podpira te podmnožice specifikacij.

3.3.3 Procedura

Procedure v jeziku SDL so podobne proceduram v programskih jezikih. Omogočajo združevanje kompleksnih delov v celoto z enkratno definicijo in večkratno uporabo. Predstavljajo samostojno enoto z lastnimi podatkovnimi tipi in spremenljivkami. Definicija procedure lahko vsebuje tudi formalne parametre, ki omogočajo splošnejšo definicijo procedur in prenos vrednosti med proceduro in kličočim procesom. Definicija formalnih parametrov določa seznam podatkovnih tipov spremenljivk, ki se bodo lahko uporabile ob klicu procedure. Poznamo dva tipa formalnih parametrov: vhodne in vhodno/izhodne. Prvi so namenjeni izključno prenosu vrednosti v proceduro, drugi pa določajo spremenljivke na nivoju kličočega procesa, ki jih lahko procedura s svojim izvajanjem spreminja. Procedura se lahko kliče tudi iz druge procedure, vendar se v nadaljevanju besedila vedno sklicujemo samo na kličoči proces.

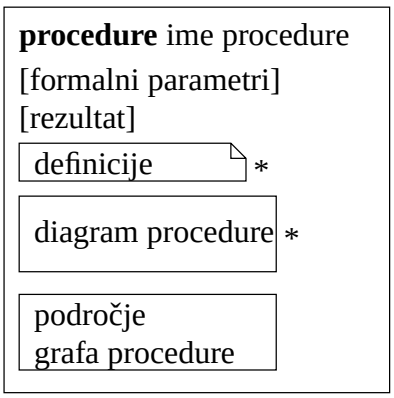
Slika 3.31 podaja EBNF zapis sintakse grafične in tekstovne specifikacije procedure. Naloga ne obravnava oddaljenih procedur, saj je zaradi neučinkovitega modela izvajanja priporočljivo poiskati drugačne rešitve.

Definicija procedure se lahko nahaja na različnih hierarhičnih nivojih, in sicer v opisu sistema, bloka ali procesa. Prav tako lahko nastopi znotraj opisa druge procedure. Mesto definicije določa vidno območje procedure. Skladno s strukturnimi pravili se lahko procedura, ki je definirana v okviru bloka, uporablja v vseh strukturno nižjih konstruktih — podblokih oz. procesih.

Ker lahko procedure vsebujejo tudi lastna stanja, jih obravnavamo kot razširitev procesa in jih ne modeliramo v samostojnih konstruktih, temveč jih primerno dopolnjene vključujemo neposredno na mesto klica procedure v procesu. Takšna obravnava omogoča zagotavljanje atomarnosti prehodov med stabilnimi stanji procesa ob klicu procedure s stanji. Zato pri tvorbi modela ne podpiramo rekurzivnih klicev procedur. Podpora bi se lahko izvedla z uporabo zunanje kode v jeziku C (slika 2.1), vendar naloga tega ne obravnava.

Klic procedure lahko delimo na samostojni klic ali klic znotraj izraza. V slednjem procedura vrača vrednost, ki jo določa konstrukt `return` oz. skrajno desni vhodno/izhodni formalni parameter. Procedura je ob izvajanju pridružena procesu, ki proceduro kliče. Torej

Grafična predstavitev



Tekstovni zapis

<procedura> :=
procedure <ime procedure>;
[<formalni parametri procedure>]
[<rezultat procedure>]
{<definicija podatkov>
| <definicija spremenljivk>
| <definicija procedure>}
[<telo procedure>]
endprocedure [<ime procedure>];

<formalni parametri procedure> := **fpar** <formalni parametri>{,<formalni parametri>}*
<formalni parametri> := <vrsta parametra> <tip parametrov>
<vrsta parametra> := [IN/OUT | IN]
<tip parametrov> := <ime spremenljivke>{,<ime spremenljivke>}* <ime podatkovnega tipa>
<rezulat procedure> := **returns** [<ime spremenljivke>] <ime podatkovnega tipa>

Slika 3.31: EBNF zapis sintakse specifikacije procedure

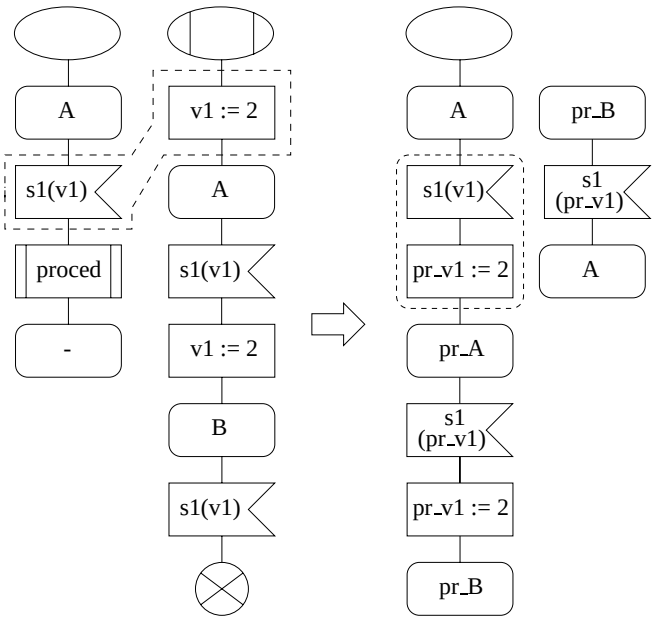
se izvaja v časovni reži, ki je namenjena izvajanju procesa. Proceduro opisuje telo procedure, ki ima podobne elemente kot telo procesa ali storitve. Klic procedure se lahko izvede v poljubnem prehodu procesa med stanji. V tej točki se izvajanje procesa na logičnem nivoju ustavi. Tvori in izvede se lokalna kopija procedure, ki ima dostop do signalov v vhodni vrsti procesa.

Skladno s semantiko jezika SDL je prehod med dvema eksplicitno definiranima stanjema neprekinljiv. Zato je potrebno poskrbeti za primerno modeliranje v primeru klica procedure z lastnimi stanji. Slika 3.32 prikazuje razširitev procesa za ta primer. Neprekinljivost je potrebno zagotoviti med začetnim stanjem procesa in prvim eksplicitnim stanjem v proceduri — del, ki je označen s prekinjeno črto. Podobno velja ob izhodu iz procedure, kjer se nadaljuje izvajanje procesa z naslednjim konstruktom v aktivnem prehodu. Atomarni prehod se zaključi v naslednjem eksplicitnem stabilnem stanju procesa.

Analizo procedur prikazuje algoritem 3.16. Definicija 3.32 določa elemente, ki v celoti opisujejo posamezno proceduro v specifikaciji sistema in predstavljajo vhod pri tvorbi modela.

Definicija 3.32 Procedura $proced$, $proced \in spec.stru$, je sedmerica $(sdlime, pmlime, fpar, sprem, rezul, stan, telo)$, kjer je:

$sdlime$ ime procedure v specifikaciji;
$pmlime$ ime procedure v modelu sistema;
$fpar$ množica formalnih parametrov;
$sprem$ množica spremenljivk;
$rezul$ podatki o rezultatu, ki se vrne v primeru klica znotraj izraza;
$stan$ boolova konstanta, ki zavzame vrednost 1, če ima procedura stanja;
$telo$ pretvorjeno telo procedure.♦



Slika 3.32: Primer vključevanja procedure s stanji v telo procesa

Definicija 3.33 Formalni parameter $fpar$, $fpar \in spec.stru.proced$, je četverica $(sdlime, sdltip, pmltip, ven)$, kjer je:

$sdlime$ ime formalnega parametra v specifikaciji;
$sdltip$ ime podatkovnega tipa, ki omejuje zalogo vrednosti parametra;
$pmltip$ ime podatkovnega tipa v modelu sistema;
ven boolova konstanta, ki zavzame vrednost 1, če je <vrsta parametra> enaka IN/OUT.♦

Definicija 3.34 Rezultat $rezul$, $rezul \in spec.stru.proced$, je trojica $(sdlime, sdltip, pmltip)$, kjer je:

$sdlime$ ime lokalne spremenljivke v specifikaciji;
$pmlime$ ime lokalne spremenjljivke v modelu sistema;
$pmltip$ ime podatkovnega tipa, ki omejuje zalogo vrednosti spremenljivke.♦

Imenu procedure v specifikaciji dodamo mesto deklaracije v strukturi sistema z uporabo funkcije $pot()$. Tako v prvem koraku algoritma pridobimo $pmlime$, ki predstavlja edinstveno ime v modelu sistema. Sledi analiza formalnih parametrov procedure, ki se izvaja v zanki z oznako R2. V primeru, ko je v definiciji <vrsta parametra> naštetih več elementov <ime spremenljivke>, se vsak obravnava samostojno. V algoritmu ne ločimo med sintaktično enakovrednimi zapisi in podajamo splošen postopek analize. Za vsak parameter se shrani ime spremenljivke in pridružen podatkovni tip. Ker podatkovni tip v primeru večkratne uporabe <ime podatkovnega tipa> določa mesto definicije, se shrani tudi njegovo edinstveno ime $pmltip$. Funkcija $poišči()$ pridobi veljaven podatkovni tip iz množice podatkovnih tipov. Zapis množic v oklepajih ponazarja, da se sklicujemo na rezultate predhodnih analiz. Nadaljnja obravnava parametra je odvisna od vrste parametra. Kadar je parameter vhodno/izhodni, ob klicu procedure ni potrebno definirati lokalnih spremenljivk, saj se zaradi

START ;
R1:
DECISION $\exists a$: $a \in spec.stru.proced$;
(TRUE):
ELSE:
END ;
ENDDECISION;
TASK $a.pmlime$:= $pot()//sdlime$;
R2:
DECISION $\exists b$: $b \in proced.fpar$;
(TRUE):
TASK $a.b.sdlime$:= <ime spremenljivke>;
TASK $a.b.sdltip$:= <ime podatkovnega tipa>;
TASK i := $a.b.sdltip$;
TASK $a.b.pmltip$:= $poišči(i, (spec.def\{stype \vee ntype \vee ptype\}))$;
DECISION $\exists j$: j :=<vrsta parametra>, j =IN/OUT;
(TRUE):
TASK $a.b.ven$:= TRUE;
ELSE:
TASK $a.b.ven$:= FALSE;
ENDDECISION;
TASK $shrani(a.b.sdlime, a.b.sdltip)$;
TASK $shrani(a.b.pmltip, a.b.ven)$;
JOIN **R2**;
ELSE:
DECISION $\exists k$: k :=<rezultat procedure>;
(TRUE):
DECISION $\exists m$: m :=<ime spremenljivke>;
(TRUE):
TASK $shrani(a.k.m)$;
ELSE:
ENDDECISION;
TASK $a.k.sdltip$:= <ime podatkovnega tipa>;
TASK $a.k.pmltip$:= $poišči((spec.def\{stype \vee ntype \vee ptype\}))$;
TASK $shrani(a.k.sdltip, a.k.pmltip)$;
ELSE:
ENDDECISION;
TASK $stanja(a.stan)$;
TASK $shrani(a.sdlime, a.pmlime, a.stan)$;
ENDDECISION;
ELSE:
ENDDECISION;
CALL 3.17($a.pmlime$); // (str. 75)
JOIN R1;

Algoritem 3.16: Algoritem analize procedur

načina vključevanja procedure v kličoči proces, lahko uporabijo spremenljivke procesa. S tem zmanjšamo velikost vektorja stanj, ki opisuje model sistema med formalno verifikacijo.

Po analizi vseh formalnih parametrov preverimo, ali definicija procedure vsebuje <rezultat procedure>. Skladno s pravili zapisa se lahko na tem mestu definira nova lokalna spremenljivka ali pa samo podatkovni tip. Ta omejuje zalogo vrednosti, ki jo procedura vrača v primeru klica znotraj izraza. Prenos vrednosti med proceduro in kličočim procesom se, ne glede na način klica procedure, izvaja z uporabo parametrov vrste IN/OUT. V primeru klica procedure znotraj izraza se priredi spremenljivki na levi strani izraza vrednost, ki jo določa:

- izraz v konstruktu <izhod>, ki je omejen z definicijo podatkovnega tipa v okviru <rezultat procedure>,
- lokalna spremenljivka, ki je bila definirana v okviru <rezultat procedure> ali
- skrajno desni parameter vrste IN/OUT.

Vrstni red določa prioriteto izbire. V prvem primeru mora rezultat izraza zadoščati veljavni definiciji podatkovnega tipa. Tega določa <ime podatkovnega tipa> v okviru definicije <rezultat procedure>. Če ta ne obstaja, predvideno zalogo vrednosti predpisuje tip skrajno desnega formalnega parametra vrste IN/OUT. Izvajanje se modelira tako, da se tik pred zaključkom procedure izbrani spremenljivki priredi rezultat izraza. Orodje OG pri definiciji procedur postavlja strožje omejitve, saj dopušča uporabo procedure v izrazu samo v primerih, ko je definicija <rezultat procedure> prisotna.

V okviru algoritma 3.16 se v naslednjem odločitvenem stavku preverja prisotnost definicije <rezultat procedure>. Ker je deklaracija lokalne spremenljivke neobvezna, se njen obstoj preverja z ločenim odločilnim stavkom. Predpisani podatkovni tip se z uporabo funkcije $poišči()$ pridobi iz množice podatkovnih tipov. Območje veljavnosti podatkovnega tipa določa mesto definicije procedure in ne mesto klica. S klicem funkcije $stanja()$ postavimo parameter $stan$ na boolovo vrednost 1 v primeru, ko telo procedure vsebuje stanja. Sledi pretvorba telesa procedure, ki jo opisujeta algoritma 3.17 in 3.18. Telo procedure se v celoti shrani v $spec.stru.proced.telo$ in se ob klicu procedure primerno doda v model sistema.

```
FPAR a ∈ (spec.stru.proc ∨ spec.stru.proced);
START ;
 R1:
  DECISION ∃i : i ∈ spec.kon,
   ( TRUE ):
    DECISION i;
     ( START ): CALL 3.18(a,START) // (str.75)
     ( STATE ): CALL 3.19(a); // (str. 79)
     ELSE:
    ENDDECISION;
    JOIN R1;
   ELSE:
    TASK "end: skip";
    TASK "}";
    END:
  ENDDECISION;
```

Algoritem 3.17: Algoritem za pretvorbo telesa procesa ali procedure

```
FPAR a ∈ (spec.stru.proc ∨ spec.stru.proced);
FPAR b ∈ {(a.stanje),START};
TASK c ∈ {(b.odloč),NONE};
TASK i ∈ <vprašanje>;
TASK k ∈ BOOLEAN;
START ;
 R1:
  DECISION ∃i : i ∈ spec.kon
   ( TRUE ):
    DECISION i;
     ( LABEL ): CALL 3.34(a); // (str. 106)
     ( TASK ): CALL 3.36; // (str. 108)
     ( CREATE ): CALL 3.32; // (str. 104)
     ( OUTPUT ): CALL 3.31(a); // (str. 98)
     ( DECISION ): CALL 3.38(a,b,c,k); // (str. 110)
     ( (<odgovor>): ): CALL 3.39(a,b,c,k); // (str. 110)
     ( ELSE ): CALL 3.39; // (str. 110)
     ( ENDDECISION ):
      CALL 3.40(a,b,c); // (str. 111)
     ( SET ): CALL 3.26; // (str. 95)
     ( RESET ): CALL 3.27; // (str. 95)
     ( ALTERNATIVE ): STOP;
     ( NEXTSTATE ): CALL 3.33; // (str. 105)
     ( JOIN ): CALL 3.35(a); // (str. 107)
     ( STOP ):
      TASK "table_//a.pmlime//_free–;";
      TASK "goto end;";
     ENDDECISION;
     JOIN R1;
    ELSE:
     RETURN:
  ENDDECISION;
```

Algoritem 3.18: Algoritem za pretvorbo konstruktov v telesu procesa ali procedure

V tem razdelku smo opisali konstrukte, ki omogočajo strukturiran opis specifikacije. Poudariti velja, da ni mogoče definirati optimalnega postopka delitve sistema na posamezne konstrukte, ki bi veljal za vse opise. Končna odločitev je vedno na strani razvojnega inženirja. V pomoč je lahko razvojna metodologija, ki upošteva lastnosti skupine načrtovanih sistemov in postopke simulacije ter formalne verifikacije specifikacije.

V nadaljevanju sledi opis konstruktov, s katerimi opisujemo razširjene končne avtomate, ki določajo delovanje sistema.

3.4 Razširjen končni avtomat

Jedro specifikacije sistema predstavlja skupek vseh razširjenih končnih avtomatov, ki opisujejo procese in procedure sistema. Med procesi ni nobene hierarhične stukture in vsi tečejo sočasno in neodvisno. Pri klicu procedure se izvajanje procesa ustavi v točki klica. Ko se program vrne iz procedure, se izvajanje nadaljuje v točki za klicem procedure. Procedure znotraj procesa se ne izvajajo sočasno.

Vsak proces lahko komunicira z drugim procesom, če ju povezuje komunikacijska pot. Med izvajanjem lahko primerek procesa tvori drug primerek procesa v okviru istega bloka. Vsak proces je sestavljen iz več eksplicitnih stanj in še večje množice implicitnih stanj, ki jih določajo spremenljivke procesa. Spremembo eksplicitnega stanja povzroči prispeli signal. Vse signale proces pridobi iz pridružene vhodne vrste (razdelek 3.2.1). Ob prehodu med stanji se lahko izvedejo različne naloge. Izvrši se lahko računska operacija, starta časovnik, pošlje signal, tvori proces, kliče procedura ali izvaja vejitev izvajanja na osnovi trenutne vrednosti izbrane spremenljivke. Izvajanje procesa se zaključi ob uporabi konstrukta <izhod>. V naslednjih razdelkih sledi opis konstruktov, s katerimi opisujemo razširjene končne avtomate v specifikaciji sistema.

3.4.1 Start in stop

Vsak končni avtomat ima začetno stanje in prvi prehod. Slika 3.33 prikazuje EBNF zapis sintakse specifikacije konstrukta <start>, ki ponazarja začetno stanje razširjenega končnega avtomata z opisom delovanja procesa ali procedure. Izvajanje končnega avtomata se zaključi s konstruktom <stop>.

Grafična predstavitev	**Tekstovni zapis**
	<start> ::= **start** <prvi prehod>
	<prvi prehod> ::= <prehod>
	<stop> ::= **stop**

Slika 3.33: EBNF zapis sintakse specifikacije konstruktov start in stop

Začetno stanje v modelu sistema označuje prvi prehod, izvajanje procesa pa se zaključi z oznako `end:`.

3.4.2 Prehod

Prehod je del opisa procesa ali procedure, ki se izvede med prehodom iz trenutnega eksplicitnega stabilnega stanja do končnega stavka. Končni stavek predstavljajo konstrukti naslednje stanje, stop in vrni. Prehod se lahko z uporabo konstrukta odločitev deli na več možnih vej izvajanj, ob uporabi konstrukta združi pa se izvajanje nadaljuje na drugem mestu v specifikaciji, ki ga označuje konstrukt oznaka z enakim imenom.

Jedro prehoda je zaporedje stavkov, ki lahko izvajajo manipulacije nad podatki procesa, ustvarijo nov proces, postavljajo in brišejo časovnike, kličejo procedure ter pošljejo signal. Slika 3.34 prikazuje EBNF zapis sintakse tekstovne specifikacije prehoda. Nazoren prikaz grafične sintakse je zaradi številčnosti elementov nemogoč. Posamezni konstrukti so predstavljeni v svojih razdelkih.

<prehod>::={{<stavek>}+ [<končni stavek>]} | <končni stavek>

<stavek> ::= [<oznaka>] <akcija>

<akcija>::= <naloga> | <izhod> | <ustvari> | <odločitev> | <izbira> | <postavi časovnik>
| <zbriši časovnik> | <kliči proceduro>

<končni stavek> ::= [<oznaka>] {<naslednje stanje> | <združi> | <stop> | <vrni>}

<naslednje stanje> ::= <stanje>

Slika 3.34: EBNF zapis sintakse specifikacije prehoda

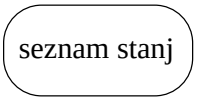
Algoritma 3.17 in 3.18 opisujeta pretvorbo telesa procesa ali procedure. Zaradi obsežnosti je algoritem predstavljen s skupino manjših algoritmov, ki opisujejo analizo in pretvorbo posameznih konstruktov in so podrobneje predstavljeni v drugih razdelkih.

3.4.3 Stanje

Konstrukt stanje opisuje eksplicitna stabilna stanja v grafu procesa ali procedure. V okviru lastnega grafa so stanja edinstveno poimenovana. Slika 3.35 prikazuje sintakso specifikacije stanja.

Vsako stanje ima pridružene vhode, ki določajo, pod kakšnimi pogoji in kateri signali se v tem stanju lahko obdelajo (algoritem 3.13). Sprejemu signala sledi predviden prehod v naslednje stanje. Nabor signalov, ki jih lahko proces v stanju obravnava, določa unija signalov v okviru konstruktov vhod, prioritetni vhod in shrani. Signali, ki se v posameznem stanju ne morejo obravnavati, se implicitno zavržejo. Posebni primer predstavljata trajni signal in spontani prehod. Prvi se lahko izvede le v primeru, ko v vhodni vrsti ni nobenega drugega signala, drugi pa se lahko izvede v vsakem trenutku. Različna obravnava signalov, predvidene prioritete in dodatne omejitve narekujejo poseben postopek modeliranja vhodne vrste, ki ni skladna z mehanizmi komunikacije v Promeli. Zato se v okviru algoritma 3.19 za vsako stanje pripravi posebno programsko ogrodje za polno podporo vsem komunikacijskim

Grafična predstavitev **Tekstovni zapis**




```
<stanje> ::=
      state  <seznam stanj> {
             <seznam vhodov> [ <omogočitveni pogoj> ]
             | <prioritetni vhod> | <shrani>
             | <spontani prehod> | <trajen signal>}
      [endstate [ <ime stanja> ] ]

<seznam stanj> ::={<ime stanja>{,<ime stanja>}*} |
                  <seznam stanj zvezdica>
```

Slika 3.35: EBNF zapis sintakse specifikacije konstrukta stanje

gradnikom SDL. V primeru stanj, ki ne vsebujejo posameznih vhodnih konstruktov, se lahko ob tvorbi modela uporabijo samo potrebni deli modela SDL-stroja.

Poseben pomen ima stanje "*". To predstavlja vsa stanja končnega avtomata in opisuje prehode, ki so skupni vsem stanjem. Slika 3.36 prikazuje sintakso zapisa. Kadar želimo določena stanja izključiti iz skupine stanj, jih je potrebno eksplicitno navesti v seznamu stanj zvezdica. Izključena stanja morajo biti v grafu izvajanja definirana s svojimi prehodi. V stanju zvezdica se vsi definirani vhodni konstrukti in pridruženi prehodi eksplicitno modelirajo. V algoritmu 3.19 se za vsa stanja obravnavajo vhodni konstrukti, ki so pridruženi trenutnemu stanju in stanju zvezdica. Formalni zapis množice je podan v drugi vrstici algoritma.

```
<seznam stanj zvezdica> ::= <zvezdica> [(<ime stanja> {,<ime stanja>}*)]
```

Slika 3.36: EBNF zapis sintakse specifikacije seznama stanj zvezdica

V prvem koraku algoritma 3.19 s pomočjo funkcije $poišči()$ pridobimo kazalec na stanje v specifikaciji procesa. Določa ga trenutni proces in ime stanja v njegovem telesu. Sledi oznaka stanja, ki jo določa ime stanja. Na to mesto v modelu sistema se lahko vrnemo z brezpogojnim skokom v primeru, ko se to stanje pojavi v konstruktu naslednje stanje. Če stanje vsebuje spontani prehod, je potrebno pred nadaljnjim izvajanjem izvesti naključno izbiro, ki je definirana na sliki 3.47. Ob analizi procesa (algoritem 3.13) dobijo končna stanja posebno oznako. Če sodi trenutno stanje v množico končnih stanj, dodamo še oznako s predpono `end_`. Takšne oznake označujejo veljavna končna stanja procesa in so podrobneje obravnavane v poglavju 4.

Sledi prvi del modela komunikacijskega podsistema SDL-stroja (1-6). Posamezne vrstice so označene z zaporednim številom, na katero se v besedilu sklicujemo v oklepajih. V razširjeni verziji orodja Spin za podporo diskretnemu času se vrednost časovnikov lahko

```
FPAR a ∈ (spec.stru.proc ∨ spec.stru.proced);
TASK b = poišči(<ime stanja>);
TASK množica := ((c ∈ ((b.sigsta) ∨ (a.časo)) ∨
(c ∈ (a.stanje.sigsta) ∧ a.stanje.pmlime = *))
START;
  TASK "b.pmlime//:";
  DECISION b.spon;
   ( TRUE ):
     TASK "random();";
   ELSE:
  ENDDECISION
  DECISION b.konč;
   ( TRUE ):
     TASK "end_//b.pmlime:";
   ELSE:
  ENDDECISION
  TASK 1: "do";
  TASK 2: "::table_channum_ptr[input] > pv_cur −>";
  TASK 3: "   table_channum_prio[input]=false;";
  TASK 4: "   pv_cur++;";
  TASK 5: "   pv_ptr=0;";
  TASK 6: "   atomic{";
  TASK 7: "   do ";
  TASK 8: "   :: pv_ptr <= cv_buff-1 −> ";
  TASK 9: "      if";
Analiza vrste, R1:
  DECISION ∃c : množica ∧ c.shrani = TRUE;
   ( TRUE ):
     DECISION c.pmlime = *;
      ( TRUE ):
      R2:
          DECISION ∃d : d ∈ (a.sigin\b.sigsta);
           ( TRUE ):
             TASK "::set_save(//d.pmlime//)"; JOIN R2;
           ELSE:
          ENDDECISION;
      ELSE:
          TASK "::set_save(//c.pmlime//)"; JOIN R1;
     ENDDECISION;
   ELSE:
  ENDDECISION;
R3:
  DECISION ∃c : množica ∧ c.prio = TRUE;
   ( TRUE ):
     TASK "::set_prio(//c.pmlime//)"; JOIN R3;
   ELSE:
  ENDDECISION;
  TASK 10: "    :: else −> set_clear()";
  TASK 11: "    fi;";
  TASK 12: "    pv_ptr++";
  TASK 13: " :: else −> goto b.pmlime//_start";
  TASK 14: " od;";
  TASK 15: " }";
  TASK 16: "b.pmlime//_start:";
  TASK 17: "if";
  TASK 18: "::table_channum_prio[input]==true −>";
  TASK 19: "   pv__ptr=0;";
  TASK 20: "   do";
  TASK 21: "   :: (pv_ptr <= cv_buff-1) ";
  TASK 22: "      && (table_channum_nsp[input].
                 data[pv__ptr].prio==true) −>";
  TASK 23: "      if";
Prioritetni vhod, R4:
  DECISION ∃c : množica ∧ c.prio = TRUE;
   ( TRUE ):
     CALL 3.22(a,b,c); JOIN R4; // (str. 90)
   ELSE:
  ENDDECISION;
  TASK 24: "      :: else −> skip;";
  TASK 25: "      fi";
  TASK 26: "    :: (pv_ptr == cv_buff) −> break;";
  TASK 27: "    :: else −> pv_ptr++";
  TASK 28: "    od;";
  TASK 29: "::else −>";
  TASK 30: "   pv_ptr=0;";
  TASK 31: "   do";
  TASK 32: "   :: (pv__ptr <= cv_buff-1) −>";
  TASK 33: "      if";
  TASK 34: "      :: skip_save()";
  TASK 35: "      :: else −>";
  TASK 36: "         if";
Vhod, R5:
  DECISION ∃c : množica ∧ c.shrani != TRUE
                          ∧ c.prio != TRUE;
   ( TRUE ):
     CALL 3.20(a,b,c); JOIN R5; // (str. 84)
   ELSE:
  ENDDECISION;
  TASK 37: "         :: else −>";
  TASK 38: "            if";
Implicitni prehod, R6:
  DECISION b.zvez != TRUE;
   ( TRUE ):
     DECISION ∃c : c ∈ (a.sigin\b.sigsta)
      ( TRUE ):
          CALL 3.23(a,b,c); JOIN R6; // (str. 91)
      ELSE:
     ENDDECISION;
   ELSE:
  ENDDECISION;
  TASK 39: "            :: else −> goto b.pmlime;";
  TASK 40: "            fi";
  TASK 41: "         fi;";
  TASK 42: "      fi";
  TASK 43: "   od;";
  TASK 44: " fi;";
  TASK 45: " goto b.pmlime;";
Spontani prehod, R7:
  DECISION ∃c : množica ∧ c.pmlime = NONE;
   ( TRUE ):
     CALL 3.24(a,b); JOIN R7; // (str. 92)
   ELSE:
  ENDDECISION;
Časovnik, R8:
  DECISION ∃c : množica ∧ (c ∈ (a.časo));
   ( TRUE ):
     CALL 3.25(a,c); JOIN R8; // (str. 94)
   ELSE:
  ENDDECISION;
  Trajen signal, R9:
   DECISION ∃c : c ∈ (b.trajen);
    ( TRUE ):
      CALL 3.28(a,c); JOIN R9; (str. 96)
    ELSE:
   ENDDECISION;
  TASK 46: "od;";
  RETURN;
```

Algoritem 3.19: Algoritem pretvorbe konstrukta stanje

zmanjša samo v trenutkih, ko se vsi drugi aktivni procesi v modelu blokirajo. Zato je potrebno analizo pridružene vhodne vrste modelirati statično — preverjanje se izvaja samo takrat, ko se v vhodni vrsti pojavi nov signal, sicer se proces blokira. Takšen pristop narekuje tudi tvorba učinkovitega modela.

Vhodno vrsto je zaradi podpore vsem komunikacijskim konstruktom jezika SDL potrebno pred vsakim sprejemom signala analizirati. Posameznim signalom se pred obravnavo vhodne vrste dodelijo zastavice, ki določajo njihov status v trenutnem stanju procesa (7-14). V zanki se pregledajo vse reže pridruženega kanala. Slika 3.39 prikazuje definicije konstruktov, ki posamezen signal, na osnovi pridobljenih podatkov v okviru analize (algoritem 3.13, str. 62), označijo kot prioritetni (`set_prio()`, zanka z oznako R3), neprioritetni (`set__clear()` (10)) ali signal, ki ga je potrebno v trenutnem stanju shraniti za kasnejšo obravnavo (`set__save()`). Zastavice se za vse signale, ki čakajo na obravnavo, postavijo v globalnem polju `table_channum_nsp`, ki je definirano v okviru algoritma 3.13. Ime ponazarja, da se vrši preslikava med številko kanala in imenom signala (“name”) ter zastavicami za “save” in “priority”. Podrobnejši opis je podan v razdelkih 3.4.4 in 3.4.6.

Kadar je ime signala predstavljeno z zvezdico, je potrebno pregledati zastavice za vse signale $\in spec.stru.proc.sigin \backslash spec.stru.proc.stanje.sigsta$. Obravnava se torej množica vseh signalov, ki jih lahko sprejme proces in njihov sprejem v trenutnem stanju ni eksplicitno naveden.

V nadaljevanju se izvede sprejem najstarejšega prioritetnega signala (17-28), vendar le v primeru, ko je v vhodni vrsti ta prisoten (20) — sicer se ne vrši pregled vhodne vrste. V delu z oznako R4 se z algoritmom 3.22 (str. 90) pripravi sprejem za vse prioritetne signale v obravnavanem stanju. Vsakemu sprejemu signala sledi prehod, ki se pretvori s klicem algoritma 3.18.

Kadar v vhodni vrsti ni prioritetnega signala, se preveri, ali so prisotni signali, ki so zapisani v okviru konstrukta vhod (36-37). V okviru zanke z oznako R4 se s klicem algoritma 3.20 pripravi vse potrebno za sprejem signalov in modeliranje pridruženih prehodov. Podprta je tudi uporaba omogočitvenega pogoja (razdelek 3.4.5).

Zadnji del prvega sklopa algoritma (37-45) modelira implicitni sprejem signalov (razdelek 3.4.7), ki v obravnavanem stanju niso zavedeni v nobenem izmed vhodnih konstruktov, vendar samo v primeru, ko v stanju ni uporabljen signal zvezdica. Po implicitnem sprejemu signala se vrnemo v isto stanje in postopek sprejema se ponovi.

V zanki z oznako R7 se modelira sprejem spontanih prehodov, ki se lahko izvedejo v poljubnih trenutkih in so podrobneje predstavljeni v razdelku 3.4.8.

Podpora diskretnemu času v orodju Spin omogoča modeliranje časovnikov v obliki lokalnih spremenljivk. Njihova vrednost se zmanjša za ena v trenutku, ko se vsi procesi v sistemu blokirajo. Časovnik se izteče v trenutku, ko je vrednost pridružene spremenljivke nič. V okviru zanke R7 se dodajo stražarji, ki ob izteku časovnika v lastni pridruženi kanal

posredujejo signal z imenom časovnika (razdelek 3.4.9). Signal se nato obdela po enakem postopku kot signali, ki se sprejmejo iz okolja procesa.

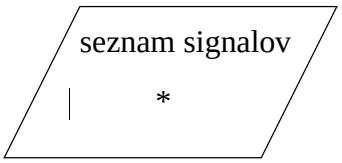
Algoritem 3.19 določa tudi mesto modeliranja trajnega signala. Trajen signal ima nižjo prioriteto in se lahko izvede samo, kadar je izpolnjen omogočitveni pogoj in v pridruženem kanalu ni signala, ki bi ga v trenutnem stanju lahko obdelali. Prioriteto posameznega prehoda določa <celoštevilčni literal> (slika 3.50). Podrobnejši opis je podan v razdelku 3.4.10.

Posamezno stanje se lahko v opisu grafa izvajanja pojavi večkrat. Takšen zapis je enakovreden zapisu s samo enim stanjem in je namenjen preglednejši grafični predstavitvi. V modelu sistema so vsi prehodi združeni v okviru istega stanja.

3.4.4 Shrani

Konstrukt shrani omogoča spremembo vrstnega reda sprejema signalov iz vhodne vrste procesa. Kadar prispelega signala pri opisu avtomata stanj ne želimo obravnavati v trenutnem stanju, ga lahko z uporabo tega konstrukta shranimo za obravnavo v kasnejšem stanju. Prehod v naslednje stanje tako sproži prvi veljavni signal, ki ni naveden znotraj konstrukta shrani. Konstrukt je veljaven samo v trenutnem stanju, zato ga je potrebno definirati v vsakem stanju razširjenega avtomata. Slika 3.37 prikazuje sintakso specifikacije konstrukta shrani. Seznam shranjenih signalov zvezdica opisuje večjo skupino signalov, ki jo sestavlja razlika množice veljavnih vhodnih signalov in množice signalov, ki jih proces v trenutnem stanju lahko obdela. Kadar je v stanju definiran vhod zvezdica, ni mogoče uporabiti seznama shranjenih signalov zvezdica.

Grafična predstavitev



Tekstovni zapis

```
<shrani> ::=
      save  <seznam signalov> |
            <seznam shranjenih signalov zvezdica>

<seznam shranjenih signalov zvezdica> ::= <zvezdica>
<zvezdica> ::= *
```

Slika 3.37: EBNF zapis sintakse specifikacije konstrukta shrani

Sočasnega modeliranja konstrukta shrani, običajnih in prioritetnih vhodov, trajnih signalov, spontanih prehodov in časovnikov, ki bi bilo skladno s semantičnimi pravili jezika SDL, ni mogoče izvesti izključno s komunikacijskimi konstrukti jezika Promela. Zaradi dodatnih lastnosti posameznega signala, ki določajo vrstni red sprejema signalov iz pridruženega kanala, se sprejem modelira z dodatnim poljem in naključnim sprejemom. Ob uporabi naključnega sprejema [36] se lahko sprejme poljubni signal iz vhodne vrste, ki zadosti kriterijem sprejema — npr. pravilno ime signala. Če kriterijem sprejema zadosti več signalov, se sprejme najstarejši.

V globalnem polju se za vsak proces oz. pridružen kanal hranijo podatki o trenutnem vrstnem redu signalov in njihove dodatne lastnosti — shrani ali prioritetni signal. Definicija polja `table_channum_nsp` je prikazana na sliki 3.38, operacije nad poljem pa so definirane na sliki 3.39. Ključ za dostop do podatkov o kanalu je edinstvena številka kanala v modelu sistema. Vsak proces hrani številko pridruženega kanala v lokalni spremenljivki `input`. Število polj je enako največjemu predvidenemu številu procesov v modelu. Podpolje posameznega procesa definira zapis `pt__chandata`, ki je sestavljen iz edinstvenega imena signala ("name") in zastavic "save" in "prio". Zastavico "save" je za posamezen signal potrebno postaviti na vrednost 1 takrat, ko je v trenutnem stanju zapisan v okviru konstrukta shrani. Zastavica "prio" določa, da sodi signal v trenutnem stanju v množico prioritetnih signalov. Vsak proces ima dodatno pridruženo polje `table_channum_prio`. Zastavica je postavljena na 1 le v primerih, ko je v pridruženem kanalu vsaj en prioritetni signal.

```
1 #define none 0
2 typedef pt__chandat {pt__sig name;bool save;bool prio};
3 typedef pt__chandata {pt__chandat data[cv__buff]};
4 byte
5  table_channum_ptr[cv__procnum];
6 pt__chandata
7  table_channum_nsp[cv__procnum];
8 bool
9  table_channum_prio[cv__procnum];
```

Slika 3.38: Definicija polja za opis stanja kanala

Velikost polja je enaka $cv__procnum * cv__buff * (8 + 1 + 1)$ in je torej zmnožek največjega predvidenega števila procesov, števila rež pridruženih kanalov, osmih bitov za predstavitev imena signala in dveh dodatnih bitov za zastavici. V primeru privzetega števila rež 3 se v modelu sistema za vsak proces porabi dodatnih 30 bitov. Alternativni pristop, ki ga opisuje [47], rešuje uporabo konstrukta save s podvojitvijo pridruženih kanalov. S tem razpolovimo število procesov, ki jih lahko modeliramo. Dodatna slabost se pokaže v primerih, ko kanali vsebujejo veliko število parametrov, saj se poraba pomnilnika za definicijo kanalov podvoji. Predlagana rešitev v [47] je torej s stališča porabe pomnilnika optimalnejša le v primerih, ko kanal vsebuje parametre skupne dolžine 10 bitov ali manj. V vseh drugih primerih je naša rešitev optimalnejša. Opozoriti velja, da se polja podatkovnih tipov bit in $bool$ v internih podatkovnih strukturah v orodju Spin hranijo kot vrednosti tipa byte — zasedejo torej osem bitov pomnilnika. Dodatna omejitev rešitve iz [47] je v tem, da ne predvideva sprejema prioritetnih signalov, implicitnih prehodov, spontanega prehoda in trajnega signala.

Na sliki 3.39 so podani konstrukti za dostop in urejanje polja `table_channum_nsp`. Inline `update_chan_tab()` (1-30) se uporabi po vsakem sprejemu signala za osvežitev polja. Vsi zapisi o signalih, ki so v pridruženem kanalu za sprejetim kanalom, se premaknejo za eno mesto naprej. Zadnje mesto v polju se sprosti. Pred sprejemom signala se skladno z algoritmom 3.19 izvede analiza trenutnega stanja pridruženega kanala. Za vsak signal, ki je v obravnavanem stanju naveden v konstruktu shrani, se kliče

inline `set__save(sig)` (44-53), kjer je `sig` edinstveno ime signala. Ob pregledu vhodne vrste se v okviru zanke R1 v algoritmu 3.19 za takšen signal postavi zastavica "save" na ena. V primeru signala zvezdica se ta status dodeli signalom, ki so v množici $spec.stru.proc.sigin \backslash spec.stru.proc.stanje.sigsta$. Ob sprejemu neprioritetnih signalov (algoritem 3.19, (34)), se sprejem shranjenih signalov z uporabo konstrukta `skip__save()` (slika 3.39 (68-72)) preskoči.

```
 1 inline update_chan_tab(){
 2 atomic{
 3 do
 4 ::(pv__ptr < cv__buff-1) ->
 5   table_channum_nsp[input].\
 6         data[pv__ptr].name=\
 7         table_channum_nsp[input].\
 8         data[pv__ptr+1].name;
 9   table_channum_nsp[input].\
10         data[pv__ptr].save=\
11         table_channum_nsp[input].\
12         data[pv__ptr+1].save;
13   table_channum_nsp[input].\
14         data[pv__ptr].prio=\
15         table_channum_nsp[input].\
16         data[pv__ptr+1].prio;
17   pv__ptr++
18 :: else ->
19   table_channum_nsp[input].\
20         data[pv__ptr].name=none;
21   table_channum_nsp[input].\
22         data[pv__ptr].save=false;
23   table_channum_nsp[input].\
24         data[pv__ptr].prio=false;
25   break
26 od;
27 table_channum_ptr[input]--;
28 pv__cur=0;
29 }
30 }
31
32 inline set__prio(sig){
33   table_channum_nsp[input].\
34         data[pv__ptr].name==sig ->
35   atomic{
36     table_channum_nsp[input].\
37         data[pv__ptr].save=false;
38     table_channum_nsp[input].\
39         data[pv__ptr].prio=true;
40     table_channum_prio[input]=true
41   }
42 }
43
44 inline set__save(sig){
45   table_channum_nsp[input].\
46         data[pv__ptr].name==sig ->
47   atomic{
48     table_channum_nsp[input].\
49         data[pv__ptr].save=true;
50     table_channum_nsp[input].\
51         data[pv__ptr].prio=false;
52   }
53 }
54
55 inline set__clear(){
56   atomic{
57     table_channum_nsp[input].\
58         data[pv__ptr].save=false;
59     table_channum_nsp[input].\
60         data[pv__ptr].prio=false;
61   }
62 }
63
64 #define recv__sig(pog,sig)\
65  ((pog) && table_channum_nsp[input].\
66           data[pv__ptr].name==sig)
67
68 inline skip__save(){
69   table_channum_nsp[input].\
70           data[pv__ptr].save==true\
71           -> pv__ptr++
72 }
```

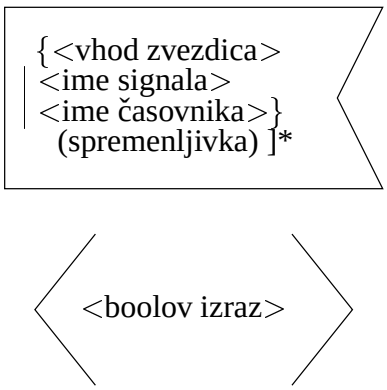
Slika 3.39: Dodatni konstrukti za modeliranje sprejema

3.4.5 Vhod in vhod zvezdica

Obnašanje procesa opisuje razširjen končni avtomat. Iz začetnega stanja in prvotnega prehoda, ki je namenjen inicializacijskim operacijam, se izvajanje procesa zaključi ob prehodu v novo stanje. Nadaljevanje izvajanja je mogoče ob sprejetju pričakovanega signala. Vsi signali, ki jih proces lahko obdela, so zapisani v enem ali več vhodnih konstruktih. Slika 3.40 prikazuje grafično in tekstovno sintakso specifikacije konstrukta vhod.

Tekstovnega opisa v tem primeru ni mogoče zajeti z enim grafičnim konstruktom, saj vključuje tudi konstrukt omogočitveni pogoj. V seznamu vhodov se lahko navede vhod zvezdica, ki predstavlja vse signale ali enega ali več signalov oz. časovnikov. Kadar signal nosi dodatne podatke z uporabo parametrov, se ti shranijo v lokalne spremenljivke procesa.

Grafična predstavitev



Tekstovni zapis

<vhod> ::=
input <seznam vhodov> [<omogočitveni pogoj>]

<seznam vhodov> ::= (<vhod zvezdica> | <drugo>
{, <drugo>}*

<drugo> ::= {<ime signala> | <ime časovnika>}
[([<spremenljivka> {,[<spremenljivka>]}*)]

<vhod zvezdica> ::= <zvezdica>
<omogočitveni pogoj> ::= **provided** <boolov izraz>

Slika 3.40: EBNF zapis sintakse specifikacije konstruktov vhod in omogočitveni pogoj

Vrstni red in podatkovni tipi spremenljivk, navedeni v konstruktu vhod, se morajo ujemati s prenašanimi vrednostmi. Posamezni parametri se lahko ob sprejemu izpustijo, kar povzroči izgubo prenašanih vrednostih. Če posamezen parameter ni bil določen ob pošiljanju signala, postane vrednost lokalne spremenljivke ob sprejemu nedefinirana. Skladno s standardom [57] ob uporabi nedefiniranih spremenljivk izvajanja sistema ni več mogoče predvideti. Ker Promela ne podpira takšne interpretacije sprejema nedefiniranih parametrov, se pri modeliranju sprejema na nedefinirana mesta zapišejo začasne spremenljivke primernega podatkovnega tipa. S tem se zagotovi pravilen sprejem preostalih parametrov in zadosti semantičnim pravilom Promele.

FPAR $a \in (spec.stru.proc \vee spec.stru.proced)$;
FPAR $b \in a.stanje$;
FPAR $((c \in (b.sigsta)) \vee$
$(c \in (a.stanje.sigsta) \wedge a.stanje.pmlime = *))$
START ;
TASK $e \in spec.kom.sig$;
DECISION $c.pmlime = *$;
(TRUE):
R1:
DECISION $\exists d : d \in (a.sigin \backslash b.sigsta)$;
(TRUE):
TASK e := $poišči(d.pmlime, spec.kom.sig)$;
CALL 3.21(a,b,e); // 85
TASK “goto //$b.pmlime$//__asteriks;”;
JOIN **R1**;
ELSE:
ENDDECISION;
ELSE:
TASK e := $poišči(c.pmlime, spec.kom.sig)$;
CALL 3.21(a,b,e); // 85
ENDDECISION;
DECISION $c.pmlime = *$;
(TRUE):
TASK “$b.pmlime$//__asteriks:”;
ELSE:
ENDDECISION;
CALL 3.18(a,b) // (str. 75)
RETURN;

Algoritem 3.20: Algoritem pretvorbe konstrukta vhod

Z uporabo boolovega izraza v omogočitvenem pogoju lahko dodatno omejimo izvajanje posameznega prehoda tudi v primeru, ko je proces prejel predvideni signal. Uporabi se lahko več zaporednih konstruktov, ki se pred izvedbo prehoda preverjajo v poljubnem vrstnem redu. Če kateri od dodatnih pogojev ni izpolnjen, se signal shrani za kasnejšo obravnavo. Dodatni omogočitveni pogoj se lahko uporabi tudi v povezavi s spontanim prehodom (razdelek 3.4.8).

Postopek pretvorbe konstrukta vhod je razdeljen na algoritma 3.20 in 3.21. Prvi se kliče med obdelavo vsakega stanja procesa. S parametri se prenesejo podatki o trenutnem procesu, stanju in signalu. V prvem odločitvenem stavku se preveri, ali smo prejeli zahtevo za pretvorbo signala zvezdica. V tem primeru se v okviru zanke z oznako R1 modelira sprejem za vse signale $\in spec.stru.proc.sigin \backslash spec.stru.proc.stanje.sigsta$. Obravnava se torej množica vseh signalov, ki jih lahko sprejme proces in njihov sprejem v trenutnem stanju ni eksplicitno definiran.

```
FPAR a ∈ (spec.stru.proc ∨ spec.stru.proced);
FPAR b ∈ a.stanje;
FPAR c ∈ spec.kom.sig;
START ;
R1:
DECISION ∃e : e ∈ c.param,
 ( TRUE ):
  DECISION f =<spremenljivka>,
                f ∈ (a.sprem);
   ( TRUE ):
    DECISION a ∈ spec.stru.proc ;
     ( TRUE ):
      TASK dodaj(par, f);
     ELSE:
      TASK i := poišči(f, (a.sprem));
      TASK dodaj(par, i);
    ENDDECISION;
   ELSE:
    DECISION e ∈ (spec.def.ntip);
     ( TRUE ):
      TASK dodaj(par,undefined_//e);
     ELSE:
      TASK dodaj(par,_);
    ENDDECISION;
  ENDDECISION;
  JOIN R1;
 ELSE:
ENDDECISION;
DECISION ∃ <omogočitveni pogoj>
 ( TRUE ):
  TASK pog := <boolov izraz>;
 ELSE:
  TASK pog := TRUE;
ENDDECISION;
TASK "  ::atomic{recv_sig(//pog//,//c.pmlime//) ->";
DECISION par = ∅;
 ( TRUE ):
  TASK "  input??//c.pmlime//(sender);";
 ELSE:
  TASK "  input??//c.pmlime//(sender,//par//);";
ENDDECISION;
TASK "  update_chan_tab();}";
RETURN;
```

Algoritem 3.21: Modeliranje sprejema signala

Ker ima vsak proces potencialno drugačen pridružen kanal, je potrebno podatke o parametrih signala hraniti na dveh mestih. Prvo, $spec.kom.sig$, hrani informacije o podatkovnem tipu posameznega parametra. Drugo, $spec.stru.proc.sigin$, pa vsebuje zaporedno mesto posameznega parametra v pridruženem kanalu procesa. S pomočjo funkcije $poišči()$ se na osnovi imena obravnavanega signala pridobi kazalec na njegov zapis v množici $spec.kom.sig$. Sledi klic algoritma 3.21, ki modelira sprejem signala. V zanki z oznako R1 se vrši analiza vseh dejanskih parametrov signala. Če se na posameznem mestu nahaja spremenljivka, se ta doda v seznam vseh dejanskih parametrov par na mesto, ki ga določa definicija kanala. V primeru procedure je potrebno z uporabo funkcije $poišči()$ pridobiti dejansko ime spremenljivke, ki je odvisno od mesta klica procedure.

Kadar je v konstruktu vhod dejanski parameter izpuščen, je potrebno zagotoviti primerno modeliranje izgube podatkov. Skladno s semantičnimi pravili Promele je ob sprejemu signala potrebno zagotoviti, da so uporabljeni vsi parametri signala in da so njihovi podatkovni tipi skladni z definicijo kanala. Zato smo v okviru algoritma 3.13 (str. 62) za vse

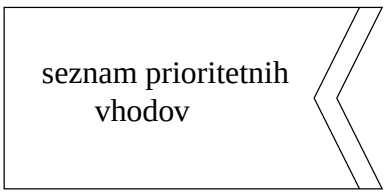
podatkovne tipe $\in$ *spec.def.ntip* pripravili skrite globalne spremenljivke z imenom podatkovnega tipa in predpono `undefined__`. Te se sedaj uporabijo za modeliranje izgube vrednosti parametra. Pri drugih podatkovnih tipih se lahko uporabi predefinirana spremenljivka _, v katero se lahko samo piše, branje pa ni dovoljeno.

Po pripravi dejanskih parametrov se preveri prisotnost omogočitvenega pogoja, ki določa vrednost začasne spremenljivke pog. V primeru, ko je omogočitveni pogoj prisoten, se pri sprejemu v modelu sistema doda dodatna vrstica, ki modelira preskok pri obravnavi signalov v vhodni vrsti v primeru, ko dodatni pogoj ni izpolnjen. Makro `recv__sig(pog,sig)` je predstavljen na sliki 3.39. V zadnjem delu algoritma se pripravi vse potrebno za dejanski sprejem signala. Postopek sprejema signala je opisan v naslednjem razdelku.

3.4.6 Prioritetni vhod

Posebna oblika vhoda je prioritetni vhod. Z njim lahko opišemo primere izvajanja, kjer želimo posameznemu signalu dati prednost pred drugimi signali — na primer signalu, ki sporoča kritično napako v sistemu. Vsak prioritetni vhod se v SDL-stroju modelira tako, da se tvorita dve novi stanji. Na prvo stanje so priključeni prioritetni signali, na drugo pa vsi signali. V prvem stanju se vsi neprioritetni signali zapišejo v konstrukt shrani, spontani prehodi pa se zavedejo v obeh stanjih. Slika 3.41 prikazuje grafično in tekstovno sintakso specifikacije konstrukta.

Grafična predstavitev



Tekstovni zapis

<prioritetni vhod> ::=
priority input <seznam prioritetnih vhodov>
<seznam prioritetnih vhodov> ::= <drugo>

<drugo> ::={<ime signala> | <ime časovnika>}
[([<spremenljivka> {,[<spremenljivka>]}*)]

Slika 3.41: EBNF zapis sintakse specifikacije konstrukta prioritetni vhod

Slika 3.42 prikazuje poskus modeliranja sprejema prioritetnih signalov. Osnovno stanje iz specifikacije sistema je razdeljeno na dve stanji. V prvem se lahko sprejmeta prioritetna signala, v drugem pa še dodatno neprioritetna signala. Predstavljeni model je na prvi pogled podoben modelu, ki ga opisuje [57] (str. 105).

Pošiljanje signalov se v prikazanem modelu izvede v atomarnem zaporedju. Ker število rež omogoča sprejem vseh predvidenih signalov, se pridružen kanal "napolni" v enem neprekinljivem koraku. Uporabljen naključni sprejem dopušča sprejem prioritetnih signalov ne glede na njihovo mesto v kanalu. To omogoča sprejem prioritetnega signala tudi v primeru, ko je v vhodni vrsti prisoten kakšen starejši signal (prioritetni ali neprioritetni). Tako

```
chan chan_client = [4] of {mtype,pid};
bool s_1,s_2,p_1,p_2;
mtype = {sig1,sig2,prio1,prio2};
/* sig - normalni signal
   prio - prioritetni signal */

proctype client(chan input; pid parent){
pid sender;
stanje:
end_stanje:
do
::input??[prio1,sender] ->
  input??prio1(sender)-> p_1=true;p_1=false
::input??[prio2,sender] ->
  input??prio2(sender)-> p_2=true;p_2=false
::!(input??[prio1,sender] ||
  input??[prio2,sender]) ->
end_NOPRIORITY_stanje:
  if
  :: input?sig1(sender) -> s_1=true;s_1=false
  :: input?sig2(sender) -> s_2=true;s_2=false
  :: input??prio1(sender)-> p_1=true;p_1=false
  :: input??prio2(sender)-> p_2=true;p_2=false
  fi;
od
}

init{
pid offspring;
offspring = run client(chan_client,_pid);
atomic{
 chan_client!sig1(_pid);
 chan_client!sig2(_pid);
 chan_client!prio1(_pid);
 chan_client!prio2(_pid)
}
}
```

Slika 3.42: Poskus modeliranja sprejema prioritetnega signala

```
(Spin Version 4.1.1 -- 2 January 2004)

Full statespace search for:
        never claim             - (not selected)
        assertion violations    - (disabled by -A flag)
        cycle checks            - (disabled by -DSAFETY)
        invalid end states      +

State-vector 32 byte, depth reached 22, errors: 0
      50 states, stored
       5 states, matched
      55 transitions (= stored+matched)
       6 atomic steps
hash conflicts: 0 (resolved)
(max size 2^14 states)

Stats on memory usage (in Megabytes):
0.002   equivalent memory usage for states (stored*(State-vector + overhead))
0.009   actual memory usage for states (unsuccessful compression: 487.78%)
        State-vector as stored = 172 byte + 4 byte overhead
0.066   memory used for hash table (-w14)
0.028   memory used for DFS stack (-m1000)
0.102   total actual memory usage

unreached in proctype client
        line 25, "pan_in", state 27, "-end-"
        (1 of 27 states)
unreached in proctype :init:
        (0 of 7 states)
```

Slika 3.43: Rezultat formalne verifikacije modela na sliki 3.42

v modelu na sliki 3.42 ni mogoče predvideti, kateri prioritetni signal se bo sprejel prvi. Skladno z modelom v [57] se mora sprejeti najstarejši prioritetni signal. Dodatni pregled modela pokaže, da je mogoče tudi izvajanje, pri katerem se kot prvi sprejme signal sig1.

Celovito analizo vseh možnih izvajanj sistema omogočajo sonde, ki so nameščene na mesto sprejema vsakega signala. Vsa možna zaporedja posameznih parov signalov smo preverili s formulo LTL $\Box(p \rightarrow \Diamond q) \wedge \Diamond p$. Z uporabo dodatnih makrojev se trditvama p in q dodeli vrednost posamezne sonde. Rezultati so pokazali, da so v vseh izvajanjih mogoča naslednja zaporedja: {sig1, sig2},{prio1, sig2} in {prio2, sig2}. Vrstni red sprejema signalov sig1, prio1 in prio2 torej ni določen. Slika 3.43 prikazuje rezultat verifikacije modela. Za opis posameznega stanja je bilo uporabljenih 32 zlogov, najdaljša pot izvajanja sistema

ima 22 stanj, vseh edinstvenih stanj v sistemu je 50 in šest korakov se je izvedlo v okviru atomarnega zaporedja.

V Promeli z obstoječimi konstrukti ni mogoče modelirati sprejema najstarejšega prioritetnega signala. Pravilni sprejem se lahko zagotovi samo z lastnim modeliranjem sprejema. Rešitev prikazanega primera prikazuje slika 3.44. Formalna verifikacija formule LTL $\Box(p_1 \rightarrow \Diamond p_2) \wedge \Box(p_2 \rightarrow \Diamond s_1) \wedge \Box(s_1 \rightarrow \Diamond s_2) \wedge \Diamond p_1$ dokaže, da je edino mogoče zaporedje v modelu sistema {prio1, prio2, sig1, sig2}.

Slika 3.45 prikazuje rezultat verifikacije modela. Za opis posameznega stanja je bilo uporabljenih 52 zlogov, najdaljša pot izvajanja sistema ima 125 stanj, vseh edinstvenih stanj v sistemu je 51 in 75 korakov se je izvedlo v okviru atomarnega zaporedja.

Algoritem 3.22 opisuje pretvorbo konstrukta prioritetni vhod. S parametri se prenesejo podatki o trenutnem procesu, stanju in signalu. Ker ima vsak proces potencialno drugačen pridružen kanal, je potrebno podatke o parametrih signala hraniti na dveh mestih. Prvo, $spec.kom.sig$, hrani informacije o podatkovnem tipu posameznega parametra. Drugo, $spec.stru.proc.sigin$, pa vsebuje zaporedno mesto posameznega parametra v pridruženem kanalu procesa. S pomočjo funkcije $poišči()$ se na osnovi imena obravnavanega signala pridobi kazalec na njegov zapis v množici $spec.kom.sig$.

V zanki z oznako R1 se vrši analiza vseh dejanskih parametrov signala. Če se na posameznem mestu nahaja spremenljivka, se ta doda v seznam vseh dejanskih parametrov par. V primeru procedure je potrebno z uporabo funkcije $poišči()$ pridobiti dejansko ime spremenljivke, ki je odvisno od mesta klica procedure.

Kadar je v konstruktu vhod dejanski parameter izpuščen, je potrebno zagotoviti primerno modeliranje izgube podatkov. Skladno s semantičnimi pravili Promele je ob sprejemu signala potrebno zagotoviti, da so uporabljeni vsi parametri signala in da so njihovi podatkovni tipi skladni z definicijo kanala. Zato smo v okviru algoritma 3.13 (str. 62) za vse podatkovne tipe $\in$ *spec.def.ntip* pripravili skrite globalne spremenljivke z imenom podatkovnega tipa in predpono `undefined__`. Te se sedaj uporabijo za modeliranje izgube vrednosti parametra. Pri drugih podatkovnih tipih se lahko uporabi predefinirana spremenljivka _, v katero se lahko samo piše, branje pa ni dovoljeno.

V zadnjem delu algoritma se pripravi vse potrebno za dejanski sprejem signala in pretvorbo telesa prehoda s klicem algoritma 3.18. Sprejem signala je omogočen samo za najstarejši signal v pridruženem kanalu s postavljeno prioritetno zastavico.

3.4.7 Implicitni prehod

Ob implicitnem prehodu končni avtomat ne izvaja nobenih nalog, temveč preide v izvorno stanje. Takšen prehod se pridruži vsem signalom, ki niso eksplicitno navedeni v konstruktih vhod, prioritetni vhod ali shrani in predstavlja njihovo izgubo. V modelu sistema je potrebno dodati posameznemu procesu implicitni sprejem za vse signale, ki jih

```
#define BUFF 4
#define none 0
typedef tab {mtype name;bool prio};
tab chan__tab[BUFF];

inline update_chan__tab(){
atomic{
do
::(ptr < BUFF-1) ->
  chan__tab[ptr].name=chan__tab[ptr+1].name;
  chan__tab[ptr].prio=chan__tab[ptr+1].prio;
  ptr++
:: else ->
  chan__tab[ptr].name=none;
  chan__tab[ptr].prio=false;
  break
od;
chan__sprejemnik_ptr--;
cur=0; } }

inline set__prio(sig){
  chan__tab[ptr].name==sig ->
  atomic{
    chan__tab[ptr].prio=true;
    chan__sprejemnik_prio=true } }

inline set__clear(){
  atomic{
    chan__tab[ptr].prio=false } }

#define recv__sig(sig)\
 (chan__tab[ptr].name==sig)

chan chan__sprejemnik =
     [BUFF] of {mtype,pid};
byte chan__sprejemnik_ptr;
bool chan__sprejemnik_prio;
bool s_1,s_2,p_1,p_2;
mtype = {sig1,sig2,prio1,prio2};

proctype sprejemnik(chan input;
                    pid parent){
byte cur;
pid sender;
byte ptr;
stanje:
end_stanje:
do
::chan__sprejemnik_ptr > cur  ->
    chan__sprejemnik_prio=false;
    cur++;
    ptr=0;
    atomic{
    do
    ::ptr <= BUFF-1 ->
      if
      :: set__prio(prio1)
      :: set__prio(prio2)
      :: else -> set__clear()
      fi;
      ptr++
    ::else -> goto stanje__start
    od; }
stanje__start:
    if
    ::chan__sprejemnik_prio==true ->
      ptr=0;
      do
      ::(ptr <= BUFF-1)
        && (chan__tab[ptr].prio==true) ->
        if
        ::atomic{recv__sig(prio1) ->
          input??prio1(sender);
          update_chan__tab(); }
          p_1=true;p_1=false;
          goto stanje
        ::atomic{recv__sig(prio2) ->
          input??prio2(sender);
          update_chan__tab(); }
          p_2=true;p_2=false;
          goto stanje
        fi;
      ::(ptr == BUFF) -> break;
      ::else -> ptr++
      od;
    ::else ->
      ptr=0;
      do
      ::(ptr <= BUFF-1) ->
        if
        ::atomic{recv__sig(sig1) ->
          input??sig1(sender);
          update_chan__tab(); }
          s_1=true;s_1=false;
          goto stanje
        ::atomic{recv__sig(sig2) ->
          input??sig2(sender);
          update_chan__tab(); }
          s_2=true; s_2=false;
          goto stanje
        fi;
      od;
    fi;
goto stanje;
od; }

init{
pid offspring;
offspring = run sprejemnik(chan__sprejemnik,_pid);
atomic{
  chan__sprejemnik!sig1(_pid);
  chan__tab[chan__sprejemnik_ptr].name=sig1;
  chan__sprejemnik_ptr++;

  chan__sprejemnik!sig2(_pid);
  chan__tab[chan__sprejemnik_ptr].name=sig2;
  chan__sprejemnik_ptr++;

  chan__sprejemnik!prio1(_pid);
  chan__tab[chan__sprejemnik_ptr].name=prio1;
  chan__sprejemnik_ptr++;

  chan__sprejemnik!prio2(_pid);
  chan__tab[chan__sprejemnik_ptr].name=prio2;
  chan__sprejemnik_ptr++ } }
```

Slika 3.44: Modeliranje sprejema prioritetnega signala

```
(Spin Version 4.1.1 -- 2 January 2004)

Full statespace search for:
        never claim             - (not selected)
        assertion violations    - (disabled by -A flag)
        cycle checks            - (disabled by -DSAFETY)
        invalid end states      +

State-vector 52 byte, depth reached 125, errors: 0
      51 states, stored
       0 states, matched
      51 transitions (= stored+matched)
      75 atomic steps
hash conflicts: 0 (resolved)
(max size 2^13 states)

Stats on memory usage (in Megabytes):
0.003   equivalent memory usage for states (stored*(State-vector + overhead))
0.042   actual memory usage for states (unsuccessful compression: 1454.76%)
        State-vector as stored = 811 byte + 4 byte overhead
0.033   memory used for hash table (-w13)
0.028   memory used for DFS stack (-m1000)
0.102   total actual memory usage

unreached in proctype sprejemnik
        line 105, "pan_in", state 139, "-end-"
        (1 of 139 states)
unreached in proctype :init:
        (0 of 15 states)
```

Slika 3.45: Rezultat formalne verifikacije modela na sliki 3.44

FPAR $a \in (spec.stru.proc \vee spec.stru.proced)$;
FPAR $b \in a.stanje$;
FPAR $((c \in (b.sigsta)) \vee$
$(c \in (a.stanje.sigsta) \wedge a.stanje.pmlime = *))$
START ;
TASK $d \in spec.kom.sig$;
TASK d := $poišči(c.pmlime, spec.kom.sig)$;
R1:
DECISION $\exists e : e \in d.param$,
(TRUE):
DECISION f =<spremenljivka>,
$f \in (a.sprem)$;
(TRUE):
DECISION $a \in spec.stru.proc$;
(TRUE):
TASK $dodaj(par, f)$;
ELSE:
TASK i := $poišči(f, (a.sprem))$;
TASK $dodaj(par, i)$;
ENDDECISION;
ELSE:
DECISION $e \in (spec.def.ntip)$;
(TRUE):
TASK $dodaj(par$,undefined_//$e)$;
ELSE:
TASK $dodaj(par$,_);
ENDDECISION;
ENDDECISION;
JOIN **R1**;
ELSE:
ENDDECISION;
TASK “ ::atomic{recv_sig(true,//$b.pmlime$//) $->$”;
DECISION $par = \emptyset$;
(TRUE):
TASK “ input??//$b.pmlime$//(sender);”;
ELSE:
TASK “ input??//$b.pmlime$//(sender,//par//);”;
ENDDECISION;
TASK “ update_chan_tab();”;
TASK “ }”;
CALL 3.18(a,b) // (str. 75)
RETURN;

Algoritem 3.22: Algoritem pretvorbe konstrukta prioritetni vhod

proces lahko sprejme, njihove neposredne obravnave pa nima definirane. To velja samo za stanja, kjer ni uporabljen vhod zvezdica, saj se v nasprotnem vsi signali iz množice $spec.stru.proc.sigin \backslash spec.stru.proc.stanje.sigsta$ sprejmejo v okviru tega vhoda.

Algoritem 3.23 za vsak tak signal definira prehod, ki signal prebere iz pridruženega kanala ter osveži polja za analizo trenutnega stanja kanala. Podobno kot v algoritmih 3.20 in 3.22 se tudi pri implicitnem sprejemu izvede sprejem signala, ki je prilagojen parametrom kanala. Vrednosti parametrov signala se v primeru seznamov zapišejo v globalne spremenljivke s predpono `undefined_`. Te so bile v okviru algoritma 3.1 (str. 27) deklarirane za vse uporabniške podatkovne tipe, ki določajo sezname. V primeru implicitnega sprejema signala se modelira izguba vseh dejanskih parametrov.

```
FPAR a ∈ (spec.stru.proc ∨ spec.stru.proced);
FPAR b ∈ a.stanje;
FPAR ((c ∈ (b.sigsta)) ∨
(c ∈ (a.stanje.sigsta) ∧ a.stanje.pmlime = *))
START ;
TASK d ∈ spec.kom.sig;
TASK d := poišči(c.pmlime, spec.kom.sig);
R1:
 DECISION ∃e : e ∈ d.param,
  ( TRUE ):
   DECISION e ∈ (spec.def.ntip);
    ( TRUE ):
     TASK dodaj(par,undefined_//e);
    ELSE:
     TASK dodaj(par,_);
   ENDDECISION;
   JOIN R1;
  ELSE:
ENDDECISION;
TASK "  ::atomic{recv_sig(true,//b.pmlime//) ->";
DECISION par = ∅;
 ( TRUE ):
  TASK "  input??//b.pmlime//(_);";
 ELSE:
  TASK "  input??//b.pmlime//(_,//par//);";
ENDDECISION;
TASK "  update_chan_tab();";
TASK "  goto //b.pmlime//;";
TASK "  }";
RETURN;
```

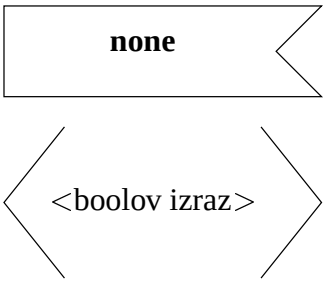
Algoritem 3.23: Algoritem modeliranja implicitnega prehoda

3.4.8 Spontani prehod

Spontani prehod predstavlja posebnost, saj lahko sproži izvajanje procesa brez prejetja signala iz vhodne vrste. Na logičnem nivoju je enakovreden drugim vhodom in ni definirane nobene prioritete pri izvajanju. Spremenljivka `sender`, ki določa pošiljatelja, se v tem primeru postavi na `_pid`. Slika 3.46 prikazuje definicijo konstrukta, algoritem 3.24 pa njegovo pretvorbo.

Če je v vhodni vrsti neobdelan signal, je potrebno izbiro med sprejetjem signala in spontanega prehoda, ki ima izpolnjen omogočitveni pogoj, izvršiti nedeterministično. Standard dodatno predvideva naključno aktivacijo spontanega prehoda tudi v primerih, ko je vhodna vrsta prazna. Pri formalni verifikaciji se preverjajo vsa možna izvajanja. Med mogoča izvajanja sodi tudi primer, ko se v neskončnem zaporedju izvaja samo spontani prehod. Brez dodatnih poštenostnih omejitev je tako formalna verifikacija zahtev po živosti omejena. Odločitev o izvajanju se modelira z naključno izbiro, ki podvoji število dosegljivih stanj sistema. Za podporo naključni izbiri je dodatno potrebno dopolniti procesni tip

Grafična predstavitev



Tekstovni zapis

<spontani prehod> ::=
input <oznaka spontanega vhoda> [<omogočitveni pogoj >]

<oznaka spontanega vhoda> ::= **none**

Slika 3.46: EBNF zapis sintakse specifikacije konstrukta spontani prehod

`__Timers()`, ki sodi v del razširitve orodja Spin za podporo diskretnemu času (slika 3.47). S postavitvijo prireditve `pv__random` na prvo mesto vplivamo na postopek pregledovanja najprej v globino pri formalni verifikaciji tako, da se najprej preverijo vse poti, kjer je spontani prehod onemogočen. Dodatno je potrebno vršiti naključno izbiro ob zaključku vsakega prehoda, zato se naključna izbira vrši takoj po prehodu v novo stanje — pred morebitno oznako s predpono `end_`, ki dodeli stanju status veljavnega končnega stanja.

FPAR $a \in \{(spec.stru.proc),$
$(spec.stru.proced)\}$;
FPAR $b \in (a.stanje)$
START ;
DECISION ∃ <omogočitveni pogoj>
(TRUE):
TASK pog := <boolov izraz>;
ELSE:
TASK pog := TRUE;
ENDDECISION;
TASK dopolni(__Timers);
TASK " :: atomic{(pv__random && (//pog//)) −>
random(); sender = _pid;}";
CALL 3.18(a,b) // (str. 75)
RETURN;

Algoritem 3.24: Algoritem za pretvorbo spontanega prehoda

```
bool pv__random;

inline random(){
if
:: pv__random = true
:: pv__random = false
fi
}

active proctype __Timers()
{
 end__Timers: do :: timeout -> random() od;
}
```

Slika 3.47: Naključna izbira in dopolnitev procesnega tipa `__Timers`

3.4.9 Časovnik

Ob specifikaciji sistemov v jeziku SDL se za opis odvisnosti izvajanja posameznih delov sistema od časa uporabi konstrukt časovnik. Časovnik je neaktiven do prve aktivacije. Aktivira

se s s konstruktom postavi, ki mu priredi vrednost tipa $TIME$. Ob izteku časovnika se skladno z [57] v vhodno vrsto procesa vstavi signal z enakim imenom. Enak dogodek se izvede v primeru, ko se časovniku priredi vrednost, ki je manjša ali enaka sistemski spremenljivki NOW, ki hrani trenutni absolutni čas sistema.

Še pred iztekom se lahko časovnik ponovno postavi z novim konstruktom postavi ali resetira s konstruktom resetiraj. Sistemska spremenljivka $sender$, ki določa pošiljatelja, se v tem primeru postavi na $self$. Signal, ki se prejme ob izteku časovnika, ima vse lastnosti signala SDL. Ob definiciji časovnika, katere sintakso prikazuje slika 3.48, se lahko definira več izrazov. Ob izteku časovnika prejeti signal vsebuje rezultate izrazov kot parametre signala v enakem vrstnem redu, kot so bili definirani. Časovnik postane neaktiven ob izteku s sprejetjem signala ali ob resetu. Kadar se resetira neaktiven časovnik, ta ostane neaktiven. Ob ponovnem aktiviranju že aktivnega časovnika se najprej izvrši reset in nato takoj ponovna postavitev z novimi parametri. Skladno z definicijo je časovnik ob tem ves čas aktiven.

Grafična predstavitev	**Tekstovni zapis**
set (<stavek “set”> +)	<postavi> ::= **set** (<stavek “set”>{,<stavek “set”>}∗) <stavek “set”> ::= ([<izraz tipa time>,]<ime časovnika> [(<seznam izrazov>)])
reset (<stavek reset> +)	<resetiraj> ::= **reset** (<stavek reset>{,<stavek reset>}∗) <stavek reset> ::= <ime časovnika>[(<seznam izrazov>)] <seznam izrazov> ::= <izraz> {, <izraz> }∗ >

Slika 3.48: EBNF zapis sintakse specifikacije konstrukta časovnik

Čas se v orodju DT Spin modelira z diskretnimi intervali, ki se indeksirajo z naravnimi števili in se merijo v tikih. Izjave med dvema zaporednima tikoma imajo enako časovno oznako. Čas v razširjenem modelu sistema teče samo v trenutkih, ko se vsi procesi blokirajo [7]. V teh trenutkih se sproži prekinitev $timeout$ v dodatno definiranem aktivnem procesu $__Timers()$ (slika 3.11, str. 39), ki se izvaja vzporedno z drugimi procesi v modelu sistema in zagotavlja, da se izvajanje modela (čas) nikoli ne ustavi. Ob zaključku posameznega časovnega intervala se vsem časovnikom v modelu zmanjša vrednost za ena. Časovniki so v modelu sistema predstavljeni s spremenljivkami tipa $timer$, ki je zapis z enim elementom tipa $short$ (slika 3.11). Ker se ob postavitvi časovnika uporabijo samo pozitivne vrednosti, je velikost posameznega časovnika omejena na $2^{15} - 1$ tikov[1]. Neaktiven časovnik določa vrednost -1. DT Spin ob tiku vsem časovnikom v modelu zmanjša vrednost za ena. Časovnik se izteče, ko ima vrednost nič. V tem trenutku postane izvršljiv stavek `expire` (sliki 3.49 in 3.11).

[1]Natančen razpon vrednosti določa uporabljena strojna oprema in prevajalnik za programski jezik C.

```
#define set(tmr,value) tmr.val=value
#define expire(tmr) (tmr.val==TOVAL)
#define reset(tmr) tmr.val=OFF
```

Slika 3.49: Dodatne definicije v modelu sistema za podporo časovnikom

V algoritmu 3.19 se v okviru zanke z oznako R8 doda vse potrebno za obravnavo izteka vseh časovnikov trenutnega procesa. Opis priprave modela za posamezen časovnik podaja algoritem 3.25. Signali, s katerimi modeliramo iztek časovnika, nosijo originalno ime časovnika iz specifikacije sistema. Spremenljivkam, ki predstavljajo trenutno vrednost časovnika, pa se doda predpona `tmr__`.

V zanki z oznako R1 se za vsak parameter uporabi začasna spremenljivka z imenom časovnika in zaporedno številko parametra. Uporaba začasnih spremenljivk je neizbežna, saj se vrednost parametrov določi ob postavitvi časovnika in ne ob izteku. Zato ni mogoče uporabiti originalnih izrazov, ki so se uporabili ob postavitvi časovnika, saj se je njihova vrednost lahko v vmesnem izvajanju spremenila. Definicija začasnih spremenljivk se izvede v algoritmu 3.15 (str. 70). Ob izteku časovnika pošlje proces v lastni pridružen kanal signal z imenom časovnika. Kot drugi parameter signala, ki vedno določa pošiljatelja, se zapiše lastni PId. Vrednost naslednjih parametrov določajo pomožne spremenljivke, ki hranijo dejanske vrednosti izrazov ob postavitvi časovnika. S funkcijo $preslikaj()$ se izvede preslikava parametrov časovnika na predvidena mesta v kanalu obravnavanega procesa.

FPAR $a \in (spec.stru.proc \vee spec.stru.proced)$;
FPAR $b \in a.časo$;
START ;
R1:
  TASK i:=0;
  DECISION $\exists c : c \in b.param$
    (TRUE):
      TASK $dodaj(par, b.pmlime//__//i)$;
      TASK $i := i + 1$;
      JOIN **R1**;
    ELSE:
ENDDECISION;
TASK $chanpar = preslikaj(a, par)$;
TASK ":: atomic{expire(tmr__//$b.pmlime$//) $->$";
TASK " input!//$b.pmlime$//(_pid,//$chanpar$//);";
TASK " table_channum_nsp[input].\";
TASK " data[table_channum_ptr[input]].name=";
TASK " //$b.pmlime$//;";
TASK " table_channum_ptr[input]++;";
TASK " reset(tmr__//$b.pmlime$//)};";
RETURN;

Algoritem 3.25: Algoritem za obravnavo izteka časovnika

Algoritem 3.26 opisuje pretvorbo konstrukta postavi. V prvem koraku pridobimo kazalec na časovnik v množici $spec.stru.proc.časo$. Sledi odločitveni stavek, ki preveri, ali definicija vsebuje element <izraz tipa time>, ki določa čas izteka časovnika. Če ta ni definiran, se uporabi privzeta vrednost, ki je bila podana ob definiciji časovnika (algoritem 3.6, str. 40). V zanki z oznako R1 se v atomarnem zaporedju priredi vrednost začasnim spremenljivkam, ki hranijo parametre časovnika. Posamezen izraz se pretvori s klicem algoritma 3.2. Prvi parameter je tipa IN/OUT in po zaključku algoritma hrani pretvorjen izraz v jeziku Promela. Kot drugi parameter podamo izraz, ki ga želimo pretvoriti. Sledi postavitev časovnika z makrojem `set(tmr,value)` (slika 3.49) in zaključek algoritma.

```
FPAR a ∈ (spec.stru.proc ∨ spec.stru.proced);
START;
 TASK b := poišči(<ime časovnika>, a.časo);
 DECISION ∃c : c = <izraz tipa time>;
  ( TRUE ):
  ELSE:
   DECISION b.izraz != NONE
    ( TRUE ):
     TASK c := b.izraz;
    ELSE:
     STOP;
   ENDDECISION;
 ENDDECISION;
 TASK i := 0;
R1:
 DECISION ∃d : d ∈ <seznam izrazov>;
  ( TRUE ):
   CALL 3.2(e,d); // (str. 30)
   TASK "atomic{"
   TASK "b.pmlime//_//i = e";
   JOIN R1;
  ELSE:
 ENDDECISION;
 CALL 3.2(f,c); // (str. 30)
 TASK "set(tmr_//b.pmlime//,//f//);"
 TASK "}"
 RETURN
```

Algoritem 3.26: Algoritem pretvorbe konstrukta postavi

Algoritem 3.27 opisuje pretvorbo konstrukta resetiraj. Ob resetu časovnika se v atomarnem zaporedju vsem pomožnim spremenljivkam priredi vrednost nič[2]. Če je v vhodni vrsti prisoten signal od časovnika, se zavrže. Z uporabo makroja `reset(tmr)` se časovnik deaktivira.

```
FPAR a ∈ (spec.stru.proc ∨ spec.stru.proced);
START;
 TASK b := poišči(<ime časovnika>, a.časo);
 TASK i := 0;
R1:
 DECISION ∃d : d ∈ <seznam izrazov>;
  ( TRUE ):
   TASK "atomic{
   TASK "b.pmlime//_//i = none";
   JOIN R1;
  ELSE:
 ENDDECISION;
 TASK i:=0;
 R2:
  DECISION ∃c : c ∈ (spec.stru.proc.časo.pmltip)
   ( TRUE ):
    DECISION c ∈ (spec.def.ntip);
     ( TRUE ):
      TASK dodaj(par,undefined_//e);
     ELSE:
      TASK dodaj(par,_);
    ENDDECISION;
    TASK i := i + 1;
    JOIN R2;
   ELSE:
 ENDDECISION;
 TASK chanpar = preslikaj(a, par);
 TASK "if";
 TASK ":: atomic{"
 TASK "   input??[//b.pmlime//(_,//chanpar//)] ->
          input??//b.pmlime//(_,//chanpar//)}"
 TASK ":: else -> skip";
 TASK "fi;";
 TASK "reset(tmr_//b.pmlime);"
 RETURN
```

Algoritem 3.27: Algoritem pretvorbe konstrukta resetiraj

V okviru konstrukta postavi ali resetiraj je lahko definiranih več stavkov "set" oz. stavkov reset. Model takšnega zapisa je enakovreden zaporednim stavkom v enakem vrstnem redu.

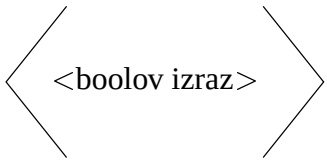
3.4.10 Trajen signal

Trajen signal se uporabi pri opisu, kadar želimo izvesti posamezen prehod v trenutku, ko postane veljaven izbran boolov izraz. Slika 3.50 prikazuje EBNF zapis sintakse specifikacije

[2] V primeru polj ali zapisov s takšnim prirejanjem ne postavimo vseh elementov na vrednost nič.

konstrukta. Prehod se lahko izvede le takrat, ko v pridruženem kanalu ni drugih signalov. Signali imajo torej prioriteto izvajanja. Zato se v modelu stanja, ki ga določa algoritem 3.19 (str. 79), obravnava le v primeru, ko na vhodu ni nobenega signala, ki bi ga proces v trenutnem stanju lahko obdelal. Tudi iztek časovnika ima prednost pred trajnim signalom.

Grafična predstavitev



Tekstovni zapis

<trajen signal> ::=
provided <boolov izraz>
[**priority** <celoštevilčni literal>]

Slika 3.50: EBNF zapis sintakse specifikacije konstrukta trajen signal

V primeru večjega števila trajnih signalov, ki so sočasno omogočeni, se izvede tisti prehod, ki sledi trajnemu signalu z najvišjo prioriteto. Prioriteto določa najnižje celo število, ki se zavede za rezervirano besedo **priority**. Če prioriteta trajnih signalov ni določena, je izbira nedeterministična. Prioriteta se določa tako, da se v stražarju signala postavi kot pogoj za izvajanje neizvršljivost vseh signalov z višjo prioriteto.

Algoritem 3.28 v zanki z oznako R1 shrani v začasno spremenljivko $traj$ kazalce na vse druge trajne signale v okviru obravnavanega stanja. Sledi modeliranje stražarja z lastnim boolovim izrazom. V zankah z oznako R2 in R3 se stražarju dodajo dodatni pogoji glede na prioritete drugih trajnih signalov in časovnikov v okviru obravnavanega procesa — iztek časovnikov ima prioriteto pred trajnim signalom. Kot zadnji pogoj za omogočitev prehoda je "prazna" vhodna vrsta. Sledi klic algoritma za pretvorbo telesa prehoda in zaključek algoritma.

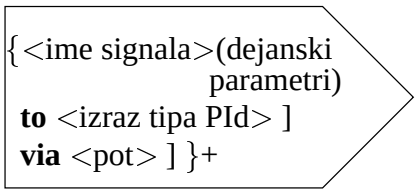
FPAR $a \in (spec.stru.proc \vee spec.stru.proced)$;
FPAR $b \in a.stanje.trajen$;
FPAR $f \in a.stanje$;
START;
R1:
DECISION $\exists c : c \in b.trajen, c \mathrel{!}= b$;
(TRUE):
TASK $dodaj(traj, c)$;
JOIN **R1**;
ELSE:
ENDDECISION;
TASK ":: atomic{//($b.pogoj$)//
R2:
DECISION $\exists d : d \in traj, d.prio < b.prio$;
(TRUE):
TASK " && !(//$d.pogoj$//)
JOIN **R2**;
ELSE:
ENDDECISION;
R3:
DECISION $\exists e : e \in a.časo$;
(TRUE):
TASK " && !(//expire(tm__//$a.časo.sdlime$//)//)
JOIN **R3**;
ELSE:
ENDDECISION;
TASK " !(table_channum_ptr[input] > pv__cur)) −>";
CALL 3.18(a,f) // (str. 75)
RETURN;

Algoritem 3.28: Algoritem za pretvorbo trajnega signala

3.4.11 Izhod

Signal se v grafu procesa ali procedure pošlje z uporabo konstrukta izhod, ki je prikazan na sliki 3.51. V zapisu konstrukta izhod se lahko navede več signalov s pridruženimi parametri, ki so ločeni z vejico. Takšen zapis se interpretira kot množica zaporednih konstruktov izhod, ki imajo skupnega naslovnika in predvideno pot.

Grafična predstavitev



Tekstovni zapis

```
<izhod> ::=
      output <ime signala> [<dejanski parametri>]
        {, <ime signala> [<dejanski parametri>]}*
        [to <naslov>][ via [ all ] <pot> ]
<naslov> ::= <izraz tipa PId> | <ime procesa>
<dejanski parametri> ::= (<izraz predvidenega tipa>,
                          {,<izraz predvidenega tipa>})*
<pot> ::= <element poti>{,<element poti>}*
<element poti> ::= <ime signalne poti> | <ime kanala>
```

Slika 3.51: EBNF zapis sintakse specifikacije konstrukta izhod

Grafični opis pozna dva dodatna konstrukta — notranji vhod in izhod. Njuna uporaba se odsvetuje in je prisotna v [57] zaradi skladnosti s starejšimi specifikacijami. Tekstovni opis obeh konstruktov je enak opisu konstruktov vhod in izhod. Ob uporabi izrazov na mestu dejanskih parametrov model predvideva samo enkratno interpretacijo izraza. Vrednost se shrani v začasno spremenljivko in se uporabi v nadaljnjih korakih.

Predstavitev komunikacije je podana v razdelkih 3.2 in 3.3.2. Nadaljujemo z analizo vseh potencialnih naslovnikov posameznega konstrukta izhod, ki predstavlja osnovo za celovito modeliranje komunikacije.

Postopek analize je razdeljen na dva dela. V algoritmu 3.29 se izvede analiza vseh konstruktov izhod v vseh procesih v specifikaciji sistema. Vsakemu procesu $\in spec.stu.proc$ se za vsak signal $\in spec.kom.sig$, ki se uporabi v katerem izmed konstruktov izhod v telesu procesa, pridruži pot od obravnavanega procesa do vseh veljavnih prejemnikov in njihovo edinstveno ime ($spec.stru.proc.sigout$, str. 59).

V primeru uporabe priključkov se pri analizi uporabi rekurzivni algoritem 3.30. Priključki se uporabijo v primeru mednivojskih povezav, kot je to prikazano na sliki 3.52. Slika je sestavljena iz treh delov. V zgornjem delu je podan grafični opis bloka SDL z imenom B. Blok sestavljata podbloka B1 in B2, ki ju povezujeta dva kanala — k1 in k2. Blok B1 ima dva procesa, P1 in P2, ki sta povezana med sabo in s kanaloma k1 in k2. Za povezavo se uporabi priključek. Blok B2 sestavljata podbloka B21 in B22. Prvi ima dva procesa, P1 in P2, drugi pa samo enega — P1. Poimenovanje poudarja veljavnost ponavljajočih se imen v strukturno različnih elementih. Bloka B21 in B22 sta povezana z nadblokom vsak s svojim kanalom. Prav vsi komunikacijski gradniki lahko prenašajo signal z imenom “sig1”. To je

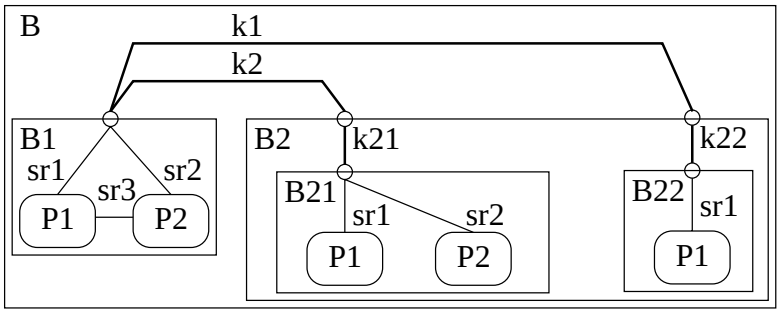
```
START ;
 R1:
  DECISION ∃a : a ∈ spec.stru.proc
   ( TRUE ):
   ELSE:
    END:
  ENDDECISION;
 R2:
  DECISION ∃b : b ∈ spec.kon, b = <izhod>;
   ( TRUE ):
   R3:
    DECISION ∃c : c ∈ b, c = <ime signala>;
     ( TRUE ):
      TASK i := poišči(c, (spec.kom.sig));
      TASK a.sigout.pmlsig := i;
     R4:
      DECISION ∃f : f ∈ (spec.kom.sigpot),
            f.izvor = a.pmlime, i ∈ f.signal;
       ( TRUE ):
        TASK dodaj(pot, f.ime);
        DECISION f.ponor;
         ( ∈ (spec.stru.proc) ):
          TASK g := proc(f.ponor);
          DECISION (a.sigout.pmlsig ∈ g.sigin;
           ( TRUE ):
            TASK dodaj(a.sigout.proc.pmlime,
                       g.pmlime);
            TASK dodaj(a.sigout.proc.pot,pot);
           ELSE:
            TASK dodaj(a.sigout.proc.pmlime,
                       SKIP);
            TASK dodaj(a.sigout.proc.pot,pot);
          ENDDECISION;
          JOIN R4
         ( ENV )
          CALL 3.30(a, f, pot, none);
        ENDDECISION;
       ELSE:
      ENDDECISION;
     ELSE:
      TASK shrani(a.sigout);
    ENDDECISION;
```

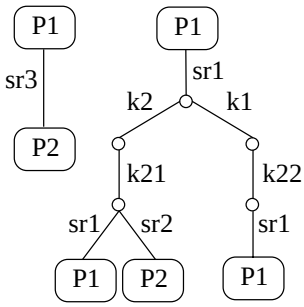
Algoritem 3.29: Algoritem analize vseh veljavnih naslovnikov za vse konstrukte izhod

razvidno iz spodnjega dela slike 3.52, ki prikazuje specifikacijo v tekstovni obliki, na osnovi katerega se vrši analiza. Analiza ne upošteva morebitnih dodatnih omejitev, ki ju podajata elementa definicije konstrukta izhod <naslov> in <pot> (slika 3.51). Naslovniki in veljavne poti se obravnavajo ob modeliranju konstrukta izhod v algoritmu 3.31. Analiza torej podaja vse veljavne naslovnike, ki jih določajo komunikacijski gradniki specifikacije.

Za namene predstavitve algoritmov se bo izvedla miselna analiza konstrukta izhod v procesu B/B1/P1, ki pošilja signal z imenom sig1. Na zgornjem desnem delu slike 3.52 sta prikazani drevesi vseh veljavnih poti za podani primer. Iz dreves poti je razvidno, da lahko signal prejmejo vsi preostali procesi v okviru bloka B. Posamezni deli besedila, ki se sklicujejo na specifikacijo, so označeni s številkami v oklepajih. Enako so označene vrstice v tekstovni obliki specifikacije v spodnjem delu slike 3.52. Zaradi večje preglednosti so iz opisa izpuščena telesa procesov.

V algoritmu 3.29 zanka z oznako R1 zagotavlja, da se izvede analiza vseh procesov v okviru specifikacije sistema. V naslednjem koraku se posamezno obravnavajo vsi konstrukti izhod, ki so definirani v okviru procesa. Ker se lahko v okviru konstrukta zapiše več signalov, se v zanki z oznako R3 izvede analiza za vse navedene signale. S pomočjo funkcije $poišči()$ se pridobi edinstveno ime signala ($spec.kom.sig.pmlime$), ki je veljavno na mestu definicije procesa. V nadaljevanju je potrebno poiskati vse procese, ki lahko sprejmejo obravnavani signal. V zanki z oznako R4 se posamezno pregledajo vse signalne poti, ki se zaključujejo v trenutnem procesu in lahko prenašajo obravnavani signal. Vsak veljaven element poti se med analizo doda v začasno spremenljivko pot.






```
BLOCK B;
SUBSTRUCTURE ;
 CHANNEL k1 (5)
  FROM B1 TO B2 WITH sig1;
  FROM B2 TO B1 WITH sig1;
 ENDCHANNEL;
 CHANNEL k2 (3)
  FROM B1 TO B2 WITH sig1;
  FROM B2 TO B1 WITH sig1;
 ENDCHANNEL;
 BLOCK B1;
  SIGNALROUTE sr3 (1)
   FROM P1 TO P2 WITH sig1;
   FROM P2 TO P1 WITH sig1;
  SIGNALROUTE sr1 (2)
   FROM P1 TO ENV WITH sig1;
   FROM ENV TO P1 WITH sig1;
  SIGNALROUTE sr2
   FROM P2 TO ENV WITH sig1;
   FROM ENV TO P2 WITH sig1;
 CONNECT k1,k2 AND sr1, sr2; (4)
 PROCESS P1;
 PROCESS P2;
 ENDBLOCK;
 BLOCK B2;
  SUBSTRUCTURE ;
   CHANNEL k22
    FROM B22 TO ENV WITH sig1;
    FROM ENV TO B22 WITH sig1;
   ENDCHANNEL;
   CHANNEL k21 (7)
    FROM B21 TO ENV WITH sig1;
    FROM ENV TO B21 WITH sig1;
   ENDCHANNEL;
   CONNECT k1 AND k22;
   CONNECT k2 AND k21;(6)
   BLOCK B21;
    SIGNALROUTE sr1 (9)
     FROM P1 TO ENV WITH sig1;
     FROM ENV TO P1 WITH sig1;
    SIGNALROUTE sr2
     FROM P2 TO ENV WITH sig1;
     FROM ENV TO P2 WITH sig1;
   CONNECT k21 AND sr1, sr2;(8)
   PROCESS P1;
   PROCESS P2;
   ENDBLOCK;
   BLOCK B22;
    SIGNALROUTE sr1
     FROM P1 TO ENV WITH sig1;
     FROM ENV TO P1 WITH sig1;
    CONNECT k22 AND sr1;
    PROCESS P1;
   ENDBLOCK;
```

Slika 3.52: Primer povezav med procesi bloka

Signalna pot se lahko zaključuje na sosednjem procesu, kot v primeru signalne poti B/B1/sr3 (1), ali pa se z uporabo priključka poveže na višjeležni element. Pri neposredni povezavi na sosednji proces se preveri, če lahko obravnavani signal ciljni proces obdela. Vsi signali, ki jih proces lahko dejansko sprejme in obdela, so $\in spec.proc.sigin$. V nasprotnem primeru se signal zavrže že na strani pošiljatelja. To se označi z rezervirano besedo “SKIP”. Če se signalna pot priključuje na višjeležni element, kot v primeru B/B1/sr1 (2), s klicem algoritma 3.30 sledimo komunikacijskim konstruktom vse do končnega procesa. Uporabi se postopek pregledovanja najprej v globino. Osnovni element nadaljnje analize je priključek, ki ga opisuje definicija 3.18 in analizira algoritem 3.12 (str. 52).

Posamezen element poti se lahko zaključuje na procesu ali priključku. Zaradi splošnosti algoritma ne ločimo med signalno potjo in kanalom. S formalnimi parametri algoritma se izvede prenos kazalcev na obravnavani proces in trenutni element poti ter inicializacija začasnih seznamov pot in obd. V primeru kanala k2 (3), ki je povezan na obeh straneh na priključek, se ob prvem klicu algoritma v prvem odločitvenem stavku v seznam obdelanih priključkov obd shrani priključek, ki povezuje B/B1/sr1 in B/B1/sr2 z B/k1 in B/k2 (4). V zanki R1 sledi analiza vseh komunikacijskih elementov, ki so na drugi strani priključka, ne glede na mesto definicije — na desni ali levi strani rezervirane besede AND.

```
FPAR a ∈ (spec.stru.proc)
FPAR j ∈ (spec.kom.(sigpot ∨ kan))
FPAR pot, obd
START ;
 DECISION ∃o : o ∈ (spec.kom.prik),
               j ∈ o.sez1, j ∉ obd;
  ( TRUE ):
   TASK dodaj(obd, j);
  R1:
   DECISION ∃p : p ∈ o.sez2;
    ( TRUE ):
     TASK dodaj(pot, p);
     DECISION p.ponor;
      ( ∈ (spec.stru.proc) ):
       TASK r := proc(p.ponor);
       DECISION (a.sigout.pmlsig ∈ r.sigin;
        ( TRUE ):
         TASK dodaj(a.sigout.proc.pmlime,
                    r.pmlime);
         TASK dodaj(a.sigout.proc.pot,pot);
        ELSE:
         TASK dodaj(a.sigout.proc.pmlime,
                    SKIP);
         TASK dodaj(a.sigout.proc.pot,pot);
       ENDDECISION;
       JOIN R1
      ELSE:
       CALL 3.30(a, p, pot, obd);
     ENDDECISION;
    ELSE:
     RETURN
   ENDDECISION;
ELSE:
 ENDDECISION;
 DECISION ∃o : o ∈ (spec.kom.prik),
               j ∈ o.sez2, j ∉ obd;
  ( TRUE ):
   TASK dodaj(obd, j);
  R2:
   DECISION ∃p : p ∈ o.sez1;
    ( TRUE ):
     TASK dodaj(pot, p);
     DECISION p.ponor;
      ( ∈ (spec.stru.proc) ):
       TASK r := proc(p.ponor);
       DECISION (a.sigout.pmlsig ∈ r.sigin;
        ( TRUE ):
         TASK dodaj(a.sigout.proc.pmlime,
                    r.pmlime);
         TASK dodaj(a.sigout.proc.pot,pot);
        ELSE:
         TASK dodaj(a.sigout.proc.pmlime,
                    SKIP);
         TASK dodaj(a.sigout.proc.pot,pot);
       ENDDECISION;
       JOIN R2
      ELSE:
       CALL 3.30(a, p, pot, obd);
     ENDDECISION;
    ELSE:
     RETURN
   ENDDECISION;
  ELSE:
 ENDDECISION;
```

Algoritem 3.30: Algoritem analize vseh povezav

V predstavljenem primeru sta to kanala B/k1 (5) in B/k2 (3). Najprej se preveri, ali se morebiti zaključujeta na procesu. V tem primeru to ni mogoče, zato se izvajanje nadaljuje s ponovnim klicem algoritma 3.30. V tem primeru se v prvem prehodu recimo odločimo za analizo kanala B/k2, ki ga hrani spremenljivka p.

Izvede se identična analiza kot v primeru signalne poti sr1. V prvem koraku se pridobi kazalec na priključek, ki zaključuje kanal. Ker se kanal lahko priključi samo na dva priključka in je prvi že na seznamu obdelanih priključkov, v tem prehodu algoritem obravnava (6). Na drugi strani priključka se nahaja kanal B/B2/k21 (7). Ker se kanal ne zaključuje na procesu, se izvede ponovni klic algoritma 3.30. Tokrat se v seznamu naslednjih komunikacijskih elementov pojavita signalni poti B/B2/B21/sr1 in B/B2/B21/sr2 (8). Najprej obdelamo pot sr1. V prvi odločitvi v okviru zanke R1 ugotovimo, da se signalna pot B/B2/B21/sr1 (9) zaključuje na procesu B/B2/B21/P1. Če se obravnavani signal nahaja v množici B/B2/B21/P1.$sigin$, se proces doda v seznam potencialnih prejemnikov signala. Zaradi morebitnih omejitev poti ob pošiljanju signala se shrani tudi celotna pot med izvornim in ponornim procesom. Izvajanje algoritma se nadaljuje v prejšnjem klicu, in sicer z

obravnavo signalne poti B/B2/B21/sr2. S tem se zaključi analiza potencialnih prejemnikov, ki se priključujejo na kanal B/k2. Algoritem nadaljuje z analizo kanala B/k1. Po obravnavi procesa B/B2/B22/P1 se analiza za signal sig1 zaključi. Izvajanje se nadaljuje v algoritmu 3.29 z naslednjim signalom v okviru konstrukta izhod, naslednjim konstruktom izhod ali telesom naslednjega procesa.

Po zaključeni analizi sledi modeliranje posameznih konstruktov izhod z upoštevanjem vseh dodatnih omejitev, ki jih lahko določajo elementi zapisov <naslov> in <pot> (slika 3.51). Za razliko od algoritma analize konstruktov, ki se izvaja samostojno, se algoritem 3.31 kliče iz algoritma 3.17 (str. 75) med pretvorbo grafa procesa. Kot dejanski parameter se ob klicu algoritma poda ime obravnavanega procesa.

Če se v konstruktu izhod navede več signalov, se ti obravnavajo posamezno v okviru zanke z oznako R1 v enakem vrstnem redu, kot so navedeni v specifikaciji. Sledi analiza vseh morebitnih omejitev poti, ki so podane v zapisu <pot> (slika 3.51). Navedeni elementi poti se shranijo v začasno spremenljivko pot. Naslovnik se lahko navede kot <izraz tipa PId>, <ime procesa> ali pa se izpusti.

V primeru neposrednega naslavljanja z uporabo <izraz tipa PId> se naslovnik določa dinamično med izvajanjem modela. Pri modeliranju je potrebno upoštevati morebitne dodatne omejitve, ki jih narekuje seznam dovoljenih komunikacijskih elementov pot. Ker je signal potrebno prilagoditi ciljnemu procesu oz. njegovemu pridruženemu kanalu, moramo v tem primeru pripraviti izhodne stavke s stražarji za vse mogoče naslovnike. V odločitvenem stavku z oznako R3 poiščemo vse potencialne naslovnike za obravnavani signal, ki so skladni z omejitvami poti.

V zanki z oznako R4 se tvori seznam vseh dejanskih parametrov za uporabo pri modeliranju pošiljanja signala. Zaporedno mesto parametra je odvisno od definicije naslovnikovega pridruženega kanala. Če kateri izmed parametrov ni naveden, se na njegovo mesto zapiše `null` oz., v primeru zapisov, pomožna spremenljivka z imenom podatkovnega tipa spremenljivke in predpono `undefined__`. To se opravi v okviru funkcije $dodaj()$.

Na osnovi dejanskega PId-a se pridobi številka imena kanala, ki v povezavi s stražarjem zagotovi, da je oddaja signala prilagojena pridruženemu kanalu naslovnika. Podrobnejša razlaga dinamične izbire naslovnika je podana s primerom sistema na sliki 3.29 in njegovim modelom na sliki 3.30 (str. 67).

Pri naslavljanju z uporabo <ime procesa> v izhodnem stavku se v prvem koraku preveri, ali navedeno ime predstavlja relativni ali absolutni identifikator procesa. Prvi se lahko uporabi samo znotraj istega bloka in ne vsebuje neposrednega opisa mesta v strukturi sistema. Rezultat funkcije poišči je edinstveno ime naslovnika, ki je lahko tudi skupina procesov — v primeru, ko se proces med izvajanjem lahko pojavi v več primerkih. Tudi v tem primeru se iskanje ciljnih procesov omeji glede na morebitne dodatne omejitve poti v konstruktu <pot>.

```
FPAR a ∈ (spec.stru.proc)
START ;
R1:
 DECISION ∃z : z ∈<izhod>,
                z =<ime signala>;
  ( TRUE ):
   TASK i := poišči(z, (spec.kom.sig));
  R2:
   DECISION ∃b : b ∈<pot>;
    ( TRUE ):
     TASK i=poišči(b, (spec.kom.(sigpot
                    ∨ kan)));
     TASK dodaj(pot, i);
     JOIN R2;
    ELSE:
   ENDDECISION;
  DECISION ∃d : d ∈<naslov>, d;
   ( <izraz tipa PId> ):
    TASK "if";
   R3:
    DECISION ∃e : e ∈ (a.sigout.proc), pot ⊂ e.pot
     ( TRUE ):
      DECISION e.pmlime = SKIP;
       ( TRUE ):
        TASK s := TRUE;
       ELSE:
       R4:
        DECISION ∃c : c ∈ i.param;
         ( TRUE ):
          TASK dodaj(par, e.pmlime, c);
          JOIN R4;
         ELSE:
        ENDDECISION;
        TASK m := 0;
       R5:
        DECISION m < a.proc.maks;
        + ( TRUE ):
         TASK ":: atomic{table_pid_channame[//d//]
            == chan_//e.pmlime//_select + //m// −>";
         TASK "table_pid_channum[//d//]!//i//(_pid,//par//)"
         TASK j := table_pid_channum[//d//];
         TASK k := table_channum_ptr[//j//];
         TASK "table_channum_nsp[//j//].data[//k//].name=//i//;
         TASK "table_channum_ptr[//j//]++;"
         TASK "}"
         TASK m := m + 1;
         JOIN R5;
        ELSE :
       ENDDECISION;
      JOIN R3;
     ENDDECISION;
    ELSE:
     DECISION s = TRUE;
      ( TRUE ):
       TASK ":: skip";
      ELSE:
     ENDDECISION;
    TASK ":: else −> pv__runtime_error = true";
    TASK "fi";
   ENDDECISION;
  ( <ime procesa> ):
   TASK nasl = poišči(<ime procesa>,(spec.stru.proc));
   DECISION ∃e : e ∈ (a.sigout.proc),
                 e.pmlime = nasl,
                 pot ⊂ e.pot
    ( TRUE ):
    R6:
     DECISION ∃c : c ∈ i.param;
      ( TRUE ):
       TASK dodaj(par, e.pmlime, c);
       JOIN R6;
    ELSE:
   ENDDECISION;
   TASK "atomic{";
   TASK "d_step{";
   TASK "pfv_tmp=0;";
   TASK "do";
   TASK ":: pfv_tmp < table_//e.pmlime//_free −>";
   TASK "   pfv_tmp++";
   TASK ":: break;"
   TASK "od;";
   TASK "}";
   TASK "chan_//e.pmlime//[pfv_tmp]!//i//(_pid,//par//);";
   TASK j :=  table_channame_channum[
                chan__//e.pmlime//__select + pfv_tmp];
   TASK k := table_channum_ptr[//j//];
   TASK "table_channum_nsp[//j//].data[//k//].name=//i//;";
   TASK "table_channum_ptr[//j//]++;";
   TASK "}";
  ELSE:
   TASK "skip;";
  ENDDECISION;
 ELSE:
  TASK "if";
  R7:
   DECISION ∃e : e ∈ (a.sigout.proc),
                 pot ⊂ e.pot
    ( TRUE ):
     DECISION e.pmlime = SKIP;
      ( TRUE ):
       TASK s := TRUE;
      ELSE:
       R8:
        DECISION ∃c : c ∈ i.param;
         ( TRUE ):
          TASK dodaj(par, e.pmlime, c);
          JOIN R8;
         ELSE:
        ENDDECISION;
        TASK ":: atomic{true −>";
        TASK "d_step{";
        TASK "pfv_tmp=1;";
        TASK "do";
        TASK ":: pfv_tmp < table_//e.pmlime//_free −>";
        TASK "   pfv_tmp++";
        TASK ":: break;"
        TASK "od;";
        TASK "pfv_tmp−−;";
        TASK "};";
        TASK "chan_//e.pmlime//[//pfv_tmp//]!//i//(_pid,//par//);";
        TASK j :=  table_channame_channum[
                     chan__//e.pmlime//__select + //pfv_tmp//];
        TASK k := table_channum_ptr[//j//];
        TASK "table_channum_nsp[//j//].data[//k//].name=//i//;";
        TASK "table_channum_ptr[//j//]++;";
        TASK "}";
      ENDDECISION;
      JOIN R7;
     ELSE:
      DECISION s = TRUE;
       ( TRUE ):
        TASK ":: skip";
       ELSE:
      ENDDECISION;
      TASK "fi";
    ENDDECISION;
  ENDDECISION;
  JOIN R1;
 ELSE:
  END;
ENDDECISION;
```

Algoritem 3.31: Algoritem pretvorbe konstrukta izhod

Če ne obstaja veljavna pot do naslovnika, se signal zavrže na mestu pošiljanja s stavkom skip. V nasprotnem primeru se pripravi izhodni stavek, ki v prvem koraku izvede nedeterministično izbiro med aktivnimi primerki procesa. Ime kanala v modelu sistema se sestavi z uporabo rezervirane besede “chan”, edinstvenega imena procesa in izbrane številke primerka. Za znakom “!” sledi edinstveno ime signala, PId pošiljatelja in morebitni dejanski parametri signala.

Zadnji del algoritma obravnava naslavljanje, ki ne vsebuje elementa <naslov>. V tem primeru sodijo v množico veljavnih naslovnikov vsi procesi, do katerih obstaja veljavna pot za obravnavani signal. Če se v imenu procesa pojavi rezervirana beseda “SKIP”, se postavi zastavica z imenom s. V nasprotnem se v modelu sistema tvori vedno izvršljiva veja, ki modelira nedeterministično izbiro veljavnega primerka procesa. Takšna izbira se pripravi za vse veljavne naslovnike. Ker so vse veje pogojnega stavka “if” izvršljive, se med njimi vrši nedeterministična izbira. V tem primeru je nedeterminizem vpeljan v dveh korakih. V prvem se izvede izbira ciljnega procesnega tipa, v drugem pa se vrši izbira aktivnega primerka. Zanka z oznako R5 se zaključi, ko so modelirani izhodni stavki za vse veljavne naslovnike. Sledi modeliranje izgube signala za primere, ko veljavni naslovnik signala ne more obdelati. V tem primeru se pošiljanje signala nadomesti z vedno izvršljivim izrazom `skip`.

S takšnim modeliranjem komunikacije je v celoti podprt standard [57]. Podprte so tudi vse specifikacije sistemov z orodjem OG, ki predstavljajo podmnožico rešitev, ki jih predpisuje standard. Orodji $sdl2if$ in $if2pml$ podpirata bistveno manjši nabor rešitev, ki ne vključujejo nedeterministične izbire, omejitev poti, komunikacije z različnimi primerki procesa in naslavljanja z uporabo PId-a.

3.4.12 Ustvari

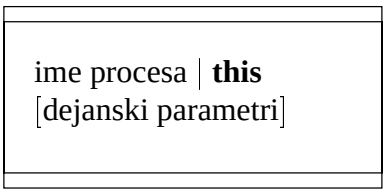
Med izvajanjem sistema lahko vsak proces poda zahtevo za tvorbo novega primerka procesa, ki je definiran v okviru skupnega bloka. Ob opisu se takšna zahteva poda z uporabo konstrukta ustvari. Slika 3.53 prikazuje grafični in tekstovni zapis konstrukta. Novemu procesu se dinamično dodeli edinstvena identifikacijska številka tipa PId. Številka zadnjega tvorjenega procesa se hrani v spremenljivki `offspring`. V novem procesu se spremenljivki `parent` priredi PId procesa, ki je podal zahtevo po tvorbi. S tem se omogoči sledenje posameznim procesom in nedvoumno naslavljanje posameznih primerkov procesa.

Ob tvorbi se procesu dodeli vhodna vrsta, tvorijo se lokalne spremenljivke in interpretirajo izrazi v dejanskih parametrih, ki so podani ob zahtevi. Izrazi se interpretirajo v istem vrstnem redu, kot so bili podani. Če kateri izmed dejanskih parametrov ni podan, pridružena lokalna spremenljivka nima definirane vrednosti. Sledi izvedba prvega prehoda. V nadaljevanju se proces izvaja asinhrono in sočasno s preostalimi procesi sistema.

Kadar število primerkov procesa preseže največjo dovoljeno vrednost, ki je bila podana ob definiciji, se zahteva po tvorbi ignorira. Procesu, ki je podal takšno zahtevo, se

spremenljivka `offspring` postavi na vrednost `Null`, interpretacija izvajanja procesa pa se nadaljuje nemoteno. Ob tvorbi modela z orodjem $sdl2pml$ lahko spremenimo obnašanje modela tako, da se takšni poskusi ob formalni verifikaciji interpretirajo kot napake (algoritem 3.32). Rezervirana beseda **this** se lahko uporabi ob ustvarjanju novega primerka trenutnega procesa.

Grafična predstavitev



Tekstovni zapis

<ustvari> ::=
create {<ime procesa> | **this**} [<dejanski parametri>]

Slika 3.53: EBNF zapis sintakse specifikacije konstrukta ustvari

```
FPAR a ∈ (spec.stru.proc ∨ spec.stru.proced);
START ;
 TASK b ∈ spec.stru.proc;
 TASK c ∈ spec.stru.blok;
 DECISION ∃b : (b ∈ c) ∧ (a ∈ c)
               ∧ b.sdlime = <ime procesa>
  ( TRUE ):
  ELSE:
   STOP;
 ENDDECISION;
 R1:
  DECISION ∃d : d ∈ b.fpar;
   ( TRUE ):
    DECISION ∃f : f =<izraz predvidenega tipa>
     ( TRUE ):
      TASK dodaj(par, f);
     ELSE:
      DECISION d.sdltip ∈ (spec.def.ntip);
       ( TRUE ):
        TASK dodaj(par,undefined__//d.pmltip);
       ELSE:
        TASK dodaj(par,_);
      ENDDECISION;
    ENDDECISION;
    JOIN R1;
   ELSE:
  ENDDECISION;
  TASK " atomic{";
  TASK " if";
  TASK " :: table__//b.pmlime//__free <
  table__//b.pmlime//__max ->";
  DECISION b.sigin = ∅;
   ( TRUE ):
    DECISION par = ∅;
     ( TRUE ):
      TASK " offspring = run //b.pmlime//(_pid)";
     ELSE:
      TASK " offspring = run //b.pmlime//(_pid,//par//)";
    ENDDECISION;
   ELSE:
    TASK " offspring = run";
    DECISION par = ∅;
     ( TRUE ):
      TASK " //b.pmlime//(chan__//b.pmlime//\
              [table__//b.pmlime//__free], _pid);";
     ELSE:
      TASK " //b.pmlime//(chan__//b.pmlime//\
              [table__//b.pmlime//__free], _pid,//par//);";
    ENDDECISION;
    TASK " table_pid_channum[offspring] =";
    TASK " chan__//b.pmlime//\
              [table__//b.pmlime//__free];";
    TASK " table_pid_channame[offspring] =";
    TASK " chan__//b.pmlime//__select +
              table__//b.pmlime//__free;";
    TASK " table_channame_channum\
              [chan__//b.pmlime//__select
              + table__//b.pmlime//__free] =";
    TASK " chan__//b.pmlime//\
              [table__//b.pmlime//__free]";
   ENDDECISION;
   TASK " if";
   TASK " ::(offspring==0) -> error = true";
   TASK " ::(offspring!=0) -> \
              table__//b.pmlime//__free++";
   TASK " fi";
   TASK " :: else -> error = true";
   TASK " fi;";
   TASK " }";
   RETURN;
```

Algoritem 3.32: Algoritem pretvorbe konstrukta ustvari

Jezik Promela loči med terminacijo procesa in njegovo smrtjo. Proces zaključi z izvajanjem, ko pride do konca svojega telesa. V avtomatsko tvorjenem modelu se to modelira s skokom na `end: skip`. Stavek $skip$ se med formalno verifikacijo obravnava prednostno, zato ni mogoče, da proces nikoli ne bi zaključil z izvajanjem. Težave nastopijo ob verifikaciji specifikacij, ki ne predpisujejo tvorbe in terminacije procesov v obratnem vrstnem redu. Proces se lahko ubije le takrat, ko ni aktiven noben proces, ki se je med izvajanjem sistema tvoril kasneje. Šele ob smrti se sprosti tudi PId, ki ga proces zaseda. V takšnih primerih je potrebno, zaradi tabel za preslikavo med PId-om procesa, sinonimom njegovega imena in številko kanala, konstanto `cv_procnum` postaviti na 256. S tem se poveča prostorska zahtevnost formalne verifikacije.

Večjo omejitev predstavlja postopek sproščanja PId-ov, saj pri določeni podskupini veljavnih specifikacij onemogoča izvedbo algoritmov formalne verifikacije v celoti. To velja za vse specifikacije, ki dinamično tvorijo primerke procesov in ti ne zaključijo z delovanjem v obratnem vrstnem redu, kot so bili tvorjeni. V teh primerih, se izredno poveča tudi množica mogočih izvajalnih poti, kar poveča tako prostorsko, kot časovno zahtevnost formalne verifikacije. Delno rešitev tega problema predstavlja dopolnitev orodja Spin, ki prepreči prekinitev formalne verifikacije ob poskusu prekoračitve maksimalnega števila dovoljenih procesov, temveč, skladno s semantiko jezika Promela, tvorbo samo zavrne. Rešitev problema, ki bi v celoti poenotila izvajanje modela sistema z obravnavo specifikacije v SDL-stroju, zahteva spremembo semantičnih pravil jezika Promela in dopolnitev mnogih algoritmov v orodju Spin.

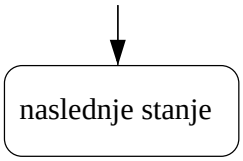
3.4.13 Naslednje stanje

Po zaključenem prehodu se v specifikaciji določi naslednje eksplicitno stanje. Kadar želimo, da se avtomat po izvedenem prehodu vrne v prejšnje stanje, torej v stanje, kjer se je prehod pričel izvajati, se lahko uporabi za opis naslednjega stanja posebna oznaka "`-`". Enakovreden opis bi predstavljal opis s polnimi imeni stanj. Takšen opis prispeva k preglednosti specifikacije. Slika 3.54 prikazuje EBNF zapis sintakse specifikacije konstrukta. Pretvorba je definirana z algoritmom 3.33. V primeru uporabe "`-`" se s klicem funkcije $stanje()$ pridobi ime trenutnega eksplicitnega stanja.

START ;
DECISION a = <telo konstrukta>, a;
(-):
  TASK i := $stanje()$;
ELSE:
  TASK i := <ime stanja>";
ENDDECISION;
TASK " goto //i";
RETURN;

Algoritem 3.33: Algoritem pretvorbe konstrukta naslednje stanje

Grafična predstavitev **Tekstovni zapis**



<naslednje stanje> ::=
nextstate <telo konstrukta>

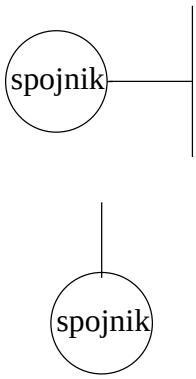
<telo konstrukta> ::={<ime stanja> | −}

Slika 3.54: EBNF zapis sintakse specifikacije konstrukta naslednje stanje

3.4.14 Oznaka in združi

Skok med prehodi v specifikaciji omogočata konstrukta oznaka in združi. Z oznako se označi del specifikacije, kjer želimo, da se nadaljuje izvajanje prehoda, ki se zaključi s konstruktom združi z enako oznako spojnika. Slika 3.55 prikazuje EBNF zapis specifikacije konstruktov.

Grafična predstavitev **Tekstovni zapis**



<oznaka> ::= <ime spojnika>:

<združi> ::= **join** <ime spojnika>

Slika 3.55: EBNF zapis sintakse specifikacije konstruktov oznaka in združi

Imena vseh oznak morajo biti znotraj definicije delovanja procesa ali procedure edinstvena. Za vsak konstrukt združi mora obstajati natančno en konstrukt oznaka. Prenos izvajanja je mogoč samo v okviru posameznega procesa ali procedure.

Pretvorba obeh konstruktov je trivialna. Algoritma 3.34 in 3.35 v prvem odločitvenem stavku preverita ali se vrši pretvorba v okviru procedure. V tem primeru je potrebno, glede na mesto klica, določiti edinstveno ime spojnika v okviru procesa.

```
FPAR a ∈ (spec.stru.proc ∨ spec.stru.proced);
START ;
TASK i := <ime spojnika>;
DECISION a ∈ spec.stru.proc ;
 ( TRUE ):
  TASK j := pot()//i;
 ELSE:
  TASK j := i;
ENDDECISION;
TASK "j//:";
RETURN;
```

Algoritem 3.34: Algoritem za pretvorbo konstrukta oznaka

```
FPAR a ∈ (spec.stru.proc ∨ spec.stru.proced);
START ;
TASK i := <ime spojnika>;
DECISION a ∈ spec.stru.proc ;
 ( TRUE ):
   TASK j := pot()//i;
 ELSE:
   TASK j := i;
ENDDECISION;
TASK "goto //j//;";
RETURN;
```

Algoritem 3.35: Algoritem za pretvorbo konstrukta združi

3.4.15 Naloga

Konstrukt naloga lahko vsebuje enega ali več prireditvenih stavkov ali neformalnih besedil. Uporaba več prireditvenih stavkov ali neformalnih besedil znotraj enega konstrukta je enakovredna njihovi zaporedni uporabi. Slika 3.56 prikazuje EBNF zapis sintakse specifikacije naloge. Dodatne definicije prikazujeta sliki 3.5 (str. 31) in 3.57.

Grafična predstavitev

```
 prireditveni stavek *
| neformalno besedilo *
```

Tekstovni zapis

```
<naloga> ::=
      task <telo naloge>;

<telo naloge> ::= {<prireditveni stavek> {, <prireditveni stavek>}*}
                | {<neformalno besedilo> {,<neformalno besedilo>}*}

<prireditveni stavek> ::= <ime spremenljivke> := <izraz>

<neformalno besedilo> ::= <niz znakov>
```

Slika 3.56: EBNF zapis sintakse specifikacije konstrukta naloga

Pretvorbo konstrukta opisuje algoritem 3.36. V prvem delu algoritma se definirata dve začasni spremenljivki. Prva lahko hrani podatkovni tip obravnavane spremenljivke, druga pa pretvorjen izraz v jeziku Promela. V prvem odločitvenem stavku preverimo, ali telo naloge vsebuje neformalno besedilo. V tem primeru se vsebina zapiše v modelu sistema kot komentar. Drugače je v konstruktu naloga uporabljen prireditveni stavek. S klicem funkcije $poišči()$ pridobimo podatkovni tip obravnavane spremenljivke. V naslednjem koraku algoritem 3.2 shrani pretvorjen prireditveni izraz v začasno spremenljivko l.

Pri modeliranju prirejanja je potrebno upoštevati posebnosti posameznih podatkovnih tipov, ki izhajajo iz izbrane predstavitve v razdelku 3.1, in pomanjkanja podpore neposrednemu prirejanju zapisov v jeziku Promela. V primeru polja prireditveni stavek vsebuje dva izraza. Dodatni izraz opisuje uporabljeni indeks polja. Pretvorba se izvede s klicem algoritma 3.2 in se shrani v začasno spremenljivko n, ki se v naslednjem koraku uporabi ob modeliranju prirejanja. Pri spremenljivkah tipa $pt__charstring$ se prirejanje izvede s klicem definicije inline $strcpy()$ (slika 3.1, str. 22).

Algoritem 3.37 izvaja prirejanje elementov zapisa. V prvem koraku z uporabo funkcije $poišči()$ pridobimo kazalec na strukturo obravnavanega zapisa. V zanki z oznako R1

```
<niz znakov> ::= <opuščaj> {<alfanumerični znak> | <drugi znak> | <posebni znak> | <pika> |
                 <podčrtaj> | <presledek> | <opuščaj> <opuščaj>}* <opuščaj>

<alfanumerični znak> ::= <črka> | <cifra> | <nacionalni znak>

<črka> ::= A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z
           | a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z

<cifra> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<nacionalni znak> ::= # | ‘ | ¤ | @ |  | <levi oglati oklepaj> | <desni oglati oklepaj>
                      | <levi zaviti oklepaj> | <navpična črta> | <desni zaviti oklepaj>
                      | <vijuga> | <strešica>

<opuščaj> ::= ’
<levi oglati oklepaj> ::= [
<desni oglati oklepaj> ::= ]
<levi zaviti oklepaj> ::= {
<navpična črta> ::= |
<desni zaviti oklepaj> ::= }
<vijuga> ::= ˜
<strešica> ::= ˆ
<pika> ::= .
<podčrtaj> ::= _
<drugi znak> ::= ? | & | %
<posebni znak> ::= + | - | ! | / | > | * | ( | ) | ”| , | ; | < | = | :
```

Slika 3.57: EBNF zapis sintakse niza znakov

```
FPAR a ∈ (spec.stru.proc ∨ spec.stru.proced);
TASK k ∈ spec.def.(stip ∨ ntip ∨ ptip);
TASK l ∈ <izraz promela>;
DECISION i = <telo naloge>, i;
 ( <neformalno besedilo> ):
  TASK komentar(i)”;
 ELSE:
  TASK j := <ime spremenljivke>;
  TASK k := poišči(j, a.sprem);
  TASK o := <izraz>;
  CALL 3.2(l,o); (str. 30)
  DECISION k.tip;
   ( zapis ):
    CALL 3.37(k,o); (str. 109)
   ( polje ):
    TASK m ∈ spec.def.ntip.polje.ind
    CALL 3.2(n,m); (str. 30)
    TASK “j//.val[//n//] = //l//;”;
   ( unsigned ):
    TASK “j//.val = //l//;”;
   ( int, short, bool, bit, chan, mtype ):
    TASK “j// = //l//;”;
   ( pt__charstring ):
    TASK “strcpy(//j//,//l//)”;
   ( pt__character ):
    TASK “j//.fchar=//l//;”;
   ELSE:
  ENDDECISION;
ENDDECISION;
RETURN;
```

Algoritem 3.36: Algoritem za pretvorbo konstrukta naloga

priredimo vrednosti vsem elementom zapisa z uporabo funkcije $priredi()$. Še pred tem se iz celotnega izraza s klicem funkcije $izraz()$ pridobi podizraz, ki določa novo vrednost obravnavanega elementa. V primeru, ko je element tudi zapis, je izvajanje algoritma rekurzivno.

3.4.16 Odločitev

Reaktivni sistemi se odzivajo na pobude iz okolja. Posamezen odziv je odvisen od specifikacije sistema, trenutnega globalnega stanja sistema in vhodnih pobud. Z uporabo konstrukta

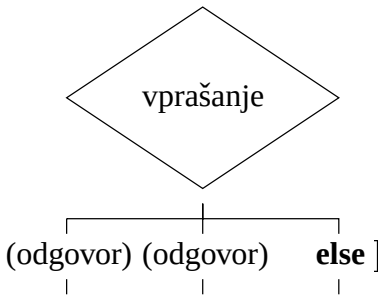
```
FPAR k ∈ a.sprem;
FPAR o ∈ <izraz>;
TASK m := poišči(a.sprem.pmltip, (spec.def.ntip.zapis));
START ;
R1:
 DECISION ∃n : n ∈ m;
  ( TRUE ):
   DECISION n.tip ∈ (spec.def.ntip.zapis);
    ( TRUE ):
     TASK p := izraz(o);
     CALL 3.37(n,p);
    ELSE:
     TASK p := izraz(o);
     TASK priredi(n.ime, p);
   ENDDECISION;
   JOIN R1;
  ELSE:
   RETURN ;
 ENDDECISION;
```

Algoritem 3.37: Algoritem za prirejanje vrednosti zapisom

odločitev lahko posamezen prehod razdelimo na več vej. Odločitev lahko vsebuje več konstruktov odgovor. Vsak odgovor predstavlja svojo pot nadaljevanja izvajanja prehoda.

Izjemo predstavlja vprašanje, kjer smo uporabili rezervirano besedo **any**. Interpretacija takšne odločitve mora zagotoviti nedeterministično izbiro med podanimi vejami, odgovori pa ne smejo biti prisotni.

Grafična predstavitev



Tekstovni zapis

<odločitev> ::=
decision <vprašanje> <telo odločitve>
enddecision

<vprašanje> ::= <izraz> | <neformalno besedilo> | **ANY**

<telo odločitve> ::={<del odgovora> <del "else">}
| {<del odgovora> {<del odgovora>}+
[<del "else">]}

<del odgovora> ::= ([<odgovor>]) : [<prehod>]
<odgovor> ::= <intervalni pogoj> | <neformalno besedilo>
<del "else"> ::= **ELSE** : [<prehod>]

<intervalni pogoj> ::= {<zaprti interval> | <odprti interval>} {,{<zaprti interval> | <odprti interval>}}*
<odprti interval> ::= <konstanta> | { = | /= | < | > | <= | >=} <konstanta>
<zaprti interval> ::= <konstanta> : <konstanta>

Slika 3.58: EBNF zapis sintakse specifikacije konstrukta odločitev

Presek intervalnih pogojev, ki so podani v odgovorih, mora biti prazna množica. Tip vrednosti v odgovorih in vprašanju mora biti enak. Odločitev prenese tok izvajanja na prehod, ki sledi pravilnemu odgovoru. Pravilni odgovor v svoji definiciji vsebuje interval vrednosti, ki vključuje vrednost, pridobljeno z interpretacijo vprašanja.

Modeliranje konstrukta odločitev je razdeljeno v tri korake, ki jih opisujejo algoritmi 3.38, 3.39 in 3.40. Prvi dodeli konstruktu z uporabo funkcije $edinstveno()$ edinstveno ime v okviru trenutnega procesa. To se uporabi ob analizi vseh odgovorov v algoritmu 3.39 in kasnejši tvorbi dodatnega odgovora v algoritmu 3.40. V nadaljevanju se preveri, ali se je v vprašanju uporabilo neformalno besedilo, ki pri tvorbi modela ni podprto. V tem primeru se s konstruktom stop tvorba modela neregularno zaključi. Če se v vprašanju uporabi rezervirana

beseda **ANY**, je potrebna posebna obravnava odgovorov, zato se postavi vhodno/izhodni formalni parameter k na TRUE. Konstrukt odločitev se v jeziku Promela modelira z uporabo stavka if.

```
FPAR a ∈ {(spec.stru.proc), (spec.stru.proced)};
FPAR b ∈ {(a.stanje), START};
FPAR IN/OUT c ∈ (b.odloč);
FPAR IN/OUT k ∈ BOOLEAN;
START ;
  TASK i := <vprašanje>;
  TASK c.ime := edinstveno(b, i);
  DECISION i;
   ( <neformalno besedilo> ):
     STOP;
   ( ANY ):
     TASK k := TRUE;
   ELSE:
     CALL 3.2(j,i); (str. 30)
     TASK c.vpraš := j;
   TASK k := FALSE;
ENDDECISION;
TASK "if";
RETURN;
```

Algoritem 3.38: Algoritem pretvorbe konstrukta odločitev

Algoritem 3.39 se izvede za vse odgovore v okviru konstrukta odločitev. Kliče se v okviru algoritma 3.18 (str. 75). V prvem odločitvenem stavku se vrednost odgovora oz. rezervirana beseda **ELSE** priredi začasni spremenljivki i. Formalni parameter k določa, ali se med prehodi vrši naključna izbira. Če parameter ni postavljen, se s klicem algoritma 3.2 pretvorjen odgovor shrani v začasno spremenljivko m. Sledi primerno modeliranje stražarja prehoda, vsak odgovor pa se shrani v množico $spec.stru.proc.stanje.odloč.odgov$ (definicija 3.27, str. 59). Če odgovor ne vsebuje relacijskega operatorja, se pri tvorbi stražarja doda binarni operator $==$. Tako je stražar veljaven takrat, ko je odgovor enak vprašanju, drugače pa se uporabi relacijski operator iz odgovora.

```
FPAR a ∈ {(spec.stru.proc), (spec.stru.proced)};
FPAR b ∈ (a.stanje);
FPAR c ∈ (b.odloč);
FPAR k ∈ BOOLEAN;
TASK i ∈ {<odgovor>, ELSE};
TASK m ∈ <izraz promela>;
START ;
  DECISION i = <odgovor>;
   ( TRUE ):
     TASK i := <odgovor>;
   ELSE:
     TASK i := ELSE;
  ENDDECISION;
  DECISION k = TRUE;
   ( TRUE ):
   ELSE:
     CALL 3.2(m,i); (str. 30)
  ENDDECISION;
  DECISION m = else;
   ( TRUE ):
    TASK ":: else ->";
   ELSE:
    DECISION k = TRUE;
     ( TRUE ):
       TASK ":: true ->";
     ELSE:
       TASK dodaj(c.odgov, m);
       DECISION > ∨ => ∨ < ∨
                <= ∨ ==∈ m
         ( TRUE ):
           TASK ":: (//c.vpraš// //m//) ->";
         ELSE:
           TASK ":: (//c.vpraš// == //m//) ->";
       ENDDECISION;
    ENDDECISION;
  ENDDECISION;
RETURN;
```

Algoritem 3.39: Algoritem pretvorbe odgovora v konstruktu odločitev

Definicija konstrukta v [57] zahteva, da so odgovori določeni tako, da eden in samo eden vedno vsebuje pravilni odgovor. Zaradi uporabe izrazov v vprašanju in odgovorih je

nemogoče izvesti statično analizo, ki bi preverila skladnost zapisa z omenjeno zahtevo. Zato se pri modeliranju konstrukta odločitev vedno uporabi dodaten odgovor, ki preverja ali sta v danem trenutku hkrati veljavna dva odgovora iz specifikacije. V algoritmu 3.40 se z uporabo funkcije $pogoj()$ za vsak odgovor tvorijo vsi mogoči pari odgovorov. Začasna spremenljivka pog hrani najmanjšo množico parov odgovorov. V zanki z oznako R2 se tvori stražar, ki je veljaven vedno, kadar sta hkrati veljavna vsaj dva odgovora. V takšnem primeru se z uporabo globalne spremenljivke `pv__runtime_error` in nadzornega procesa sproži prekinitev formalne verifikacije (razdelek 4.2.2). Če ni pravilen noben odgovor, pride na izvajalni poti do zagatnega stanja, ki ob formalni verifikaciji sproži prekinitev izvajanja. Zato dodatno modeliranje za takšen primer ni potrebno.

START ;
  FPAR $a \in \{(spec.stru.proc), (spec.stru.proced)\}$;
  FPAR $b \in (a.stanje)$;
  FPAR $c \in (b.odloč)$;
R1:
  DECISION $\exists i : i \in (c.odgov)$;
    (TRUE):
      TASK $j := pogoj(i)$;
      TASK $dodaj(pog, j)$;
      JOIN **R1**;
    ELSE:
  ENDDECISION;
  TASK “:: false”;
R2:
  DECISION $\exists k : k \in pog$;
    (TRUE):
      TASK “|| //k//”;
      JOIN **R2**;
    ELSE:
      TASK “−> pv__runtime_error = true”;
  ENDDECISION;
  TASK “fi;”;
  RETURN;

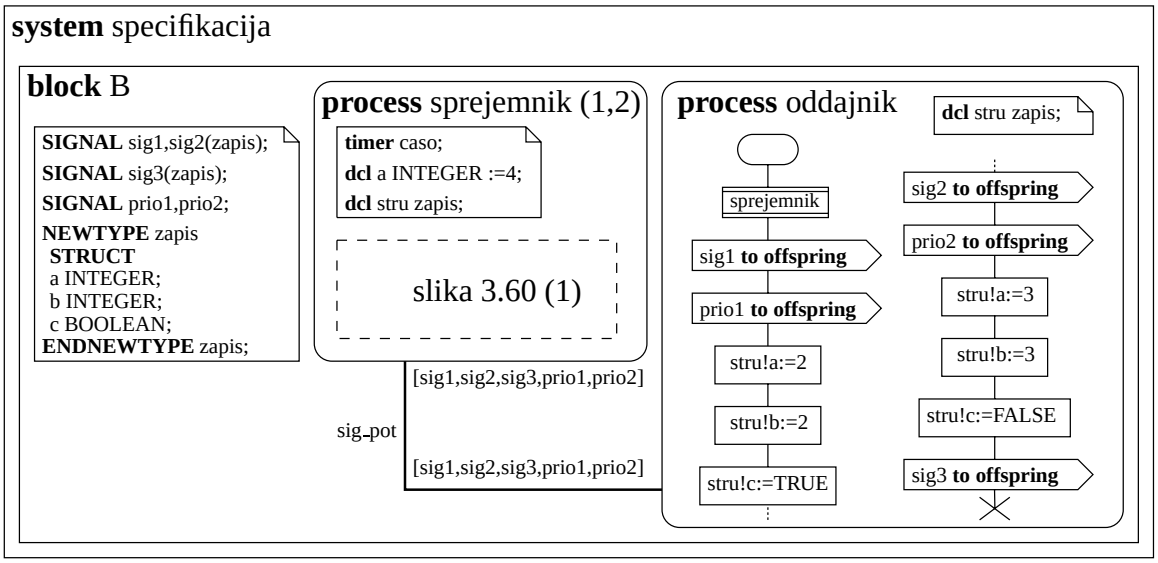
Algoritem 3.40: Algoritem pretvorbe zaključka konstrukta odločitev

3.5 Primer modeliranja komunikacijskih konstruktov

Sistem na sliki 3.59 ima en blok in dva procesa. Ob zagonu sistema se tvori en primerek vsakega procesa. V prvem koraku izvajanja procesa “oddajnik” se ustvari nov primerek procesa “sprejemnik”, ki s tem doseže največje predvideno število primerkov med izvajanjem sistema. Vsak dodatni poskus tvorbe primerka procesa bi predstavljal napako. V okviru makroja `run__proc()` se ob tvorbi procesa preveri skladnost števila aktivnih primerkov s specifikacijo. Če je največje predvideno število primerkov preseženo, se postavi spremenljivka `pv__run_time_error` na boolovo vrednost 1. To med formalno verifikacijo sproži napako v procesnem tipu “monitor”.

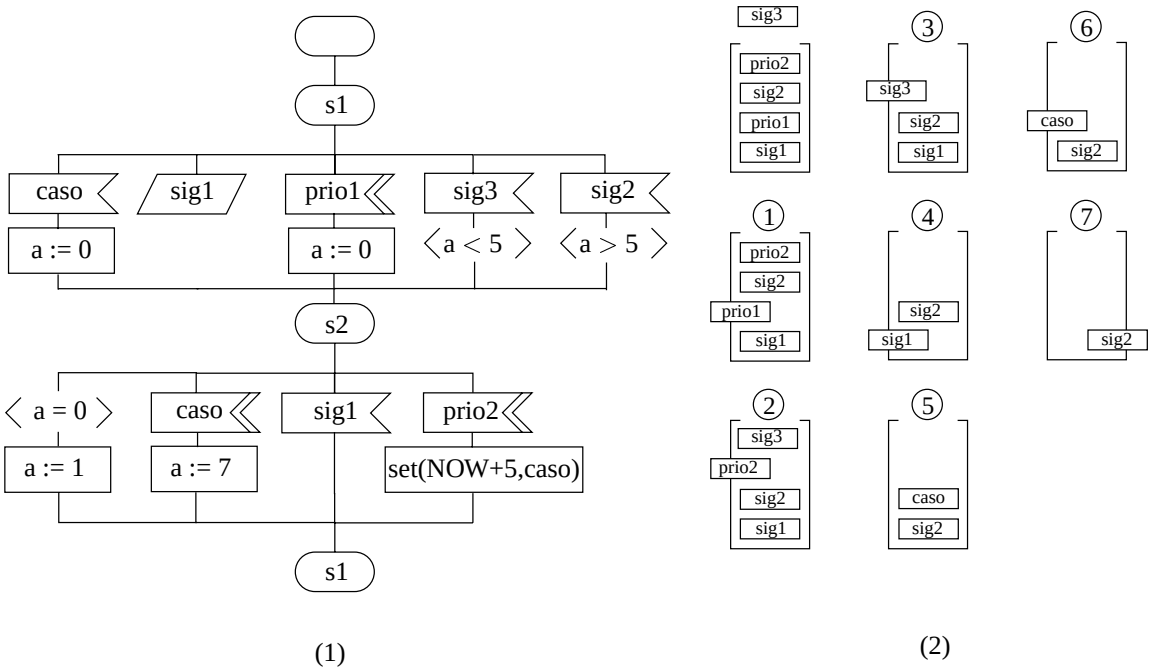
V preostalih korakih se pošlje tvorjenemu procesu zaporedje signalov. Prehod se obravnava kot atomarno zaporedje, zato so vse reže pridruženega kanala procesa “sprejemnik” že pred obravnavo prvega signala zasedene. Ker zaporedje vsebuje večje število signalov, kot jih kanal lahko sprejme, proces “oddajnik” prekine z izvajanjem, saj se blokira. Zadnji signal se tako pošlje v poljubnem kasnejšem obdobju.

Slika 3.60 prikazuje nekoliko poenostavljen opis SDL procesa “sprejemnik” z dvema stanjema in pridruženo vhodno vrsto. Natančno specifikacijo podaja tekstovni opis (priloga



Slika 3.59: Primer sistema z dvema procesoma

B.1, str. 177). V grafu procesa je uporabljena večina podprtih vhodnih konstruktov. Zaradi lažje predstavitve delovanja v primeru ni uporabljen spontani prehod.



Slika 3.60: Primer modeliranja vhodnih konstruktov

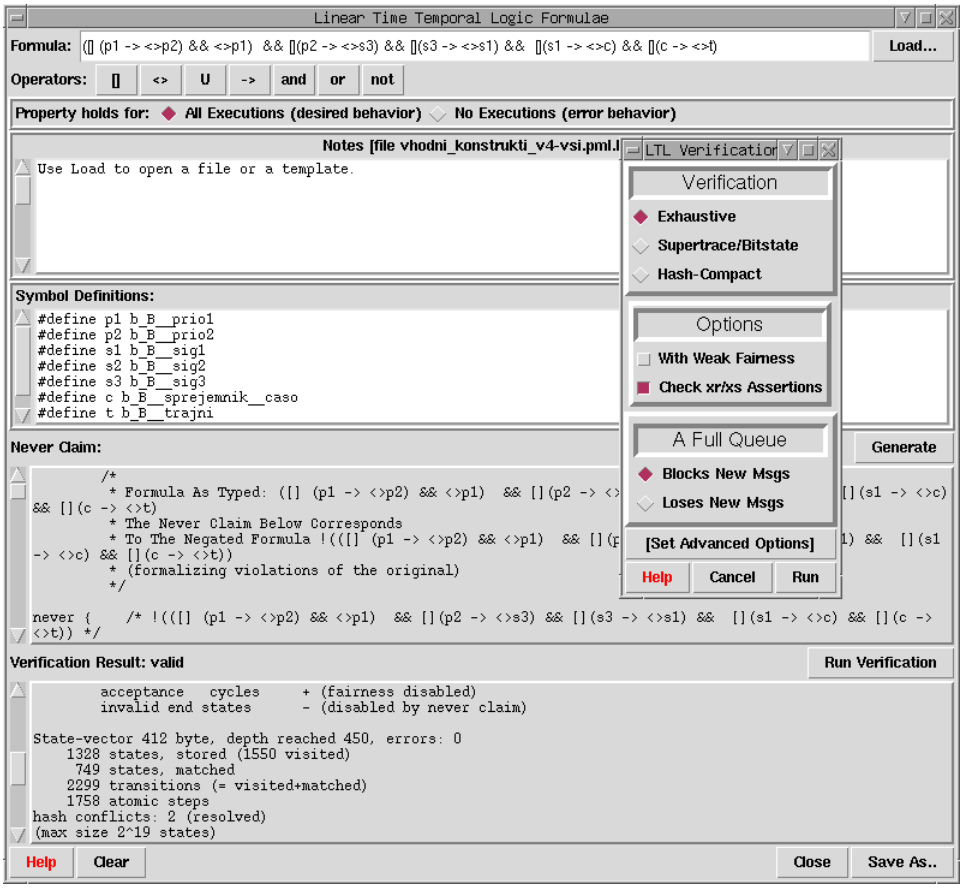
Model sistema sestavljajo procesni tipi “init”, “oddajnik”, “sprejemnik”, “monitor” in “__Timers”. Zaradi večje preglednosti izpisa smo del modela sistema (priloga B.2) predstavili z dvema vhodnima datotekama. Prva vsebuje makro za tvorbo novega procesa, v drugi pa so definirani pomožni komunikacijski elementi.

Pravilnost modeliranja se lahko preveri z miselnim izvajanjem, ki narekuje sprejem signalov v vrstnem redu, kot je prikazan na sliki 3.60. Zaradi velikosti modela je primerneje,

da pričakovane lastnosti podamo v obliki trditev in formul LTL. Edina trditev, ki je prisotna v modelu sistema, je vključena v proces "monitor". Zaradi asinhronega in sočasnega izvajanja procesov se lahko ob formalni verifikaciji proces izvede v kateremkoli koraku izvajanja. Če je omogočen stražar v telesu procesa, se sproži napaka pri simulaciji in formalni verifikaciji. Stražar preverja vrednost spremenljivke `pv_run_time_error`.

Za formalno verifikacijo so v modelu sistema nameščene sonde, ki nosijo ime signala s predpono `b_`. Ob vsakem sprejemu signala se sonda postavi na 1 in v naslednjem koraku na 0. Tako lahko s formulo $\Box(p \rightarrow \Diamond q) \wedge \Diamond p$ preverimo, ali sprejemu signala p vedno sledi tudi sprejem signala q. Razširitev formule na vse prisotne signale, $(\Box(p1 \rightarrow \Diamond p2) \wedge \Diamond p1) \wedge \Box(p2 \rightarrow \Diamond s3) \wedge \Box(s3 \rightarrow \Diamond s1) \wedge \Box(s1 \rightarrow \Diamond c) \wedge \Box(c \rightarrow \Diamond t)$, potrdi skladost modela s specifikacijo sistema in semantiko jezika SDL in dokazuje pravilnost algoritmov za pretvorbo vhodnih konstruktov. Preslikava med simboli v formuli in dejanskimi sondami v modelu sistema je prikazana v oknu "Symbol Definitions" na sliki 3.61. Orodje Spin iz formule LTL avtomatsko tvori Büchijev avtomat v obliki trditve "never claim" (okno "Never Claim"). Pripravi in pretvorbi specifikacije zahtev sledi formalna verifikacija. Pogovorno okno na sliki 3.61 prikazuje izbiro izčrpne verifikacije z blokiranjem procesov v primeru polnih kanalov. V zadnjem oknu se izpišejo rezultati formalne verifikacije. Na sliki 3.61 je prikazana količina uporabljenega pomnilnika, v nadaljevanju pa podajamo celoten izpis rezultatov:

```
warning: for p.o. reduction to be valid the never claim must be stutter-invariant
(never claims generated from LTL formulae are stutter-invariant)
depth 62: Claim reached state 27 (line 373)
depth 291: Claim reached state 31 (line 377)
depth 334: Claim reached state 35 (line 381)
depth 250: Claim reached state 39 (line 385)
depth 109: Claim reached state 43 (line 389)
depth 334: Claim reached state 77 (line 417)
depth 445: Claim reached state 47 (line 393)
(Spin Version 4.1.1 -- 2 January 2004
+ Discrete-Time Extension Version 0.1.2 (Multiple Tick) -- 24 December 2002)
        + Partial Order Reduction

Full statespace search for:
        never claim                 +
        assertion violations        + (if within scope of claim)
        acceptance   cycles         + (fairness disabled)
        invalid end states        - (disabled by never claim)

State-vector 412 byte, depth reached 450, errors: 0
    1328 states, stored (1550 visited)
     749 states, matched
    2299 transitions (= visited+matched)
    1758 atomic steps
hash conflicts: 2 (resolved)
(max size 2^19 states)

Stats on memory usage (in Megabytes):
0.558         equivalent memory usage for states (stored*(State-vector + overhead))
0.752         actual memory usage for states (unsuccessful compression: 134.80%)
        State-vector as stored = 558 byte + 8 byte overhead
2.097         memory used for hash table (-w19)
0.320         memory used for DFS stack (-m10000)
3.134         total actual memory usage
```



Slika 3.61: Primer formalne verifikacije modela sistema z orodjem `xdtspin`

Tabela 3.2 vsebuje dodatne primere formul, njihov pomen in rezultate verifikacije. Primer neizpolnjene zahteve predstavlja formula $\Diamond s2$, ki dokazuje, da obstaja izvajanje, pri katerem se signal sig2 ne sprejme. Naslednje besedilo prikazuje rezultat formalne verifikacije:

```
warning: for p.o. reduction to be valid the never claim must be stutter-invariant
(never claims generated from LTL formulae are stutter-invariant)
pan: acceptance cycle (at depth 444)
pan: wrote pan_in.trail
(Spin Version 4.1.1 -- 2 January 2004
+ Discrete-Time Extension Version 0.1.2 (Multiple Tick) -- 24 December 2002)
Warning: Search not completed
        + Partial Order Reduction

Full statespace search for:
        never claim                 +
        assertion violations        + (if within scope of claim)
        acceptance   cycles         + (fairness disabled)
        invalid end states        - (disabled by never claim)

State-vector 412 byte, depth reached 444, errors: 1
     142 states, stored (143 visited)
       1 states, matched
     144 transitions (= visited+matched)
     162 atomic steps
hash conflicts: 0 (resolved)
(max size 2^19 states)

Stats on memory usage (in Megabytes):
0.060         equivalent memory usage for states (stored*(State-vector + overhead))
0.246         actual memory usage for states (unsuccessful compression: 413.21%)
```

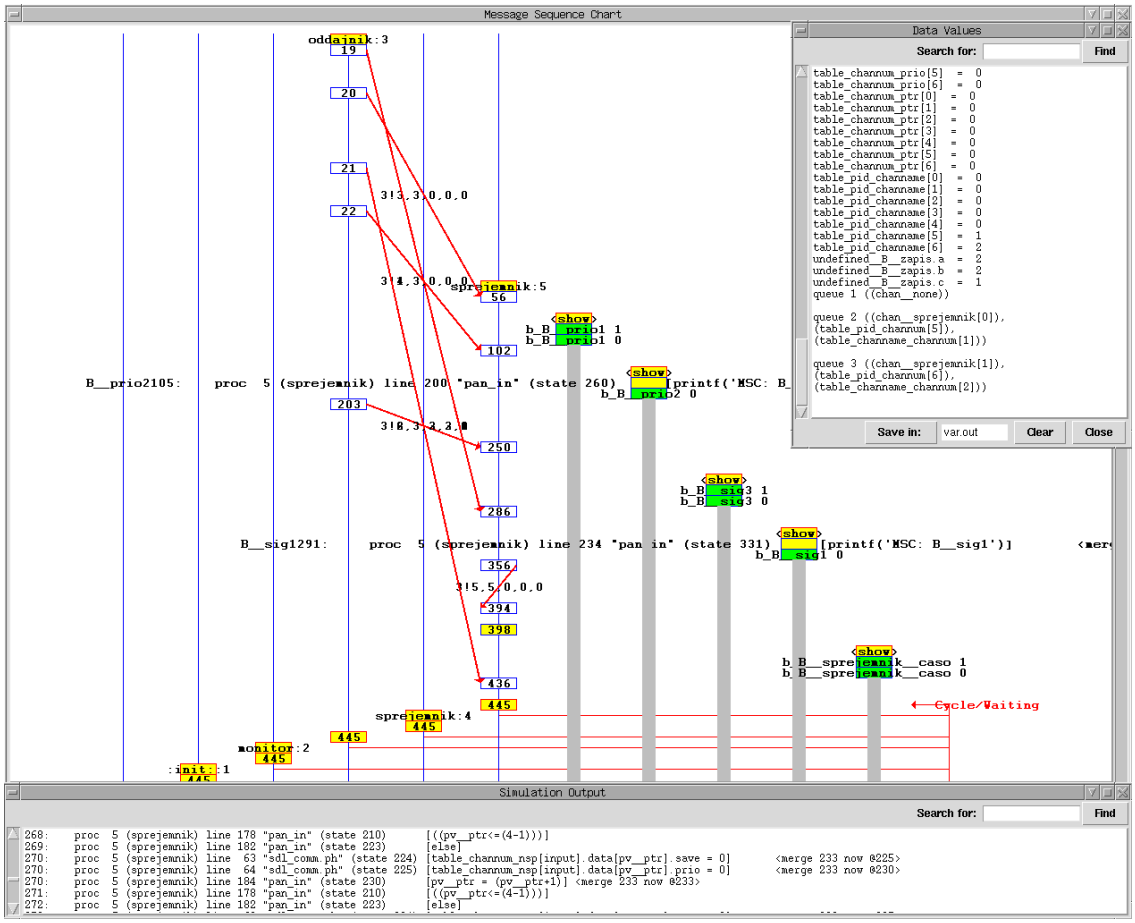
```
        State-vector as stored = 1727 byte + 8 byte overhead
2.097         memory used for hash table (-w19)
0.320         memory used for DFS stack (-m10000)
2.622         total actual memory usage
```

Tabela 3.2: Rezultati formalne verifikacije modela

formula	**opis formule**	**rezultat**
$\Diamond s1$	v vseh izvajanjih se sprejme signal b_B__sig1	drži
$!\Diamond s2$	signal b_B__sig2 se ne sprejme v nobenem izvajanju	drži
$\Diamond s3$	v vseh izvajanjih se sprejme signal b_B__sig3	drži
$\Diamond p1$	v vseh izvajanjih se sprejme signal b_B__prio1	drži
$\Diamond p2$	v vseh izvajanjih se sprejme signal b_B__prio2	drži
$\Diamond c$	v vseh izvajanjih se izteče časovnik	drži
$\Diamond t$	v vseh izvajanjih se izvede trajni prehod	drži
$\Box(p1 \rightarrow \Diamond s1) \wedge \Diamond p1$	ali sprejemu prio1 vedno nekoč sledi sprejem sig1	drži
$\Box(p2 \rightarrow \Diamond s1) \wedge \Diamond p2$	ali sprejemu prio2 vedno nekoč sledi sprejem sig1	drži
$!(\Box(s1 \rightarrow \Diamond p1) \wedge \Diamond s1)$	sig1 se nikoli ne sme sprejeti pred prio1	drži
$!(\Box(p2 \rightarrow \Diamond p1) \wedge \Diamond p2)$	prio2 se nikoli ne sme sprejeti pred prio1	drži

Orodje Spin v takšnih primerih omogoča prikaz neveljavnega izvajanja s pomočjo vodene simulacije. Na sliki 3.62 je podana grafična predstavitev izvajanja z uporabo časovnih diagramov, prikazom rež kanalov, lokalnih in globalnih spremenljivk (okno “Data Values”) ter natančnega opisa poteka izvajanja posameznega procesa v oknu “Simulation Output”. Analiza izvajanja pokaže, da je v tem izvajanju signal sig2 implicitno sprejet. Orodje Spin omogoča nadaljevanje preverjanja pravilnosti tudi po odkriti napaki. Tako se lahko potencialno pridobijo poteki izvajanj za vse kršitve podane zahteve v modelu sistema. V obravnavanem primeru je prikazano edino možno izvajanje. Vsaka sprememba vrednosti spremenljivk, ki so v modelu sistema označene s posebno oznako `show`, se lahko ob vodeni simulaciji prikazuje v lastnem stolpcu. V primeru na sliki 3.62 tako prikazujemo vrednosti vseh sond za sledenje sprejema signalov.

*



Slika 3.62: Primer vodene simulacije sistema z orodjem `xdtspin`

4. poglavje

Formalna verifikacija

Cilj formalne verifikacije je ugotoviti, ali se lahko v kateremkoli primeru krši specifikacija zahtev. Ne ukvarja se torej s pogostostjo kršitev ali njihovo verjetnostjo, temveč z matematično natančnostjo potrdi ali ovrže skladnost s specifikacijo. Specifikacija zahtev se lahko poda na različne načine. V večini primerov je ni mogoče podati v celoti, zato se verifikacija modela razdeli na manjše dele, ki preverjajo posamezno lastnost sistema. Preverjanje logične pravilnosti sistema se torej ukvarja z možnostjo napake v vseh mogočih izvajanjih sistema in ne z njeno verjetnostjo [36].

Orodje Spin uporablja tehniko preverjanja modelov, ki temelji na teoriji ω-avtomata. Ta se od klasične teorije o končnih avtomatih razlikuje v tem, da lahko obravnava tudi neskončna izvajanja. Izkaže se, da ima ω-avtomat primerno izrazno moč tako za modeliranje obnašanja procesov v distribuiranih sistemih, kot za zapis širokega spektra njihovih lastnosti, ki jih želimo preveriti. Formalni zapis končnega avtomata je predstavljen v definiciji 4.1.

Definicija 4.1 Končni avtomat je peterica (S, s_0, L, T, F), kjer je:
S množica stanj;
s_0 začetno stanje, $s_0 \in$ S;
L končna množica oznak;
T množica prehodov, T $\subseteq$ (S $\times$ L $\times$ S);
F množica končnih stanj, F $\subseteq$ S.♦

V najpreprostejšem primeru je avtomat determinističen — naslednje stanje je edinstveno določeno s trenutnim stanjem in oznako prehoda. Pri mnogih sistemih se zaporedje stanj ne more pridobiti iz zaporedja prehodov. Takšne sisteme opisujemo z nedeterminističnimi ω-avtomati.

Definicija 4.2 Končni avtomat (S, s_0, L, T, F) je determinističen, če in samo če velja $\forall s \forall l, ((s,l,s') \in T \wedge (s,l,s'') \in T) \rightarrow s' \equiv s''$.♦

Kadar preverjamo določeno lastnost sistema, preverjamo, ali je njegovo obnašanje skladno s podano zahtevo [28]. Obnašanje sistema je torej sinonim za vse možne izvajalne poti. Izvajalno pot sistema opisuje urejena množica prehodov (definicija 4.3).

Definicija 4.3 Izvajalna pot σ končnega avtomata (S, s_0, L, T, F) je urejena množica prehodov $\{(s_0,l_0,s_1), (s_1,l_1,s_2), (s_2,l_2,s_3), \ldots\}$, kjer velja $\forall i, (i \geq 0) \rightarrow (s_i,l_i,s_{i+1}) \in$ T.♦

Stanje v modelu sistema edinstveno določajo vrednosti globalnih in lokalnih spremenljivk, trenutna stanja aktivnih procesov in vsebina vseh komunikacijskih kanalov.

Definicija 4.4 Dosegljiva stanja v modelu sistema določajo mogoče izvajalne poti. Izvajalna pot σ je mogoča v modelu sistema M, če zadosti naslednjima kriterijema:

1. prvo stanje zaporedja je začetno stanje M — vse spremenljivke hranijo vrednost nič, vsi komunikacijski kanali so prazni, edini aktivni proces $init()$ se nahaja v začetnem stanju;
2. če je M v stanju z zaporednim številom i, potem obstaja vsaj en prehod, ki vodi v stanje z zaporednim številom $i + 1$.♦

Poznamo dve posebni obliki zaporedij: končna in ciklična. Ciklična zaporedja lahko opisujejo neskončne izvajalne poti, ki jih imenujemo ω-izvajalne poti. Vsa končna in ciklična zaporedja, ki se lahko pridobijo z izvajanjem modela M, določajo njegovo obnašanje. Množica dosegljivih stanj modela je unija vseh stanj, ki določajo obnašanje modela.

Definicija 4.5 Izvajalna pot je končna, če se nobeno stanje sistema v zaporedju ne pojavi več kot enkrat in model sistema M v končnem stanju zaporedja nima izvršljivih stavkov. Formalno: mogoča končna izvajalna pot končnega avtomata (S, s_0, L, T, F) je izvajalna pot σ, pri kateri ima zadnji prehod (s_{n-1}, l_{n-1}, s_n) lastnost, da je $s_n \in F$.♦

Definicija 4.6 Izvajalna pot je ciklična, če so vsa stanja v zaporedju, razen zadnjega, edinstvena, zadnje stanje pa je enako enemu izmed stanj, ki se je v zaporedju že pojavilo. Formalno: mogoča ω-izvajalna pot končnega avtomata (S, s_0, L, T, F) je izvajalna pot σ, pri kateri $\exists s_f, s_f \in F \wedge s_f \in \sigma^\omega$ ♦

Končni avtomat, v katerem so veljavne izvajalne poti, ki so skladne z definicijo 4.6, imenujemo Büchijev avtomat[1]. Veljavne izvajalne poti se lahko predstavijo tudi z regularnima izrazoma U_i in V_i nad prehodi končnega avtomata.

$$\bigcup_{i=1}^{N} U_i \cdot V_i^\omega$$

U opisuje končno zaporedje, ki se izvede v prvem delu izvajalne poti. V opisuje končno zaporedje, ki se v nadaljevanju izvajanja ponavlja v neskončnost. Takšne izraze imenujemo ω-regularni izrazi. N je število ω-regularnih izrazov, ki določajo vse mogoče izvajalne poti. Lastnosti, ki jih lahko s takšnimi izrazi opišemo, imenujemo ω-regularne lastnosti.

Na osnovi Büchijeve definicije ω-izvajalne poti končnega avtomata (definicija 4.6) se lahko za posamezen avtomat preveri, ali je množica mogočih izvajalnih poti prazna. Preveri se lahko tudi presek ω-izvajalnih poti množice Büchijevih avtomatov — tvori se skupni Büchijev avtomat, v katerem so mogoče samo tiste izvajalne poti, ki so mogoče v vseh elementih množice. Algoritmi za formalno verifikacijo v orodju Spin preverjajo, ali je množica mogočih izvajalnih poti skupnega Büchijevega avtomata prazna [36]. Navadno je prvi element množice model sistema, drugi pa specifikacija zahtev.

Definicija 4.7 Σ naj bo množica vseh mogočih izvajalnih poti ($\sigma \in \Sigma$) sistema. Pravimo, da sistem zadosti pravilnostni zahtevi ϕ takrat in samo takrat, ko je ϕ veljavna v vseh mogočih izvajanjih:

$$(\Sigma \models \phi) \iff \forall \sigma, (\sigma \in \Sigma \rightarrow \sigma \models \phi) \tag{4.1}$$

[1]Osnove teorije ω-avtomata je postavil Büchi v letu 1960.

Pravilnost rezultatov formalne verifikacije je v prvi vrsti odvisna od pravilne priprave modela in formalnega zapisa zahtev. V tem poglavju se posvečamo načinom zapisa specifikacije zahtev in spremljajočim dopolnitvam modela, ki omogočajo formalno verifikacijo zahtev z orodjem Spin. Pravilnostne trditve v Promeli se nanašajo na lastnosti dosegljivih stanj modela M. Vključujejo lahko vse elemente, ki določajo trenutno stanje sistema. V posameznem stanju je lahko trditev pravilna ali nepravilna. Specifikacija zahtev podaja stanja, v katerih mora dana trditev držati.

4.1 Pravilnostne trditve

Pravilnostne trditve delimo v dve skupini. V prvi so tiste, ki opisujejo varnost — nikoli se ne bo pripetilo nič slabega. V to skupino torej sodijo trditve, ki jih ne sme biti veljavna na nobeni mogoči izvajalni poti. Trditve v drugi skupini opisujejo živost — nekoč bo želena lastnost držala. V to množico sodijo trditve, ki morajo biti veljavne v vseh mogočih izvajanjih. S stališča algoritmov za formalno verifikacijo tako obstajata dve vrsti trditev:

1. trditve o dosegljivih in nedosegljivih stanjih sistema (lastnosti stanj) in
2. trditve o veljavnih in neveljavnih izvajanjih (lastnosti poti).

Primer iz prve skupine je trditev, ki mora veljati v vseh dosegljivih stanjih sistema (invarianta). Šibkejša različica sistemske invariante je trditev, ki mora veljati samo v določenih dosegljivih stanjih (invarianta procesa). Lastnosti poti se opisujejo z množico lastnosti stanj. Primer takšne lastnosti je zahteva, da se mora izvajanje po vsakem obisku stanja z oznako P nekoč pojaviti v stanju Q, z dodatnim pogojem, ki predpisuje, da se na poti med stanjema ne sme pojaviti stanje R. Promela omogoča opis obeh skupin trditev na pet različnih načinov[2]:

- osnovne pravilnostne trditve (“basic assertions”),
- oznake veljavnih končnih stanj sistema (“end states”),
- oznake stanj, t. i. stanj napredka, sistema, ki morajo biti prisotna v vseh neskončnih izvajalnih poteh modela (“progress states”),
- oznake dopustnih stanj, ki se v nobeni izvajalni poti sistema ne smejo pojaviti neskončnokrat (“accept states”),
- trditve, ki ne smejo veljati na nobeni izvajalni poti (“never claim”),

4.1.1 Osnovne pravilnostne trditve

Osnovne pravilnostne trditve se zapišejo v okviru stavka `assert(`$izraz$`)`. Področje veljavnosti trditve določa mesto v modelu sistema. Uporabnost takšnih trditev sta v svojem prispevku iz leta 1947 opisala že Goldstein in von Neumann [36]. Osnovne trditve so vedno izvršljive in so v primeru, ko so veljavne, brez stranskih učinkov. Veljavna trditev je tista trditev, katere izraz se ovrednoti na boolovo vrednost 1 oz. celoštevilčno vrednost, ki je različna od nič. Neveljavna osnovna trditev predstavlja napako v izvajanju in sproži prekinitev simulacije in formalne verifikacije. Vse druge oblike pravilnostnih trditev sprožijo prekinitev samo ob formalni verifikaciji. Tudi veljavnost osnovne pravilnostne trditve v vseh mogočih izvajanjih lahko dokaže samo formalna verifikacija.

[2] Obstajajo še trditve sledi, ki jih zaradi predpisane omejitve, da so elementi kanala lahko samo podatkovnega tipa $mtype$, ne uporabljamo.

4.1.2 Končna stanja

Orodje Spin lahko preveri, ali katera mogoča izvajalna pot vsebuje kakšno zagatno stanje.

Definicija 4.8 Zagatno stanje je stanje, ki nima izvršljivih stavkov in ni v množici veljavnih končnih stanj.♦

Po privzetem je sistem v veljavnem končnem stanju ob zaključku izvajanja vseh procesov. Dodatna veljavna končna stanja se lahko dodajo v model sistema z uporabo posebnih oznak s predpono `end`. To omogoča preverjanja zagatnih stanj modelov, ki vsebujejo procese z neskončnimi izvajalnimi potmi (npr. proces čaka na stimulus iz okolja). Število oznak končnih stanj ni omejeno. Edino omejitev predstavlja edinstvenost oznak v okviru procesa. V množico končnih oznak sodijo vse oznake s predpono `end` (`end:`, `endhere:`, `end_state:`,...).

Med avtomatsko tvorbo modela iz specifikacije sistema v jeziku SDL se vsakemu procesu pred zaviti oklepaj, ki označuje zaključek njegovega opisa, doda osnovna končna oznaka (`END:`). V primeru zaključka izvajanja procesa s konstruktom stop se izvede brezpogojni skok na to oznako. Po privzetem se torej sistem nahaja v veljavnem končnem stanju samo ob zaključku izvajanja vseh procesov, kar je skladno z osnovno definicijo Promele. Dodatna veljavna končna stanja je potrebno eksplicitno definirati s primernimi nastavitvami programa $sdl2pml$.

Orodje Spin omogoča tudi bolj strogo definicijo veljavnega končnega stanja sistema, ki vključuje zahtevo, da so ob zaključku izvajanja prazni vsi komunikacijski kanali. To velja samo za primere, ko se niso zaključili vsi procesi. V primerih, ko je izvajanje vseh procesov zaključeno, komunikacijski kanali pa niso prazni, Spin ne javi neveljavnega končnega stanja.

4.1.3 Stanja napredka

Stanja napredka označujejo stavke, ki opisujejo želene dogodke v modelu sistema. Prehod takšnega stanja označuje pričakovano aktivnost procesa in omogoča opis živosti. Ob formalni verifikaciji orodje Spin samostojno tvori proces "never claim", ki v primeru obstoja neskončne izvajalne poti, ki ne vsebuje stanja napredka, sproži napako. Število oznak s predpono `progress` ni omejeno, pri poimenovanju pa veljajo enaka pravila kot pri oznakah končnih stanj.

Definicija 4.9 Neskončna izvajalna pot ne vključuje stanja napredka, če velja $\forall s_f$, $s_f \in \sigma^\omega \rightarrow s_f \notin P$, kjer je $s_f \in F$, P množica stanj napredka in velja $P \subseteq S$.♦

Definicija 4.9 opisuje samo ω-izvajalne poti. Vsaka končna izvajalna pot se lahko predstavi kot ω-izvajalna pot z vpeljavo jecljanja (definicija 4.10). Takšna ω-izvajalna pot zadosti pogoju v definiciji 4.6, če in samo če je končno stanje v množici F. Z vpeljavo jecljanja postanejo izvajalne poti, ki jih opisuje definicija 4.5, samo posebna oblika ω-izvajalnih poti (definicija 4.6).

Definicija 4.10 Končna izvajalna pot σ s končnim stanjem s_n je po vpeljavi jecljanja ω-izvajalna pot σ, $(s_n, \varepsilon, s_n)^\omega$. ε je v Promeli predstavljen s stavkom $skip$.♦

Orodje Spin ne predpisuje relativnih hitrosti izvajanja posameznih procesov. Zato sodi med veljavna izvajanja tudi primer, ko se izvršljiv stavek v procesu nikoli ne izvede. To je mogoče samo v primerih, ko je v modelu sistema vedno na voljo kakšen drug izvršljiv stavek.

Kadar želimo takšna izvajanja izključiti, je to mogoče z uporabo poštenostnih omejitev. V to skupino omejitev sodi tudi t. i. končno izvajanje — vsak proces, ki lahko izvede prehod v naslednje stanje, bo ta prehod nekoč tudi izvedel. Poznamo definicijo stroge in šibke poštenosti.

Končno izvajanje opisuje šibko poštenost. Stroga poštenost predpisuje, da bo proces lahko izvedel vsak stavek, ki postane neskončnokrat izvršljiv. Orodje Spin ima vključeno podporo šibki poštenosti, ki poveča zahtevnost formalne verifikacije za faktor N, kjer je N število aktivnih procesov. Ostale poštenostne omejitve se lahko zapišejo z uporabo linearne temporalne formule ali z lastno tvorbo procesa "never claim". Podrobnejša razlaga algoritmov in njihove prostorske ter časovne zahtevnosti je opisana v [36].

4.1.4 Dopustna stanja

Oznake dopustnih stanj predstavljajo še zadnjo obliko posebnih oznak stanj. Navadno se oznake s predpono `accept` uporabljajo samo v okviru procesov "never claim", ki se avtomatično tvorijo iz specifikacije zahtev v linearni temporalni formuli. Uporabijo pa se lahko tudi v modelu sistema, ki ne vključuje procesa "never claim". Med verifikacijo algoritmi iščejo tista ciklična izvajanja, ki vključujejo vsaj eno dopustno stanje. Ob formalni verifikacije se pričakuje, da nobena veljavna izvajalna pot ne vsebuje neskončnega števila dopustnih stanj. Orodje Spin podpira omejitve, ki jih narekuje šibka poštenost, tudi pri iskanju nedopustnih ciklov.

4.1.5 Nikoli veljavne trditve

Vse dosedanje oblike zapisa pravilnostnih trditev posegajo v model sistema. Kljub številnim pravilnostnim zahtevam, ki se z njihovo uporabo lahko izrazijo, ne moremo preveriti zahteve: "v vseh veljavnih izvajanjih sistema vsakemu stanju sistema, v katerem velja trditev p, nekoč sledi stanje, v katerem velja trditev q". Za preverjanje trditev takšnega tipa je potrebno zagotoviti mehanizem vpogleda v stanje sistema po vsakem koraku. Korak predstavlja prehod iz trenutnega stanja sistema v naslednje stanje sistema. Tega ni mogoče zagotoviti z lastnim procesom, saj v množico veljavnih izvajanj sodi tudi takšno, kjer proces poljubno dolgo ne izvede omogočenega prehoda[3]. Za te namene se v Promeli uporabi proces "never claim", ki se po definiciji izvaja izmenično s preostalim sistemom — en korak proces, en korak sistem. Prvi korak naredi proces "never claim". Skladno z imenom, ta proces opisuje končno ali neskončno izvajalno pot, ki ne sme biti v množici veljavnih izvajalnih poti modela sistema. Zaradi podrobnega pogleda v izvajanje sistema omogoča proces "never claim" zapis kompleksnih pravilnostnih trditev. Tvorijo se lahko ročno ali avtomatsko. Ob avtomatski tvorbi se kot vhod lahko uporabi zapis lastnosti v linearni temporalni logiki, izhod pa predstavlja Büchijev avtomat pravilnostne zahteve. Algoritmi formalne verifikacije preverijo, ali je presek veljavnih izvajalnih poti obeh avtomatov prazna množica. Kadar presek ni prazna množica, pridobljene mogoče poti predstavljajo protiprimere za podano pravilnostno zahtevo.

Slika 4.1 prikazuje tri različne zapise enostavne trditve, ki preverja, ali v kateremkoli koraku sistema postane veljavna trditev p. V tem trenutku proces zaključi z izvajanjem in sproži prekinitev formalne verifikacije. Prekinitev se sproži tudi v primeru, ko je trditev p veljavna v začetnem stanju sistema. Stavki, ki se uporabijo v procesu "never claim" ne

[3] Ob predvidevanju, da je v vmesnem obdobju vedno izvršljiv še vsaj en stavek v modelu sistema.

smejo vplivati na potek izvajanja sistema — t. i. stavki brez stranskih učinkov. Izbrani primer opisuje enostavno sistemsko invarianto. Njeno veljavnost lahko preverimo tudi brez uporabe procesa "never claim", kot je to prikazano v tretjem zapisu na sliki 4.1. Takšen proces je uporabljen ob avtomatski tvorbi modela. Trditev p je v tem primeru enaka globalni spremenljivki `pv__runtime_error`, ki zavzame boolovo vrednost 1, ko izvajanje prekrši semantična pravila jezika SDL.

```
never {
        do
        :: p -> break
        :: else
        od
}
```

A

```
never {
        do
        :: assert(p)
        od
}
```

B

```
active proctype monitor()
{
  atomic{
          p -> assert(false)
  }
}
```

C

Slika 4.1: Primer enostavnega procesa "never claim"

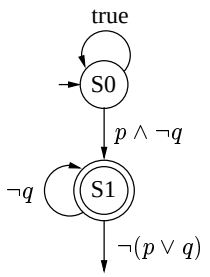
Med primeroma A in B na sliki 4.1 ni bistvenih razlik s stališča formalne verifikacije. V obeh primerih se pravilnost trditve preverja *med vsakim* korakom sistema. Proces `monitor()` v primeru C pa predstavlja del sistema in se *lahko* izvede v kateremkoli koraku, kjer je trditev p resnična. Če je takšno izvajanje *mogoče*, ga formalna verifikacija tudi najde, saj se preverijo vsa mogoča izvajanja. V primeru preproste pravilnostne trditve se torej lahko izognemo uporabi procesa "never claim", ki prispeva k računski kompleksnosti sistema.

To ne drži za kompleksnejše pravilnostne izjave, ki vključujejo neskončna izvajanja. Primer takšne pravilnostne izjave predstavlja LTL formula $\Box(p \rightarrow (p \,\mathrm{U}\, q))$[4]. Ker s Spinom vedno iščemo neveljavna izvajanja, nas zanima, ali je mogoča takšna izvajalna pot, na kateri trditev p najprej postane resnična, nato pa trditev q za vedno ostane neresnična ali pa trditev p postane neresnična, preden postane resnična trditev q. Takšna izvajalna pot ni veljavna in mora sprožiti prekinitev formalne verifikacije. Preverjanje izjave, da je trditev q neveljavna *za vedno*, vključuje neskončno izvajalno pot. Ker obravnavamo končne sisteme, to opisuje ciklično izvajalno pot in sodi v skupino lastnosti, ki opisujejo živost sistema. Takšne lastnosti se ne morejo izraziti z osnovnimi trditvami, temveč narekujejo uporabo procesa "never claim".

Slika 4.2 prikazuje proces "never claim" za formulo $\neg\Box(p \rightarrow (p \,\mathrm{U}\, q))$. Izvajanje nikoli veljavne trditve se prične v stanju z oznako `S0`. Zanka ima dva stražarja. Drugi stražar je vedno veljaven in modelira poljubno število izvajanj — formula ne predpisuje, kdaj postane veljavna trditev p. Prvi stražar postane veljaven v trenutku, ko velja $p \wedge \neg q$. V tem trenutku je postala veljavna trditev p, trditev q pa je neveljavna. Izvajanje se nadaljuje v stanju `S1`, ki sodi v množico dopustnih stanj. Tudi v tem stanju je zanka z dvema stražarjema. V tem stanju lahko proces ostane samo takrat, ko trditev q za vedno ostane neveljavna. Takšno izvajanje sproži prekinitev formalne verifikacije, saj predstavlja kršitev zahteve, da nobena izvajalna pot ne vključuje neskončnega števila dopustnih stanj. Obravnavano pravilnostno trditev bi kršila tudi vsaka izvajalna pot, na kateri bi trditev p postala neresnična preden bi postala resnična trditev q. Takšen primer opisuje drugi stražar. Proces "never claim" v tem primeru zaključi izvajanje in sproži prekinitev formalne verifikacije zaradi veljavnosti nikoli veljavne trditve.

V primeru, ko se proces "never claim" nahaja v stanju `S1` in postaneta veljavni obe trditvi p in q, se proces v naslednjem koraku blokira. S tem je obravnavana izvajalna pot

[4] "U" predstavlja operator "strong until", "weak until" se označuje z "U".




```
never {
S0:
  do
  :: p && !q -> break
  :: (1)
  od;
S1:
accept:
  do
  :: !q
  :: !(p || q) -> break
  od;
}
```

Slika 4.2: Proces “never claim” za trditev $\neg\Box(p \rightarrow (p \text{ U } q))$

zadostila pogojem trditve $\neg\Box(p \rightarrow (p \text{ U } q))$ in preverjanje se lahko zaključi.

Prejšnji odstavek nakazuje dodatno uporabnost procesov “never claim” za omejevanje mogočih izvajanj. Takoj, ko določena izvajalna pot zadosti nikoli veljavni trditvi, se formalna verifikacija nadaljuje z analizo naslednje izvajalne poti. S tem mehanizmom se lahko postopek formalne verifikacije omeji samo na izbrani del sistema. Proces `never { do:: p od }` se lahko uporabi za omejevanje formalne verifikacije na samo tista izvajanja, kjer v vseh stanjih trditev p drži.

Nikoli veljavne trditve imajo izredno veliko izrazno moč. Ročna priprava procesov “never claim” se izkaže kot zahtevna naloga, zato orodje Spin vključuje avtomatsko tvorbo procesa na osnovi podane formule v linearni temporalni logiki. Za opis enostavnejših pravilnostnih trditev, ki vključujejo časovno zaporedje trditev, je na voljo grafično orodje “timeline editor”. Specifikacijo zahtev je pogosto enostavneje zapisati z LTL formulo, poudariti pa velja, da je takšen zapis strogo manj izrazen kot proces “never claim”. Podoben odnos velja med uporabo časovnega zaporedja trditev in LTL formulami.

4.1.6 Temporalne formule

Osnovna pravilnostna trditev omogoča zapis formule stanja — podaja trditve, ki morajo držati v predvidenem stanju. Primera takšnih trditev sta p in q:

p: vrednost spremenljivke x je liho število,
q: vrednost spremenljivke x je enaka 42.

Pravilnost takšne trditve lahko preverimo v vsakem razširjenem stanju. Razširjeno stanje v jeziku Promela določajo vrednosti globalnih in lokalnih spremenljivk, trenutna stanja aktivnih procesov in vsebina vseh komunikacijskih kanalov. Večji nabor lastnosti se lahko opiše z mogočim oz. nemogočim zaporedjem formul stanja na celotni ω-izvajalni poti. Takšnim formulam pravimo temporalne formule, saj vključujejo časovno vzročnost trditev:

- p vedno drži,
- p od nekega stanja na izvajalni poti dalje za vedno ne drži,
- p vedno nekoč postane neveljavna še vsaj enkrat,
- q vedno implicira ¬p,
- p vedno implicira nekoč q.

Pravilnost temporalne formule se lahko ovrednoti samo nad celotnim zaporedjem stanj, ki sestavljajo urejeno množico izvajalne poti. V orodju Spin je podprta linearna temporalna logika. LTL formule so sestavljene iz formul stanja in temporalnih operatorjev. Semantika LTL je definirana nad neskončnimi izvajalnimi potmi. Z vpeljavo jecljanja pa se lahko obravnavajo tudi končne izvajalne poti.

Definicija 4.11 Vse formule stanja, vključno s $true$ in $false$, so veljavne temporalne formule. Če je α unarni temporalni operator, β binarni temporalni operator in sta p in q veljavni temporalni formuli, potem so veljavne temporalne formule tudi α p, p β q, (p) in $\neg$p.♦

V nadaljevanju so podane definicije temporalnih operatorjev linearne temporalne logike iz [36]. Zapis σ_i predstavlja i-ti element izvajalne poti σ, $\sigma[i]$ pa je del poti σ, ki se prične v i-tem elementu poti. Velja torej $\sigma \equiv \sigma[1] \equiv \sigma_1\sigma[2]$.

Definicija 4.12 Šibki "until", U, določa formula $\sigma[i] \models$ (p U q) $\iff$ $\sigma_i \models$ q $\vee$ ($\sigma_i \models$ p $\wedge$ $\sigma[i+1] \models$ (p U q)).♦

Definicija operatorja šibki "until" ne pričakuje, da bo formula stanja q postala veljavna. Formula p U q je torej veljavna tudi v primerih, ko q nikoli ne drži. To zahtevo doda definicija strogega operatorja "until".

Definicija 4.13 Strogi "until", U, določa formula $\sigma[i] \models$ (p U q) $\iff$ $\sigma[i] \models$ (p U q) $\wedge$ $\exists j, j \geq i, \sigma_j \models$ q.♦

Poseben primer operatorjev "until" predstavlja formula p U $false$, katere pravilnost je odvisna samo od formule p. Zahtevo, da formula p drži na celotni ω-izvajalni poti, lahko opišemo z operatorjem "always".

Definicija 4.14 Operator "always", $\Box$, določa formula $\sigma \models \Box$p $\iff$ $\sigma \models$ (p U $false$).♦

Poseben primer operatorjev "until" predstavlja tudi formula $true$ U q. Zahtevo, da formula q nekoč na ω-izvajalni poti zagotovo postane veljavna vsaj enkrat, lahko opišemo z operatorjem "eventually".

Definicija 4.15 Operator "eventually", $\Diamond$, določa formula $\sigma \models \Diamond$q $\iff$ $\sigma \models$ ($true$ U q).♦

Med operatorjema $\Diamond$ in $\Box$ veljata naslednji povezavi:

$$\neg\Diamond \text{p} \iff \Box\neg \text{p} \tag{4.2}$$

$$\neg\Box \text{p} \iff \Diamond\neg \text{p} \tag{4.3}$$

Zapis zahteve, da formula p drži v naslednjem stanju, omogoča še zadnji operator LTL — operator "next".

Definicija 4.16 Operator "next", X, določa formula $\sigma[i] \models$ X p $\iff$ $\sigma_{i+1} \models$ p).♦

Nekatere oblike LTL formul imajo zaradi pogoste uporabe posebno ime. Tabela 4.1 prikazuje pogosto uporabljene LTL formule z izgovorjavo opisane lastnosti in njenim imenom. Implikacijo in enakovrednost določata definiciji 4.17 in 4.18.

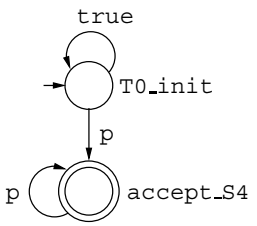
Definicija 4.17 Implikacijo, $\rightarrow$, določa formula p $\rightarrow$ q $\models$ ($\neg$ p)$\vee$ q.♦

Tabela 4.1: Pogosto uporabljene LTL formule

formula	izgovorjava	ime
$\Box p$	vedno p	invarianca
$\Diamond p$	nekoč p	zagotovilo
$p \rightarrow \Diamond q$	p implicira nekoč q	reakcija
$p \rightarrow q \; U \; r$	p implicira q dokler ne r	predhodnost
$\Box\Diamond p$	vedno nekoč p	ponavljanje (napredovanje)
$\Diamond\Box p$	nekoč vedno p	nespremenljivost (nenapredovanje)
$\Diamond p \rightarrow \Diamond q$	nekoč p implicira nekoč q	korelacija

Definicija 4.18 Enakovrednost, $\Longleftrightarrow$, določa formula $p \Longleftrightarrow q \models (p \rightarrow q) \wedge (q \rightarrow p)$.♦

V osemdesetih letih je bilo dokazano, da za vsako temporalno formulo obstaja Büchijev avtomat, v katerem so mogoče samo tiste izvajalne poti, ki zadoščajo podani formuli. Orodje Spin vključuje enega izmed obstoječih algoritmov za avtomatsko tvorbo Büchijevega avtomata iz LTL formule, vendar se lahko uporabi samo za opis specifikacije zahtev [20]. LTL se ne more uporabiti za opis modelov. Grafični vmesnik orodja Xspin je viden na sliki 3.61 (str. 114). Iz LTL formule se tvori proces "never claim". Ta predstavlja primerno označen avtomat stanj, v katerem je mogoča samo tista ω-regularna lastnost, ki jo predpisuje LTL formula. Spin podpira samo strogi "until". Slika 4.3 prikazuje primer pretvorbe LTL formule $\Diamond\Box p$ v Büchijev avtomat z avtomatsko tvorbo procesa "never claim". Za avtomatsko pretvorbo se lahko uporabijo tudi druga orodja, ki v nekaterih primerih tvorijo optimalnejšo predstavitev LTL formule [22].




```
never {    /* <>[]p */
T0_init:
        if
        :: ((p)) -> goto accept_S4
        :: (1) -> goto T0_init
        fi;
accept_S4:
        if
        :: ((p)) -> goto accept_S4
        fi;
}
```

Slika 4.3: Pretvorba LTL formule $\Diamond\Box p$ v Büchijev avtomat

4.1.7 Zaključek

Specifikacija zahtev se v orodju Spin lahko formalno zapiše z uporabo različnih pravilnostnih trditev, ki predpisujejo, katera izvajanja so dovoljena in katera ne. Verifikacijski algoritmi iščejo primere izvajalnih poti, ki dokazujejo nepravilnost podane pravilnostne trditve.

1. Osnovna pravilnostna trditev predstavlja formalni zapis trditve, da je nemogoče, da se podani izraz ovrednoti na boolovo vrednost 0.
2. Z uporabo oznak končnih stanj trdimo, da se izvajanje sistema nikoli ne bo zaključilo brez končanja vseh aktivnih procesov oz. blokade posameznih procesov v posebno označenih stanjih.

3. Z uporabo oznak stanj napredka trdimo, da ni mogoče neskončno izvajanje sistema, ki ne vsebuje vsaj eno stanje napredka neskončnokrat.
4. Z uporabo oznak dopustnih stanj trdimo, da ni mogoče neskončno izvajanje sistema, ki vsebuje vsaj eno dopustno stanje neskončnokrat.
5. Nikoli veljavna trditev pravi, da nobeno končno ali neskončno izvajanje sistema ne zadosti podani trditvi v celoti.

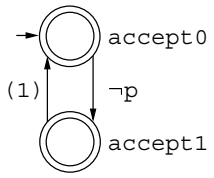
Orodje Spin ne poskuša neposredno dokazati 4.1, temveč išče izvajanja, ki s podano pravilnostno trditvijo ϕ niso skladna:

$$\neg(\Sigma \models \phi) \iff \exists\sigma, (\sigma \in \Sigma \wedge (\sigma \models \neg\phi)) \tag{4.4}$$

Če iskanje protiprimera, ki ga opisuje desna stran 4.4, uspe, smo pokazali, da leva stran 4.1 ne drži. Obnašanje sistema ni skladno s pravilnostno trditvijo ϕ. Če iskanje protiprimera ni uspešno, smo pokazali, da ne drži $\neg(\Sigma \models \phi)$, kar pomeni da je leva stran 4.1 izpolnjena in da sistem predstavlja model ϕ.

Lastnosti, ki jih lahko zapišemo z LTL, sodijo v podmnožico ω-regularnih lastnosti. Z LTL formulami lahko opišemo mnoge lastnosti obravnavanega modela. V primerih, ko to ni mogoče, pa se lahko vse ω-regularne lastnosti opišejo z neposredno uporabo procesa "never claim". Alternativni pristop v [19] predlaga uporabo Stutter-invariant Existential Quantified Linear Propositional Temporal Logic (SI-EQLTL). Če se v LTL formuli dopusti uporaba eksistenčnega kvantifikatorja pred eno izmed vključenih trditev, se lahko opišejo vse ω-regularne lastnosti.

Primer zanimive lastnosti predstavlja pravilnostna zahteva, da ne obstaja izvajalna pot, na kateri bi trditev p lahko veljala samo v lihih, nikoli pa v sodih stanjih izvajalne poti [36]. Enostavneje: ali obstaja izvajalna pot, na kateri trditev p nikoli ne drži v sodih stanjih? Poskus zapisa z LTL formulo $\Box$ X p ne predstavi lastnosti, saj predpisuje, da trditev p velja v vseh lihih stanjih. Lastnost lahko zapišemo z uporabo eksistenčnega kvantifikatorja nad navidezno trditvijo t: $\{\mathrm{E\ t}\}\ \mathrm{t} \wedge \Box(\mathrm{t} \rightarrow \mathrm{X}\ \neg\mathrm{t}) \wedge \Box(\neg\mathrm{t} \rightarrow \mathrm{X\ t}) \wedge \Box(\mathrm{p} \rightarrow \neg\mathrm{t})$[5]. Slika 4.4 prikazuje Büchijev avtomat, ki ga tvori orodje *eqltl* [19, 21]. Formula izraža lastnost, da obstaja trditev t, ki na začetku izvajalne poti σ^ω drži, v nadaljevanju pa izmenoma drži in ne drži. Zadnji del formule pa pravi, da trditev p vedno implicira $\neg$t. Izvajalna pot je torej mogoča takrat in samo takrat, ko trditev p ne drži v prvem stanju in v vsakem lihem stanju na izvajalni poti. Veljavnost trditve p se v sodih stanjih ne preverja. S tem se bistveno omejijo mogoče izvajalne poti, v katerih algoritmi formalne verifikacije preverjajo pravilnost trditve.




```
never {
accept0:
        if
        :: (!(p)) -> goto accept1
        fi;
accept1:
        if
        :: (1) -> goto accept0
        fi;
}
```

Slika 4.4: Pretvorba SI-EQLTL formule v Büchijev avtomat

[5]Če bi želeli zajeti zahtevo, da mora trditev p veljati v sodem stanju le v primeru, ko velja tudi v lihem, bi formulo morali dopolniti z $\wedge\Diamond(\neg\mathrm{p} \rightarrow \mathrm{t})$

4.2 Dopolnitev modela s sondami

Kjerkoli želimo opazovati spremembe določene količine, je potrebno namestiti merilne elemente — sonde. Pri postopku formalne verifikacije z orodjem Spin lahko sonde razdelimo na lokalne in globalne. Lokalne sonde lahko zaznajo samo spremembe v okviru posameznega procesa in so omejene na posebne oznake stanj in osnovne pravilnostne trditve. Za opisovanje ω-regularnih lastnosti pa je potrebno zagotoviti, da so sonde dostopne globalno. To narekuje zapis specifikacije zahtev v okviru zunanjega procesa "never claim". Ta mora imeti dostop do vseh elementov, ki se pojavljajo v formalnem zapisu zahteve.

Definicija 4.19 Sonda je element modela sistema, ki omogoča vpogled v obnašanje sistema.♦

Sonda je lahko primerno postavljena osnovna trditev, oznaka dovoljenega ali končnega stanja ter globalno dostopna kopija lokalnih spremenljivk. Formalno verifikacijo tvorjenega modela sistema lahko razdelimo v pet sklopov:

1. iskanje zagatnih stanj,
2. iskanje izvajalnih poti, ki niso skladne s semantiko jezika SDL,
3. iskanje potencialno nepravilnih izvajalnih poti,
4. iskanje cikličnih izvajalnih poti,
5. preverjanje temporalnih lastnosti modela.

V prvem sklopu se preverja varnost — iščejo se zagatna stanja modela sistema. Četrti sklop opisuje iskanje neželenih neskončnih izvajalnih poti, ki onemogočajo želeno izvajanje sistema — živost. Drugi preverja skladnost modela s semantičnimi pravili jezika SDL. Modeli nekaterih konstruktov za ta namen vključujejo sonde, ki ob kršitvi semantičnih pravil v povezavi z nadzornim procesom sprožijo napako med formalno verifikacijo modela sistema. V okviru tretjega sklopa iščemo izvajalne poti, ki so skladne s semantičnimi pravili jezika SDL, vendar potencialno predstavljajo neželeno obnašanje modela sistema. Primer takšnega dogodka je implicitni sprejem signala, ki lahko predstavlja nepričakovani dogodek v specifikaciji sistema ali pa namerno podspecifikacijo, ki temelji na semantičnih pravilih jezika SDL. Zadnji sklop vključuje preverjanje skladnosti modela s specifikacijami zahtev, ki izražajo časovno soodvisnost podanih trditev oz. formul stanja.

4.2.1 Iskanje zagatnih stanj

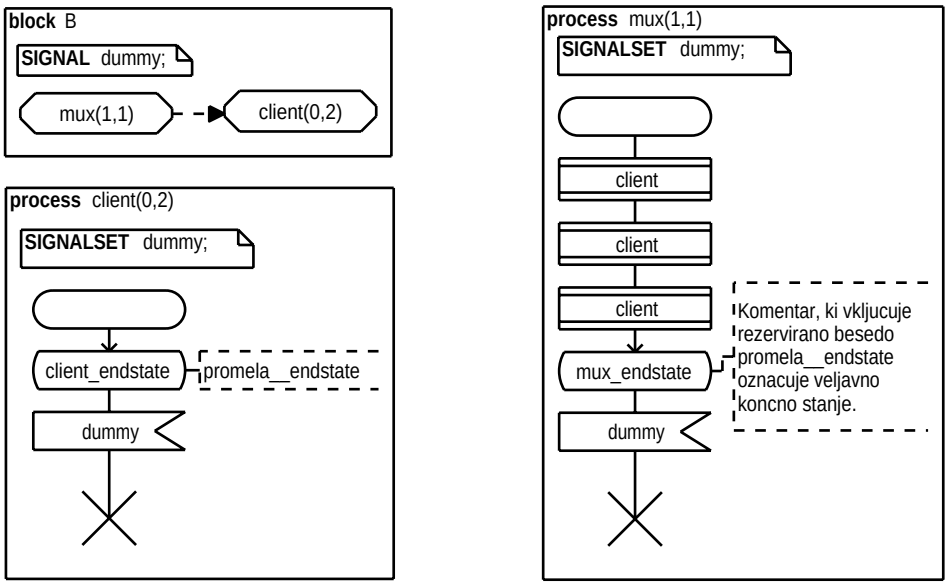
Zagatna stanja opisuje definicija 4.8. Na sliki 4.5 stanji `client_endstate` in `mux_endstate` predstavljata potencialni zagatni stanji, saj nikoli ne pride do sprejema signala `dummy`. Izvajanje procesov se v teh stanjih zaključi in oba procesa bi se blokirala. Med formalno verifikacijo se preveri, ali je v modelu sistema mogoča kakšna izvajalna pot, ki vključuje zagatno stanje. V takšnem primeru se sproži prekinitev formalne verifikacije, za analizo problema pa je na voljo protiprimer, ki natančno opisuje izvajalno pot. Vsa veljavna končna stanja je zato potrebno v specifikaciji sistema označiti s komentarjem, ki vsebuje ključno besedo `promela_endstate`. Takšna stanja se ob formalni verifikaciji obravnavajo kot veljavna končna stanja (razdelek 4.1.2).

4.2.2 Kršitve semantičnih pravil jezika SDL

Kršitve semantičnih pravil jezika SDL zaznavamo z uporabo globalne spremenljivke `pv__runtime_error` in osnovnih trditev ob koncu vsakega prehoda v grafu procesa med dvema eksplicitno definiranima stanjema.

Nekatere sonde so vključene v modele konstruktov jezika SDL in jih imenujemo trajne sonde. Te dostopajo do globalne spremenljivke `pv__runtime_error` in ji ob kršitvah dodelijo boolovo vrednost 1. Vse takšne izvajalne poti zazna samostojni nadzorni proces (slika 4.1 C, kjer velja `#define p pv__runtime_error`). Primera uporabe trajnih sond predstavljata modela konstruktov ustvari in odločitev.

Algoritem 3.32 (str. 104) omogoča zaznavo kršitev največjega predvidenega števila primerkov procesa. Pred tvorbo novega procesa predpisuje preverjanje trenutnega števila primerkov. Za ta namen sta v model sistema za vsak proces nameščeni dve sondi. Prva hrani trenutno število, druga pa v okviru definicije makroja določa največje dovoljeno število primerkov procesa. V primeru, ko ne drži relacija, ki jo predpisuje specifikacija sistema, postavi vrednost globalne spremenljivke `pv__runtime_error` na boolovo vrednost 1. Slika 4.5 prikazuje primer sistema, ki krši podano specifikacijo. Avtomatsko tvorjen model sistema je predstavljen v prilogi C.1.1.



Slika 4.5: Specifikacija sistema `ustvari`

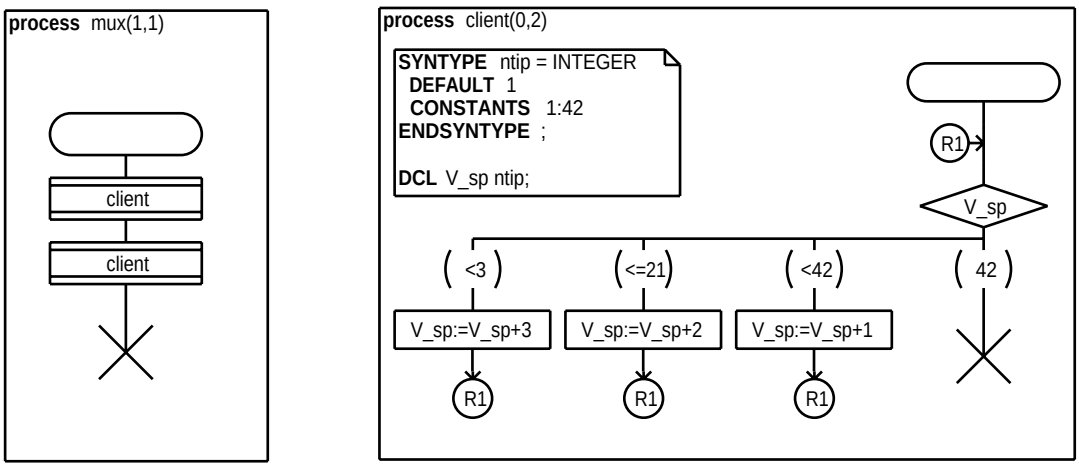
Formalna verifikacija modela najde kršitev osnovne pravilnostne trditve, ki je podana v okviru procesa `monitor`. Slika 4.6 prikazuje rezultate formalne verifikacije z orodjem Spin. S pregledom tvorjenega protiprimera hitro ugotovimo, da je prišlo do napake ob poskusu tvorbe tretjega primerka procesa `client`. Iz specifikacije procesa na sliki 4.5 je razvidno, da sta za ta proces predvidena največ dva primerka. Če odstranimo enega izmed konstruktov ustvari v procesu `mux`, formalna verifikacija ne najde nobene izvajalne poti, ki bi kršila semantična pravila jezika SDL.

Naslednji primer kršitve semantičnih pravil predstavlja nepravilna uporaba konstrukta odločitev. Odločitev sestavlja eno vprašanje in množica odgovorov. Standard [57] predpisuje, da mora biti na vsaki mogoči izvajalni poti pravilen vedno en in samo en odgovor. Kršitev te zahteve predstavlja napako v izvajanju sistema. Primerno modeliranje konstrukta

```
pan: assertion violated 0 (at depth 9)
pan: wrote pan_in.trail
(Spin Version 4.1.1 -- 2 January 2004
+ Discrete-Time Extension Version 0.1.2 (Multiple Tick) -- 24 December 2002)
Warning: Search not completed
+ Partial Order Reduction

Full statespace search for:
never claim          - (not selected)
assertion violations +
cycle checks        - (disabled by -DSAFETY)
invalid end states +

State-vector 184 byte, depth reached 9, errors: 1
       7 states, stored
       0 states, matched
       7 transitions (= stored+matched)
       3 atomic steps
hash conflicts: 0 (resolved)
(max size 2^19 states)

Stats on memory usage (in Megabytes):
0.001  equivalent memory usage for states (stored*(State-vector + overhead))
0.283  actual memory usage for states (unsuccessful compression: 21045.24%)
State-vector as stored = 40399 byte + 8 byte overhead
2.097  memory used for hash table (-w19)
0.320  memory used for DFS stack (-m10000)
2.622  total actual memory usage
```

Slika 4.6: Rezultati formalne verifikacije modela sistema `ustvari`

omogoča namestitev sonde, ki postavi globalno spremenljivko `pv_runtime_error` ob kršitvi te zahteve na boolovo vrednost 1. Slika 4.7 prikazuje primer procesa `client`, ki podano zahtevo krši, dokler je vrednost spremenljivke `V_sp` manjša od 22.



Slika 4.7: Nepravilna uporaba konstrukta odločitev v specifikaciji procesa `client`

Naključna simulacija z orodji Spin in OG te kršitve ne odkrije. Takšna specifikacija dejansko ne krši semantičnih pravil jezika Promela, zato model odločitve predstavlja veljavno nedeterministično izbiro. Orodje OG pa v primeru veljavnosti večjega števila odgovorov vedno izbere tistega, ki je naveden "prvi" (veljaven odgovor, ki je najbolj levi na sliki 4.7)[6]. Sočasne veljavnosti večjega števila odgovorov ne najde niti izčrpna simulacija (slika 4.8). Slika 4.9 prikazuje model konstrukta odločitev z vključitvijo sonde za zaznavo

[6]To velja za naključno simulacijo in generator kode v jeziku C.

njegove napačne uporabe.

```
Number of states : 5
Number of transitions : 5
Maximum depth reached : 4
Maximum breadth reached : 2
duration : 0 mn 0 s

Number of exceptions : 0
Number of deadlocks : 1
Number of stop conditions : 0
Transitions coverage rate : 100.00 (0 transitions not covered)
States coverage rate : 100.00 (0 states not covered)
Basic blocks coverage rate : 100.00 (0 basic blocks not covered)

Sub-graphs statistics :
process mux : 2 states
process client : 4 states
queue mux!queue : 2 states
queue client!queue : 3 states
```

Slika 4.8: Izčrpna simulacija z orodjem OG ne zazna kršitve semantičnih pravil

```
if
::((V_sp.val < 3) && (V_sp.val <= 21)) ||
  ((V_sp.val < 3) && (V_sp.val < 42)) ||
  ((V_sp.val < 3) && (V_sp.val == 42)) ||
  ((V_sp.val <= 21) && (V_sp.val < 42)) ||
  ((V_sp.val <= 21) && (V_sp.val == 42)) ||
  ((V_sp.val < 42) && (V_sp.val == 42))
  -> pv__runtime_error=true;
:: else ->
   if
   :: V_sp.val < 3 ->
     d_step{
        V_sp.val = V_sp.val+3;
        assert((1<=V_sp.val)&&(V_sp.val<=42))
     }
     goto R1;
   :: V_sp.val <= 21 ->
     d_step{
        V_sp.val = V_sp.val+2;
        assert((1<=V_sp.val)&&(V_sp.val<=42))
     }
     goto R1;
   :: V_sp.val < 42 ->
     d_step{
        V_sp.val = V_sp.val+1;
        assert((1<=V_sp.val)&&(V_sp.val<=42))
     }
     goto R1;
   :: V_sp.val == 42 ->
     goto end
   fi;
fi;
```

Slika 4.9: Primer modeliranja konstrukta odločitev z vključitvijo sonde

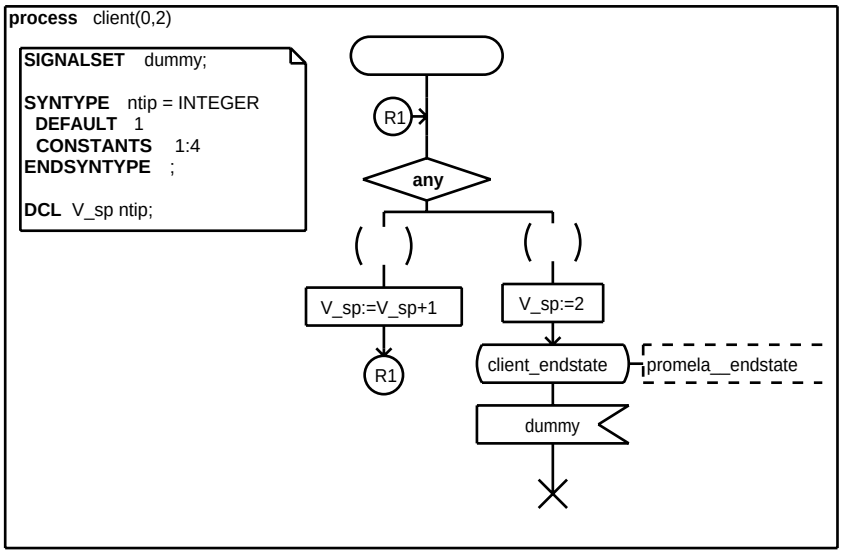
Prav tako so neveljavne izvajalne poti, kjer vprašanju ne zadosti nobeden od ponujenih odgovorov. V tem primeru se proces blokira. Ker to implicitno stanje ni označeno kot veljavno končno stanje (razdelek 4.1.2), takšne izvajalne poti med formalno verifikacijo sprožijo napako.

V skupino potencialnih kršitev semantičnih pravil jezika SDL sodi tudi preverjanje skladnosti trenutne vrednosti spremenljivke z omejitvami, ki jih določa predpisan podatkovni tip. Dosledno omejevanje zaloge vrednosti spremenljivk s primerno definicijo podintervalnih podatkovnih tipov omogoča hitro zaznavo neželenih prekoračitev vrednosti v okviru specifi-

kacije sistema.

Predvideno zalogo vrednosti in morebitno privzeto vrednost pridobimo za vsak podatkovni tip v okviru algoritma 3.3 (str. 32). Če definicija podatkovnega tipa ali deklaracija spremenljivke ne vsebuje privzete vrednosti, je ta spremenljivka do prve prireditve nedefinirana. Tako določa standard [57]. Uporaba takšne spremenljivke v kakšnem drugem izrazu predstavlja kršitev izvajanja. Izvajanje, ki se v tem primeru nadaljuje, ni več predvidljivo[7]. Semantika jezika Promela obravnava takšne primere drugače. Če privzeta vrednost ni definirana eksplicitno, se vsem spremenljivkam po privzetem dodeli vrednost nič in ne predstavlja izvora nepredvidljivosti. Model specifikacije je zato potrebno dopolniti s sondami, ki bodo omogočile zaznavo obeh kršitev — uporabo spremenljivke, ki ji še ni bila prirejena nobena vrednost, in prirejanja izven predvidene zaloge vrednosti pridruženega podatkovnega tipa.

Neposredno prirejanje vrednosti, ki je izven predvidene zaloge vrednosti, se lahko zazna s statično analizo. Zato ni potrebno vršiti preverjanja privzetih vrednosti. Slika 4.10 prikazuje primer izvajanja, ki ga s statično analizo ni mogoče zaznati. Procesu `client` smo dodali definicijo podintervalnega podatkovnega tipa `ntip` s predpisano zalogo vrednosti $\{1, 2, 3, 4\}$ in privzeto vrednostjo 1. V zanki z oznako R1 se lahko spremenljivki tega tipa vrednost poveča poljubno mnogokrat.



Slika 4.10: Razširitev specifikacije procesa `client`

Za namene iskanja takšnih kršitev se dopolni algoritem prirejanja vrednosti tako, da se v okviru neprekinljivega zaporedja `d_step` prirejanje izvede skupaj s stavkom `assert`. Vsaki prireditvi sledi preverjanje skladnosti s predvideno zalogo vrednosti, kot to prikazuje model nedeterministične odločitve na sliki 4.11. Razširitev prirejanja prikazujejo vrstice 3-6 in 9-12. Osnovni pravilnostni trditvi (5, 11) po vsaki prireditvi preverita skladnost s specifikacijo sistema.

Z množico osnovnih pravilnostnih trditev se med formalno verifikacijo zagotovo najdejo vse izvajalne poti, ki niso skladne s specifikacijo sistema. Število korakov v posamezni izvajalni poti se zaradi uporabe stavka `d_step` ne poveča, prostorska zahtevnost verifikacije se ne spremeni, poveča pa se procesna zahtevnost, saj se ob vsakem prirejanju vrši tudi preverba osnovne pravilnostne trditve. Alternativni pristop, ki bi ob izbranih trenutkih preverjal

[7] To velja tudi za deterministične izvajalne poti.

```
1 if
2 :: true ->
3   d_step{
4      V_sp.val = V_sp.val+1;
5      assert((1<=V_sp.val)&&(V_sp.val<=4))
6   }
7   goto R1;
8 :: true ->
9   d_step{
10     V_sp.val = 2;
11     assert((1<=V_sp.val)&&(V_sp.val<=4))
12  }
13  goto client_endstate;
14 fi;
```

Slika 4.11: Primer sonde za preverjanje skladnosti s predpisano zalogo vrednosti

skupino spremenljivk, je s stališča zahtevnosti operacije učinkovitejši, če se v istem prehodu posamezni spremenljivki vrednost priredi večkrat. Preverjanje skladnosti s predvideno zalogo vrednosti je potrebno izvajati ob vsakem prirejanju, saj v nasprotnem ni mogoče zaznati nepravilnih prirejanj, ki se izvedejo v vmesnih korakih.

Sledenje spremenljivkam, ki jim ob deklaraciji ali med izvajanjem še ni bila prirejena nobena vrednost, je v Promeli mogoče samo z njihovo eksplicitno označbo. Za takšne spremenljivke pravimo, da imajo nedefinirano vrednost. Primer takšnega prirejanja bi predstavljal proces `client` na sliki 4.10, če definicija podatkovnega tipa `ntip` ne bi vsebovala privzete vrednosti. Izbrali smo rešitev, kjer v vsakem podatkovnem tipu eden izmed literalov predstavlja nedefinirano vrednost. Pri standardnih podatkovnih tipih je to najvišja pozitivna vrednost, naštevnim in podintervalnim podatkovnim tipom pa dodamo en literal za predstavitev nedefinirane vrednosti. Množica vseh "nedefiniranih" vrednosti je torej $\subseteq (i \cup j), i \in$ *spec.def.ptip*, $j \in$ *spec.def.ntip* $\wedge$ j*.pmltip* $=$ *npt*. Sledenje tej zahtevi torej bistveno spremeni avtomatsko tvorjeni model sistema, saj so spremenjeni vsi naštevni in podintervalni tipi, vrednosti vseh spremenljivk, ki se uporabijo v izrazu, pa se preverjajo z osnovno pravilnostno trditvijo. Kadar je vrednost katere izmed spremenjivk enaka "nedefinirani" vrednosti, se sproži prekinitev formalne verifikacije. Prostorska zahtevnost se poveča samo v primerih, ko to narekuje uporabo dodatnega bita za predstavitev posameznega podatkovnega tipa. V primeru podintervalnega podatkovnega tipa `ntip` na sliki 4.10 temu ni tako. Za predstavitev predvidene zaloge vrednosti so potrebni trije biti ($round_up(log_2(4+1))$)[8]. Če predvideno zalogo vrednosti povečamo za en literal (0 ali 5), se število bitov, ki je potrebno za predstavitev spremenljivke podintervalnega tipa v vektorju stanj, ne poveča. Prostorska zahtevnost predstavitve modela se torej poveča samo v primerih, ko se za opis spremenljivk posameznega naštevnega ali podintervalnega tipa uporabi dodatni bit v vektorju stanj. Pri podintervalnih tipih se za predstavitev "nedefinirane" vrednosti uporabi najnižja vrednost, ki ni v predvideni zalogi vrednosti. Tako se v primeru definicije podintervalnega tipa z zalogo vrednosti $\{1, 2, 3\}$ izbere število 0. S tem se prostorska zahtevnost ne spremeni, v primeru uporabe števila 4, pa bi za vsako spremenljivko tega tipa porabili dodaten bit v vektorju stanj ($round_up(log_2(3+1)) < round_up(log_2(4+1)) \iff 2 < 3$).

Pomanjkljivost tega pristopa predstavljata:

- nezmožnost modeliranja specifikacij, ki uporabljajo celotno zalogo vrednosti standardnih podatkovnih tipov,
- sledenje spremenljivkam podatkovnega tipa $BOOLEAN$ ni mogoče.

[8]Skladno z algoritmom 3.3 (str. 32) se ob analizi podintervalnih tipov pri izračunu upošteva samo predvidena zgornja vrednost in ne dejansko število literalov. Takšna odločitev je bila sprejeta zaradi transparentnega modeliranja specifikacije, saj bi predstavitev z naštevnim tipom narekovala preslikavo vrednosti.

Omenjene pomanjkljivosti odpravi razširitev modela z dodatnimi spremenljivkami, ki hranijo podatek o trenutnem stanju spremenljivke. Ta rešitev poveča prostorsko zahtevnost formalne verifikacije, saj vnaša nove spremenljivke tipa boolean. Zato sonde za detekcijo uporabe spremenljivk z nedefinirano vrednostjo ne sodijo v množico trajnih sond.

Preverjanje skladnosti s semantiko jezika SDL se lahko izvaja z uporabo:

1. trajnih sond,
2. sond za zaznavo napačne uporabe konstrukta odločitev,
3. sond za zaznavo kršitev predvidenih zalog vrednosti,
4. sond za zaznavo uporabe spremenljivk, ki jim še ni bila prirejena nobena vrednost.

V modelu sistema so vedno prisotne trajne sonde, preostali tri vrste sond, pa se lahko dodata neodvisno. S tem se zahtevnost formalne verifikacije lahko razdeli na dva neodvisna koraka.

4.2.3 Iskanje potencialno nepravilnih izvajalnih poti

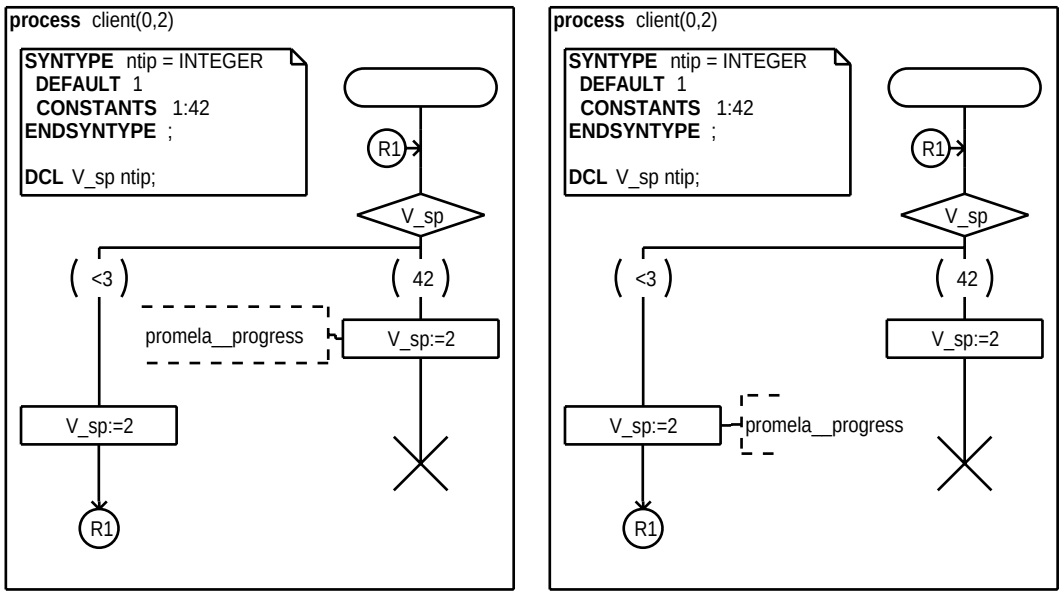
V skupino potencialno nepravilnih izvajalnih poti sodijo izvajanja, kjer se zaradi nepopolnega opisa obnašanja pri interpretaciji specifikacije uporabijo semantična pravila jezika SDL. Primer predstavlja implicitni sprejem signala, saj se pripeti le v primerih, ko sprejem ni bil eksplicitno predviden. To lahko opisuje čisto regularno izvajalno pot, ki jo je snovalec sistema predvidel, potencialno pa predstavlja podspecifikacijo. Ob tvorbi modela se lahko takšni sprejemi označijo s sondo, ki sproži prekinitev formalne verifikacije. Ker so implicitni sprejemi v nekaterih stanjih procesa lahko zaželeni, je potrebno takšna stanja pred tvorbo modela eksplicitno označiti s primernimi parametri ob klicu orodja $sdl2pml$. Podati je potrebno ime procesa, eksplicitno definirano stanje in ime signala. Tako se preverjanje izvaja samo nad želenimi procesi, stanji in signali.

Uporaba konstrukta naloga z vsebino “promela_error” omogoča piscu specifikacije, da eksplicitno označi tiste izvajalne poti, ki v specifikaciji niso zaželene. Takšen primer je konstrukt odločitev, kjer se nikoli ne sme izvesti prehod od odgovora “else”. Orodje $sld2pml$ na mesto konstrukta naloga vstavi sondo, ki ob formalni verifikaciji sproži prekinitev.

4.2.4 Iskanje cikličnih izvajalnih poti

Med formalno verifikacijo se obravnavajo tudi neskončne izvajalne poti. Če ciklična izvajalna pot ne vsebuje stanj napredka, se med formalno verifikacijo sproži prekinitev. Stanja napredka v specifikaciji sistema označujemo z rezervirano besedo `promela_progress` v komentarju konstrukta, ki predstavlja predvideno ciklično operacijo. Slika 4.12 prikazuje dve specifikaciji procesa `client` z neskončno izvajalno potjo. Prva nima stanj napredka, druga pa kot stanje napredka označuje konstrukt naloga. Stanje napredka torej lahko označuje poljuben konstrukt v specifikaciji in ni omejen samo na eksplicitno definirana stanja procesa. Ob formalni verifikaciji sproži prekinitev izvajanja samo prvi primer.

V specifikaciji sistema se lahko z rezervirano besedo `promela_accept` označijo tudi dopustna stanja. Ob formalni verifikaciji se iščejo takšne izvajalne poti, kjer se vsaj eno dopustno stanje pojavlja neskončnokrat (razdelek 4.1.4). Te oznake se navadno uporabljajo samo znotraj avtomatsko tvorjenih procesov “never claim”.



Slika 4.12: Specifikacija procesa `client` brez in z oznako stanja napredka

4.2.5 Preverjanje temporalnih lastnosti modela

V orodju Spin se za preverjanje temporalnih lastnosti modela uporabljajo procesi "never claim", kot to opisuje razdelek 4.1.5. Priporočljiva je avtomatska tvorba procesov z uporabo vgrajenega algoritma ali drugih orodij. Ročna priprava Büchijevih avtomatov z zahtevnimi lastnostmi lahko hitro vodi do lažnih potrditev specifikacije zahtev.

Med avtomatsko tvorbo modela je potrebno omogočiti dostop do vseh spremenljivk (sond), ki se uporabijo v pridruženem Büchijevem avtomatu. Orodje $sdl2pml$ to zagotavlja z globalno definicijo vseh uporabljenih spremenljivk. V jeziku SDL so spremenljivke pridružene procesu, zato lahko globalna definicija vsebuje rezervirano besedo **local**. Ta določa, da do te globalne spremenljivke dostopa samo en proces. Če se proces med izvajanjem lahko pojavi v več primerkih, je potrebno zagotoviti primerno preslikavo z uporabo polja sond. Ime vsake sonde vključuje natančno pot do mesta deklaracije v specifikaciji sistema.

Poseben primer predstavlja pošiljanje in sprejem signalov. Obravnavanemu signalu se med tvorbo modela pridružita sondi — ena za oddajo in ena za sprejem. Ob tvorbi modela sistema je potrebno upoštevati specifikacijo zahtev. Pogost primer specifikacije zahtev opisuje LTL formula $(\Box(p \rightarrow \Diamond q)) \wedge (\Diamond p)$. Sonda p sledi oddaji, sonda q pa sprejemu signala. Specifikacija zahtev pravi, da v kolikor se pošlje signal p, se nekoč sprejme tudi signal q. Ob odposlanju izbranega signala je potrebno vrednost pridružene sonde (p) postaviti na boolovo vrednost 1. To se stori v okviru istega atomarnega zaporedja. V naslednjem koraku se sondi priredi boolova vrednost 0. Drugo prirejanje se mora izvesti izven tega atomarnega zaporedja, sicer proces "never claim" ne more zaznati spremembe vrednosti. Ponovna postavitev vrednosti na boolovo vrednost 0 omogoča zaznavo večkratne oddaje signala. Sondi q se na enak način prirejajo vrednosti ob sprejemu signala. Asinhrono izvajanje procesov omogoča izvajalno pot, na kateri v nekem stanju po sprejemu signala velja $p \wedge \neg q$. Do tega lahko pride tudi ob enkratni oddaji in sprejemu signala. Skladno s specifikacijo zahtev, bi temu stanju nekoč moralo slediti stanje, kjer bi imela sonda q pridruženo boolovo vrednost 1. Do tega lahko pride samo pri cikličnem izvajanju, ki vključuje oddajo in sprejem signala. Potencialna nepravilnost se odpravi tako, da se sonda p ob vsakem sprejemu signala postavi na boolovo vrednost 0. To je potrebno storiti v okviru istega atomarnega zaporedja, kjer se

vrši sprejem signala in postavitev sonde q na boolovo vrednost 1. Obravnava sond za sledenje oddaji in sprejemu signalov je odvisna od specifikacije zahtev, zato se ob tvorbi modela določa eksplicitno. V kolikor se proces med izvajanjem lahko pojavi v več primerkih, je potrebno zagotoviti primerno preslikavo z uporabo polja sond. V nasprotnem primeru ena sonda spremlja delo vseh primerkov procesa.

5. poglavje

Primer verifikacije specifikacije protokola V.76

Primer avtomatske tvorbe modela s sondami in formalne verifikacije protokola V.76 temelji na specifikaciji protokola v [17]. Specifikacija opisuje poenostavljeno verzijo protokola, ki ga določa ITU standard V.76. Protokol svojim uporabnikom omogoča vzpostavljanje več sočasnih nadzornih ali podatkovnih povezav. Podrobnejši opis funkcionalnosti protokola je opisan v [45].

5.1 Popravki izvorne specifikacije

V obravnavani specifikaciji je uporabljen ASN.1 podatkovni tip "choice", ki ga definira [60]. Uporaba "choice" omogoča zelo eleganten opis komunikacije v primerih, ko se ob prenosu strukture vedno obravnava samo en element. V orodju $sdl2pml$ njegova uporaba ni podprta, zato smo ga v specifikaciji nadomestili z opisom na nižjem abstraktnem nivoju. Dodali smo naštevni podatkovni tip `T_CHOICE`, katerega literali opisujejo elemente okvirja protokola V.76 (slika 5.1). Ob uporabi podatkovnega tipa "choice" se ta definicija izvede implicitno. Ob izvajanju sistema se aktivni element zapisa `V76paramTyp` hrani v dodanem elementu `presen`. V povezavi z dodatnimi prireditvenimi stavki pred pošiljanjem signalov smo v celoti nadomestili uporabo rezerviranega elementa **present**, ki je implicitno definiran v okviru podatkovnega tipa "choice".

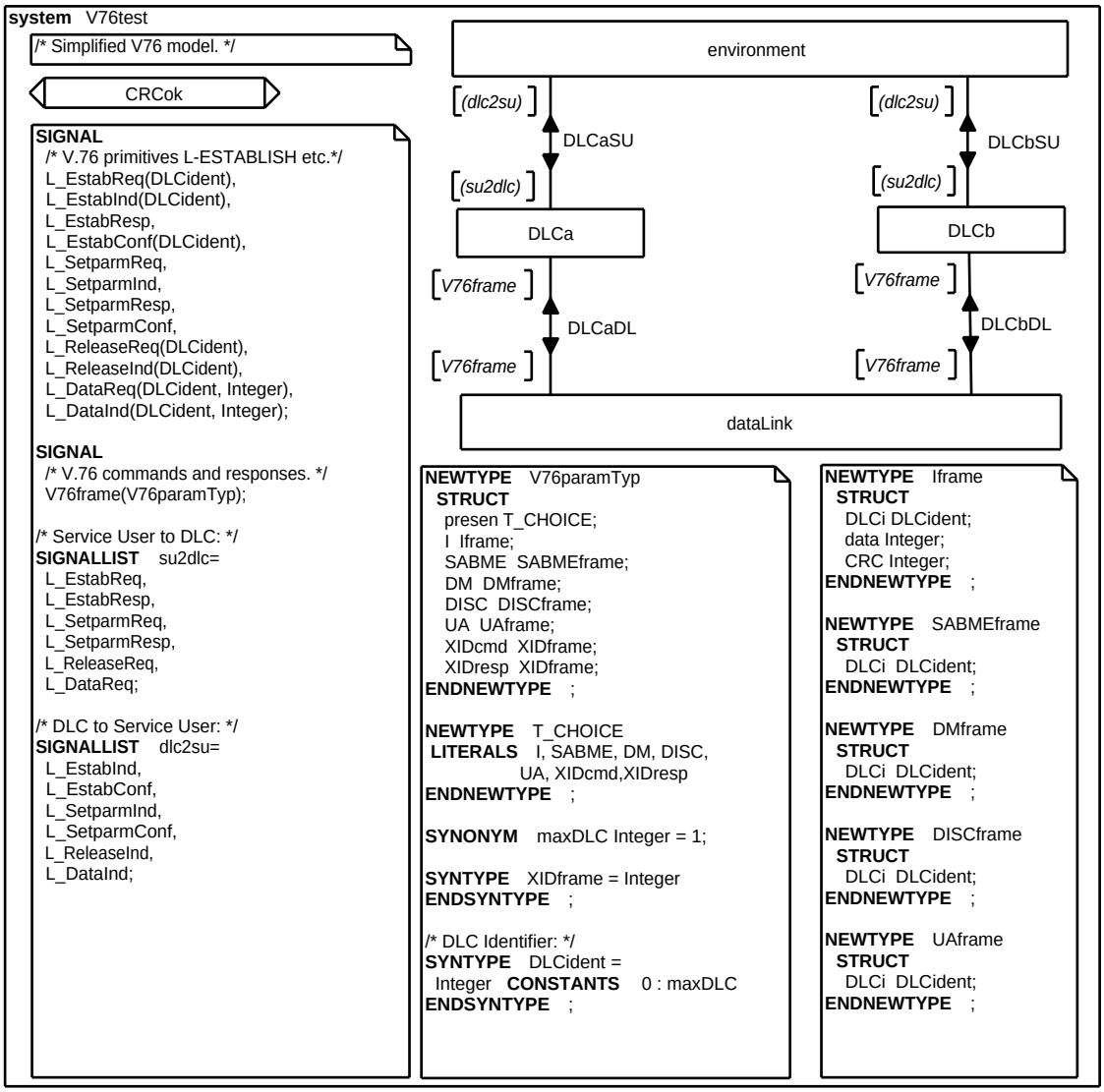
Odločitev, ki je bila v okviru procesov `AtoB` in `BtoA` podana z neformalnim besedilom, smo nadomestili z nedeterministično izbiro (slika 5.4).

Vhodna specifikacija je v celoti zapisana v prilogi C.2.1. Komentarje in zapise tipa CIF (Common Interchange Format) smo zaradi preglednosti in kompaktnosti zapisa odstranili.

5.2 Specifikacija v jeziku SDL

Slika 5.1 prikazuje grafično specifikacijo sistema in vmesnik do okolja. Protokol V.76 je specificiran v okviru bloka DLCx. Sistem `V76test` sestavljata dve specifikaciji protokola V.76 (bloka `DLCa` in `DLCb`) in model nezanesljivega sloja podatkovne povezave (blok `dataLink`). Predvidena sta dva uporabnika sistema, ki za komunikacijo uporabljata protokol V.76. Uporabnikov specifikacija ne opisuje, pričakovano obnašanje pa določa [45]. Pričakovani potek komunikacije prikazujejo diagrami MSC na sliki 5.5.

Bloka `DLCa` in `DLCb` sta enaka. Slika 5.2 prikazuje blok `DLCa`. Sestavljen je iz



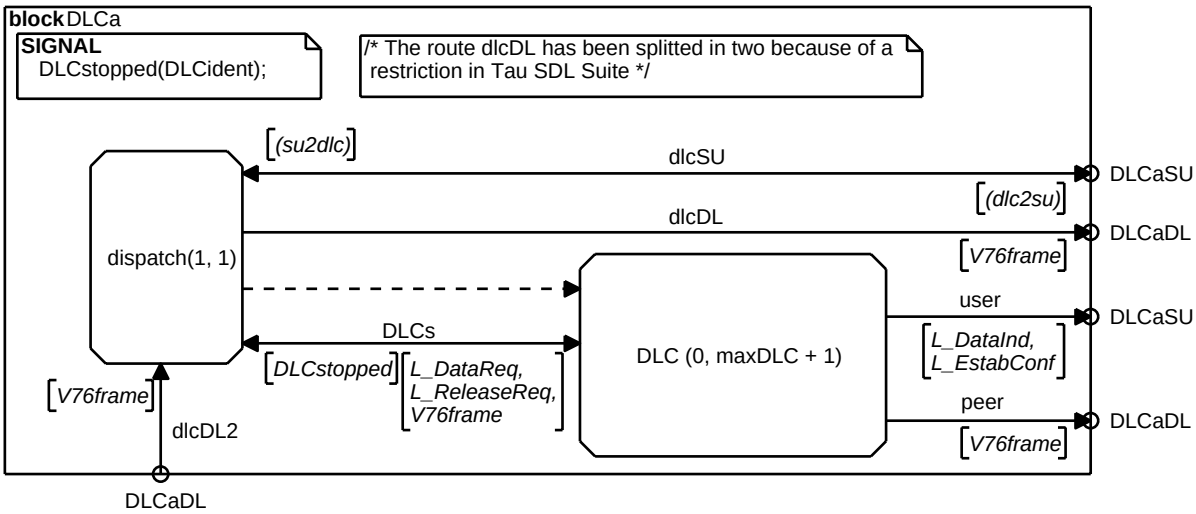
Slika 5.1: Grafični opis specifikacije sistema `V76test`

dveh procesov. Proces `dispatch` je upravljalni proces, ki se tvori že ob zagonu sistema. Največje predvideno število primerkov je ena. Proces `DLC` se tvori na zahtevo procesa `dispatch` za vsako vzpostavljeno podatkovno povezavo. Tvorbo procesa nakazuje prekinjena črta na sliki 5.2. Največje dovoljeno število primerkov procesa, in s tem sočasnih povezav med uporabnikoma, določa izraz `maxDLC + 1`. Sinonim `maxDLC` je definiran v okviru definicij na sliki 5.1. Avtomata stanj obeh procesov sta podana v prilogi C.2.1.

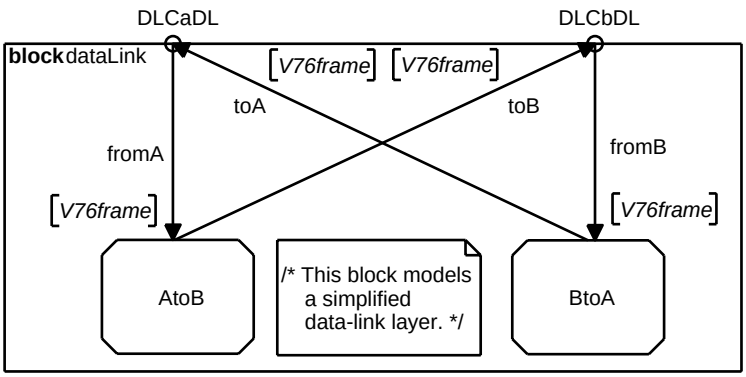
Blok `DLCa` je povezan z modelom nezanesljivega sloja podatkovne povezave s kanalom `DLCaDL`, ki lahko prenaša vrednosti podatkovnega tipa `V76paramTyp` (slika 5.1). Podatkovni tip predstavlja okvir protokola V.76. Slika 5.3 prikazuje povezavo procesov `AtoB` in `BtoA` z mejami bloka `dataLink`. Signalne poti se povezujejo na kanale z uporabo priključkov.

Slika 5.4 prikazuje graf procesa `AtoB`. Z uporabo rezervirane besede **ANY** v vprašanju konstrukta odločitev se modelira nedeterministična izbira med posredovanjem prejetega okvirja ali njegovo izgubo.

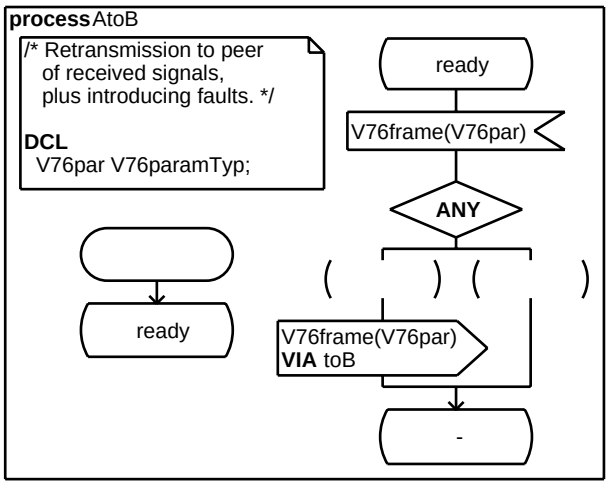
Sledi opis predvidenega obnašanja uporabnikov sistema (SU — System User) [45, 17]. Uporabnik lahko od sistema zahteva:



Slika 5.2: Grafični opis specifikacije bloka `DLCa`



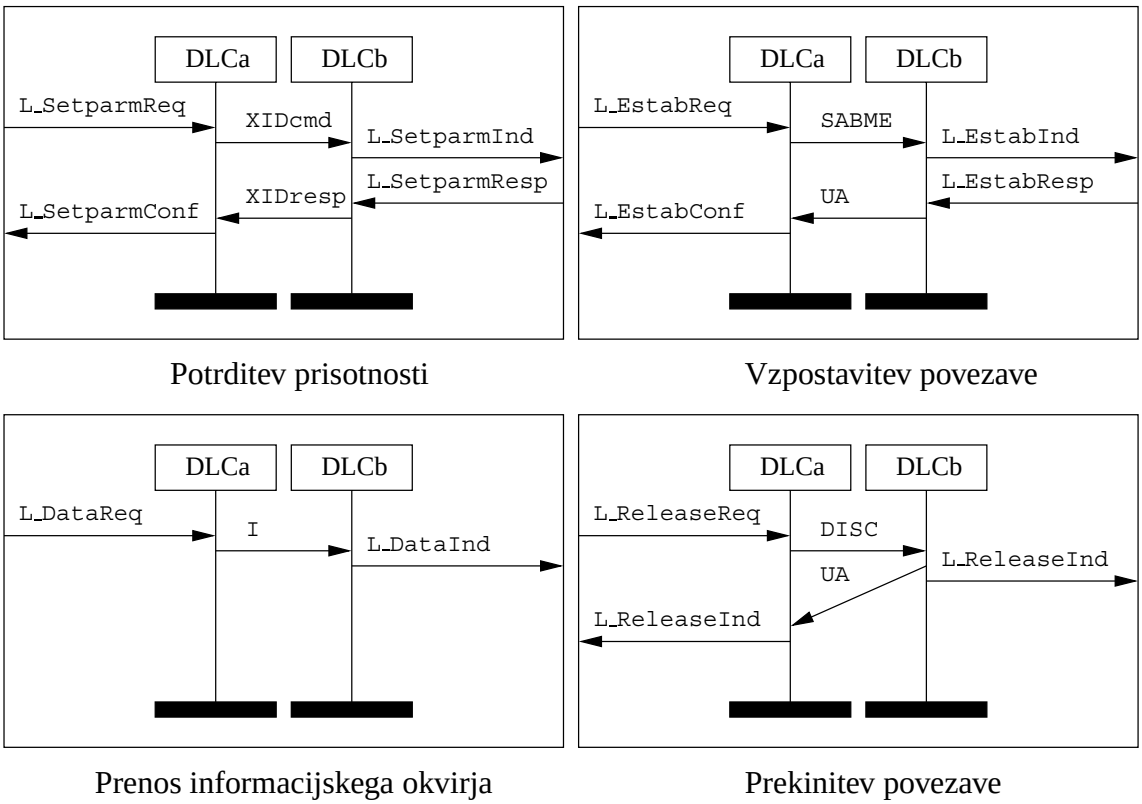
Slika 5.3: Grafični opis specifikacije bloka `dataLink`



Slika 5.4: Delna specifikacija grafa procesa `AtoB`

1. izmenjavo parametrov,
2. vzpostavitev podatkovne povezave,
3. prenos informacijskih okvirjev in
4. prekinitev povezave.

Uporabnika sistema `V76test` lahko imata `maxDLC + 1` sočasnih povezav. Podprti primitivi protokola in njihovi podtipi (request, indication, response, confirm) so predstavljeni v okviru definicije signalov na sliki 5.1. Primitivi so predstavljeni z množico signalov — po en signal za vsak podtip. Ob definiciji komunikacijskih gradnikov so razdeljeni na tiste, ki jih SU pošilja oz. sprejema — signalni listi `dlc2su` in `su2dls`. Ob večjem številu sočasnih povezav je potrebno ločiti med primitivi različnih povezav. Primitivi so torej pridruženi posamezni podatkovni povezavi. Za primerno preslikavo med številko podatkovne povezave in pridruženim procesom `DLC` je zadolžen proces `dispatch`. Komunikacija med elementoma protokola V.76 poteka z izmenjavo ukaznih okvirjev. Standard predvideva trinajst ukazov, obravnavana specifikacija pa opisuje podmnožico šestih. Zbrani so v okviru zapisa `V76paramTyp` (slika 5.1) in se prenašajo s signalom `V76frame`.

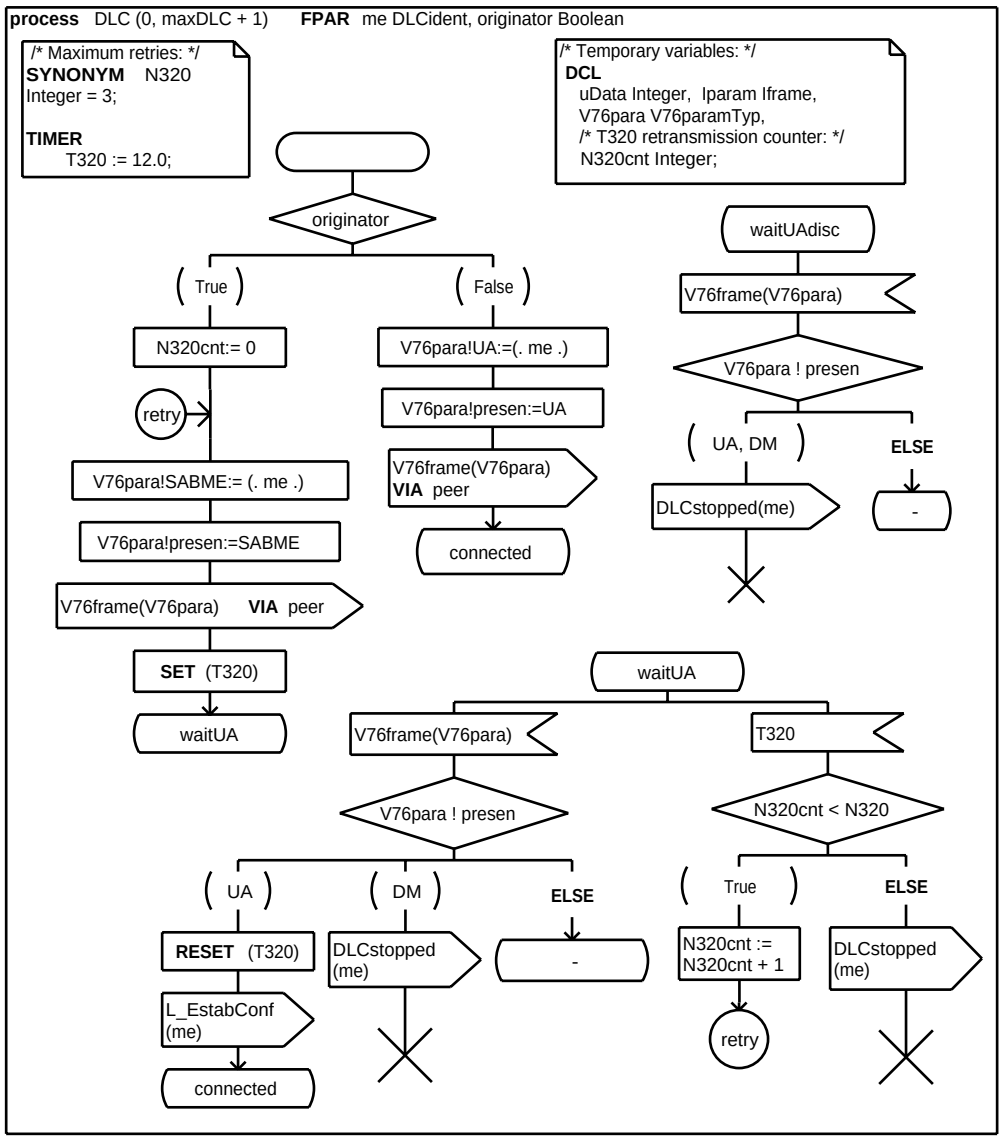


Slika 5.5: Diagrami MSC

Potrditev prisotnosti partnerjev v komunikaciji (SU-jev) in izmenjavo parametrov predstavlja neobvezen del vzpostavljanja povezave. Prikazana je na prvem diagramu MSC na sliki 5.5. Drugi diagram MSC prikazuje vzpostavljanje podatkovne povezave. Iniciator (SUa) pošlje zahtevo za vzpostavitev povezave s signalom `L_EstabReq` (primitiv L-ESTABLISH). Proces `dispatch` v tem trenutku poskusi vzpostaviti novo povezavo s tvorbo novega procesa `DLC`. Ta s prenosom ukaza `SABME` (Set Asynchronous Balanced Mode Extended) posreduje zahtevo svojemu partnerju v komunikaciji — procesu `DLC`, ki

je zadolžen za to povezavo na ponorni strani. Slika 5.6 prikazuje del grafa procesa DLC, ki se izvede ob tvorbi procesa in definira sprejem ukaza UA. Še pred oddajo signala se resetira števec ponovnih oddaj N320cnt. Po oddaji signala V76frame se postavi časovnik ponovne oddaje T320.

Proces dispatch na sprejemni strani preveri, ali lahko uporabnik (SUb) sprejme novo povezavo (L_EstabInd). V primeru pozitivnega odgovora (L_EstabResp) se tvori nov proces DLC, ki je zadolžen za to povezavo. Z dejanskimi parametri se procesu dodeli številka povezave (me) in vloga ponornega procesa (originator postavi na False). Po tvorbi se v prvem prehodu s signalom V76frame pošlje neoštevilčena potrditev UA (Unnumbered Acknowledge) (Slika 5.6). Ob sprejemu neoštevilčene potrditve se resetira časovnik ponovne oddaje. O uspešnosti vzpostavitve povezave se s signalom L_EstabConf obvesti iniciatorja (SUa), proces pa se postavi v stanje connected.



Slika 5.6: Delna specifikacija grafa procesa DLC

V nasprotnem primeru (sprejem primitiva L-RELEASE) se na zahtevo po novi povezavi odgovori z ukazom DM (Disconnect Mode). Če se med vzpostavljanjem povezave pred

prejetjem odgovora UA ali DM izteče časovnik T320, se postopek vzpostavljanja povezave ponovi. Število ponovitev določa števec N320, ki ima po privzetem prirejeno vrednost 3. Po treh neuspelih poskusih se s signalom L_ReleaseInd o neuspehu obvesti uporabnika.

Po uspešni vzpostavitvi povezave se lahko prične s prenosom informacijskih okvirjev. Na oddajni stani SUa pošlje primitiv L-DATA s signalom L_DataReq procesu dispatch. Ta ga posreduje primerku procesa DLC, ki je zadolžen za to povezavo. Vsebina se nato skladno s pravili protokola V.76 posreduje sprejemniku z ukazom I. Na sprejemni strani se vrši preverjanje pravilnosti sprejema, vsebina okvirja pa se posreduje uporabniku SUb s signalom L_DataInd.

Prekinitev posamezne povezave lahko zahteva katerakoli stran s signalom L_Release-Req. Proces DLC, ki je na strani izvora zahteve po prekinitvi povezave, zahteva prekinitev od svojega partnerja v komunikaciji z ukazom DISC. Ta sprejem potrdi z ukazom UA in se ubije. O svoji nameri obvesti tudi skrbnika procesov dispatch, ki osveži tabelo preslikav med številko povezave in procesom DLC. Vsi neobdelani primitivi L-Data in pripadajoči informacijski okvirji se zavržejo. Uspešna prekinitev se iniciatorju potrdi s primitivom L-RELEASE (signal L_ReleaseInd). Za podrobnejši opis specifikacije in protokola priporočamo [17] in [45].

5.3 Formalna verifikacija

V okviru formalne verifikacije modela sistema lahko preverjamo različne lastnosti sistema (poglavje 4). Specifikacijo protokola lahko razdelimo na pet delov [28]. Celovita specifikacija mora vsebovati:

1. opis storitve, ki jo nudi protokol,
2. opis predvidevanj o obnašanju okolja, v katerem se bo protokol izvajal,
3. sporočila, ki so uporabljena pri implementaciji protokola,
4. format zapisa sporočil,
5. postopke izmenjave sporočil.

Najtežji del specifikacije in verifikacije predstavlja opis postopkov izmenjave sporočil. To še posebno velja za sisteme s sočasnostjo, kjer se lahko pojavi izredno veliko število možnih izvajanj sistema.

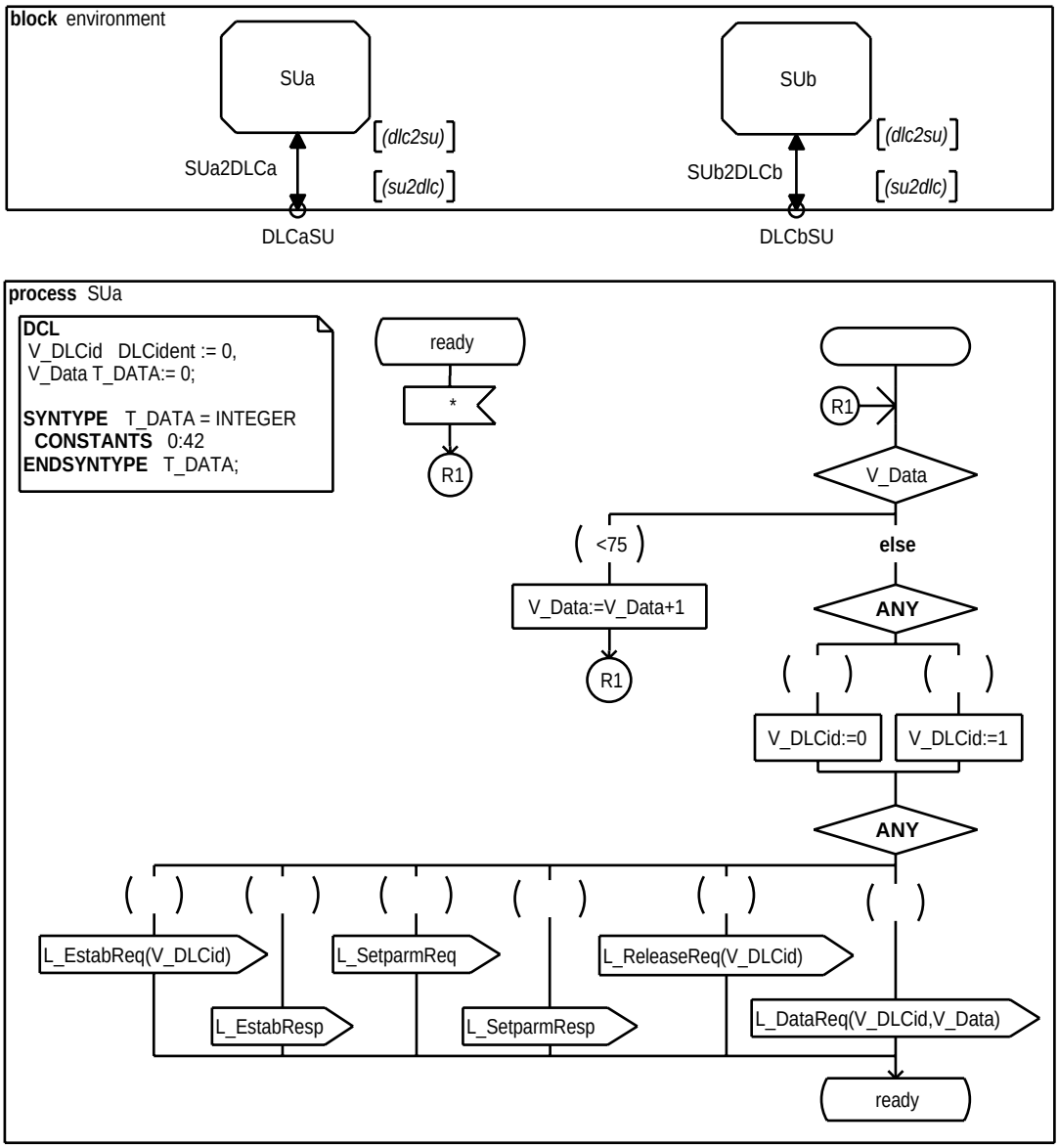
Z orodjem Spin lahko verificiramo samo zaprt sistem. Pred verifikacijo je torej potrebno zagotoviti tudi model okolja. Okolje definira najmanj dva uporabnika storitve (SU — Service User) in prenosni kanal. Uporabniki lahko izmenjajo parametre, vzpostavljajo povezave, prenašajo podatke in prekinjajo povezave. Pri specifikaciji protokola se ni priporočljivo zanašati na posebne omejitve okolja, saj se le-te lahko spremenijo. Temu sledimo tudi pri verifikaciji protokola in v pripravljeno okolje vključujemo samo nujno potrebne omejitve brez dodatnih predvidevanj, ki nimajo formalne osnove. Načrtovanje protokolov, ki vedno delujejo v "normalnih" razmerah, je enostavno. Šele nepričakovani dogodki postavijo protokol na resnično preizkušnjo [28]. Z minimalnim omejevanjem okolja iz verifikacije ne izključimo malo verjetna izvajanja.

Specifikacija protokola V.76 že vključuje model nižjega sloja, ki nudi storitev nezanesljivega prenosa sporočil z ohranjanjem vrstnega reda. Model uporabnikov storitve protokola ni definiran, zato ga moramo pred pričetkom formalne verifikacije pripraviti sami. Uporaba

orodja $sdl2pml$ omogoča dva pristopa — z dopolnitvijo specifikacije sistema pred avtomatsko tvorbo modela ali z dopolnitvijo modela z dodatnim procesom v jeziku Promela. Med avtomatsko tvorbo modela se pripravi vse potrebno za komunikacijo z okoljem.

5.3.1 Model okolja

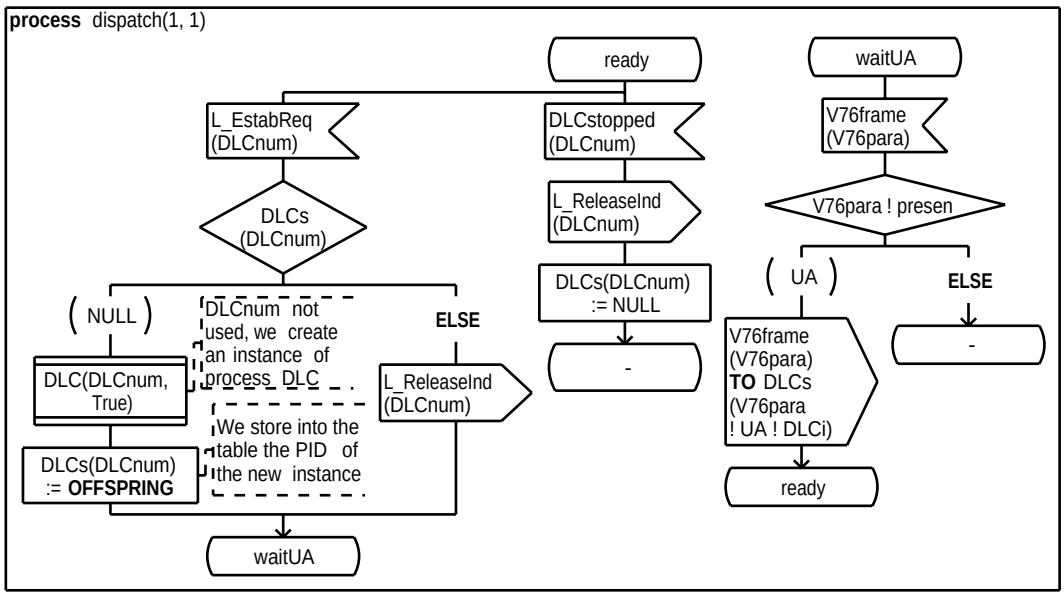
V prvem koraku formalne verifikacije je potrebno pripraviti model okolja. Za prikaz delovanja smo izbrali prvi pristop z opisom okolja v jeziku SDL. Slika 5.7 prikazuje pripravljeno okolje, ki je opisano z dvema enakima procesoma. V prvem koraku okolja nismo omejevali z vrstnim redom pošiljanja signalov. Edino omejitev predstavlja zahteva, da se pred odposlanjem naslednjega signala prejme odgovor od nižjega sloja — protokola V.76. S tem zagotovimo, da okolje ni preveč agresivno in da se ne zapolnijo pridruženi kanali. Število, ki predstavlja prenešeni podatek, se vrši v okviru zanke z oznako R1. Izbira prve ali druge povezave se vrši nedeterministično.



Slika 5.7: Prvotna specifikacija okolja

5.3.2 Zagatna stanja

Formalna verifikacija tvorjenega modela se ustavi v 909. koraku s sporočilom o nepravilnem končnem stanju. Vodena simulacija protiprimera tvori diagram MSC in natančen prikaz vseh korakov, ki so privedli do neželenega stanja. Pregled izvajanja pokaže, da sta se pravilno zaključila samo procesa `DLC`. Procesi `SUa`, `SUb`, `AtoB` in `BtoA` so v stanju `ready`, procesa `dispatch` pa v stanju `waitUA` (slika 5.8).

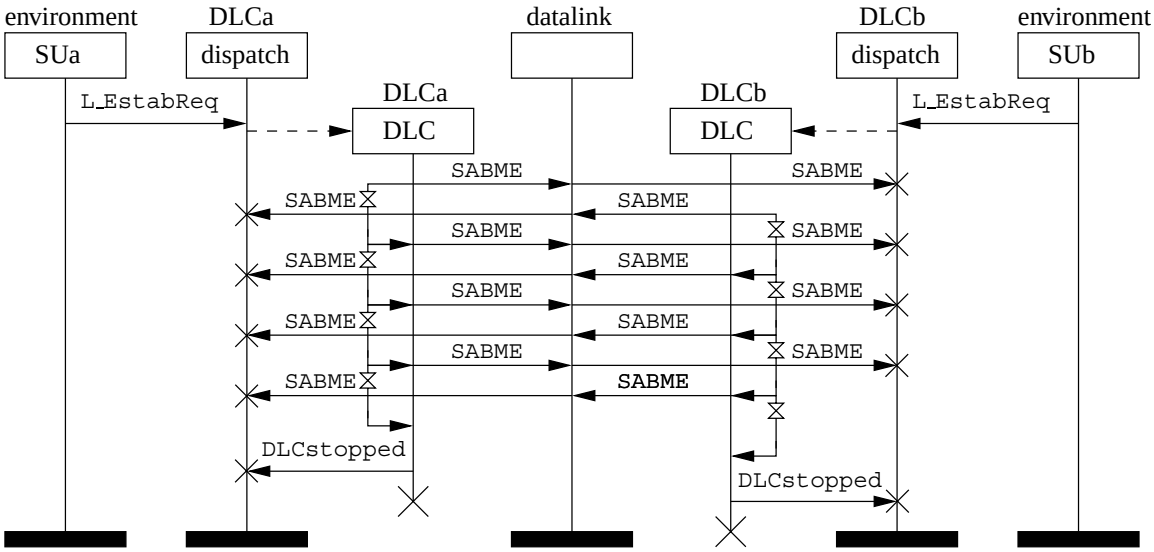


Slika 5.8: Delna specifikacija grafa procesa `dispatch`

V sistemu so predvidene neskončne izvajalne poti, zato je potrebno v vseh procesih označiti veljavna končna stanja — torej stanja, v katerih lahko proces čaka neskončno dolgo na vhod, ki sproži nadaljevanje izvajanja. Ob prvi formalni verifikaciji modela so bili vsi procesi, razen obeh procesov `dispatch`, v veljavnem končnem stanju. Specifikacijo sistema in okolja zato v teh stanjih dopolnimo s komentarjem, ki vsebuje rezervirano besedo `promela_endstate` (razdelek 4.1.2). Pri obeh procesih `dispatch` to oznako dodamo v stanju `ready`.

Ponovna izčrpna formalna verifikacija da isti rezultat, saj sta procesa `dispatch` v neveljavnem končnem stanju. V stanju `waitUA` oba procesa pričakujeta sprejem signala `V76frame` z ukazom `UA`, ki bi potrdil sprejem zahteve po novi povezavi. Protiprimer pokaže izvajalno pot, na kateri v tem stanju ostaneta v nedogled (slika 5.9). To se pripeti v primeru, ko oba uporabnika poskusita vzpostaviti povezavo skoraj sočasno.

Po sprejemu signala `L_EstabReq` procesa `dispatch` tvorita pridružena procesa `DLC`, prehod pa zaključita v stanju `waitUA`. V tem stanju je predviden samo sprejem potrditve z ukazom `UA`. Ostali signali se implicitno zavržejo ali zanemarijo — prehod nazaj v stanje `waitUA`. Po izteku časovnikov se v procesih `DLC` sproži mehanizem ponovne oddaje in po treh neuspelih poskusih oba procesa `DLC` zaključita z delovanjem, medtem ko procesa `dispatch` v nedogled ostaneta v stanju `waitUA`. Zavrže se tudi obvestilo o zaključitvi izvajanja procesov `DLC` s signalom `DLCstopped`. Do enakega zagatnega stanja bi prišlo ob zaporedni izgubi signalov v nižjem sloju. Podrobnejši pregled procesa `dispatch` je na voljo v prilogi C.2.1 in [17].



Slika 5.9: Razširjen MSC diagram prvega protiprimera

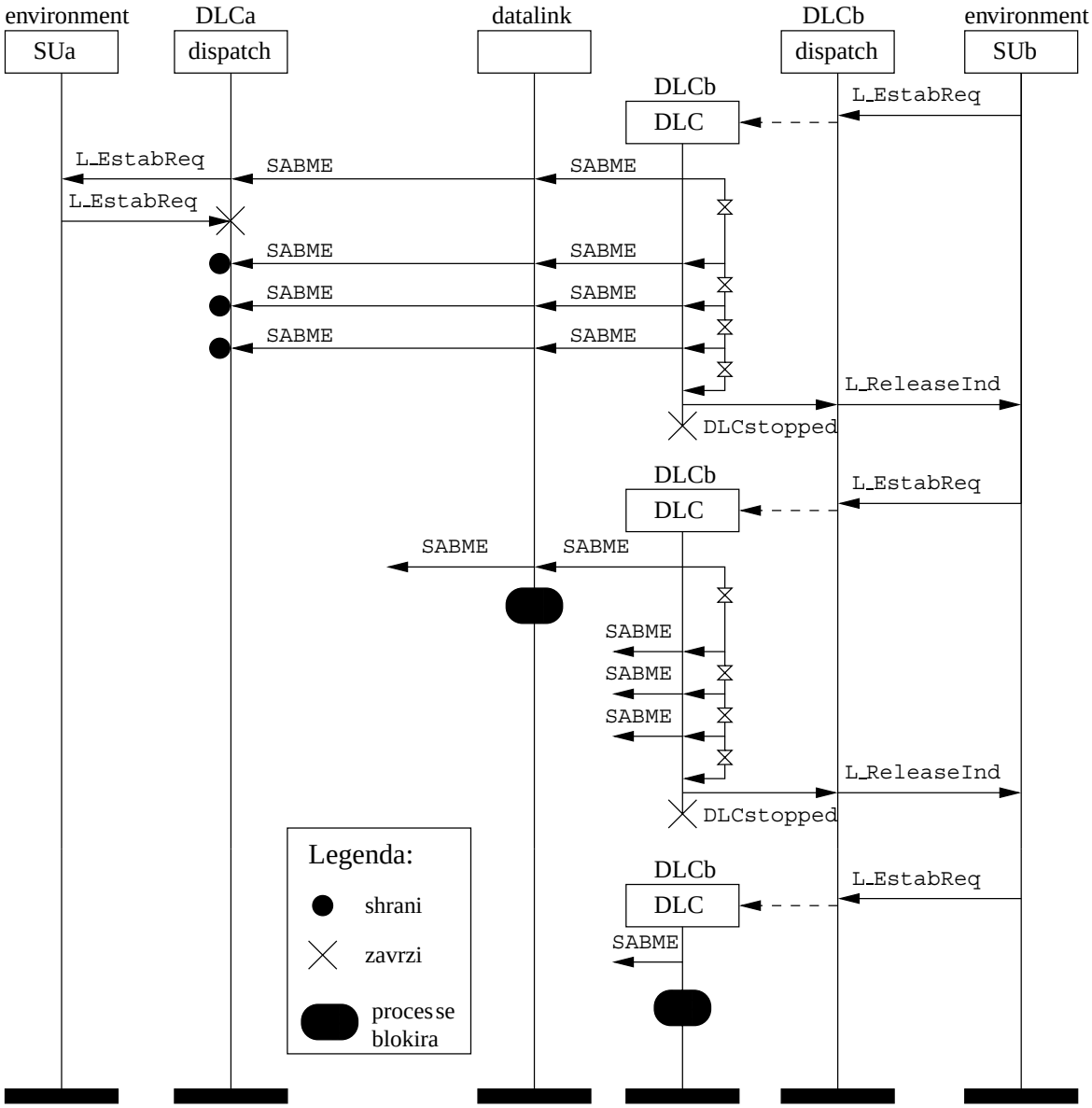
5.3.2.1 Prva dopolnitev sistema

Trenutna specifikacija protokola V.76 ne podpira skoraj sočasne zahteve po vzpostavitvi povezave in izgube signalov v nižjih slojih povezave. To ni skladno s standardom, zato smo specifikacijo dopolnili s sprejemom signala `DLCstopped` v stanju `waitUA` z enakim prehodom, kot je v stanju `ready` na sliki 5.8. Ob večkratni izgubi signala ali sočasnem poskusu vzpostavljanja povezave se obvesti uporabnika o neuspeli vzpostavitvi povezave, proces `dispatch` pa se, po osvežitvi zapisov o aktivnih procesih `DLC`, vrne v stanje pripravljenosti (`ready`).

Formalna verifikacija dopolnjene specifikacije se zaključi z opozorilom o nepravilnem končnem stanju. Tokrat v 2590. koraku. V primeru, ki ga prikazuje slika 5.10, se zapolnijo vse reže pridruženega kanala procesa `dispatch` v bloku `DLCa`. Po privzetem imajo procesi v tvorjenem modelu pridružene kanale s tremi režami, kar je razvidno tudi iz razširjenega diagrama MSC. Proces `dispatch` ob prejemu ukaza `SABME` o tem obvesti uporabnika `SUa` in se postavi v stanje `waitEstabResp` (slika 5.11).

V stanju `waitEstabResp` lahko sprejme samo potrditev uporabnika o sprejemu povezave s signalom `L_EstabResp`. Vsi signali `V76frame` se hranijo za kasnejšo obdelavo. Uporabnik `SUa` torej nima mehanizma, s katerim bi lahko zavrnil povezavo. Ob nepričakovanem odzivu okolja se prejeti signali implicitno zavržejo. V naslednjem koraku se v procesu `DLC` v bloku `DLCb` sproži mehanizem ponovne oddaje, ki zapolni pridružen kanal procesa `dispatch` v bloku `DLCa`. Po treh neuspelih poskusih proces `DLC` zaključi z izvajanjem in o tem obvesti proces `dispatch` v bloku `DLCb`, ki obvestilo o neuspeli vzpostavitvi povezave posreduje uporabniku `SUb` s signalom `L_ReleaseInd`. Ta ponovi zahtevo po vzpostavitvi povezave. Tvori se nov primerek procesa `DLC` in prične se pošiljanje ukaza `SABME`. Že pri prvem signalu se zaradi polnega pridruženega kanala procesa `dispatch` v bloku `DLCa` blokira proces `BtoA` v bloku `datalink`, saj ne more posredovati prejetega signala. Blok `DLC` po postopku ponovne oddaje zaključi z izvajanjem in o tem obvesti uporabnika `SUb`. Ta ponovno poskusi vzpostaviti povezavo. V tretjem poskusu blokira tudi novi proces `DLC`, saj zaradi polnega pridruženega kanala procesa `BtoA` ne more poslati ukaza `SABME`.

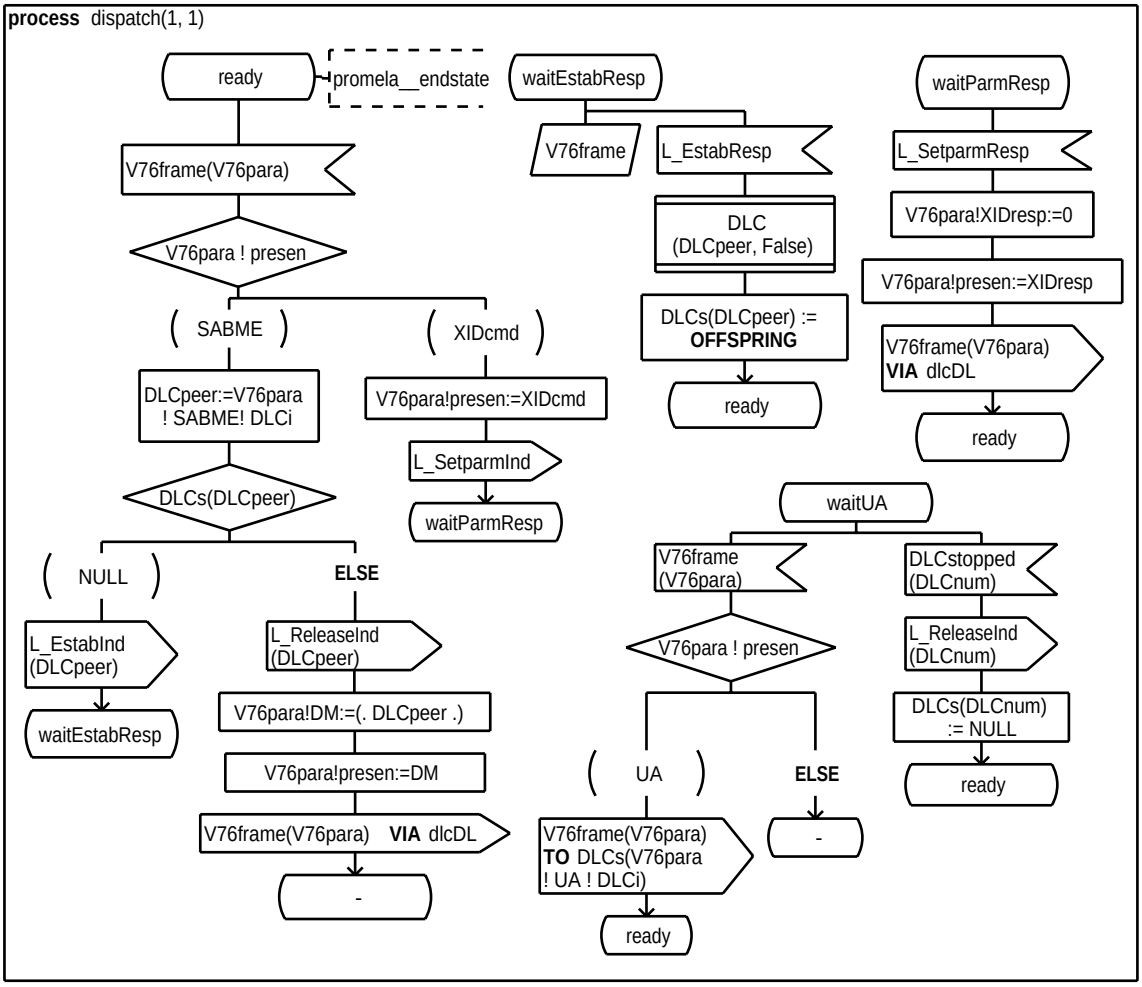
Vsaka implementacija specifikacije ima omejeno število rež pridruženih kanalov.



Slika 5.10: Razširjen MSC diagram drugega protiprimera

Kljub temu, da prikazana izvajalna pot velja za privzeto število rež (3), lahko pride do zagatnega stanja v implementaciji s poljubno velikim številom rež. Do zagatnega stanja ne pride samo tedaj, kadar uporabnik sistema vedno sprejme zahtevo po povezavi — tudi v primeru, ko je sam želel vzpostaviti povezavo, a ni dobil odgovora od sistema.

Prikazana izvajalna pot je bila pridobljena z uporabo privzetega algoritma, ki išče najprej v globino. Tako se je pred zagatnim stanjem izvedlo kar 2590 korakov. Orodje Spin vključuje tudi algoritem za iskanje najprej v širino. Dobra lastnost tega algoritma je priprava protiprimera z najkrajšo izvajalno potjo. Če na istem modelu izvedemo iskanje najprej v širino, dobimo protiprimer s samo 262 koraki. Slika 5.12 prikazuje diagram MSC, ki ga ob vodeni simulaciji izriše orodje *xdtspin*. Prikazana je komunikacija med procesi v modelu sistema z vsebino vseh signalov oz. sporočil. Vsak kanal in signal ima v okviru modela pridruženo unikatno število, ki se v diagramu MSC uporabi namesto imena. Iz slike je razvidno, da ima proces `DLCb_dispatch` pridružen kanal z identifikacijsko številko 4 in da mu je proces, ki modelira uporabnika `SUb`, poslal signal z identifikacijsko številko 5.

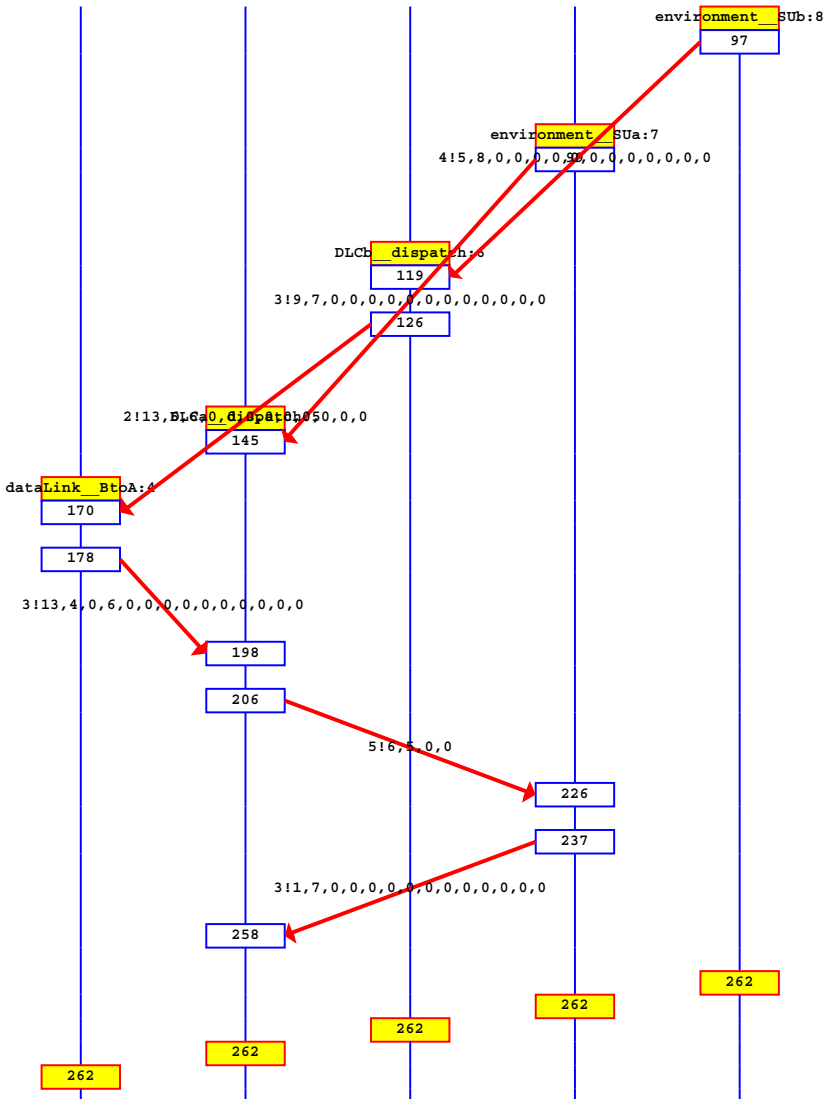


Slika 5.11: Delna specifikacija procesa `dispatch`

Pregled modela (priloga C.2.2.1) pokaže, da je to signal `L_SetparmReq`. V naslednjem trenutku pošlje svojo zahtevo tudi uporabnik `SUa`. Specifikacija procesa `dispatch` v stanju `ready` ne predvideva odziva na signal `L_ReleaseReq` (identifikacijska številka 9), zato proces `environment_SUa` obtiči v veljavnem končnem stanju `ready` in ne pošlje nobenega dodatnega povpraševanja.

Medtem proces `DLCb_dispatch` posreduje zahtevo po izmenjavi parametrov z ukazom `XIDcmd` (slika 5.5). Proces `BtoA` ukaz uspešno posreduje procesu `DLCa_dispatch`. Ta obvesti model uporabnika `SUa` s signalom `L_SetparmInd`. Uporabnik se v tem primeru nepričakovano odzove z zahtevo po vzpostavitvi povezave (signal `L_EstabReq`). Proces `DLCa_dispatch` v stanju `waitParmResp` lahko sprejme samo signal `L_SetparmResp` (slika 5.11), zato se signal implicitno sprejme. Model uporabnika `environment_SUa` ni prejel nobenega odziva, zato ob povpraševanju obtiči v veljavnem končnem stanju `ready` in ne pošlje nobenega dodatnega signala. Preverjani sistem je v zagatnem stanju, saj je proces `DLCa_dispatch` v neveljavnem končnem stanju.

Orodje Spin omogoča iskanje množice protiprimerov in ni omejeno samo na en primer napačne izvajalne poti, kar omogoča nadaljevanje formalne verifikacije s trenutnim modelom. Vsaka napaka v specifikaciji se lahko prikaže z množico napačnih izvajalnih poti. Zato je primerneje, da se pred nadaljevanjem formalne verifikacije odpravi znane pomanjkljivosti specifikacije.



Slika 5.12: MSC diagram protiprimera, ki ga izriše orodje $xdtspin$

5.3.2.2 Druga dopolnitev sistema

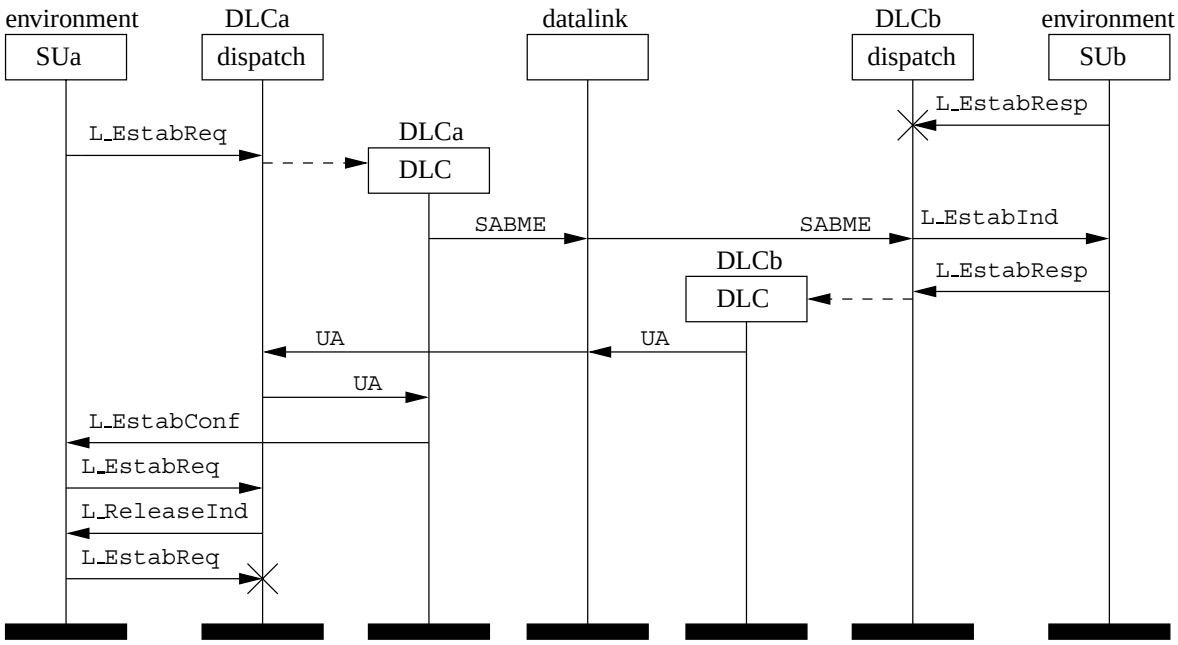
Kljub temu, da sta razloga za zagatno stanje v obeh protiprimerih različna, oba nakazujeta na pomanjkanje mehanizma za vrnitev v veljavna stanja, ki omogočajo nadaljevanje izvajanja sistema. V prvem primeru se od okolja pričakuje, da bo vedno sprejelo ponujeno povezavo. V primeru, da je ne sprejeme oz. da se ne odzove na povpraševanje, pride sistem v takšno stanje, da nadaljnje izvajanje ni več mogoče. V drugem primeru je vzrok zagatnega stanja prav tako nepričakovani odziv okolja. Oba protiprimera veljata tudi za popravljeno specifikacijo sistema v [17]. Preverjanje pravilnosti delovanja v [17] je vključevalo samo podmnožico vseh mogočih izvajalnih poti. Uporabnika sta bila omejena na naslednje signale:

- SUa: L_SetparmReq,
- SUa: L_EstabReq(1),
- SUa: L_EstabReq(0),
- SUa: L_EstabResp,
- SUa: L_DataReq(0, 39),

- SUa: L_DataReq(0, 86),
- SUa: L_ReleaseReq(1),
- SUb: L_SetparmResp,
- SUb: L_EstabResp,
- SUb: L_ReleaseReq(0).

Iz množice signalov je razvidno, da je bila iniciacija izmenjave parametrov in vzpostavitve povezave predvidena samo s strani uporabnika SUa. Uporabnik SUb je omejen na odziv na zahtevo po izmenjavi parametrov, sprejem povezave in zahtevo za prekinitev povezave. Standard V.76 [45] ne podaja omejitev okolja, ki bi opravičile spremembo modela uporabnikov storitve. Kljub temu smo, zaradi primerljivosti z rezultati iz [17], dopolnili procesa `SUa` in `SUb`.

Formalna verifikacija tvorjenega modela se ustavi v 476. koraku. Slika 5.13 prikazuje pridobljeni protiprimer. V prvem koraku uporabnik `SUb` poskusi potrditi neobstoječo zahtevo po vzpostavitvi povezave. Proces `dispatch` v bloku `DLCb` se na to ne odzove in signal `L_EstabResp` implicitno sprejme. Sledi uspešna vzpostavitev povezave na pobudo procesa `SUa`. Uspešna tvorba verige procesov `DLC` se potrdi s signalom `L_EstabConf`. Okolje se odzove s ponovno zahtevo po vzpostavitvi iste povezave — povezave št. 0. Proces `dispatch` se odzove s signalom `L_ReleaseInd`, ki predstavlja zavrnitev zahteve po novi povezavi, in se postavi v stanje `waitUA` (slika 5.11). Težava nastopi v naslednjem koraku, saj v stanju `waitUA` ne more obravnavati ponovnega sprejema signala `L_EstabReq`. Ta se implicitno zavrže. V tem trenutku se izvajanje formalne verifikacije ustavi, saj so procesi `dispatch` v bloku `DLCa` in oba procesa `DLC` v neveljavnih končnih stanjih. Do zagatnega stanja pride, ker se ob zavrnitvi zahteve po novi povezavi proces `dispatch` ne vrne v stanje pripravljenosti.

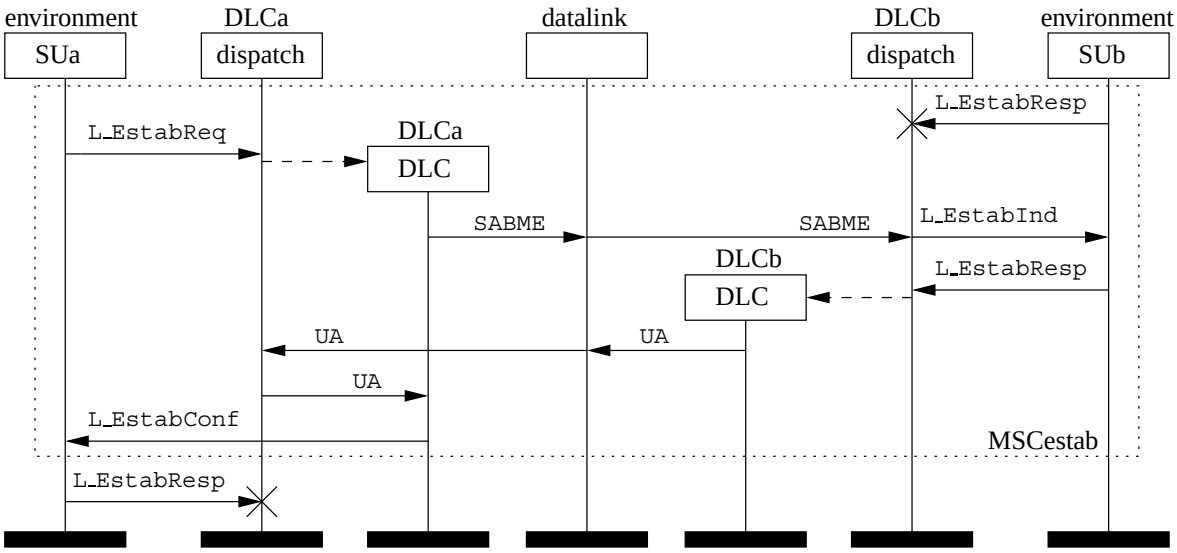


Slika 5.13: Razširjen MSC diagram tretjega protiprimera

5.3.2.3 Tretja dopolnitev sistema

Tretja dopolnitev sistema vključuje popravek procesov `dispatch`. V stanju `ready` se ob sprejemu signala `L_EstabReq` za že obstoječo povezavo pošlje okolju signal `L_ReleaseInd`, proces pa se postavi nazaj v stanje `ready`. Delna specifikacija procesa `dispatch` pred spremembo je prikazana na sliki 5.11. Sprememba zagotavlja odziv sistema na napačno zahtevo po vzpostavitvi povezave in nadaljevanje izvajanja.

Formalna verifikacija predstavi protiprimer s 417 koraki. Slika 5.14 prikazuje izvajalno pot, kjer procesa `DLC` ostaneta v neveljavnem končnem stanju `connected`. To se pripeti v primeru, ko se na obeh straneh povezave implicitno sprejmejo uporabnikove zahteve. Do enakega izvajanja pride tudi v primeru, ko se kljub uspešni vzpostavitvi povezave nikoli ne izvrši prenos podatkov ali regularna prekinitev povezave. Standard V.76 za takšne primere predvideva uporabo neobveznega časovnika T403. Ta predpisuje maksimalno dovoljeno obdobje brez prenosa podatkov v aktivni povezavi. Prikazano izvajanje je veljavno, ker obravnavana specifikacija takšnega časovnika ne vključuje. Zato je potrebno v obeh procesih `DLC` dopolniti stanje `connected` z oznakami veljavnega končnega stanja in ponoviti formalno verifikacijo.

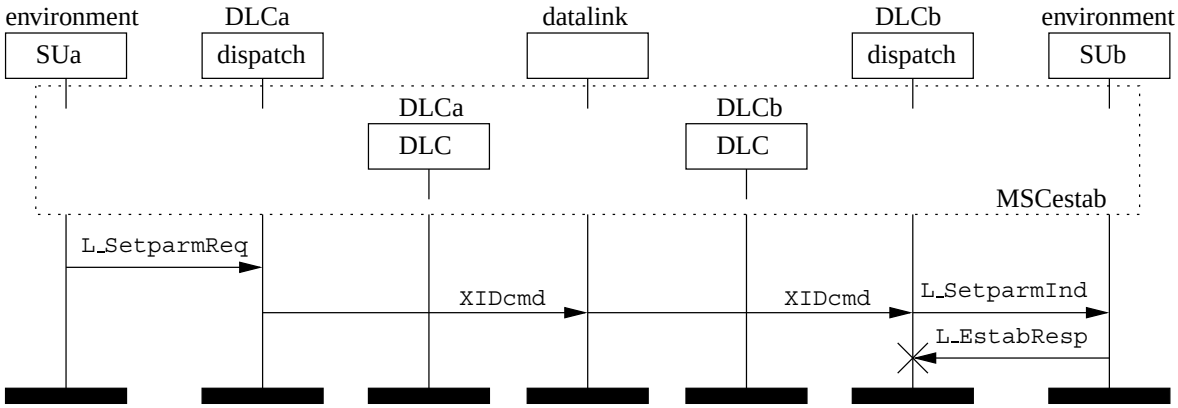


Slika 5.14: Razširjen MSC diagram četrtega protiprimera

Del izvajalne poti, ki opisuje uspešno vzpostavitev povezave med uporabnikoma `SUa` in `SUb`, se na sliki 5.15 sklicuje na del izvajalne poti na sliki 5.14. Prikazano je samo nadaljevanje, ki se v pridobljenem protiprimeru razlikuje.

Do nepravilnega zaključka izvajanja pride, ko po uspešni vzpostavitvi povezave, uporabnik `SUa` poda zahtevo po izmenjavi parametrov. V tem primeru se proces `dispatch` v bloku `DLCb` po posredovanju signala `L_SetparmInd` postavi v stanje `waitParmResp` (slika 5.11). Če se uporabnik `SUb` nikoli ne odzove na podano zahtevo, ostane proces `dispatch` v neveljavnem končnem stanju. Vsi nadaljnji signali iz okolja se implicitno sprejmejo. Popravljena specifikacija v [17] v tem stanju doda samo konstrukt shrani s signalom `V76frame`. Takšna rešitev ni zadovoljiva, saj proces `dispatch` ostane v stanju `waitParmResp` vse do prejema signala `L_SetparmResp`. Če se ta nikoli ne prejme, se prenos na obstoječi povezavi blokira, saj konstrukt shrani zapolni poljubno veliko vhodno vrsto procesa.

Pri pošiljanju zahteve po izmenjavi parametrov standard V.76 predpisuje uporabo



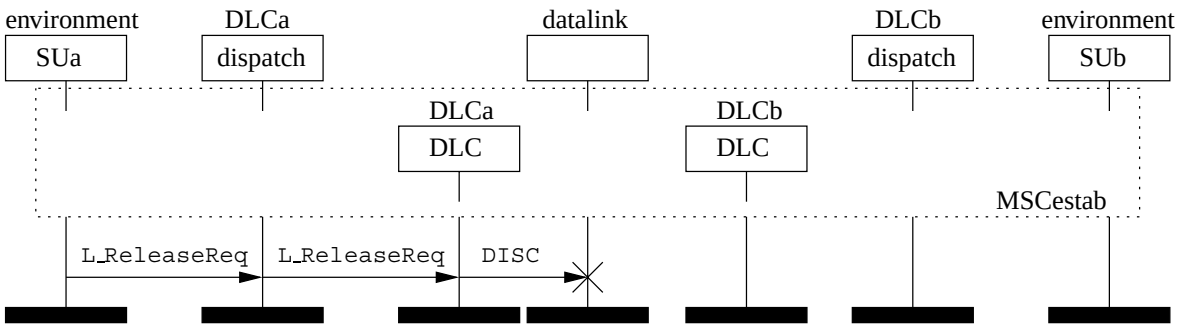
Slika 5.15: Razširjen MSC diagram petega protiprimera

časovnika T401 in števca N400[1] [45]. Če v predvidenem času T401 ne pride do potrditve izmenjave parametrov s signalom `L_SetparmConf`, se po N400 poskusih o neuspehu obvesti iniciatorja zahteve. Poenostavljena specifikacija protokola V.76 tega mehanizma ne vključuje.

5.3.2.4 Četrta dopolnitev sistema

Zaradi neregularnega izvajanja procesa `dispatch` ob zahtevi po izmenjavi parametrov med aktivno povezavo, smo specifikacijo razširili s časovnikom T401. Ob izteku se proces vrne v stanje `ready` in lahko nadaljuje z izvajanjem. V primeru pravočasnega sprejema odgovora se časovnik resetira, odgovor pa se posreduje iniciatorju zahtevka (priloga C.2.2.3). Dopolnjena specifikacija ne vključuje mehanizma ponovne oddaje.

Formalna verifikacija se ustavi v 9393. koraku zaradi nedovoljenega končnega stanja. Zaradi izredno dolge izvajalne poti smo ob ponovnem zagonu algoritma poiskali najkrajšo izvajalno pot z napako. Slika 5.16 prikazuje izvajalno pot s 470 koraki. Takoj po uspešni vzpostavitvi povezave, uporabnik `SUa` zahteva njeno prekinitev s signalom `ReleaseReq`. Proces `dispatch` posreduje zahtevo po prekinitvi povezave z ukazom `DISC` do nižjeležnega bloka `datalink`. Med prenosom se signal izgubi, saj proces `AtoB` ne nudi storitve zanesljivega prenosa (slika 5.4).



Slika 5.16: Razširjen MSC diagram šestega protiprimera

Standard V.76 v tem primeru predpisuje ponovno oddajo ukaza `DISC` po izteku časovnika T401. Po N400 neuspešnih oddajah ukaza mora proces `DLC` zaključiti z izvajanjem in o tem obvestiti nadzorni proces `dispatch`. Obravnavana specifikacija teh postop-

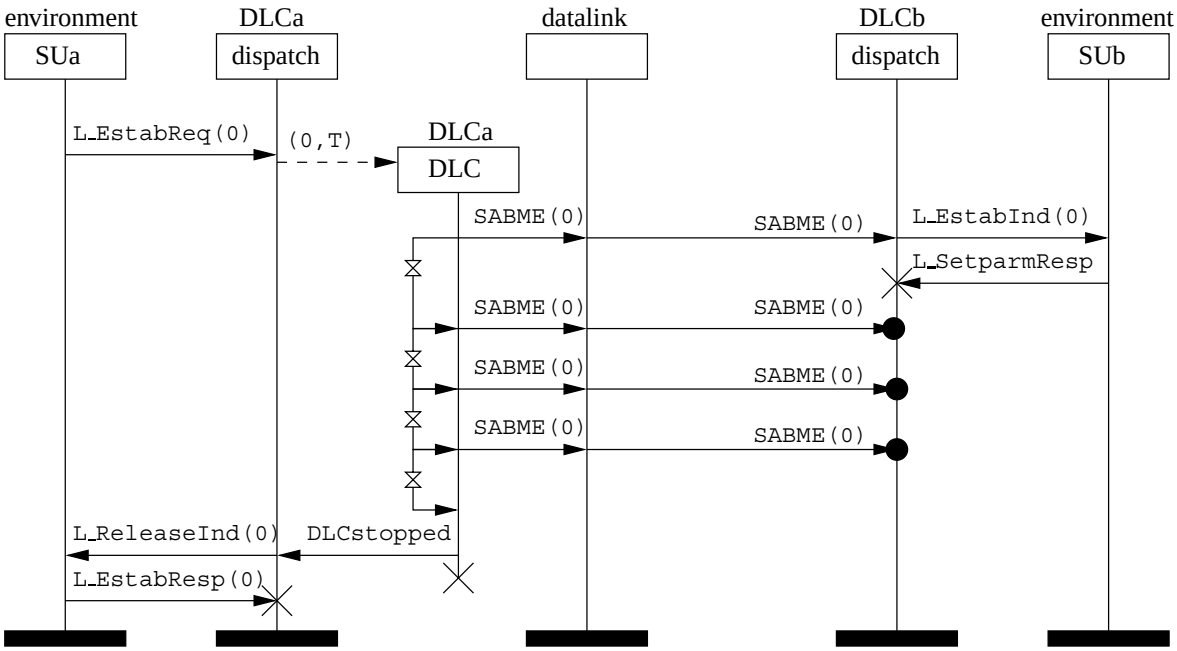
[1] Vrednosti časovnika T401 in števca N400 sta odvisni od implementacije in jih standard ne predpisuje.

kov ne vključuje, zato smo ob formalni verifikaciji našli protiprimer, ki prikazuje mogoče zagatno stanje sistema.

5.3.2.5 Peta dopolnitev sistema

Pri drugi dopolnitvi sistema smo poenotili model okolja s [17]. S tem smo omejili vsa mogoča izvajanja sistema. Tokratni popravek sodi v isto skupino, saj posega v izvajanje sloja podatkovne povezave. Namesto dopolnitve sistema s postopki za primerno obravnavo neuspelih pošiljanj ukazov smo odpravili nedeterministično izbiro v procesih `AtoB` in `BtoA` (slika 5.4). Sloj podatkovne povezave sedaj nudi storitev zanesljivega prenosa[2].

Slika 5.17 prikazuje protiprimer, ki se zaključi po 2000 korakih. Uporabnik `SUa` prične z vzpostavljanjem povezave številka 0. Specifikacija predvideva uporabo dveh povezav z enoumnima številkama — 0 in 1. Izbiro vrši okolje, torej uporabnika `SUa` in `SUb`. Proces `dispatch` v bloku `DLCa` tvori proces za nadzor nad povezavo. Prvi dejanski parameter ob tvorbi procesa `DLC` je številka povezave. Proces bo iniciator povezave, zato se drugi formalni parameter postavi na boolovo vrednost 1[3].



Slika 5.17: Razširjen MSC diagram sedmega protiprimera

Proces `DLC` prične z vzpostavljanjem povezave v svojem prvem prehodu z ukazom `SABME`. Proces `dispatch` v bloku `DLCb` posreduje zahtevo po vzpostavitvi povezave številka 0 s signalom `L_EstabInd`. V tem trenutku proces pričakuje pozitivni odgovor od uporabnika `SUb` (slika 5.11). V stanju `waitEstabResp` se lahko sprejme samo signal `L_EstabResp` in shrani signal `V76frame`. Vsi ostali signali se sprejmejo implicitno. Tako se obravnava tudi signal `L_SetparmResp`, s katerim uporabnik odgovori na zahtevo po izmenjavi parametrov.

V procesu `DLC` se po izteku časovnika sproži mehanizem ponovne oddaje. Ukazi `SABME` se prenašajo s signalom `V76frame`. Zato jih proces `dispatch` v bloku `DLCb` hrani v pridruženi vhodni vrsti za kasnejšo obravnavo. Po treh ponovitvah proces `DLC`

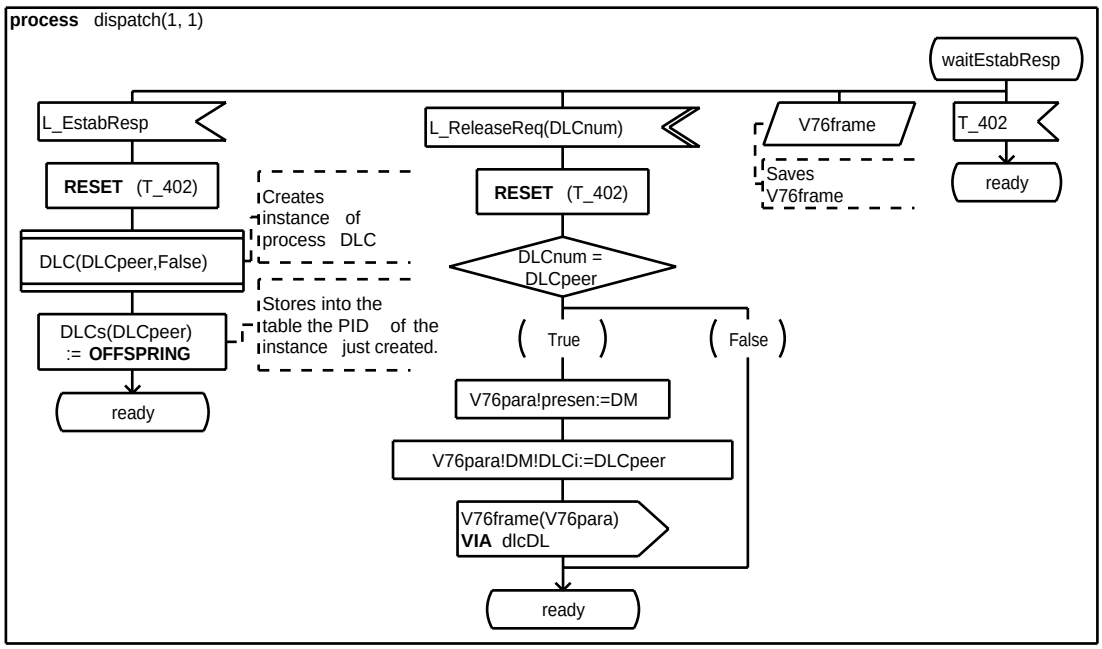
[2]Ta omejitev okolja se uporabi tudi v [17] v kasnejši fazi preverjanja pravilnosti z uporabo filtrov.
[3]Na sliki 5.17 je boolova vrednost 1 označena s črko `T`.

zaključi z izvajanjem in o tem obvesti proces `dispatch` v bloku `DLCa` s signalom `DLCstopped`. Sledi obvestilo o prekinitvi povezave s signalom `L_ReleaseInd`. Izvajalna pot se zaključi s potrditvijo neobstoječe povezave, ki jo proces `dispatch` implicitno sprejme.

Proces `dispatch` v bloku `DLCb` ni uspešno zaključil izvajanja, saj je v nedovoljenem končnem stanju. To se pripeti v primeru, ko uporabnik `SUb` zavrne zahtevo po povezavi ali nanjo nikoli ne odgovori.

5.3.2.6 Šesta dopolnitev sistema

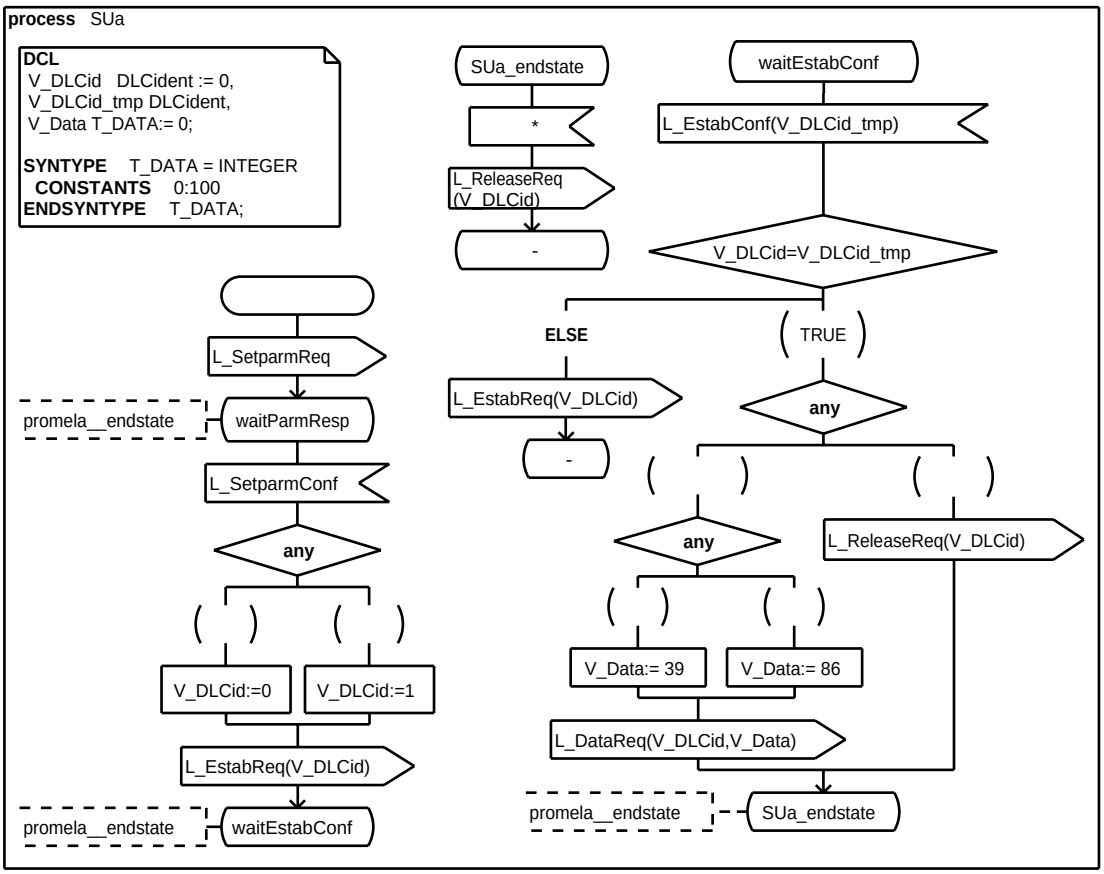
Sedmi protiprimer na sliki 5.17 je razkril, da zavrnitev zahteve po povezavi vodi do zagatnega stanja. Zato smo specifikacijo procesa `dispatch` dopolnili v stanju `waitEstabResp` s prioritetnim sprejemom signala `L_ReleaseReq`. V tem prehodu proces posreduje zahtevo procesu `DLC` z ukazom `DM` in se vrne v stanje `ready`. Če pride zahteva po prekinitvi povezave z drugo številko, se zahteva prezre, proces `dispatch` pa se vrne v stanje pripravljenosti. Protiprimer je dodatno pokazal, da je sistem, če se uporabnik ne odzove na zahtevo po povezavi, v zagatnem stanju. Standard V.76 predvideva nadzor neaktivnosti povezave z uporabo časovnika T402. Po izteku časovnika se proces `dispatch` postavi v stanje pripravljenosti tudi v primeru izvajalne poti, ki jo prikazuje protiprimer na sliki 5.17. Slika 5.18 prikazuje dopolnjeno specifikacijo procesa `dispatch` v stanju `waitEstabResp`.



Slika 5.18: Dopolnjena specifikacija procesa `dispatch`

V izogib težavam pri formalni verifikaciji zaradi omejitev, ki so opisane v razdelku 3.4.12, smo poenostavili model uporabnika SUa. Proces `SUa` prikazuje slika 5.19. Obnašanje iniciatorja povezave smo omejili na omejeno zaporedje signalov. Najprej se vedno izvede izmenjava parametrov. Sledi zahteva za novo povezavo z naključno izbrano številko povezave. Po prejemu potrditve o uspešni vzpostavitvi povezave se pošlje ali podatek ali zahteva po prekinitvi povezave. Prenašani podatek se izbere naključno. V veljavnem končnem stanju `SUa_endstate`, se uporabnik na vsako povpraševanje odzove s signalom

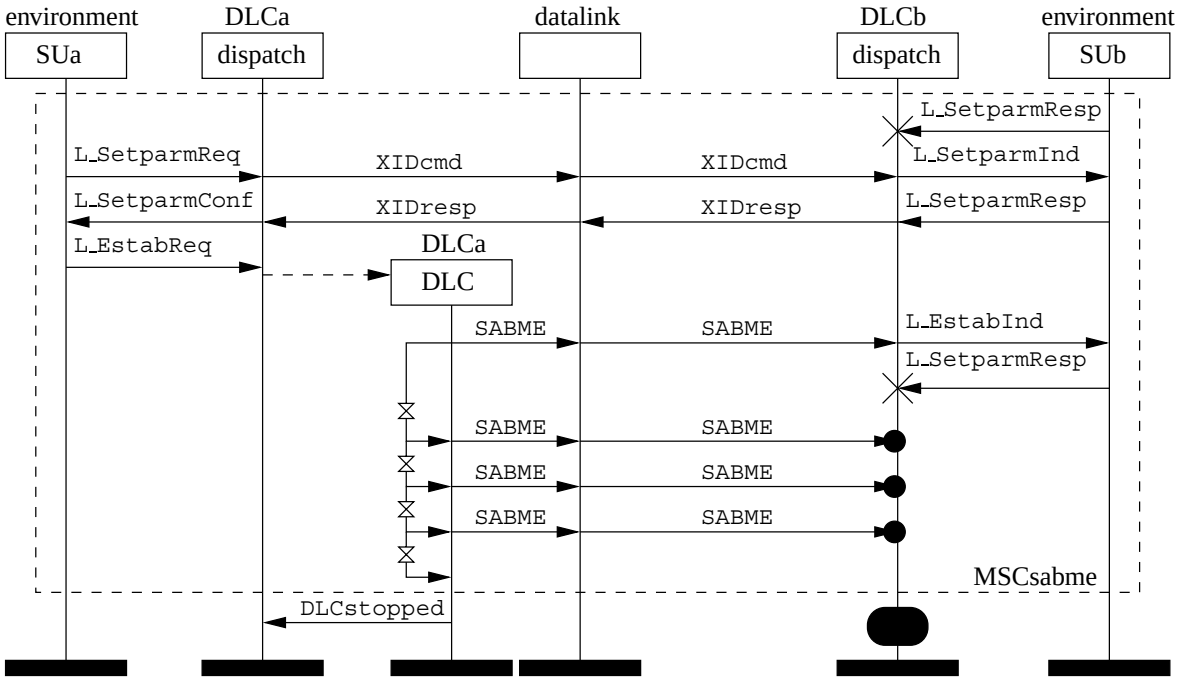
L_ReleaseReq. Avtomatsko tvorjen model procesa je podan v prilogi C.2.2.2.



Slika 5.19: Spremenjena specifikacija procesa SUa

Slika 5.20 prikazuje izvajalno pot, ki v trenutnem sistemu vodi do zagatnega stanja. V prvem delu razširjenega diagrama MSC se izvede uspešen prenos parametrov povezave. Sledi poskus vzpostavitve nove povezave. Proces DLC v svojem prvem prehodu pošlje ukaz SUBME do procesa dispatch v bloku DLCb. Ta obvesti uporabnika SUb s signalom L_EstabInd. Če se uporabnik ne odzove na zahtevo oz. pošlje signal L_SetparmResp, ki se implicitno sprejme, poteče časovnik T320 in proces DLC prične s postopkom ponovne oddaje. Proces dispatch je v stanju waitEstabResp, zato se ponovne oddaje ukaza SUBME shranijo za kasnejšo obravnavo. Ker so vse reže zapolnjene s signali, ki jih proces v trenutnem stanju ne more sprejeti, mehanizem zaznave novih signalov v pridruženem kanalu blokira izvajanje procesa. Privzeto število rež v avtomatsko tvorjenem modelu sistema je tri, kar je slučajno enako številu ponovitev oddaje ukaza SUBME. Privzeto število rež se lahko določi ob tvorbi modela. Do zagatnega stanja pride, ker proces dispatch ne more sprejeti signala, ki označuje potek časovnika T402. Če spremenimo število rež pridruženih kanalov na štiri, omogočimo potek časovnika in nadaljevanje izvajanja sistema.

Formalna verifikacija se sedaj ustavi zaradi uporabe prevelikega indeksa v tabeli table_pid_channame. Velikost tabele je omejena na število predvidenih procesov. Protiprimer, ki ni prikazan na sliki, pokaže, da se ob sprejemu shranjenih ukazov SUBME uporabniku SUb ponovno posreduje zahteva po vzpostavitvi povezave. Če se uporabnik odloči sprejeti povezavo in se odzove s signalom L_EstabResp, proces dispatch tvori nov primerek procesa DLC. Skladno s specifikacijo ponorni proces DLC ob uspešni tvorbi v svojem



Slika 5.20: MSC diagram osmega protiprimera

prvem prehodu pošlje pozitiven odgovor na zahtevo po vzpostavitvi nove povezave z ukazom `UA`. Ko proces `dispatch` v bloku `DLCa` poskusi prejeti ukaz posredovati do procesa `DLC`, ki je v bloku `DLCa` zadolžen za to povezavo, pride do prekoračitve velikosti indeksa tabele `table_pid_channame`. Zaradi treh neuspešnih poskusov oddaje signala `SUBME`, je iskani proces zaključil z izvajanjem (slika 5.20). Zato polje `DLCs` za obravnavano povezavo nima veljavnega zapisa (slika 5.11). Namesto PId-a od pridruženega procesa `DLC`, je na tem mestu zapisana vrednost 255, ki je sinonim za nedefinirano vrednost spremenljivk tipa PId (razdelek 4.2.2). To predstavlja kršitev semantičnih pravil jezika SDL, zato sproži prekinitev formalne verifikacije.

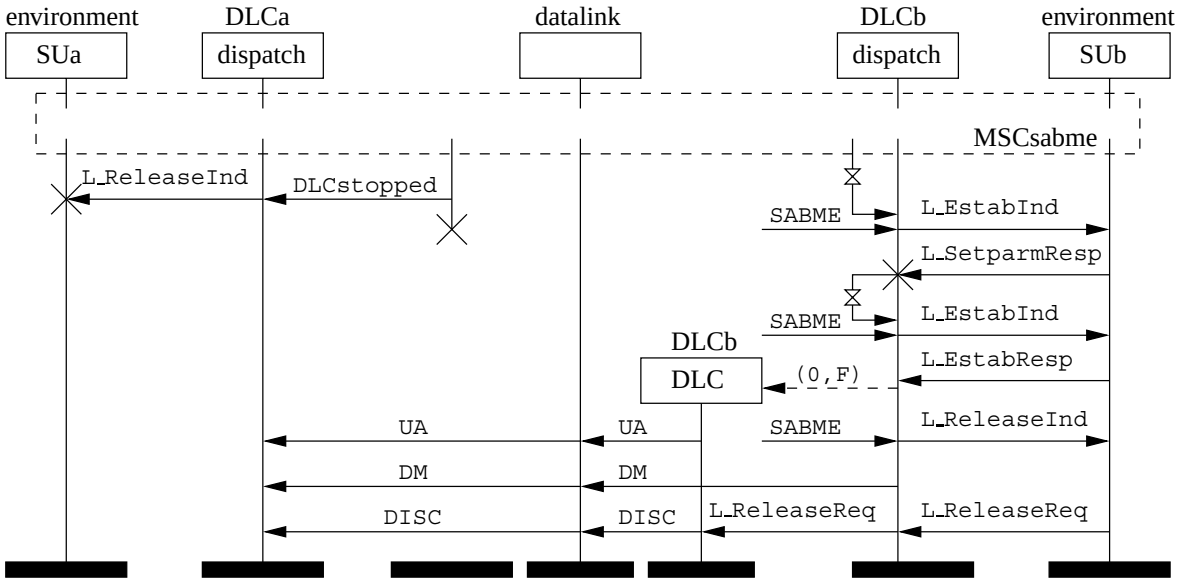
5.3.2.7 Sedma dopolnitev sistema

V specifikaciji sistema smo dopolnili vse prehode, ki vsebujejo konstrukt pošlji, s signalom `V76frame`. Pred odposlanjem signala se preveri veljavnosti naslovnika. Če naslovnik ni veljaven, torej vsebuje vrednost NULL[4], se pošiljanje signala ne izvede, proces pa ostane v istem stanju.

Slika 5.21 prikazuje protiprimer s 1988 koraki. V tem primeru ostane v neveljavnem končnem stanju proces `DLC`. Po poteku časovnika T402 se proces `dispatch` v bloku `DLCb` vrne v stanje `ready`. Sledi obravnava shranjenih signalov `V76frame`.

Proces `dispatch` ukaz `SABME` posreduje z zahtevo po vzpostavitvi nove povezave. Uporabnik `SUb` se na prvo zahtevo odzove s signalom `L_SetparmResp`, ki se implicitno sprejme. Proces `dispatch` se po ponovnem poteku časovnika T402 vrne v stanje pripravljenosti, kjer obravnava naslednji ukaz `SABME`. Uporabnik `SUb` tokrat potrdi zahtevo po novi povezavi s signalom `L_EstabResp`. Tvorjeni proces `DLC` v svojem prvem prehodu pošlje potrditev o uspešni povezavi z ukazom `UA`. Proces `dispatch` v bloku `DLCb` medtem obravnava še zadnji shranjen ukaz `SUBME`. Zaradi zahteve po vzpostavitvi obstoječe

[4]V opisu specifikacije sistema z jezikom SDL. V modelu sistema se vrednost primerja s konstanto `pid_undefined_value`, ki ima vrednost 255.



Slika 5.21: Razširjen MSC diagram devetega protiprimera

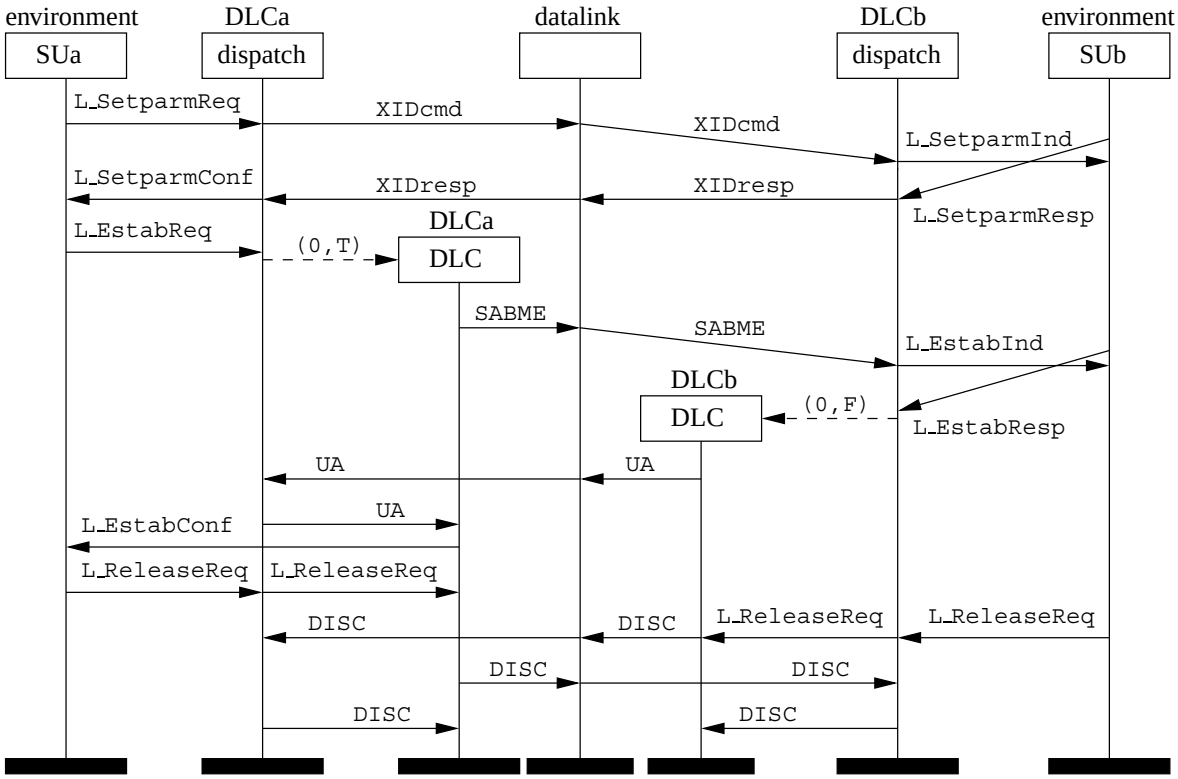
povezave sproži postopek prekinitve povezave s signalom `L_ReleaseInd` in ukazom `DM`. Uporabnik `SUb` se odzove s potrditvijo prekinitve, kar proces `dispatch` posreduje procesu `DLC`. Ta nadaljuje postopek z ukazom `DISC`, ki ga pošlje svojemu neobstoječemu partnerju v bloku `DLCa`. Preverjanje veljavnosti naslovnika v procesu `dispatch` v bloku `DLCa` prepreči posredovanje signala. Proces `DLC` ostane v neveljavnem končnem stanju `waitUAdisc`.

5.3.2.8 Osma dopolnitev sistema

Za vse signale, katerih sprejem je potrebno potrditi, standard V.76 predvideva uporabo časovnika T401. Če se časovnik izteče, se element protokola vrne v prvotno stanje. Proces `DLC` smo razširili s postavitvijo časovnika ob odposlanju ukaza `DISC`. V kolikor časovnik poteče pred sprejemom odgovora, proces zaključi z delovanjem in o tem obvesti nadzorni proces s signalom `DLCstopped` (slika 5.23).

MSC diagram na sliki 5.22 prikazuje izvajalno pot, na kateri proces `DLC` v stanju `waitUAdisc` ne obravnava ukaza `DISC`. Uporabnik `SUa` prične komunikacijo z zahtevo po izmenjavi parametrov. Proces `dispatch` z ukazom `XIDcmd` posreduje zahtevo do svojega partnerja v komunikaciji v bloku `DLCb`. Uporabnik `SUb` v svojem prvem prehodu pošlje signal `L_SetparmResp` že pred prejetjem signala `L_SetparmInd`. Proces `dispatch` zaradi predstavljega zaporedja signalov obravnava signal `L_SetparmResp` kot odgovor na na zahtevo po izmenjavi parametrov.

Uporabnik `SUa` prične z vzpostavljanjem nove povezave takoj po prejetju signala `L_SetparmConf`. Proces `dispatch` v bloku `DLCb` posreduje zahtevo uporabniku `SUb` s signalom `L_EstabInd`. Uporabnik `SUb` malo pred tem pošlje signal `L_EstabResp`. Kljub temu, da signal ni neposredni odgovor na zahtevo po vzpostavitvi nove povezave, proces `dispatch` tvori nov primerek procesa `DLC`. Ta z ukazom `UA` potrdi zahtevo po novi povezavi. V nadaljevanju oba uporabnika skoraj sočasno pošljeta zahtevo za prekinitev povezave. Procesa `DLC` se po sprejemu signala `L_ReleaseReq` postavita v stanje `waitUAdisc`, kjer lahko sprejmeta samo ukaza `UA` in `DM`. Prejem ukaza `DISC` povzroči vrnitev v neveljavno končno stanje `waitUAdisc` (slika 5.6).



Slika 5.22: Razširjen MSC diagram desetega protiprimera

5.3.2.9 Deveta dopolnitev sistema

Slika 5.23 prikazuje dopolnitev specifikacije procesa `DLC` z obravnavo ukaza `DISC` tudi v stanju `waitUAdisc`. Ponovitev formalne verifikacije ne najde neveljavnih končnih stanj (slika 5.24).

5.3.3 Kršitve semantičnih pravil jezika SDL

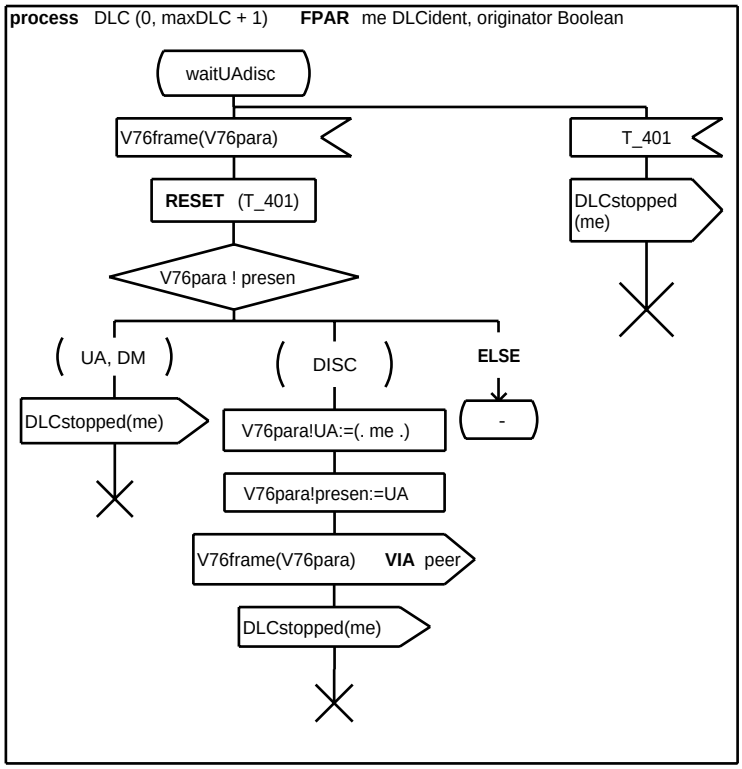
Preverjanje kršitev semantičnih pravil jezika opisuje razdelek 4.2.2. Trajne sonde med formalno verifikacijo ne pokažejo nobene dodatne kršitve. Specifikacija tudi ne vsebuje napačne uporabe konstrukta odločitev.

Delovanje sond za preverjanje zalog vrednosti pokaže formalna verifikacija modela, ki ga opisuje razdelek 5.3.2.5. Protiprimer razkrije, da je pred spremembo modela okolja prav vsaka izvajalna pot kršila specifikacijo sistema. V okviru zanke R1 na sliki 5.7 se v 149. koraku v modelu sistema spremenljivki `V_Data` priredi vrednost 43. Skladno z definicijo podintervalnega tipa `T_DATA` vrednost ne sme preseči 42.

Množica kršitev se dobi ob formalni verifikaciji uporabe spremenljivk, ki jim še ni bila prirejena vrednost. Do tega pride zaradi modeliranja ASN.1 podatkovnega tipa “choice” z zapisom. Pred odposlanjem signala se vrednost priredi samo nekaterim elementom zapisa. Ostali ostanejo nedefinirani, kar je skladno s semantiko jezika SDL. Takšni uporabi konstrukta izhod se je med specifikacijo sistema potrebno izogibati.

5.3.4 Iskanje potencialno nepravilnih izvajalnih poti

Vključitev sond za iskanje potencialno nepravilnih izvajalnih poti omogoča zaznavo implicitnih sprejemov signalov. Implicitni sprejem ne vpliva na izvajanje ciljnega procesa. Pisec



Slika 5.23: Dopolnjena specifikacija procesa `DLC` s sprejemom ukaza `DISC`

```
(Spin Version 4.1.1 -- 2 January 2004
+ Discrete-Time Extension Version 0.1.2 (Multiple Tick) -- 24 December 2002)
        + Partial Order Reduction
        + Compression

Full statespace search for:
        never claim             - (not selected)
        assertion violations    +
        cycle checks            - (disabled by -DSAFETY)
        invalid end states      +

State-vector 2276 byte, depth reached 1932, errors: 0
1.74157e+07 states, stored
3.90672e+07 states, matched
5.64829e+07 transitions (= stored+matched)
1.13318e+08 atomic steps
hash conflicts: 6.49636e+08 (resolved)
(max size 2^19 states)

Stats on memory usage (in Megabytes):
39916.876       equivalent memory usage for states (stored*(State-vector + overhead))
708.182 actual memory usage for states (compression: 1.77%)
        State-vector as stored = 25 byte + 16 byte overhead
2.097   memory used for hash table (-w19)
0.080   memory used for DFS stack (-m2500)
711.115 total actual memory usage

nr of templates: [ globals chans procs ]
collapse counts: [ 16409 2 2 96 77 399 638 2691 194 92 69 1 ]
```

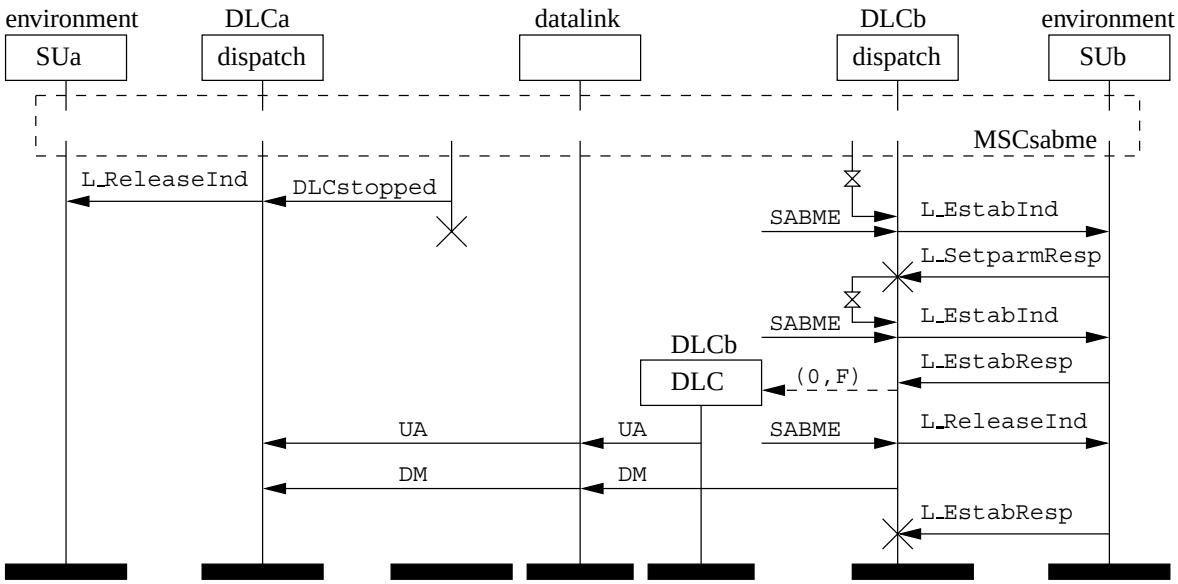
Slika 5.24: Formalna verifikacija modela brez neveljavnih končnih stanj

Tabela 5.1: Primeri implicitnih sprejemov signalov

blok	proces	stanje	signal
environment	SUa	waitEstabConf	L_ReleaseReq
DLCb	dispatch	ready	L_SetparmResp
DLCb	dispatch	ready	L_EstabResp
DLCb	dispatch	waitParmResp	L_EstabResp
DLCb	dispatch	waitParmResp	L_ReleaseReq
DLCb	dispatch	waitEstabResp	L_SetparmResp
DLCb	DLC	waitUAdisc	L_ReleaseReq

specifikacije lahko takšen sprejem uporabi namerno, v praktičnih specifikacijah pa implicitni sprejem pogosto nakazuje premalo podroben opis sistema.

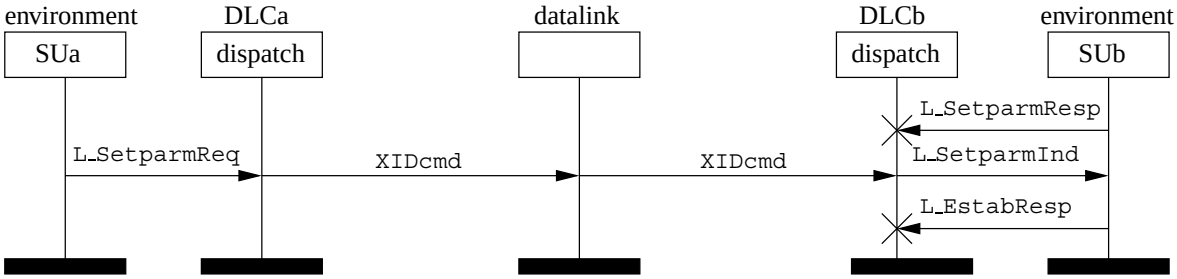
Zaradi morebitne namerne uporabe implicitnega sprejema se lahko pri avtomatski tvorbi sprejem izbranega signala izključi iz obravnave. Določi se proces in eksplicitno stanje, v katerem je implicitni sprejem dovoljen. Takšen primer je sprejem signalov L_SetparmResp in L_EstabResp v procesu dispatch, ko je ta v stanju ready (slika 5.25). S trenutnim modelom okolja to velja samo za proces v bloku DLCb. Oba signala predstavljata odgovor na povpraševanje, zato njun implicitni sprejem ne vpliva na obnašanje sistema. Sonde za njuno zaznavo se izvzamejo iz nadaljnje verifikacije modela. Tabela 5.1 prikazuje vse mogoče implicitne sprejeme v trenutnem modelu sistema.



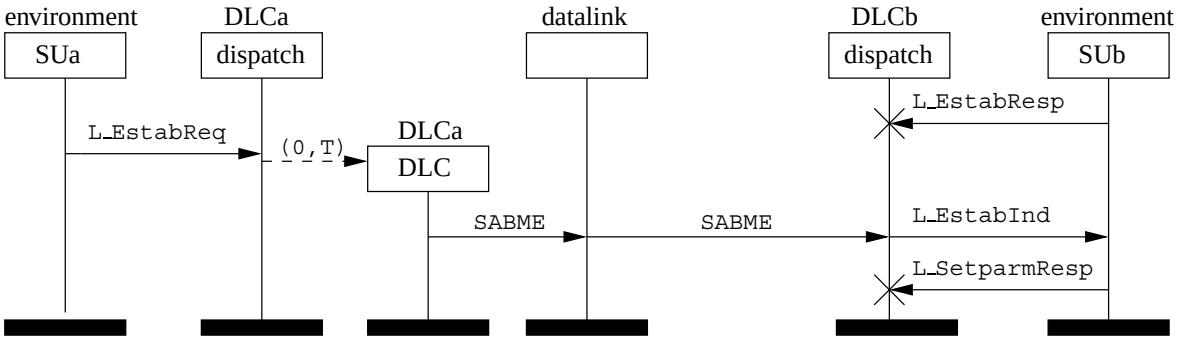
Slika 5.25: Razširjen MSC diagram implicitnega sprejema signalov L_SetparmResp in L_EstabResp pri procesu dispatch v stanju ready

Proces dispatch lahko v stanju waitParmResp implicitno sprejme signala L_EstabResp in L_ReleaseReq. Uporabnik SUb se na katerokoli povpraševanje odzove z nedeterministično izbiro med signali L_SetparmResp, L_EstabResp in L_ReleaseReq. Slika 5.26 prikazuje izvajalno pot, na kateri se na zahtevo po izmenjavi parametrov povezave odzove s signalom L_EstabResp. Skladno z modelom uporabnika je veljaven tudi odziv s signalom L_ReleaseReq. Oba signala se v stanju waitParmResp

implicitno sprejmeta, saj ne predstavljata veljavnega odziva okolja. Slika 5.27 prikazuje podoben primer za sprejem signala L_SetparmResp v stanju waitEstabResp. Implicitni sprejem signala L_SetparmResp v zadnjih dveh primerih ne sproži prekinitve formalne verifikacije, saj smo ob tvorbi modela pridruženo sondo eksplicitno izključili.



Slika 5.26: Razširjen MSC diagram implicitnega sprejema signala L_EstabResp pri procesu dispatch v stanju waitParmResp



Slika 5.27: Razširjen MSC diagram implicitnega sprejema signala L_SetparResp pri procesu dispatch v stanju waitEstabResp

Deveti protiprimer na sliki 5.21 prikazuje izvajalno pot, na kateri model okolja SUa implicitno sprejme signal L_ReleaseInd. Implicitni sprejem signala L_ReleaseReq v procesu DLC predstavlja malo bolj zapletena izvajalna pot na sliki 5.28.

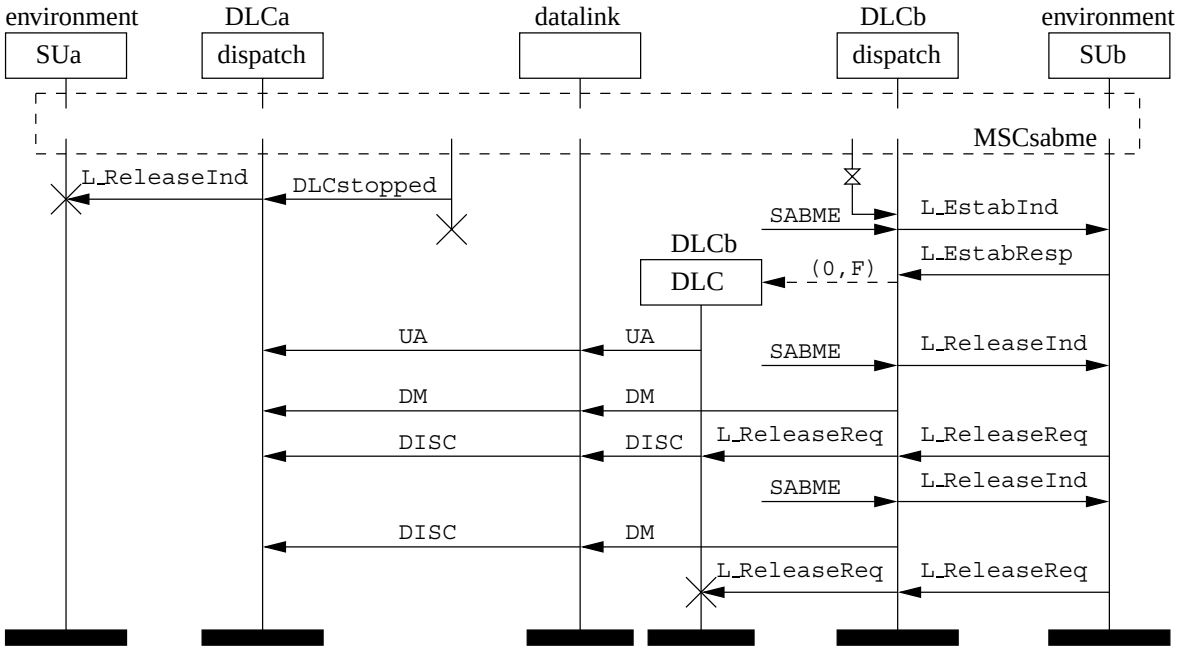
Nepravilne izvajalne poti lahko pisec specifikacije izbere tudi sam. Takšen primer so odločitve, kjer se veja z odgovorom "else" izvede samo v neželenih izvajanjih. Če vsem takšnim prehodom dodamo sondo, se formalna verifikacija prekine v 1932. koraku. Slika 5.29 prikazuje protiprimer, na katerem se v procesu dispatch ne obravnava sprejem ukaza DM (slika 5.8). Specifikacijo procesa smo dopolnili tako, kot to prikazuje slika 5.30.

Formala verifikacija popravljene specifikacije pokaže podobno kršitev v procesu DLC. V stanju waitUAdisc se ne obravnava ukaza I. Ob čakanju na potrditev prekinitve povezave se završejo vsi podatki, ki jih prenaša informacijski okvir.

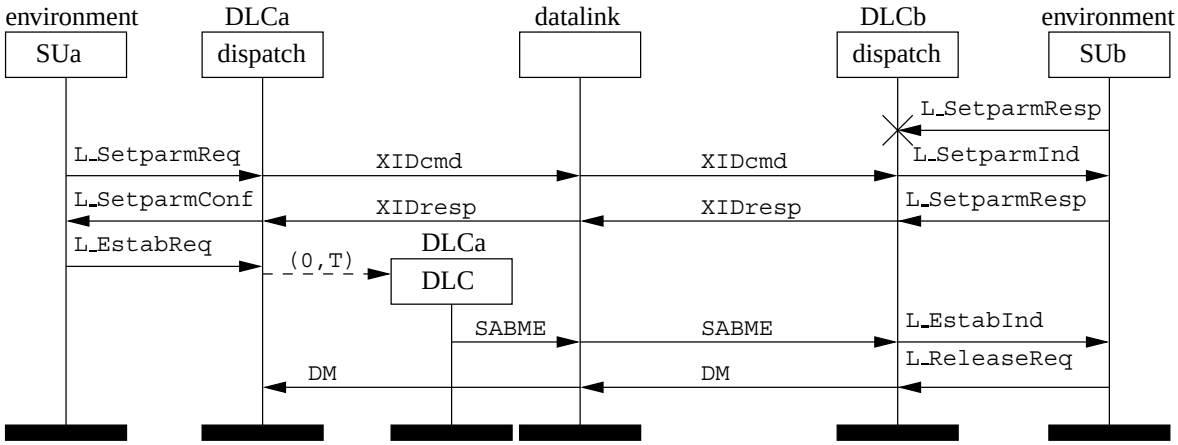
Primer je pokazal, da se lahko s preverjanjem potencialno nepravilnih izvajalnih poti odkrijejo izvajalne poti, ki jih pisec specifikacije ni predvidel.

5.3.5 Iskanje cikličnih izvajalnih poti

V modelu uporabnika SUa smo predvideno končno stanje označili s komentarjem promela_progress. S tem smo podali zahtevo, da mora proces SUa na vsaki neskončni izvajalni poti neskončnokrat priti v stanje SUa__endstate. Slika 5.19 prikazuje



Slika 5.28: Razširjen MSC diagram implicitnega sprejema signala `L_ReleaseReq` pri procesu `DLC` v stanju `waitUAdisc`



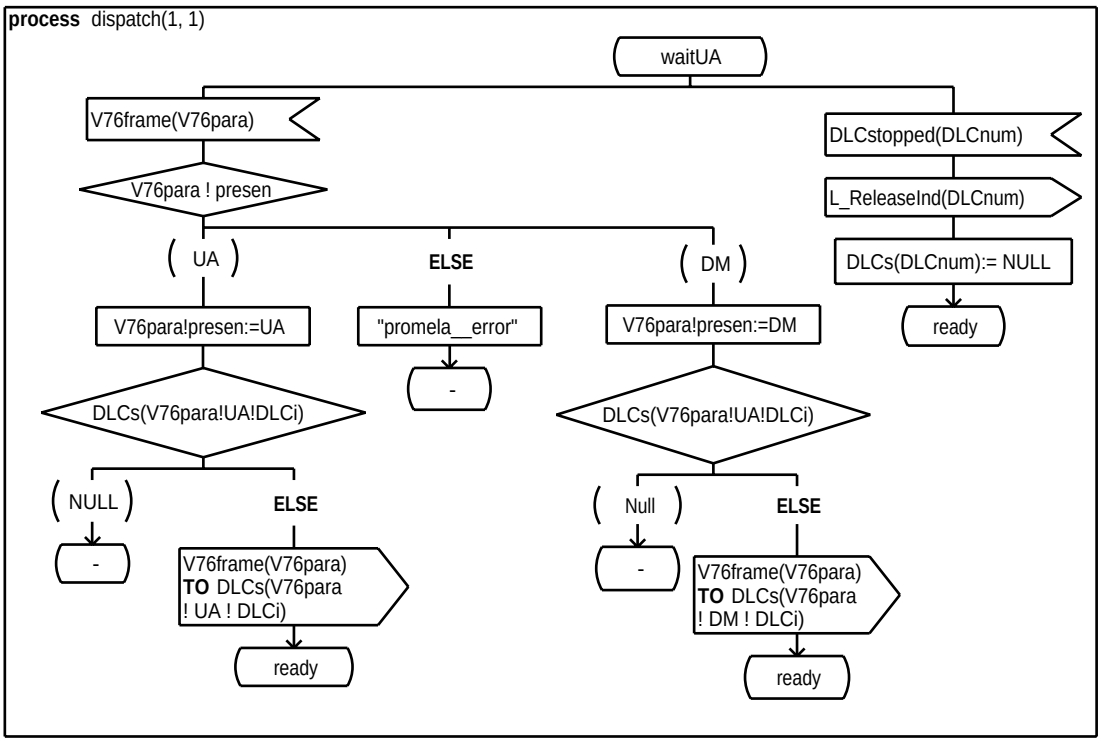
Slika 5.29: Razširjen MSC diagram sprejema ukaza `DM` pri procesu `dispatch` v stanju `waitUA`

specifikacijo uporabnika brez dodane oznake. Definicija stanja napredka 4.9 (str. 120) opisuje samo ω-izvajalne poti. Z vpeljavo jecljanja v razdelku 4.1.3 se lahko vsaka končna izvajalna pot predstavi kot ω-izvajalna pot. Končna izvajalna pot je veljavna ω-izvajalna pot samo takrat, ko se zaključi v veljavnem končnem stanju. Formalna verifikacija potrdi, da je stanje `SUa_endstate` del vseh cikličnih izvajalnih poti.

5.3.6 Preverjanje temporalnih lastnosti modela

S stališča prostorske in časovne zahtevnosti je najzahtevnejše preverjanje skladnosti s specifikacijo zahtev, ki je podana s formulo LTL. V razdelkih 4.1.5 in 4.1.6 je podrobneje predstavljen postopek verifikacije z uporabo procesa “never claim”, ki opisuje želeno lastnost modela.

Modeli specifikacije pri preverjanju z uporabo procesa “never claim” ne vključujejo

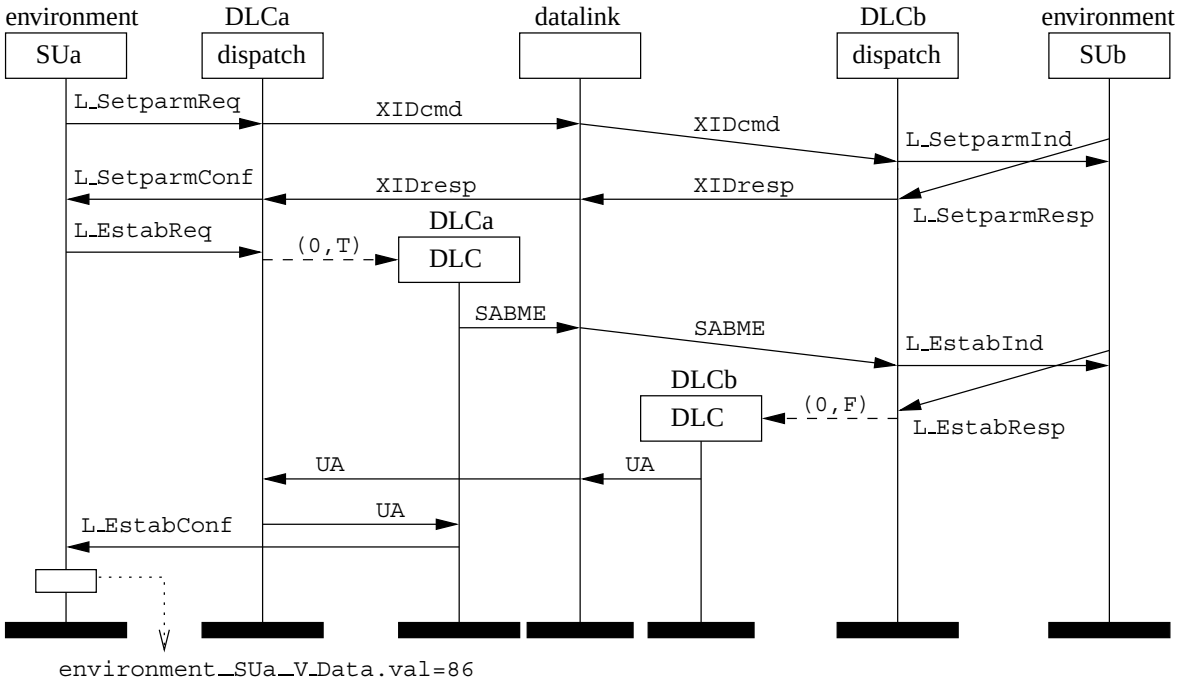


Slika 5.30: Dopolnjena specifikacija procesa `dispatch` s sprejemom ukaza `DM` v stanju `waitUA`

procesa `monitor` (slika 4.1, str. 122). Preverjanje morebitne kršitve semantičnih pravil jezika SDL lahko preveri tudi proces "never claim", ki nastane s pretvorbo formule LTL $\neg\Box\neg$`pv__runtime_error`. Proces "never claim" išče izvajalne poti, ki kršijo specifikacijo zahtev. Zato se pri avtomatski tvorbi procesa "never claim" uporabi negacija zahteve. Kadar želimo izraziti zahtevo, da p nikoli ne drži ($\Box\neg p$), se proces "never claim" pridobi z `dtspin -f '![]!p'`. Iščemo torej izvajalno pot, na kateri velja $\Diamond p$. Formalna verifikacija prekine izvajanje, če najde izvajalno pot, na kateri nekoč velja `pv__runtime_error` (razdelek 4.2.2, str. 128). Formalna verifikacija trenutnega modela, skladno s pričakovanji, ne najde kršitev semantičnih pravil jezika SDL.

Ob preverjanju zahteve $\Box\neg p$, kjer je p formula stanja `environment__SUa__V_Data.val == 86`, najde formalna verifikacija kršitev v 865. koraku. Pred odkritjem napake se je preverilo $9.71828 * 10^6$ stanj in $2.16093 * 10^7$ prehodov. Slika 5.31 prikazuje izvajalno pot, na kateri se za prenašano vrednost izbere število 86 (slika 5.19).

Preverjanje kompleksnejših temporalnih lastnosti modela opisuje formula $\Box(p \rightarrow \Diamond q) \wedge \Diamond p$, v kateri je navadno p sonda za oddajo signala in q sonda za sprejem signala. Tabela 5.2 prikazuje rezultate formalne verifikacije za različne pare signalov. Prvi zapis v tabeli potrdi, da sistem zahtevo za izmenjavo parametrov vedno uspešno posreduje do uporabnika `SUb`. Drugi zapis pokaže, da obstaja izvajalna pot, na kateri iniciator ne prejme odgovora na zahtevo po izmenjavi parametrov. Podobno izpostavi tretji zapis, saj model uporabnika, `SUa`, pred potrditvijo zahteve po izmenjavi parametrov ne prične z vzpostavljanjem nove povezave. Četrti zapis pokaže, da prenos informacijskih okvirjev ni zanesljiv. Formalna verifikacija vrne protiprimer, kjer se signal `L_DataReq` nikoli ne pošlje. Ker takšno izvajanje ne krši želene specifikacije zahtev, smo omilili zahteve s formulo $\Box(p \rightarrow \Diamond q)$. Formalna verifikacija se ustavi v 1691. koraku in pokaže izvajalno pot, ki smo jo predvideli v zadnjem



Slika 5.31: Razširjen MSC diagram izvajalne poti, ki krši specifikacijo zahtev $\Box\neg$(`environment__SUa__V_Data.val == 86`)

Tabela 5.2: Zaporedje oddaje in sprejema signalov

p (proces,stanje,signal)	q (proces,stanje,signal)	**rezultat** $\Box(p \rightarrow \Diamond q) \wedge \Diamond p$
SUa, start, L_SetparmReq	SUb, ready, L_SetparmInd	drži
SUb, ready, L_SetparmResp	SUa, waitParmResp, L_SetparmConf	ne drži
SUa, start, L_SetparmReq	SUb, ready, L_EstabInd	ne drži
SUa, waitEstabConf, L_DataReq	SUb, ready, L_DataInd	ne drži
SUa, waitParmResp, L_EstabReq	SUb, waitEstabConf, L_ReleaseReq	ne drži
SUa, waitEstabConf, L_EstabConf	SUa, (waitEstabConf, SUa_endstate), L_ReleaseReq	ne drži

primeru v razdelku 5.3.4.

V petem primeru preverjamo, ali uporabnik `SUa` po prejetem potrdilu o vzpostavljeni povezavi, vedno pošlje tudi signal `L_ReleaseReq`. Sonda za odposlanje signala `L_ReleaseReq` je nameščena v obeh stanjih, kjer se signal lahko pošlje (slika 5.19). Formalna verifikacija pokaže, da temu ni tako. Izvajalna pot se zaključi po uspešni vzpostavitvi povezave. V razdelku 5.3.2.3 smo to izvajalno pot dodali v množico veljavnih izvajalnih poti, zato v tem primeru formula LTL ne odražala specifikacije zahtev.

5.3.7 Zaključek

Odpornost protokola na nepričakovane odzive okolja predstavlja želeno lastnost. Zato bi pri dejanski specifikaciji protokola večino najdenih nepravilnosti odpravili s popravki specifikacije sistema. V tem poglavju demonstriramo delovanje avtomatske tvorbe modelov s sondami, zato smo pristali tudi na popravke modela okolja. V prvih dveh primerih okolja nismo omejevali. Predvideli smo uporabo vseh signalov, ki jih določa vmesnik protokola V.76 z uporabniki sistema. V okviru druge dopolnitve specifikacije v razdelku 5.3.2.2 smo omejitve okolja poenotili s [17]. Sledili so protiprimeri z izvajalnimi potmi, ki so se zaključile v zagatnem stanju. Uspešno formalno verifikacijo je prestal šele sistem z dodatnimi omejitvami okolja, ki jih opisuje razdelek 5.3.2.6.

Opis specifikacije zahtev s temporalno logiko nudi izredno močno orodje za formalno verifikacijo specifikacije sistema. Pri tem je potrebno biti še posebno pazljiv. Rezultat preverjanja skladnosti z zapisano formulo LTL je pravilen le, če so izpolnjeni naslednji pogoji:

- model verno predstavlja specifikacijo sistema,
- formula LTL pravilno opisuje želeno lastnost,
- sonde so vstavljene na primerno mesto in
- sondam se prirejajo vrednosti tako, da omogočajo preverjanje želene lastnosti.

6. poglavje

Zaključek

V uvodu disertacije smo pokazali potrebo po večji uporabi formalnih metod pri načrtovanju sodobnih sistemov, saj tradicionalne metode ne zagotavljajo celovitosti preverjanja pravilnosti sistemov. To še posebno velja za asinhrone sisteme s sočasnostjo, saj s testiranjem ne moremo preveriti vseh mogočih izvajalnih poti. Vzrok je v izjemno veliki množici mogočih izvajalnih poti, ki nastanejo s prepletanjem izvajanja elementov sistema, in njihovi neponovljivosti. Med testiranjem je pogosto nemogoče slediti natančnemu izvajanju sistemov s sočasnostjo, saj nimamo vpogleda v dodeljevanje časovnih rež posameznim procesom. Tudi če bi ta podatek imeli, bi bilo izvajalno pot praktično nemogoče ponoviti.

Uporaba formalnih metod odpravlja naštete pomanjkljivosti in lahko z matematično natančnostjo potrdi ali ovrže skladnost sistema s specifikacijo zahtev. Njihova uporaba je priporočljiva v vseh razvojnih fazah — od formalnega zapisa specifikacije zahtev do preverjanja pravilnosti delovanja sistema.

Funkcijska specifikacija orodja $sdl2pml$ je podana z algoritmi v poglavju 3. Algoritmi ne podajajo vseh podrobnosti implementacije in se v nekaterih delih sklicujejo na funkcije. Te so opisane v neformalni obliki v besedilu disertacije. Med nastankom produkta, ki vsebuje več kot 100.000 vrstic programske kode, je prišlo do nesporazumov izključno v neformalno zapisanih delih specifikacije. Tudi kadar so algoritmi vsebovali nepravilnosti, so se te hitro izpostavile in so bile nedvoumne. Ob napačnem delovanju programa zato v teh primerih ni bilo dvoma o tem, kje je bila storjena napaka. Te napake navadno niso bistveno vplivale na implementacijo in so se odpravile brez večjih vložkov. Pri neformalnem delu specifikacije pa smo se vedno znova spopadali z različnim razumevanjem zapisanega. Vsi dodatni vložki, ki so bili potrebni za podrobnejšo razlago delovanja in odpravo vnesenih napak v izdelek, potrjujejo, da je uporaba formalnih metod smotrna že v prvih razvojnih korakih.

Za uspešno uporabo formalnih metod na realnih sistemih je potrebno zagotoviti primerna orodja. Mednje zagotovo sodi orodje za avtomatsko tvorbo modelov. Vhod v orodje so lahko različne oblike opisa specifikacije. Od opisa na abstraktnem nivoju do podrobnega opisa, ki je osnova za implementacijo. Jezik SDL omogoča nedvoumno, jasno in natančno specifikacijo sistema. Zato obstajajo orodja za avtomatsko tvorbo izvorne kode v izbranem programskem jeziku. Mnoge raziskave se ukvarjajo z avtomatsko tvorbo modelov iz programske kode, raziskovalci pa so si enotni, da je preverjanje pravilnosti delovanja najbolje izvajati na izvorni specifikaciji, če je to mogoče.

Disertacija prispeva k celovitejši obravnavi avtomatske tvorbe modelov iz specifikacij v jeziku SDL. Te so lahko poljubno natančne, zato so rezultati disertacije uporabni v vseh razvojnih korakih sistema.

6.1 Povzetek prispevkov disertacije

V disertaciji smo predstavili:

1. algoritme za analizo specifikacij v jeziku SDL in njihovo pretvorbo v model sistema v jeziku Promela,
2. postopke za avtomatsko dopolnitev modela s sondami,
3. praktični primer uporabe rezultatov disertacije.

Prispevke lahko razvrstimo v tri osnovne skupine: pretvorba konstruktov, modeliranje komunikacije in dopolnitev modela s sondami. Osnovo za navedene prispevke predstavljajo algoritmi za analizo specifikacije sistema, vendar te ne uvrščamo v skupino pomembnejših prispevkov. V nadaljevanju poudarimo, kateri so glavni prispevki disertacije na vsakem od teh področij:

- pretvorba konstruktov:
  - celovito modeliranje zapisov in polj z minimalnim prispevkom k dolžini vektorja stanj,
  - modeliranje dinamične tvorbe procesov z reciklažo pridruženih identifikacijskih številk,
  - algoritem za prirejanje vrednosti zapisom,
  - algoritem za pretvorbo izrazov,
  - algoritem za pretvorbo telesa procesa,
  - modeliranje konstrukta shrani,
  - modeliranje stanja zvezdica,
- modeliranje komunikacije:
  - algoritem za pripravo pridruženega kanala,
  - mehanizem za dinamično spremljanje pridružene vhodne vrste,
  - modeliranje prioritetnega vhoda,
  - modeliranje implicitnega prehoda,
  - modeliranje spontanega prehoda,
  - modeliranje pogojnega sprejema signala,
  - modeliranje časovnika s parametri,
  - modeliranje trajnega signala,
  - modeliranje vhoda zvezdica,
  - modeliranje naslavljanja s PId,
  - modeliranje naslavljanja z imenom procesa,
  - modeliranje naslavljanja z uporabo rezervirane besede **via**,
  - modeliranje naslavljanja brez specifikacije naslovnika ali poti na osnovi strukture sistema,
- dopolnitev modela s sondami:

- avtomatski vnos trajnih sond za preverjanje skladnosti modela s semantičnimi pravili jezika SDL,
- avtomatski vnos sond za veljavna končna stanja,
- avtomatski vnos sond za stanja napredka,
- avtomatski vnos sond za dopustna stanja,
- avtomatski vnos sond za iskanje potencialno nepravilnih izvajalnih poti,
- avtomatski vnos sond za preverjanje temporalnih lastnosti sistema.

6.2 Zaključna razprava in usmeritev nadaljnjih raziskav

V disertaciji smo pri tvorbi modela posebno pozornost posvetili celoviti podpori uporabe spremenljivk, dinamični tvorbi procesov in komunikaciji. Prioritete raziskovalnega dela so postavljale poznane industrijske specifikacije sistema, na katerih smo preverjali rezultate implementacije algoritmov.

Podprte so vse osnovne oblike definicij podatkovnih tipov. Spremenljivke so v modelu v večini primerov predstavljene z optimalno porabo pomnilnika. To je še posebno pomembno zaradi velikosti vektorja stanj in zaznavi morebitnih kršitev semantičnih pravil jezika. Kot prvi smo podprli pravilno modeliranje prirejanja struktur, saj je jezik Promela v tem delu manj izrazen od jezika SDL. Še poseben izziv je predstavljalo modeliranje konstrukta shrani. Pravilnemu modeliranju konstrukta je podrejeno celotno telo procesa. Dosedanji poskus modeliranja konstrukta shrani je uporabil dva pridružena kanala. S tem so se podvojile pomnilniške zahteve za hrambo stanja kanalov. Posledično se je tudi razpolovilo število mogočih procesnih tipov. Naša rešitev z dinamičnim spremljanjem pridružene vhodne vrste razširi vektor stanj s *cv_procnum* $*$ *cv_buff* $*(8+1+1)$ in je torej zmnožek največjega predvidenega števila procesov, števila rež pridruženih kanalov, osmih bitov za predstavitev imena signala in dveh dodatnih bitov za zastavici. V primeru privzetega števila rež 3 se v modelu sistema za vsak proces porabi dodatnih 30 bitov[1]. Naša rešitev dodatno omogoča modeliranje prioritetnih signalov.

Pri komunikaciji jezik SDL omogoča zelo sproščeno definicijo naslovnika. V najbolj splošnem primeru, ko ni podano ne ime ciljnega procesa ne dodatne omejitve poti, lahko signal prejme katerikoli proces v sistemu, ki je s pošiljateljem povezan s komunikacijskimi gradniki — signalnimi potmi in kanali. Dosedanje rešitve niso podpirale posredne in nedeterministične izbire naslovnika med izvajanjem sistema. Prav tako ni bilo podprto naslavljanje z uporabo PId-a, kar se v realnih specifikacijah najpogosteje uporablja zaradi nedvoumnega naslavljanja posameznih primerkov procesa. Prvi smo podprli dinamično tvorbo procesov z reciklažo pridruženih identifikacijskih številk, ki omogoča minimalno porabo pomnilniškega prostora za spremljanje pridružene vhodne vrste. Tu smo naleteli na omejitve, ki jih postavlja semantika jezika Promela. Ta razlikuje med terminacijo in ubitjem procesa, kjer terminacija označuje zaključek izvajanja, ubitje pa sprostitev pridružene identifikacijske številke. Nabor specifikacij, ki jih lahko preverimo v celoti, je zato omejen na tiste, kjer se procesi tvorijo in ubijajo v obratnem vrstnem redu. Razširitev orodja Spin s dosledno podporo semantičnim pravilom jezika Promela lahko te težave delno odpravi. Kljub temu takšen model vsebuje veliko število virtualnih mogočih izvajalnih poti, ki močno povečajo časovno in prostorsko zahtevnost formalne verifikacije. Verifikacija teh specifikacij je zaradi uporabe tehnike "on

[1] Trenutna implementacija orodja Spin ne omogoča natančnega izračuna velikosti vektorja stanj zaradi podoptimalne rezervacije pomnilnika za spremenljivke podatkovnega tipa `bool`.

the fly" vseeno mogoča. Na tem področju so potrebne dodatne raziskave.

Kot prvi smo vpeljali podporo za konstrukte prioritetni vhod, vhod zvezdica ter podprli implicitni prehod, pogojni sprejem signalov, trajni signal in spontani signal. Pravilnost modeliranja konstruktov smo formalno preverili z orodjem Spin.

Pri obravnavi časovnikov smo uporabili razširitev orodja Spin za podporo diskretnemu času. Med preverjanjem rezultatov disertacije smo v razširitvi orodja odkrili napako. Na našo pobudo jo je avtor odpravil v izredno kratkem času. Kot prvi smo razširili podporo časovnikov s parametri in modelirali sprejem časovnika skladno s semantičnimi pravili jezika SDL — s sprejemom signala.

V poglavju 4 smo podali pravila za dopolnitev modela sistema s sondami. Predstavljene rešitve omogočajo zaznavo velike množice nepravilnosti v specifikaciji sitema. To dokazuje tudi praktični primer verifikacije specifikacije protokola V.76. Kjub temu, ostaja pravilen zapis specifikacije zahtev in avtomatični vnos sond v model sistema eden izmed večjih izzivov za nadaljnje raziskovalno delo.

Uporabljen praktični primer specifikacije protokola V.76 je bil pridobljen na medmrežju in zato predstavlja primerno osnovo za preverjanje delovanja algoritmov. To bi bilo bistveno manj učinkovito, če bi specifikacijo pripravili sami. Dopolnjeno specifikacijo brez komentarjev sestavlja 1304 vrstic v jeziku SDL. Velikost modela sistema v jeziku Promela je odvisna od števila vnesenih sond. V primerih, ki so predstavljeni v poglavju 5, se velikost giblje med 4627 in 5034 vrsticami. Razlika v velikosti nakaže razlike v izraznosti obeh jezikov. Prvi vtis o morebitni neučinkovitosti modela zaradi njegove dolžine, ovržejo rezultati formalne verifikacije. Protiprimere vseh odkritih napak v modelu sistema je orodje Spin tvorilo hitreje kot v eni sekundi[2]. Zaradi pogoste uporabe časovnikov smo imeli težave z dokazovanjem veljavnosti formule v razdelku 5.3.6. Za 100% potrditev veljavnosti smo morali uporabiti najmočnejšo brezizgubno kompresijo vektorja stanj in počakati 48 ur. Natančen študij obravnave časovnikov v razširitvi orodja Spin in njihova optimalnejša vključitev v algoritme formalne verifikacije je zato eden izmed večjih izzivov za nadaljnje raziskovalno delo.

Disertacija podaja množico rešitev, ki lahko pomembno prispevajo k uporabi formalnih metod v industriji. Še vedno pa ostaja mnogo izzivov. Na koncu velja izpostaviti še avtomatsko tvorbo modelov iz specifikacij v jeziku SDL, ki vključujejo zunanje gradnike v programskem jeziku C. Semantika jezika Promela in orodje Spin nudita osnovo za takšno rešitev, zato so nadaljnje raziskave na tem področju obetavne.

[2]Uporabili smo računalnik z 2 GB pomnilnika in procesorjem Intel Xeon 2.4 GHz.

Literatura

[1] Douglas Adams. *Life, the Universe and Everything*. Ballantine Books, reprint edition, 1995.

[2] Fulvio Babich and Lia Deotto. Formal Methods for Specification and Analysis of Communication Protocols. *IEEE Communications Surveys*, 2002.

[3] Jos Baeten. Esprit Project 23498 - VIRES (Verifying Industrial Reactive Systems). URL: <http://www.cordis.lu/esprit/src/23498.htm>.

[4] Ferenc Belina, Dieter Hogrefe, and Amardeo Sarma. *SDL with Applications from Protocol Specification*. Prentice Hall International (UK) Ltd., 1991.

[5] Paul E. Black, Kelly M. Hall, Michael D. Jones, Trent N. Larson, and Phillip J. Windley. A Brief Introduction to Formal Methods. In *Proceedings of the IEEE Custom Integrated Circuits Conference*, pages 377–380. IEEE, 1996.

[6] Dragan Bošnački. Extending Promela and Spin with Discrete Time. In *Proceedings of the VIII Conference on Logic and Computer Science*, 1997.

[7] Dragan Bošnački and Dennis Dams. Discrete Time Promela and Spin. *Lecture Notes in Computer Science*, pages 307–310, 1998.

[8] Dragan Bošnački, Dennis Dams, Leszek Holenderski, and Natalia Sidorova. Model Checking SDL with Spin. *Lecture Notes in Computer Science*, pages 363–377, 2000.

[9] Jonathan Bowen. Formal Methods.
URL: <http://www.afm.lsbu.ac.uk/>.

[10] Jonathan P. Bowen and Michael G. Hinchey. Seven More Myths of Formal Methods. *IEEE Software*, 12(3):34–41, 1995.

[11] Jonathan P. Bowen and Michael G. Hinchey. Ten Commandments of Formal Methods. *IEEE Computer*, 28(4):56–63, 1995.

[12] M. Bozga, J.C. Fernandez, L. Ghirvu, S. Graf, J.P. Krimm, L. Mounier, and J. Sifakis. If: An Intermediate Representation for SDL and its Applications. In *Proceedings of SDL-FORUM'99, Montreal, Canada*, June 1999.

[13] M. Bozga, L. Ghirvu, S. Graf, L. Mounier, and J. Sifakis. The Intermediate Representation IF: Syntax and semantics. Technical report, Vérimag, Grenoble, 1999.

[14] Marius Bozga, Susanne Graf, and Laurent Mounier. Automated validation of distributed software using the IF environment. *Electronic Notes in Theoretical Computer Science*, 2001.

[15] Randy Bryant. Pentium Style, An Analysis of Intel's Mistake(s), 1995. URL: <http://www.cs.cmu.edu/afs/cs.cmu.edu/user/bryant/www/home.html/>.

[16] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, third printing edition, 2001.

[17] Laurent Doldi. *Validation of Communications Systems with SDL: The Art of Simulation and Reachability Analysis*. John Wiley & Sons, Ltd, 2003.

[18] Jan Ellsberger, Dieter Hogrefe, and Amardeo Sarma. *SDL Formal Object-oriented Language for Communicating Systems*. Prentice Hall Europe, 1997.

[19] K. Etessami. Stutter-invariant languages, ω-automata, and temporal logic. In *CAV'99: Computer Aided Verification, 11th International Conference*, pages 236–248, 1999.

[20] K. Etessami and G.J. Holzmann. Optimizing buchi automata. In *Proc. CONCUR2000*, pages 153–167. Springer Verlag / LNCS 1877, Aug. 2000.

[21] K. Etessami, Th. Wilke, and R. A. Schuller. Fair simulation relations, parity games, and state space reduction for Büchi automata, 2001. To appear in *Proc. of 28th Int. Col. on Automata Languages and Programming (ICALP'2001)*.

[22] P. Gastin and D. Oddoux. Fast LTL to Büchi automata translation. In G. Berry, H. Comon, and A. Finkel, editors, *Proceedings of the 13th Conference on Computer Aided Verification (CAV'01)*, number 2102 in Lecture Notes in Computer Science, pages 53–65. Springer Verlag, 2001.

[23] Brent Hailpern and Padmanabhan Santhanam. Software Debugging, Testing, and Verification. *IBM Systems Journal*, 41(1), 2002.

[24] Anthony Hall. Seven Myths of Formal Methods. *IEEE Software*, 1990.

[25] K. Havelund and T. Pressburger. Model checking java programs using java pathfinder, 1998.

[26] Gerard J. Holzmann. Spin. URL: <http://netlib.bell-labs.com/netlib/spin/>.

[27] Gerard J. Holzmann. Algorithms for Automated Protocol Verification. *AT&T Technical Journal*, 69(2):32–44, 1988.

[28] Gerard J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1991.

[29] Gerard J. Holzmann. Proving the Value of Formal Methods. In *7th Int. Conference on Formal Description Techniques (FORTE94)*, Bern, Switzerland, 1994.

[30] Gerard J. Holzmann. Proving Properties of Concurrent Systems with SPIN. In *Proc. CONCUR95, 6th Intern. Conf. on Concurrency Theory*, 1995.

[31] Gerard J. Holzmann. Designing bug-free protocols with Spin. *Computer Communications Journal*, 20:97–105, 1997.

[32] Gerard J. Holzmann. The Model Checker SPIN. *Software Engineering*, 23(5):279–295, 1997.

[33] Gerard J. Holzmann. Logic Verification of ANSI-C Code with Spin. In *7th International SPIN Workshop on Model Checking of Software*, Lecture Notes in Computer Science. Springer Verlag, 2000.

[34] Gerard J. Holzmann. From Code to Models. In *Proc. 2nd Int. Conf. on Applications of Concurrency to System Design*, pages 3–10, 2001.

[35] Gerard J. Holzmann. The Logic of Bugs. In *Proc. ACM Foundations of Software Engineering*, November 2002.

[36] Gerard J. Holzmann. *The SPIN model checker: Primer and reference manual*. Addison Wesley, 2003. HOL g 03:1 1.Ex.

[37] Gerard J. Holzmann and Margaret Smith. Automating Software Feature Verification. *Bell Labs Technical Journal*, 5:72–87, 2000.

[38] Gerard J. Holzmann and Margaret H. Smith. An Automated Verification Method for Distributed Systems Software Based on Model Extraction. *IEEE Transactions on Software Engineering*, 28, April 2002.

[39] Nisse Husberg. Formal Models for Real Programs. In *8th Nordic Workshop on Programming Theory*, 1996.

[40] Nisse Husberg. SDL Modelling with High Level Petri Nets. In *Workshop on Concurrency, Specification & Programming*, 1996.

[41] Nisse Husberg, Markus Malmqvist, and Tero Jyrinki. Emma: A Tool for Analysis of SDL Programs. In *Workshop on Concurrency, Specification & Programming*, 1996.

[42] Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and reasoning about systems*. Cambridge University Press, 2002.

[43] ITU-T Recommendation Z.100 Appendices I and II. SDL methodology guidelines, SDL Bibliography, 1993. Series Z: Programming Languages, Specification and Description Language (SDL).

[44] IETF Network Working Group. RFC 1345 - Character Mnemonics and Character Sets, 1992.

[45] ITU-T Recommendation V.76. Generic multiplexer using V.42 LAPM-based procedures, 1996. Series V: Data communication over the telephone network.

[46] NASA. *Formal Methods Specification and Analysis Guidebook for the Verification of Software and Computer Systems*, volume Volume II: A Practitioner's Companion. National Aeronautic and Space Administration, release 1.0 edition, 1997.

[47] Armelle Prigent, Franck Cassez, Philippe Dhaussy, and Olivier Roux. Extending the translation from SDL to Promela. In *9th International SPIN Workshop on Model Checking of Software (SPIN'02)*, volume 2318 of *Lecture Notes in Computer Science*, pages 400–414, Grenoble, France, March 2002. Springer-Verlag.

[48] ITU-T Recommendation Z.100 Annexes C and D. Initial algebra model and SDL predefined data, 1993. Series Z: Programming Languages, Specification and Description Language (SDL).

[49] Ana Robnik and Uroš Srakar. *Jezik SDL in orodje GEODE*. IskraTEL d.o.o, Kranj, Slovenija, Marec 1996. Gradivo za tečaj ”Programiranje za SI2000”.

[50] Theo C. Ruys. SPIN Tutorial: How to Become a SPIN Doctor. In *Model Checking of Software: 9th International SPIN*, Lecture Notes in Computer Science. Springer Verlag, 2002.

[51] Natalia Sidorova and Martin Steffen. Verifying Large SDL-Specifications Using Model Checking. In *SDL 2001: Meeting UML: 10th International SDL Forum*, Lecture Notes in Computer Science. Springer Verlag, 2001.

[52] Heikki Tuominen. Embedding a Dialect of SDL in PROMELA. In *Theoretical and Practical Aspects of SPIN Model Checking: 5th and 6th International SPIN Workshops*, volume 1680 of *Lecture Notes in Computer Science*, pages 245–260. Springer-Verlag, 1999.

[53] Verilog. *ObjectGEODE — SDL C Code Generator, Version 4.0*. Verilog SA, 1999.

[54] Verilog. *ObjectGEODE — SDL Simulator, Version 4.0*. Verilog SA, 1999.

[55] Pierre Wolper. Temporal Logic Can Be More Expressive. *Information and Control*, 56(1–2):72–99, 1983.

[56] Pierre Wolper. Expressing Interesting Properties of Programs in Propositional Temporal Logic. In *Proc. 13th ACM Symp. on Principles of Programming Languages*, pages 184–192, St. Petersburgh, January 1986.

[57] ITU-T Recommendation Z.100. CCITT Specification and Description Language (SDL), 1993. Series Z: Programming Languages.

[58] ITU-T Recommendation Z.100. CCITT Specification and Description Language (SDL) - Addendum 1, 1996. Series Z: Programming Languages.

[59] ITU-T Recommendation Z.100. Specification and Description Language (SDL), 2002. Series Z: Languages and General Software Aspects for Telecommunication Systems.

[60] ITU-T Recommendation Z.105. SDL combined with ASN.1 (SDL/ASN.1), 1995. Series Z: Programming Languages.

A. Dodatek

Modeliranje operatorjev tipa charstring

```
/*********************************************************************/
/*                                                                   */
/*                                                                   */
/*              Copyright (c) 2004 UM FERI                           */
/*                                                                   */
/*                                                                   */
/*  Name        : sdl_data.ph                                        */
/*                                                                   */
/*  Description : Definition of the SDL datatypes                    */
/*                CHARACTER and CHARSTRING in Promela                */
/*                                                                   */
/*  Code        :                                                    */
/*                                                                   */
/*  Date        : Jan, 2004                                          */
/*                                                                   */
/*  Author      : Bostjan Vlaovic, LMS                               */
/*                                                                   */
/*  Remarks     :                                                    */
/*                                                                   */
/*  Revision    : 1.0                                                */
/*                                                                   */
/*********************************************************************/

/*********************************************************************/
/*     Type pt__character represents the SDL CHARACTER type           */
/*********************************************************************/
typedef pt__character {unsigned char : 7 = 0}


/*********************************************************************/
/*     Type pt__charstring represents the SDL CHARSTRING type         */
/*********************************************************************/
/* Predefined maximal length of the SDL CHARSTRING type 32 + 1        */
/* because in SDL index start is at 1.                                */
#define pcv__charstring_max  33

/* Arrays of unsigned type are not supported, so byte is used         */
/* typedef pt__charstring {pt_character char[pcv_charstring_max]}     */

typedef pt__charstring {byte char[pcv__charstring_max]}

/*********************************************************************/
/*                    SDL CHARSTRING operators                       */
/*********************************************************************/
/*********************************************************************/
/*                    strcpy                                         */
/*********************************************************************/
inline strcpy(pfv__stringB,pfv__stringA){
 d_step{
  pfv__tmp=1;
  do
   ::((pfv_tmp<=pcv__charstring_max) && pfv_stringA.char[pfv__tmp]!=NUL) ->
     pfv__stringB.char[pfv__tmp]=pfv__stringA.char[pfv__tmp];pfv__tmp++;
   ::else -> break
```

```
  od
 }
}

/*********************************************************************/
/*                    strcmp                                         */
/*                                                                   */
/* returns 0 if strings match                                        */
/* returns n+1 if strings match in first n elements                  */
/*********************************************************************/
inline strcmp(pfv__result,pfv__stringB,pfv__stringA){
 d_step{
  pfv__tmp=1;
  pfv__tmp1=1;
  do
   :: (pfv__tmp<=pcv__charstring_max) ->
       if
        :: pfv__stringB.char[pfv__tmp]==pfv__stringA.char[pfv__tmp]->
   pfv_tmp++;pfv_tmp1++;
:: else -> pfv__tmp++;break
       fi
   :: else -> break
  od;
  if
   :: pfv__tmp1==pfv__tmp -> pfv__result=0;
   :: else -> pfv__result=pfv__tmp1
  fi
 }
}

/*********************************************************************/
/*                     MkString                                      */
/*********************************************************************/
inline MkString(pfv__string,pfv__char){
 pfv__string.char[1]=pfv__char.char
}

/*********************************************************************/
/*                     Length                                        */
/*********************************************************************/
inline Length(pfv__string,pfv__length){
 d_step{
  pfv__length=0;
  pfv__tmp=1;
  do
   :: ((pfv__tmp<=pcv__charstring_max) && pfv__string.char[pfv__tmp]!=NUL) ->
      pfv__length++;pfv__tmp++;
   :: else -> break
  od
 }
}

/*********************************************************************/
/*                     First                                         */
/*********************************************************************/
inline First(pfv__string,pfv__char){
 pfv__char.char=pfv__string.char[1]
}

/*********************************************************************/
/*                     Last                                          */
/*********************************************************************/
inline Last(pfv__string,pfv__char){
 d_step{
  Length(pfv__string,pfv__tmp1);
  pfv__char.char=pfv__string.char[pfv__tmp1]
 }
}

/*********************************************************************/
/*                     Extract                                       */
/*********************************************************************/
inline Extract(pfv__string,pfv_position,pfv__char){
```

```
  pfv__char.char=pfv__string.char[pfv__position]
}

/*********************************************************************/
/*                     Modify                                        */
/*********************************************************************/
inline Modify(pfv__stringB,pfv__stringA,pfv__position,pfv__char){
 d_step{
  strcpy(pfv__stringB, pfv__stringA);
  pfv__stringB.char[pfv__position]=pfv__char.char
 }
}
```

B. Dodatek

Modeliranje vhodnih konstruktov

B.1 Specifikacija sistema

```
SYSTEM specifikacija;
  BLOCK B;
    SIGNAL sig1,sig2(zapis);
    SIGNAL sig3(zapis);
    SIGNAL prio1,prio2;

    NEWTYPE zapis
      STRUCT
         a  INTEGER;
         b  INTEGER;
         c  BOOLEAN;
    ENDNEWTYPE zapis;

    SIGNALROUTE sig_pot
      FROM sprejemnik TO oddajnik WITH sig1,sig2,sig3,prio1,prio2;
      FROM oddajnik TO sprejemnik WITH sig1,sig2,sig3,prio1,prio2;
    PROCESS sprejemnik
    (1,2)
    ;
      timer caso;
      dcl a INTEGER := 4;
      dcl stru  zapis;
      START ;
        TASK stru!a:=0;
        TASK stru!b:=0;
        TASK stru!c:=FALSE;
        NEXTSTATE s1
          COMMENT 'endstate';
      STATE s1
        COMMENT 'endstate';
        INPUT caso;
          TASK a:=7;
          NEXTSTATE s2;
        SAVE sig1;
        PRIORITY INPUT prio1;
          TASK a:=0;
          NEXTSTATE s2;
        INPUT sig3(stru);
          PROVIDED a<5;
          NEXTSTATE s2;
        INPUT sig2(stru);
          PROVIDED a>5;
          NEXTSTATE s2;
      ENDSTATE;
      STATE s2
        COMMENT 'endstate';
        PROVIDED a=0;
          TASK a:=1;
          NEXTSTATE s2;
        PRIORITY INPUT caso;
          TASK a:=7;
```

```
52          NEXTSTATE s1;
53        INPUT sig1;
54          NEXTSTATE s1;
55        PRIORITY INPUT prio2;
56          SET (NOW+5,caso);
57          NEXTSTATE s1;
58      ENDSTATE;
59    ENDPROCESS;
60    PROCESS oddajnik;
61      dcl stru zapis;
62      START ;
63        CREATE sprejemnik;
64        OUTPUT sig1 to offspring;
65        OUTPUT prio1 to offspring;
66        TASK stru!a:=2;
67        TASK stru!b:=2;
68        TASK stru!c:=TRUE;
69        OUTPUT sig2(stru) to offspring;
70        OUTPUT prio2 to offspring;
71        TASK stru!a:=3;
72        TASK stru!b:=3;
73        TASK stru!c:=FALSE;
74        OUTPUT sig3(stru) to offspring;
75        STOP ;
76    ENDPROCESS;
77  ENDBLOCK;
78 ENDSYSTEM;
```

B.2 Dodatne vhodne datoteke

B.2.1 sdl_run.ph

```
1 pt__chan chan__none = [1] of {bit}
2 /*vrednost 1 omogoča uporabo poštenostnih omejitev*/
3
4 #define run__proc(free,max,name,\
5                 channel,select)\
6  atomic{\
7  if\
8  :: free < max ->\
9     if\
10    :: channel == chan__none -> \
11       offspring = run name(_pid)\
12    :: else ->\
13      offspring = run\
14        name(channel[free],_pid);\
15      table_pid_channum[offspring] =\
16        channel[free];\
17      table_pid_channame[offspring] =\
18        select+free;\
19      table_channame_channum[select+free] =\
20        channel[free]\
21    fi;\
22    if\
23    ::(offspring==0) -> error = true\
24    ::(offspring!=0) -> free++\
25    fi\
26  :: else -> error = true\
27  fi;\
28  }
```

B.2.2 sdl_comm.ph

```
37 typedef pt__chandat {pt__sig name;bool save;bool prio};
38 typedef pt__chandata {pt__chandat data[cv__buff]};
39 pt__chan
40  table_pid_channum[cv__procnum];
41 pt__sig
42  table_pid_channame[cv__procnum];
43 byte
```

```
 table_channum_ptr[cv__procnum];
pt__chandata
 table_channum_nsp[cv__procnum];
bool
 table_channum_prio[cv__procnum];
pt__chan
 table_channame_channum[cv__procnum];

inline update_chan_tab(){
atomic{
do
::(pv__ptr < cv__buff-1) ->
  table_channum_nsp[input].data[pv__ptr].name=
    table_channum_nsp[input].data[pv__ptr+1].name;
  table_channum_nsp[input].data[pv__ptr].save=
    table_channum_nsp[input].data[pv__ptr+1].save;
  table_channum_nsp[input].data[pv__ptr].prio=
    table_channum_nsp[input].data[pv__ptr+1].prio;
  pv__ptr++
:: else ->
  table_channum_nsp[input].data[pv__ptr].name=none;
  table_channum_nsp[input].data[pv__ptr].save=false;
  table_channum_nsp[input].data[pv__ptr].prio=false;
  break
od;
table_channum_ptr[input]--;
pv__cur=0;
}
}

inline set__prio(sig){
  table_channum_nsp[input].data[pv__ptr].name==sig ->
  atomic{
    table_channum_nsp[input].data[pv__ptr].save=false;
    table_channum_nsp[input].data[pv__ptr].prio=true;
    table_channum_prio[input]=true
  }
}

inline set__save(sig){
  table_channum_nsp[input].data[pv__ptr].name==sig ->
  atomic{
    table_channum_nsp[input].data[pv__ptr].save=true;
    table_channum_nsp[input].data[pv__ptr].prio=false;
  }
}

inline set__clear(){
  atomic{
    table_channum_nsp[input].data[pv__ptr].save=false;
    table_channum_nsp[input].data[pv__ptr].prio=false;
  }
}

#define recv__sig(pog,sig)\
 ((pog) && table_channum_nsp[input].data[pv__ptr].name==sig)

inline skip__save(){
  table_channum_nsp[input].data[pv__ptr].save==true -> pv__ptr++
}
```

B.3 Model sistema

```
#include "dtimesdl.h"
#define cv__buff 4
#define none 0
#define NOW 0

#define pt__pid pid
#define pt__chan chan
/*št. procesov + __Timers + init + monitor + neverclaim*/
#define cv__procnum 7
```

```
10 #define pt__select byte
11 #define pt__sig byte
12 #include "sdl_comm.ph"
13 #include "sdl_run.ph"
14
15 #define B__sig1 1
16 #define B__sig2 2
17 #define B__prio1 3
18 #define B__prio2 4
19 #define B__sprejemnik__caso 5
20 #define B__sig3 6
21 #define sig4 7
22 bit error = false;
23
24 typedef B__zapis {
25   int a;
26   int b;
27   bool c
28 };
29 hidden B__zapis undefined__B__zapis;
30
31
32 #define table__sprejemnik__max 2
33 hidden byte table__sprejemnik__free;
34 #define chan__sprejemnik__select 1
35 pt__chan chan__sprejemnik[table__sprejemnik__max] = [cv__buff] of
36                                        {pt__sig,pt__pid,B__zapis};
37 #define table__oddajnik__max 1
38 hidden byte table__oddajnik__free;
39
40 show bool b_B__sig1,b_B__sig2,b_B__sig3,b_sig4,b_B__prio1;
41 show bool b_B__prio2,b_B__sprejemnik__caso,b_trajni;
42
43 proctype sprejemnik(pt__chan input;pt__pid parent){
44 pt__pid offspring;
45 pt__pid sender;
46 xr input;
47 byte pv__ptr, pv__cur;
48 timer tmr__B__sprejemnik__caso;
49 B__zapis B__sprejemnik__caso__1;
50 B__zapis stru;
51 int a=4;
52
53 stru.a=0;
54 stru.b=0;
55 stru.c=false;
56
57 s1:
58 end_s1:
59 do
60 :: table_channum_ptr[input] > pv__cur  ->
61     table_channum_prio[input]=false;
62     pv__cur++;
63     pv__ptr=0;
64     atomic{
65     do
66     :: pv__ptr <= cv__buff-1 ->
67         if
68         :: set__prio(B__prio1)
69         :: set__save(B__sig1)
70         :: else -> set__clear()
71         fi;
72         pv__ptr++
73     :: else -> goto s1_start
74     od;
75     }
76 s1_start:
77     /* konec pregleda vhodne vrste */
78     if
79     :: table_channum_prio[input]==true ->
80         pv__ptr=0;
81         /* sprejem najstarejšega prioritetnega signala */
82         do
```

```
83          :: (pv__ptr <= cv__buff-1)
84             && (table_channum_nsp[input].data[pv__ptr].prio==true) ->
85             if
86             ::atomic{recv__sig(true,B__prio1) ->
87                 input??B__prio1(sender,undefined__B__zapis);
88                 update_chan_tab();
89                 }
90                 a=0; b_B__prio1=true; b_B__prio1=false;
91               goto s2
92             fi;
93          :: (pv__ptr == cv__buff) -> break;
94          :: else -> pv__ptr++
95          od;
96          /* konec sprejema prioritetnega signala */
97      :: else ->
98          pv__ptr=0;
99          /* sprejem najstarejšega signala, ki nima oznake shrani */
100         do
101          ::(pv__ptr <= cv__buff-1) ->
102            if
103            :: skip__save()
104            :: else ->
105               if
106                 ::atomic{recv__sig(!(a > 5),B__sig2) ->
107                   pv__ptr++}
108                 ::atomic{recv__sig((a > 5),B__sig2) ->
109                   input??B__sig2(sender,stru);
110                   update_chan_tab();
111                   }
112                   printf("MSC: B__sig2:%d,%d,%d\n",stru.a,stru.b,stru.c);
113                   /* pričetek prehoda */
114                   b_B__sig2=true;
115                   b_B__sig2=false;
116                   goto s2
117                 ::atomic{recv__sig(!(a < 5),B__sig3) ->
118                   pv__ptr++}
119                 ::atomic{recv__sig((a < 5),B__sig3) ->
120                   update_chan_tab();
121                   input??B__sig3(sender,stru);
122                   }
123                   printf("MSC: B__sig3:%d,%d,%d\n",stru.a,stru.b,stru.c);
124                   b_B__sig3=true;
125                   b_B__sig3=false;
126                   goto s2
127                 ::atomic{recv__sig(true,B__sprejemnik__caso)->
128                   input??B__sprejemnik__caso(sender,stru);
129                   update_chan_tab();
130                   }
131                   printf("MSC: B__sprejemnik__caso:%d,%d,%d\n",
132                           stru.a,stru.b,stru.c);
133                   a = 0;
134                   b_B__sprejemnik__caso=true;
135                   b_B__sprejemnik__caso=false;
136                   goto s2
137                 /* implicitni sprejem preostalih signalov */
138                 :: else ->
139                    if
140                    ::atomic{recv__sig(true,B__prio2)->
141                      input??B__prio2(_,undefined__B__zapis);
142                      update_chan_tab();
143                      }
144                      printf("MSC: B__prio2 implicitno sprejet");
145                      goto s1
146                    ::else -> goto s1;
147                    fi
148                 /* konec implicitnega sprejem signalov */
149               fi;
150            fi
151         od;
152         /* konec sprejema najstarejšega signala */
153     fi;
154     goto s1;
155 /* potek časovnika s parametri */
```

```
156 ::atomic{expire(tmr__B__sprejemnik__caso) && (len(input)<cv__buff-1) ->
157     atomic{reset(tmr__B__sprejemnik__caso);
158            B__sprejemnik__caso__1.a=0;
159            B__sprejemnik__caso__1.b=0;
160            B__sprejemnik__caso__1.b=false;
161     }
162     input!B__sprejemnik__caso(_pid,B__sprejemnik__caso__1);
163     table_channum_nsp[input].data[table_channum_ptr[input]].name =
164       B__sprejemnik__caso;
165     table_channum_ptr[input]++}
166 od;
167
168
169 s2:
170 end_s2:
171 do
172 :: table_channum_ptr[input] > pv__cur  ->
173     table_channum_prio[input]=false;
174     pv__cur++;
175     pv__ptr=0;
176     atomic{
177     do
178     :: pv__ptr <= cv__buff-1 ->
179         if
180         :: set__prio(B__prio2)
181         :: set__prio(B__sprejemnik__caso)
182         :: else -> set__clear()
183         fi;
184         pv__ptr++
185     :: else -> goto s2_start
186     od;
187     }
188 s2_start:
189     if
190     :: table_channum_prio[input]==true ->
191         pv__ptr=0;
192         do
193         :: (pv__ptr <= cv__buff-1)
194            && (table_channum_nsp[input].data[pv__ptr].prio==true) ->
195            if
196            ::atomic{recv__sig(true,B__prio2) ->
197               input??B__prio2(sender,undefined__B__zapis);
198               update_chan_tab();
199              }
200               printf("MSC: B__prio2");
201               atomic{set(tmr__B__sprejemnik__caso,NOW+5);
202                      B__sprejemnik__caso__1.a=5;
203                      B__sprejemnik__caso__1.b=5;
204                      B__sprejemnik__caso__1.b=true;
205              }
206              b_B__prio2=true; b_B__prio2=false;
207              goto s1
208            ::atomic{recv__sig(true,B__sprejemnik__caso) ->
209                input??B__sprejemnik__caso(sender,stru);
210                update_chan_tab();
211              }
212              printf("MSC: B__sprejemnik__caso:%d,%d,%d\n",
213                      stru.a,stru.b,stru.c);
214              a=7;
215              b_B__sprejemnik__caso=true; b_B__sprejemnik__caso=false;
216              goto s1
217            fi;
218                break;
219         :: (pv__ptr == cv__buff) -> break;
220         :: else -> pv__ptr++;
221         od;
222     :: else ->
223         pv__ptr=0;
224         do
225         :: (pv__ptr <= cv__buff-1) ->
226            if
227            :: skip__save()
228            :: else ->
```

```
229                if
230                  ::atomic{recv__sig(true,B__sig1) ->
231                    input??B__sig1(sender,undefined__B__zapis);
232                    update_chan_tab();
233                    }
234                    printf("MSC: B__sig1");
235                    b_B__sig1=true; b_B__sig1=false;
236                    goto s1
237                  :: else ->
238                     if
239                     ::atomic{recv__sig(true,B__prio1);
240                       input??B__prio1(_,undefined__B__zapis);
241                       update_chan_tab();
242                       }
243                       printf("MSC: B__prio1 implicitno sprejet");
244                       goto s2
245                     ::atomic{recv__sig(true,B__sig3);
246                       input??B__sig3(_,undefined__B__zapis);
247                       update_chan_tab();
248                       }
249                       printf("MSC: B__sig3 implicitno sprejet");
250                       goto s2
251                     ::atomic{recv__sig(true,B__sig2);
252                       input??B__sig2(_,undefined__B__zapis);
253                       update_chan_tab();
254                       }
255                       printf("MSC: B__sig2 implicitno sprejet");
256                       goto s2
257                     ::else -> goto s2;
258                     fi
259                    fi;
260            fi
261         od;
262     fi;
263     goto s2;
264 :: atomic{expire(tmr__B__sprejemnik__caso) && (len(input)<cv__buff-1) ->
265    reset(tmr__B__sprejemnik__caso);
266    input!B__sprejemnik__caso(_pid,B__sprejemnik__caso__1);
267    table_channum_nsp[input].data[table_channum_ptr[input]].\
268      name=B__sprejemnik__caso;
269    table_channum_ptr[input]++};
270 :: atomic{ ((a==0) && !(recv__sig(true,B__prio2))
271             && !(recv__sig(true,B__sprejemnik__caso))
272             && !(expire(tmr__B__sprejemnik__caso)
273                  && (len(input)<cv__buff-1))
274                  && !(table_channum_ptr[input] > pv__cur))  ->
275    a=1;}
276    b_trajni=1;
277    b_trajni=0;
278 od;
279 }
280
281
282
283 proctype oddajnik(){
284 pt__pid offspring;
285 B__zapis stru;
286
287 run__proc(table__sprejemnik__free,
288           table__sprejemnik__max,
289           sprejemnik,chan__sprejemnik,
290           chan__sprejemnik__select);
291 atomic{
292   table_pid_channum[offspring]!B__sig1(_pid,undefined__B__zapis);
293   table_channum_nsp[table_pid_channum[offspring]].\
294   data[table_channum_ptr[table_pid_channum[offspring]]].name=B__sig1;
295   table_channum_ptr[table_pid_channum[offspring]]++;
296
297   table_pid_channum[offspring]!B__prio1(_pid,undefined__B__zapis);
298   table_channum_nsp[table_pid_channum[offspring]].\
299   data[table_channum_ptr[table_pid_channum[offspring]]].name=B__prio1;
300   table_channum_ptr[table_pid_channum[offspring]]++;
301
```

```
302   stru.a=2;
303   stru.b=2;
304   stru.c=true;
305   table_pid_channum[offspring]!B__sig2(_pid,stru);
306   table_channum_nsp[table_pid_channum[offspring]].\
307   data[table_channum_ptr[table_pid_channum[offspring]]].name=B__sig2;
308   table_channum_ptr[table_pid_channum[offspring]]++;
309
310   table_pid_channum[offspring]!B__prio2(_pid,undefined__B__zapis);
311   table_channum_nsp[table_pid_channum[offspring]].\
312   data[table_channum_ptr[table_pid_channum[offspring]]].name=B__prio2;
313   table_channum_ptr[table_pid_channum[offspring]]++;
314
315   stru.a=3;
316   stru.b=3;
317   stru.c=false;
318   table_pid_channum[offspring]!B__sig3(_pid,stru);
319   table_channum_nsp[table_pid_channum[offspring]].\
320   data[table_channum_ptr[table_pid_channum[offspring]]].name=B__sig3;
321   table_channum_ptr[table_pid_channum[offspring]]++;
322 }
323 }
324
325
326 init{
327 pid offspring;
328 atomic{
329 run__proc(table__oddajnik__free,
330           table__oddajnik__max,
331           oddajnik,chan__none,
332           chan__sprejemnik__select);
333
334 run__proc(table__sprejemnik__free,
335           table__sprejemnik__max,
336           sprejemnik,chan__sprejemnik,
337           chan__sprejemnik__select);
338 }
339 }
340
341 active proctype monitor(){
342 atomic{ (error) -> assert(false) }
343 }
```

C. Dodatek

Primeri avtomatske tvorbe modelov

C.1 Dopolnitev modela s sondami

C.1.1 Model sistema `ustvari`

```
1 #define pt__sig byte
2 #define pt__sig byte
3 #define pt__pid pid
4 #define pt__chan chan
5 #define cv__buff 3
6 #define cv__procnum 6
7 #define pcv__none 0
8 #define pcv__null 0
9
10 hidden int pfv_tmp;
11 bool pv__run_time_error=false;
12
13 /*     INCLUDE  SYNONYM PODATKOVNIH TIPOV  */
14
15 /*     INCLUDE  SYNTYPE PODATKOVNIH TIPOV  */
16
17 /*     INCLUDE  NEWTYPE PODATKOVNIH TIPOV  */
18
19 /*     INCLUDE  LITERALOV  */
20
21 /*     INCLUDE  DEFINIRANIH SIGNALOV  */
22 /*****************************************************************************/
23 /*                                                                           */
24 /*                        Sdl2Promela translator 1.0                         */
25 /*                                                                           */
26 /* AUTHOR:      A. Vreze, B. Vlaovic, University of Maribor, Slovenia        */
27 /* FILENAME:      signals.ph                                                 */
28 /*                                                                           */
29 /*                                                                           */
30 /*****************************************************************************/
31
32 #define B__dummy 0
33
34 /*     INCLUDE  DEFINIRANIH KANALOV  */
35 /*****************************************************************************/
36 /*                                                                           */
37 /*                        Sdl2Promela translator 1.0                         */
38 /*                                                                           */
39 /* AUTHOR:      A. Vreze, B. Vlaovic, University of Maribor, Slovenia        */
40 /* FILENAME:      channels.ph                                                */
41 /*                                                                           */
42 /*                                                                           */
43 /*****************************************************************************/
44
45 #define CHAN_BUFFER 3
46 chan chan__env = [CHAN_BUFFER] of {pt__sig,pt__pid}
47 #define number_of_process 6
48 #define number_of_channels 1
49
```

```
50
51 #define table__B__mux__max 1
52 hidden byte table__B__mux__free;
53 #define max__table__B__mux 1
54 #define chan__B__mux__select 1
55 chan chan__B__mux[table__B__mux__max] = [CHAN_BUFFER] of {pt__sig,pt__pid};
56
57 #define table__B__client__max 2
58 hidden byte table__B__client__free;
59 #define max__table__B__client 2
60 #define chan__B__client__select 2
61 chan chan__B__client[table__B__client__max] = [CHAN_BUFFER] of {pt__sig,pt__pid};
62
63
64
65 /***** Tables for the channel monitoring BEGIN *****/
66 typedef pt__chandat{pt__sig name; bool save; bool prio} ;
67 typedef pt__chandata{pt__chandat data[cv__buff]};
68 pt__chan table_channame_channum[cv__procnum];
69 pt__chan table_pid_channum[cv__procnum];
70 pt__sig table_pid_channame[cv__procnum];
71 byte table_channum_ptr[cv__procnum];
72 pt__chandata table_channum_nsp[cv__procnum];
73 bool table_channum_prio[cv__procnum];
74
75 /***** Tables for the channel monitoring END *****/
76
77 /**** Update of the channel monitoring tables BEGIN ****/
78 inline update_chan_tab(){
79 /* popravki vhodne tabele */
80 atomic{
81 do
82 ::(pv__ptr < cv__buff-1) ->
83   table_channum_nsp[input].data[pv__ptr].name=
84     table_channum_nsp[input].data[pv__ptr+1].name;
85   table_channum_nsp[input].data[pv__ptr].save=
86     table_channum_nsp[input].data[pv__ptr+1].save;
87   table_channum_nsp[input].data[pv__ptr].prio=
88     table_channum_nsp[input].data[pv__ptr+1].prio;
89   pv__ptr++
90 :: else ->
91   table_channum_nsp[input].data[pv__ptr].name=pcv__none;
92   table_channum_nsp[input].data[pv__ptr].save=false;
93   table_channum_nsp[input].data[pv__ptr].prio=false;
94   break
95 od;
96 table_channum_ptr[input]--;
97 pv__cur=0;
98 }
99 }
100 /**** Update of the channel monitoring tables END ****/
101
102 inline set__prio(sig){
103   table_channum_nsp[input].data[pv__ptr].name==sig ->
104   atomic{
105     table_channum_nsp[input].data[pv__ptr].save=false;
106     table_channum_nsp[input].data[pv__ptr].prio=true;
107     table_channum_prio[input]=true
108   }
109 }
110 inline set__save(sig){
111   table_channum_nsp[input].data[pv__ptr].name==sig ->
112   atomic{
113     table_channum_nsp[input].data[pv__ptr].save=true;
114     table_channum_nsp[input].data[pv__ptr].prio=false;
115   }
116 }
117
118 inline set__clear(){
119   atomic{
120     table_channum_nsp[input].data[pv__ptr].save=false;
121     table_channum_nsp[input].data[pv__ptr].prio=false;
122   }
```

```
123 }
124 #define recv__sig(pog,sig)\
125  ((pog) && table_channum_nsp[input].data[pv__ptr].name==sig)
126
127 inline skip__save(){
128   table_channum_nsp[input].data[pv__ptr].save==true -> pv__ptr++
129 }
130
131 /***** Tables for the channel monitoring END *****/
132
133
134 /*****************************************************************************/
135 /*                                                                           */
136 /*                                Process init                               */
137 /*                                                                           */
138 /*                                                                           */
139 /*****************************************************************************/
140 /* F. PAR:                                                                   */
141 /* KOMENTAR:                                                                 */
142 /* LOKACIJA:                                                                 */
143 /* MAX:      1                                                               */
144 /* MIN:      1                                                               */
145 /*                                                                           */
146 /*****************************************************************************/
147 init{
148 pt__pid offspring;
149         if
150         :: table__B__mux__free < table__B__mux__max ->
151            offspring = run
152            B__mux(chan__B__mux[table__B__mux__free],_pid);
153            table_pid_channum[offspring] = chan__B__mux[table__B__mux__free];
154            table_pid_channame[offspring] = chan__B__mux__select +
155            table__B__mux__free;
156            table_channame_channum[chan__B__mux__select+table__B__mux__free] =
157            chan__B__mux[table__B__mux__free];
158            if
159            ::(offspring==0) -> pv__run_time_error = true;
160            ::(offspring!=0) -> table__B__mux__free++;
161            fi;
162            :: else  -> pv__run_time_error = true;
163         fi;
164 }
165
166 /*****************************************************************************/
167 /*                                                                           */
168 /*                               Process B__mux                              */
169 /*                                                                           */
170 /*                                                                           */
171 /*****************************************************************************/
172 /* F. PAR:                                                                   */
173 /* KOMENTAR:                                                                 */
174 /* LOKACIJA:      B                                                          */
175 /* MAX:      1                                                               */
176 /* MIN:      1                                                               */
177 /*                                                                           */
178 /*****************************************************************************/
179 proctype B__mux(pt__chan input; pt__pid parent){
180 pt__pid offspring, sender;
181 byte pv__ptr, pv__cur;
182 xr input;
183
184
185 atomic{
186    if
187    :: table__B__client__free < table__B__client__max ->
188    offspring = run
189    B__client(chan__B__client[table__B__client__free],_pid);
190    table_pid_channum[offspring] = chan__B__client[table__B__client__free];
191    table_pid_channame[offspring] =    chan__B__client__select + \
192    table__B__client__free;
193    table_channame_channum[chan__B__client__select    + table__B__client__free] =
194    chan__B__client[table__B__client__free];
195    if
```

```
196        ::(offspring==0) -> pv__run_time_error = true;
197        ::(offspring!=0) -> table__B__client__free++;
198     fi;
199     :: else  -> pv__run_time_error = true;
200     fi;
201     }
202 atomic{
203     if
204     :: table__B__client__free < table__B__client__max ->
205     offspring = run
206     B__client(chan__B__client[table__B__client__free],_pid);
207     table_pid_channum[offspring] = chan__B__client[table__B__client__free];
208     table_pid_channame[offspring] =    chan__B__client__select + \
209     table__B__client__free;
210     table_channame_channum[chan__B__client__select    + table__B__client__free] =
211     chan__B__client[table__B__client__free];
212     if
213        ::(offspring==0) -> pv__run_time_error = true;
214        ::(offspring!=0) -> table__B__client__free++;
215     fi;
216     :: else  -> pv__run_time_error = true;
217     fi;
218     }
219 atomic{
220     if
221     :: table__B__client__free < table__B__client__max ->
222     offspring = run
223     B__client(chan__B__client[table__B__client__free],_pid);
224     table_pid_channum[offspring] = chan__B__client[table__B__client__free];
225     table_pid_channame[offspring] =    chan__B__client__select + \
226     table__B__client__free;
227     table_channame_channum[chan__B__client__select    + table__B__client__free] =
228     chan__B__client[table__B__client__free];
229     if
230        ::(offspring==0) -> pv__run_time_error = true;
231        ::(offspring!=0) -> table__B__client__free++;
232     fi;
233     :: else  -> pv__run_time_error = true;
234     fi;
235     }
236 goto mux_endstate;
237 mux_endstate:
238 end_mux_endstate:
239 do
240 :: table_channum_ptr[input] > pv__cur ->
241     table_channum_prio[input]=false;
242     pv__cur++;
243     pv__ptr=0;
244     atomic{
245     do
246     :: pv__ptr <= cv__buff-1 ->
247        if
248        :: else -> set__clear();
249        fi;
250        pv__ptr++ ;
251     :: else -> goto mux_endstate_start
252     od;
253     }
254 mux_endstate_start:
255     if
256     :: table_channum_prio[input]==true ->
257        pv__ptr=0;
258        do
259        :: (pv__ptr <= cv__buff-1)
260        && (table_channum_nsp[input].data[pv__ptr].prio==true) ->
261           if
262           /* PRIORITY INPUT */
263           :: else  -> skip;
264           fi;
265        :: (pv__ptr == cv__buff) -> break;
266        :: else -> pv__ptr++
267        od;
268     :: else ->
```

```
269        pv__ptr=0;
270        do
271        :: (pv__ptr <= cv__buff-1) ->
272           if
273           :: skip__save()
274           :: else ->
275              if
276              /* INPUT SIGNALS */
277              ::atomic{recv__sig(true, B__dummy) ->
278              input??B__dummy(sender);
279              update_chan_tab();
280              }
281              goto end;
282              :: else ->
283                 if
284                 /* IMPLICITNI PREHOD */
285                 :: else -> goto mux_endstate;
286                 fi;
287              fi;
288           fi;
289        od;
290     fi;
291     goto mux_endstate;
292     /* SPONTANEOUS TRANSITION */
293     /* TIMERS */
294     /* TRAJEN SIGNAL */
295  od;
296
297  end: skip
298  }
299
300  /*****************************************************************************/
301  /*                                                                           */
302  /*                              Process B__client                            */
303  /*                                                                           */
304  /*                                                                           */
305  /*****************************************************************************/
306  /* F. PAR:                                                                   */
307  /* KOMENTAR:                                                                 */
308  /* LOKACIJA:      B                                                          */
309  /* MAX:      2                                                               */
310  /* MIN:      0                                                               */
311  /*                                                                           */
312  /*****************************************************************************/
313  proctype B__client(pt__chan input; pt__pid parent){
314  pt__pid offspring, sender;
315  byte pv__ptr, pv__cur;
316  xr input;
317
318
319  goto client_endstate;
320  client_endstate:
321  end_client_endstate:
322  do
323  :: table_channum_ptr[input] > pv__cur ->
324     table_channum_prio[input]=false;
325     pv__cur++;
326     pv__ptr=0;
327     atomic{
328     do
329     :: pv__ptr <= cv__buff-1 ->
330        if
331        :: else -> set__clear();
332        fi;
333        pv__ptr++ ;
334     :: else -> goto client_endstate_start
335     od;
336     }
337  client_endstate_start:
338     if
339     :: table_channum_prio[input]==true ->
340        pv__ptr=0;
341        do
```

```
342       :: (pv__ptr <= cv__buff-1)
343       && (table_channum_nsp[input].data[pv__ptr].prio==true) ->
344          if
345          /* PRIORITY INPUT */
346          :: else  -> skip;
347          fi;
348       :: (pv__ptr == cv__buff) -> break;
349       :: else -> pv__ptr++
350       od;
351    :: else ->
352       pv__ptr=0;
353       do
354       :: (pv__ptr <= cv__buff-1) ->
355          if
356          :: skip__save()
357          :: else ->
358             if
359             /* INPUT SIGNALS */
360             ::atomic{recv__sig(true, B__dummy) ->
361             input??B__dummy(sender);
362             update_chan_tab();
363             }
364             goto end;
365             :: else ->
366                if
367                /* IMPLICITNI PREHOD */
368                :: else -> goto client_endstate;
369                fi;
370             fi;
371          fi;
372       od;
373    fi;
374    goto client_endstate;
375    /* SPONTANEOUS TRANSITION */
376    /* TIMERS */
377    /* TRAJEN SIGNAL */
378 od;
379
380 end: skip
381 }
382
383 /*****************************************************************************/
384 /*                                                                           */
385 /*                              Process monitor()                            */
386 /*                                                                           */
387 /*                                                                           */
388 /*****************************************************************************/
389 /* F. PAR:                                                                   */
390 /* KOMENTAR:                                                                 */
391 /* LOKACIJA:                                                                 */
392 /* MAX:      1                                                               */
393 /* MIN:      1                                                               */
394 /*                                                                           */
395 /*****************************************************************************/
396 active proctype monitor(){
397 end: do ::pv__run_time_error==true->assert(false) od
398 }
```

C.2 Protokol V.76

C.2.1 Specifikacija sistema v jeziku SDL

```
1 SYSTEM V76test;
2   SIGNAL
3     /* V.76 primitives L-ESTABLISH etc.*/
4     L_EstabReq(DLCident),
5     L_EstabInd(DLCident),
6     L_EstabResp,
7     L_EstabConf(DLCident),
8     L_SetparmReq,
9     L_SetparmInd,
```

```
10      L_SetparmResp,
11      L_SetparmConf,
12      L_ReleaseReq(DLCident),
13      L_ReleaseInd(DLCident),
14      L_DataReq(DLCident, Integer),
15      L_DataInd(DLCident, Integer);
16
17    SIGNAL
18      /* V.76 commands and responses. */
19      V76frame(V76paramTyp);
20
21    /* Service User to DLC: */
22    SIGNALLIST su2dlc=
23      L_EstabReq,
24      L_EstabResp,
25      L_SetparmReq,
26      L_SetparmResp,
27      L_ReleaseReq,
28      L_DataReq;
29
30    /* DLC to Service User: */
31    SIGNALLIST dlc2su=
32      L_EstabInd,
33      L_EstabConf,
34      L_SetparmInd,
35      L_SetparmConf,
36      L_ReleaseInd,
37      L_DataInd;
38    NEWTYPE V76paramTyp
39      STRUCT
40        presen T_CHOICE;
41        I  Iframe;
42        SABME  SABMEframe;
43        DM  DMframe;
44        DISC  DISCframe;
45        UA  UAframe;
46        XIDcmd  XIDframe;
47        XIDresp  XIDframe;
48    ENDNEWTYPE;
49
50    NEWTYPE T_CHOICE
51     LITERALS I, SABME, DM, DISC,
52                        UA, XIDcmd,XIDresp
53    ENDNEWTYPE;
54
55    SYNONYM maxDLC Integer = 1;
56
57    SYNTYPE XIDframe = Integer
58    ENDSYNTYPE;
59
60    /* DLC Identifier: */
61    SYNTYPE DLCident =
62      Integer CONSTANTS 0 : maxDLC
63    ENDSYNTYPE;
64    NEWTYPE Iframe
65      STRUCT
66        DLCi DLCident;
67        data Integer;
68        CRC Integer;
69    ENDNEWTYPE;
70
71    NEWTYPE SABMEframe
72      STRUCT
73        DLCi  DLCident;
74    ENDNEWTYPE;
75
76    NEWTYPE DMframe
77      STRUCT
78        DLCi  DLCident;
79    ENDNEWTYPE;
80
81    NEWTYPE DISCframe
82      STRUCT
```

```
83        DLCi  DLCident;
84    ENDNEWTYPE;
85
86    NEWTYPE UAframe
87      STRUCT
88        DLCi  DLCident;
89    ENDNEWTYPE;
90    /* Simplified V76 model. */
91    CHANNEL DLCaSU
92      FROM ENV TO DLCa WITH (su2dlc) ;
93      FROM DLCa TO ENV WITH (dlc2su) ;
94    ENDCHANNEL;
95    CHANNEL DLCaDL
96      FROM DLCa TO dataLink WITH V76frame ;
97      FROM dataLink TO DLCa WITH V76frame ;
98    ENDCHANNEL;
99    CHANNEL DLCbDL
100     FROM DLCb TO dataLink WITH V76frame ;
101     FROM dataLink TO DLCb WITH V76frame ;
102   ENDCHANNEL;
103   CHANNEL DLCbSU
104     FROM ENV TO DLCb WITH (su2dlc) ;
105     FROM DLCb TO ENV WITH (dlc2su) ;
106   ENDCHANNEL;
107   PROCEDURE CRCok
108     COMMENT 'This procedure
109   checks the CRC'; FPAR crc1 Integer; RETURNS Boolean;
110     START
111       COMMENT 'Simplified version,
112     returns True if crc1
113     is positive or null.';
114       RETURN
115   ENDPROCEDURE;
116   BLOCK dataLink;
117     /* This block models a
118        simplified data-link layer. */
119     SIGNALROUTE fromA
120       FROM ENV TO AtoB WITH V76frame ;
121     SIGNALROUTE toB
122       FROM AtoB TO ENV WITH V76frame ;
123     SIGNALROUTE fromB
124       FROM ENV TO BtoA WITH V76frame ;
125     SIGNALROUTE toA
126       FROM BtoA TO ENV WITH V76frame ;
127     CONNECT DLCaDL AND fromA, toA;
128     CONNECT DLCbDL AND toB, fromB;
129     PROCESS AtoB;
130       /* Retransmission to peer
131          of received signals,
132          plus introducing faults. */
133
134       DCL
135         V76par V76paramTyp;
136       START ;
137         NEXTSTATE ready;
138       STATE ready;
139         INPUT V76frame(V76par);
140           DECISION ANY;
141             (  ):
142               OUTPUT V76frame(V76par)
143               VIA toB;
144             (  ):
145           ENDDECISION;
146           NEXTSTATE -;
147       ENDSTATE;
148     ENDPROCESS;
149     PROCESS BtoA;
150       /* Retransmission to peer
151          of received signals,
152          plus introducing faults. */
153
154       DCL
155         V76par V76paramTyp;
```

```
156       START ;
157         NEXTSTATE ready;
158       STATE ready;
159         INPUT V76frame
160         (V76par);
161           DECISION ANY;
162             (  ):
163               OUTPUT V76frame
164               (V76par)
165               VIA toA;
166             (  ):
167           ENDDECISION;
168           NEXTSTATE -;
169       ENDSTATE;
170     ENDPROCESS;
171   ENDBLOCK;
172   BLOCK DLCa;
173     SIGNAL
174        DLCstopped(DLCident);
175     /* The route dlcDL has been splitted in two because of a
176      restriction in Tau SDL Suite */
177     SIGNALROUTE DLCs
178       FROM dispatch TO DLC WITH L_DataReq,
179     L_ReleaseReq,
180     V76frame;
181       FROM DLC TO dispatch WITH DLCstopped  ;
182     SIGNALROUTE dlcDL
183       FROM dispatch TO ENV WITH V76frame ;
184     SIGNALROUTE peer
185       FROM DLC TO ENV WITH V76frame ;
186     SIGNALROUTE dlcDL2
187       FROM ENV TO dispatch WITH V76frame ;
188     SIGNALROUTE dlcSU
189       FROM dispatch TO ENV WITH (dlc2su) ;
190       FROM ENV TO dispatch WITH (su2dlc) ;
191     SIGNALROUTE user
192       FROM DLC TO ENV WITH L_DataInd,
193     L_EstabConf  ;
194     CONNECT DLCaDL AND dlcDL, peer, dlcDL2;
195     CONNECT DLCaSU AND dlcSU, user;
196     PROCESS dispatch(1, 1);
197       /* Temporary variables: */
198       DCL
199         DLCnum, DLCpeer DLCident,
200         uData Integer,
201         V76para V76paramTyp;
202
203       NEWTYPE DLCsArray
204         ARRAY(DLCident, PID)
205       ENDNEWTYPE;
206
207       DCL
208         /* to store the PIDs of the instances
209         of process DLC, necessary in
210         outputs to route signals : */
211         DLCs DLCsArray;
212       START ;
213         NEXTSTATE ready;
214       STATE ready;
215         INPUT V76frame
216         (V76para);
217           DECISION V76para ! presen;
218             ( SABME ):
219               TASK DLCpeer:=
220               V76para
221               ! SABME
222               ! DLCi;
223               DECISION DLCs(DLCpeer);
224                 ( NULL ):
225                   OUTPUT L_EstabInd
226                   (DLCpeer);
227                   NEXTSTATE waitEstabResp;
228                 ELSE:
```

```
229                 OUTPUT L_ReleaseInd
230                 (DLCpeer);
231                 TASK V76para!DM:=
232                 (. DLCpeer .);
233                 TASK V76para!presen:=DM;
234                 OUTPUT V76frame
235                 (V76para)
236                 VIA dlcDL;
237                 NEXTSTATE -;
238             ENDDECISION;
239           ( UA ):
240             TASK V76para!presen:=UA;
241             OUTPUT V76frame
242             (V76para)
243             TO DLCs
244             (V76para !
245             UA ! DLCi);
246             NEXTSTATE -;
247           ( I ):
248             TASK V76para!presen:=I;
249             OUTPUT V76frame
250             (V76para)
251             TO DLCs
252             (V76para !
253             I ! DLCi)
254               COMMENT 'We pass the frame
255             to the corresponding
256             instance of process DLC';
257             NEXTSTATE -;
258           ( DM ):
259             TASK V76para!presen:=DM;
260             OUTPUT V76frame
261             (V76para)
262             TO DLCs
263             (V76para !
264             DM ! DLCi);
265             NEXTSTATE -;
266           ( DISC ):
267             TASK V76para!presen:=DISC;
268             OUTPUT V76frame
269             (V76para)
270             TO DLCs
271             (V76para !
272             DISC ! DLCi);
273             NEXTSTATE -;
274           ( XIDcmd ):
275             TASK V76para!presen:=XIDcmd;
276             OUTPUT L_SetparmInd;
277             NEXTSTATE waitParmResp;
278           ( XIDresp ):
279             TASK V76para!presen:=XIDresp;
280             OUTPUT L_SetparmConf;
281             NEXTSTATE -;
282           ELSE:
283             NEXTSTATE -;
284         ENDDECISION;
285     ENDSTATE;
286     STATE waitEstabResp;
287       INPUT L_EstabResp;
288         CREATE DLC
289         (DLCpeer,
290         False)
291           COMMENT 'Creates
292         instance of
293         process DLC';
294         TASK DLCs
295         (DLCpeer)
296         :=
297         OFFSPRING
298           COMMENT 'Stores into the
299         table the PID of the
300         instance just created.';
301         NEXTSTATE ready;
```

```
302         SAVE V76frame
303           COMMENT 'Saves
304         V76frame';
305       ENDSTATE;
306       STATE waitParmResp;
307         INPUT L_SetparmResp;
308           TASK V76para!XIDresp:=0;
309           TASK V76para!presen:=XIDresp;
310           OUTPUT V76frame
311           (V76para)
312           VIA dlcDL;
313           NEXTSTATE ready;
314       ENDSTATE;
315       STATE waitUA;
316         INPUT V76frame
317         (V76para);
318           DECISION V76para ! presen;
319             ( UA ):
320               OUTPUT V76frame
321               (V76para)
322               TO DLCs
323               (V76para
324               ! UA ! DLCi);
325               NEXTSTATE ready;
326             ELSE:
327               NEXTSTATE -;
328           ENDDECISION;
329       ENDSTATE;
330       STATE ready;
331         INPUT L_DataReq
332         (DLCnum,
333         uData);
334           DECISION DLCs
335           (DLCnum);
336             ( NULL
337             /* DLCnum
338             not
339             setup  */ ):
340             ELSE:
341               OUTPUT L_DataReq
342               (DLCnum,
343               uData)
344                TO DLCs
345               (DLCnum)
346                 COMMENT 'We pass the frame
347               to the corresponding
348               instance of process DLC';
349           ENDDECISION;
350           NEXTSTATE -;
351         PRIORITY INPUT L_ReleaseReq
352         (DLCnum)
353           COMMENT '
354
355
356
357
358
359           This is a priority input ';
360           DECISION DLCs
361           (DLCnum);
362             ( NULL
363             /* DLCnum
364             not
365             setup  */ ):
366             ELSE:
367               OUTPUT L_ReleaseReq
368               (DLCnum)
369               TO
370               DLCs
371               (DLCnum);
372           ENDDECISION;
373           NEXTSTATE -;
374         INPUT L_EstabReq
```

```
375         (DLCnum);
376           DECISION DLCs
377           (DLCnum);
378             ( NULL ):
379               CREATE DLC
380               (DLCnum,
381               True)
382                 COMMENT 'DLCnum not
383               used, we create
384               an instance of
385               process DLC';
386               TASK DLCs
387               (DLCnum)
388               :=
389               OFFSPRING
390                 COMMENT 'We store into the
391               table the PID of
392               the new instance';
393             ELSE:
394               OUTPUT L_ReleaseInd
395               (DLCnum);
396           ENDDECISION;
397           NEXTSTATE waitUA;
398         INPUT L_SetparmReq;
399           TASK V76para!XIDcmd:=0;
400           TASK V76para!presen:=XIDcmd;
401           OUTPUT V76frame
402           (V76para)
403           VIA dlcDL;
404           NEXTSTATE -;
405         INPUT DLCstopped
406         (DLCnum);
407           OUTPUT L_ReleaseInd
408           (DLCnum);
409           TASK DLCs
410           (DLCnum)
411           := NULL;
412           NEXTSTATE -;
413       ENDSTATE;
414     ENDPROCESS;
415     PROCESS DLC (0, maxDLC + 1)
416       COMMENT 'Initially 0 instances,
417     maximum maxDLC+1
418     instances in parallel,
419     created by dispatch';FPAR me DLCident, originator Boolean;
420       TIMER
421               T320 := 12.0;
422
423         DCL
424           /* T320 retransmission counter: */
425           N320cnt Integer;
426
427         /* Maximum retries: */
428         SYNONYM N320 Integer = 3;
429
430         /* Temporary variables: */
431         DCL
432           uData Integer,
433           Iparam Iframe,
434           V76para V76paramTyp;
435       START ;
436         DECISION originator;
437           ( True ):
438             TASK N320cnt:= 0;
439           retry:
440             TASK V76para!SABME:= (. me .);
441             TASK V76para!presen:=SABME;
442             OUTPUT V76frame
443             (V76para)
444             VIA peer;
445             SET (T320);
446             NEXTSTATE waitUA;
447           ( False ):
```

```
448             TASK V76para!UA:=(. me .);
449             TASK V76para!presen:=UA;
450             OUTPUT V76frame
451             (V76para)
452             VIA peer;
453             NEXTSTATE connected;
454         ENDDECISION;
455       STATE waitUA;
456         INPUT V76frame
457         (V76para);
458           DECISION V76para ! presen;
459             ( UA ):
460               RESET (T320);
461               OUTPUT L_EstabConf
462               (me);
463               NEXTSTATE connected;
464             ( DM ):
465               OUTPUT DLCstopped
466               (me);
467               STOP ;
468             ELSE:
469               NEXTSTATE -;
470           ENDDECISION;
471         INPUT T320;
472           DECISION N320cnt < N320;
473             ( True ):
474               TASK N320cnt
475               :=
476               N320cnt +
477               1;
478               JOIN retry;
479             ELSE:
480               OUTPUT DLCstopped
481               (me);
482               STOP ;
483           ENDDECISION;
484       ENDSTATE;
485       STATE connected;
486         INPUT L_DataReq
487         ( ,
488         uData)
489           COMMENT 'We do not
490         get the first
491         parameter';
492           TASK Iparam := (. me,
493            uData, 15 .);
494           TASK V76para!I:=Iparam;
495           TASK V76para!presen:=I;
496           OUTPUT V76frame
497           (V76para)
498           VIA peer;
499           NEXTSTATE -;
500         INPUT L_ReleaseReq;
501           TASK V76para!DISC:=(. me .);
502           TASK V76para!presen:=DISC;
503           OUTPUT V76frame
504           (V76para)
505           VIA peer;
506           NEXTSTATE waitUAdisc;
507         INPUT V76frame
508         (V76para);
509           DECISION V76para ! presen;
510             ( DISC ):
511               TASK V76para!UA:=(. me .);
512               TASK V76para!presen:=UA;
513               OUTPUT V76frame
514               (V76para)
515               VIA peer;
516               OUTPUT DLCstopped
517               (me);
518               STOP ;
519             ( I ):
520               DECISION CALL
```

```
521              CRCok
522              (V76para
523              ! I
524              !CRC);
525                ( True ):
526                  OUTPUT L_DataInd
527                  (me,
528                  V76para! I
529                  ! data);
530                ( False ):
531              ENDDECISION;
532              NEXTSTATE -;
533            ELSE:
534              NEXTSTATE -;
535          ENDDECISION;
536      ENDSTATE;
537      STATE waitUAdisc;
538        INPUT V76frame
539        (V76para);
540          DECISION V76para ! presen;
541            ( UA, DM ):
542              OUTPUT DLCstopped
543              (me);
544              STOP ;
545            ELSE:
546              NEXTSTATE -;
547          ENDDECISION;
548      ENDSTATE;
549    ENDPROCESS;
550  ENDBLOCK;
551  BLOCK DLCb;
552    SIGNAL
553       DLCstopped(DLCident);
554
555     /* The route dlcDL has been
556     splitted in two because of a
557     restriction in Tau SDL Suite */
558    SIGNALROUTE DLCs
559      FROM dispatch TO DLC WITH L_DataReq,
560    L_ReleaseReq,
561    V76frame;
562      FROM DLC TO dispatch WITH DLCstopped;
563    SIGNALROUTE dlcDL
564      FROM dispatch TO ENV WITH V76frame;
565    SIGNALROUTE peer
566      FROM DLC TO ENV WITH V76frame;
567    SIGNALROUTE dlcDL2
568      FROM ENV TO dispatch WITH V76frame;
569    SIGNALROUTE dlcSU
570      FROM dispatch TO ENV WITH (dlc2su);
571      FROM ENV TO dispatch WITH (su2dlc);
572    SIGNALROUTE user
573      FROM DLC TO ENV WITH L_DataInd,
574    L_EstabConf;
575    CONNECT DLCbDL AND dlcDL, peer, dlcDL2;
576    CONNECT DLCbSU AND dlcSU, user;
577    PROCESS dispatch(1, 1);
578      /* Temporary variables: */
579      DCL
580        DLCnum, DLCpeer DLCident,
581        uData Integer,
582        V76para V76paramTyp;
583
584      NEWTYPE DLCsArray
585        ARRAY(DLCident, PID)
586      ENDNEWTYPE;
587
588      DCL
589        /* to store the PIDs of the instances
590        of process DLC, necessary in
591        outputs to route signals : */
592        DLCs DLCsArray;
593      START ;
```

```
594         NEXTSTATE ready;
595       STATE ready;
596         INPUT V76frame
597         (V76para);
598           DECISION V76para ! presen;
599             ( SABME ):
600               TASK DLCpeer:=
601               V76para
602               ! SABME
603               ! DLCi;
604               DECISION DLCs(DLCpeer);
605                 ( NULL ):
606                   OUTPUT L_EstabInd
607                   (DLCpeer);
608                   NEXTSTATE waitEstabResp;
609                 ELSE:
610                   OUTPUT L_ReleaseInd
611                   (DLCpeer);
612                   TASK V76para!DM:=
613                   (. DLCpeer .);
614                   TASK V76para!presen:=DM;
615                   OUTPUT V76frame
616                   (V76para)
617                   VIA dlcDL;
618                   NEXTSTATE -;
619               ENDDECISION;
620             ( UA ):
621               TASK V76para!presen:=UA;
622               OUTPUT V76frame
623               (V76para)
624               TO DLCs
625               (V76para !
626               UA ! DLCi);
627               NEXTSTATE -;
628             ( I ):
629               TASK V76para!presen:=I;
630               OUTPUT V76frame
631               (V76para)
632               TO DLCs
633               (V76para !
634               I ! DLCi)
635                 COMMENT 'We pass the frame
636               to the corresponding
637               instance of process DLC';
638               NEXTSTATE -;
639             ( DM ):
640               TASK V76para!presen:=DM;
641               OUTPUT V76frame
642               (V76para)
643               TO DLCs
644               (V76para !
645               DM ! DLCi);
646               NEXTSTATE -;
647             ( DISC ):
648               TASK V76para!presen:=DISC;
649               OUTPUT V76frame
650               (V76para)
651               TO DLCs
652               (V76para !
653               DISC ! DLCi);
654               NEXTSTATE -;
655             ( XIDcmd ):
656               TASK V76para!presen:=XIDcmd;
657               OUTPUT L_SetparmInd;
658               NEXTSTATE waitParmResp;
659             ( XIDresp ):
660               TASK V76para!presen:=XIDresp;
661               OUTPUT L_SetparmConf;
662               NEXTSTATE -;
663             ELSE:
664               NEXTSTATE -;
665           ENDDECISION;
666       ENDSTATE;
```

```
667       STATE waitEstabResp;
668         INPUT L_EstabResp;
669           CREATE DLC
670           (DLCpeer,
671           False)
672             COMMENT 'Creates
673           instance of
674           process DLC';
675           TASK DLCs
676           (DLCpeer)
677           :=
678           OFFSPRING
679             COMMENT 'Stores into the
680           table the PID of the
681           instance just created.';
682           NEXTSTATE ready;
683         SAVE V76frame
684           COMMENT 'Saves
685         V76frame';
686       ENDSTATE;
687       STATE waitParmResp;
688         INPUT L_SetparmResp;
689           TASK V76para!XIDresp:= 0;
690           TASK V76para!presen:=XIDresp;
691           OUTPUT V76frame
692           (V76para)
693           VIA dlcDL;
694           NEXTSTATE ready;
695       ENDSTATE;
696       STATE ready;
697         INPUT L_DataReq
698         (DLCnum,
699         uData);
700           DECISION DLCs
701           (DLCnum);
702             ( NULL
703             /* DLCnum
704             not
705             setup  */ ):
706             ELSE:
707               OUTPUT L_DataReq
708               (DLCnum,
709               uData)
710                TO DLCs
711               (DLCnum)
712                 COMMENT 'We pass the frame
713               to the corresponding
714               instance of process DLC';
715           ENDDECISION;
716           NEXTSTATE -;
717         PRIORITY INPUT L_ReleaseReq
718         (DLCnum)
719           COMMENT '
720
721
722
723
724
725           This is a priority input ';
726           DECISION DLCs
727           (DLCnum);
728             ( NULL
729             /* DLCnum
730             not
731             setup  */ ):
732             ELSE:
733               OUTPUT L_ReleaseReq
734               (DLCnum)
735               TO
736               DLCs
737               (DLCnum);
738           ENDDECISION;
739           NEXTSTATE -;
```

```
740         INPUT L_EstabReq
741         (DLCnum);
742           DECISION DLCs
743           (DLCnum);
744             ( NULL ):
745               CREATE DLC
746               (DLCnum,
747               True)
748                 COMMENT 'DLCnum not
749               used, we create
750               an instance of
751               process DLC';
752               TASK DLCs
753               (DLCnum)
754               :=
755               OFFSPRING
756                 COMMENT 'We store into the
757               table the PID of
758               the new instance';
759             ELSE:
760               OUTPUT L_ReleaseInd
761               (DLCnum);
762           ENDDECISION;
763           NEXTSTATE waitUA;
764       ENDSTATE;
765       STATE ready;
766         INPUT DLCstopped
767         (DLCnum);
768           OUTPUT L_ReleaseInd
769           (DLCnum);
770           TASK DLCs
771           (DLCnum)
772           := NULL;
773           NEXTSTATE -;
774         INPUT L_SetparmReq;
775           TASK V76para!XIDcmd:=0;
776           TASK V76para!presen:=XIDcmd;
777           OUTPUT V76frame
778           (V76para)
779           VIA dlcDL;
780           NEXTSTATE -;
781       ENDSTATE;
782       STATE waitUA;
783         INPUT V76frame
784         (V76para);
785           DECISION V76para ! presen;
786             ( UA ):
787               OUTPUT V76frame
788               (V76para)
789               TO DLCs
790               (V76para
791               ! UA ! DLCi);
792               NEXTSTATE ready;
793             ELSE:
794               NEXTSTATE -;
795           ENDDECISION;
796       ENDSTATE;
797     ENDPROCESS;
798     PROCESS DLC (0, maxDLC + 1)
799       COMMENT 'Initially 0 instances,
800     maximum maxDLC+1
801     instances in parallel,
802     created by dispatch';FPAR me DLCident, originator Boolean;
803       TIMER
804               T320 := 12.0;
805
806           DCL
807             /* T320 retransmission counter: */
808             N320cnt Integer;
809
810           /* Maximum retries: */
811           SYNONYM N320 Integer = 3;
812
```

```
813          /* Temporary variables: */
814          DCL
815            uData Integer,
816            Iparam Iframe,
817            V76para V76paramTyp;
818       START ;
819         DECISION originator;
820           ( True ):
821             TASK N320cnt:= 0;
822           retry:
823             TASK V76para!SABME:=(. me .);
824             TASK V76para!presen:=SABME;
825             OUTPUT V76frame
826             (V76para)
827             VIA peer;
828             SET (T320);
829             NEXTSTATE waitUA;
830           ( False ):
831             TASK V76para!UA:=(. me .);
832             TASK V76para!presen:=UA;
833             OUTPUT V76frame
834             (V76para)
835             VIA peer;
836             NEXTSTATE connected;
837         ENDDECISION;
838       STATE waitUA;
839         INPUT V76frame
840         (V76para);
841           DECISION V76para ! presen;
842             ( UA ):
843               RESET (T320);
844               OUTPUT L_EstabConf
845               (me);
846               NEXTSTATE connected;
847             ( DM ):
848               OUTPUT DLCstopped
849               (me);
850               STOP ;
851             ELSE:
852               NEXTSTATE -;
853           ENDDECISION;
854         INPUT T320;
855           DECISION N320cnt < N320;
856             ( True ):
857               TASK N320cnt
858               :=
859               N320cnt +
860               1;
861               JOIN retry;
862             ELSE:
863               OUTPUT DLCstopped
864               (me);
865               STOP ;
866           ENDDECISION;
867       ENDSTATE;
868       STATE connected;
869         INPUT L_DataReq
870         ( ,
871         uData)
872           COMMENT 'We do not
873         get the first
874         parameter';
875           TASK Iparam := (. me,
876           uData, 15 .);
877           TASK V76para!I:= Iparam;
878           TASK V76para!presen:=I;
879           OUTPUT V76frame
880           (V76para)
881           VIA peer;
882           NEXTSTATE -;
883         INPUT L_ReleaseReq;
884           TASK V76para!DISC:=(. me .);
885           TASK V76para!presen:=DISC;
```

```
886           OUTPUT V76frame
887           (V76para)
888           VIA peer;
889           NEXTSTATE waitUAdisc;
890         INPUT V76frame
891         (V76para);
892           DECISION V76para!presen;
893             ( DISC ):
894               TASK V76para!UA:=(. me .);
895               TASK V76para!presen:=DISC;
896               OUTPUT V76frame
897               (V76para)
898               VIA peer;
899               OUTPUT DLCstopped
900               (me);
901               STOP ;
902             ( I ):
903               DECISION CALL
904               CRCok
905               (V76para
906               ! I
907               !CRC);
908                 ( True ):
909                   OUTPUT L_DataInd
910                   (me,
911                   V76para! I
912                   ! data);
913                 ( False ):
914               ENDDECISION;
915               NEXTSTATE -;
916             ELSE:
917               NEXTSTATE -;
918           ENDDECISION;
919       ENDSTATE;
920       STATE waitUAdisc;
921         INPUT V76frame
922         (V76para);
923           DECISION V76para!presen;
924             ( UA, DM ):
925               OUTPUT DLCstopped
926               (me);
927               STOP ;
928             ELSE:
929               NEXTSTATE -;
930           ENDDECISION;
931       ENDSTATE;
932     ENDPROCESS;
933   ENDBLOCK;
934 ENDSYSTEM;
```

C.2.2 Model sistema v jeziku Promela

C.2.2.1 Osnovne definicije

```
 1 #define pt__sig byte
 2 #define pt__sig byte
 3 /*****************************************************************************/
 4 /*                        dtimespin.h                                        */
 5 /*                                                                           */
 6 /*****************************************************************************/
 7
 8 #define OFF (-1)
 9 #define TOVAL (0)
10 bool pv__random;
11
12 /*discrete time macros*/
13
14 /* timer data type */
15 typedef timer
16 {
17 short val=OFF;
18 }
```

```
19
20 /* basic timing statements */
21
22 #define set(tmr,value) tmr.val=value
23 #define expire(tmr) (tmr.val==TOVAL) /*timeout*/
24 #define reset(tmr) tmr.val=OFF
25 /* derived timing statements */
26
27 #define delay(tmr,value) set(tmr,value); expire(tmr)   /* delay of val ticks */
28 #define udelay(tmr) do :: delay(tmr,1); ::break; od /* unbounded delay */
29
30 /* nondeterministic bounded delay */
31 #define bdelay(tmr,val,auxtmr) \
32           set(tmr,val);\
33           do \
34           :: expire(tmr) -> break \
35           :: else -> if :: break; :: delay(auxtmr,1); fi\
36           od
37
38 inline random(){
39 if
40 :: pv__random = true
41 :: pv__random = false
42 fi
43 }
44
45
46 active proctype __Timers()
47 {
48  end__Timers:    do :: timeout -> random() od;
49 }
50
51 #define pt__pid pid
52 #define pt__chan chan
53 #define cv__buff 4
54 #define cv__procnum 13
55 #define pcv__none 0
56 #define pcv__null 0
57
58 hidden int pfv_tmp;
59 show bool pv__runtime_error=false;
60
61 #define bit_undefined__value 1
62 #define bool_undefined__value 1
63 #define byte_undefined__value 255
64 #define chan_undefined__value 255
65 #define pid_undefined__value 255
66 #define short_undefined__value 32767
67 #define int_undefined__value 2145483647
68 #define unsigned_undefined__value 2145483647
69
70 /*     INCLUDE  SYNONYM PODATKOVNIH TIPOV  */
71 /*****************************************************************************/
72 /*                                                                           */
73 /*                          Sdl2Promela translator 1.0                       */
74 /*                                                                           */
75 /* AUTHOR:      A. Vreze, B. Vlaovic, University of Maribor, Slovenia        */
76 /* FILENAME:      synonym_dlc.log                                            */
77 /*                                                                           */
78 /*                                                                           */
79 /*****************************************************************************/
80
81 #define maxDLC 1
82 #define DLCa__DLC__N320 3
83 #define DLCb__DLC__N320 3
84
85 /*     INCLUDE  SYNTYPE PODATKOVNIH TIPOV  */
86 /*****************************************************************************/
87 /*                                                                           */
88 /*                          Sdl2Promela translator 1.0                       */
89 /*                                                                           */
90 /* AUTHOR:      A. Vreze, B. Vlaovic, University of Maribor, Slovenia        */
91 /* FILENAME:      syntype_source.log                                         */
```

```
92 /*                                                                          */
93 /*                                                                          */
94 /****************************************************************************/
95
96 #define XIDframe int
97
98 typedef DLCident {byte val}
99  typedef CommType__DLCident {byte val}
100
101 typedef environment__SUa__T_DATA {byte val}
102  typedef CommType__environment__SUa__T_DATA {byte val}
103
104
105 /*     INCLUDE  NEWTYPE PODATKOVNIH TIPOV  */
106 /****************************************************************************/
107 /*                                                                          */
108 /*                        Sdl2Promela translator 1.0                        */
109 /*                                                                          */
110 /* AUTHOR:      A. Vreze, B. Vlaovic, University of Maribor, Slovenia       */
111 /* FILENAME:      newtype_source.log                                        */
112 /*                                                                          */
113 /* STOLPEC 1:      SDL_ime_sinonima                                         */
114 /* STOLPEC 2:      Promela_ime_sinonima                                     */
115 /* STOLPEC 3:      tip_sinonima                                             */
116 /* STOLPEC 4:      ground_expresion                                         */
117 /*                                                                          */
118 /****************************************************************************/
119
120 typedef T_CHOICE {byte val}
121  typedef comm__T_CHOICE {byte val}
122
123 typedef Iframe {
124  DLCident DLCi;
125 int data;
126 int CRC
127 };
128
129 hidden Iframe undefined__Iframe;
130
131 typedef SABMEframe {
132  DLCident DLCi
133 };
134
135 hidden SABMEframe undefined__SABMEframe;
136
137 typedef DMframe {
138  DLCident DLCi
139 };
140
141 hidden DMframe undefined__DMframe;
142
143 typedef DISCframe {
144  DLCident DLCi
145 };
146
147 hidden DISCframe undefined__DISCframe;
148
149 typedef UAframe {
150  DLCident DLCi
151 };
152
153 hidden UAframe undefined__UAframe;
154
155 typedef DLCa__dispatch__DLCsArray  { pt__pid val[255] = pid_undefined__value };
156
157 typedef DLCb__dispatch__DLCsArray  { pt__pid val[255] = pid_undefined__value };
158
159 typedef V76paramTyp {
160  T_CHOICE presen;
161 Iframe I;
162 SABMEframe SABME;
163 DMframe DM;
164 DISCframe DISC;
```

```
165 UAframe UA;
166 XIDframe XIDcmd;
167 XIDframe XIDresp
168 };
169
170 hidden V76paramTyp undefined__V76paramTyp;
171
172
173 /*     INCLUDE  LITERALOV  */
174 /*****************************************************************************/
175 /*                                                                           */
176 /*                          Sdl2Promela translator 1.0                       */
177 /*                                                                           */
178 /* AUTHOR:      A. Vreze, B. Vlaovic, University of Maribor, Slovenia        */
179 /* FILENAME:      literals.ph                                                */
180 /*                                                                           */
181 /* STOLPEC 1:      #define                                                   */
182 /* STOLPEC 2:      ime_tipa__ime_literala                                    */
183 /* STOLPEC 3:      index                                                     */
184 /*                                                                           */
185 /*****************************************************************************/
186
187 #define T_CHOICE__I 1
188 #define T_CHOICE__SABME 2
189 #define T_CHOICE__DM 3
190 #define T_CHOICE__DISC 4
191 #define T_CHOICE__UA 5
192 #define T_CHOICE__XIDcmd 6
193 #define T_CHOICE__XIDresp 7
194 #define T_CHOICE__undefined__value 8
195
196 /*     INCLUDE  DEFINIRANIH SIGNALOV  */
197 /*****************************************************************************/
198 /*                                                                           */
199 /*                          Sdl2Promela translator 1.0                       */
200 /*                                                                           */
201 /* AUTHOR:      A. Vreze, B. Vlaovic, University of Maribor, Slovenia        */
202 /* FILENAME:      signals.ph                                                 */
203 /*                                                                           */
204 /*                                                                           */
205 /*****************************************************************************/
206
207 #define L_EstabReq 1
208 #define L_EstabInd 2
209 #define L_EstabResp 3
210 #define L_EstabConf 4
211 #define L_SetparmReq 5
212 #define L_SetparmInd 6
213 #define L_SetparmResp 7
214 #define L_SetparmConf 8
215 #define L_ReleaseReq 9
216 #define L_ReleaseInd 10
217 #define L_DataReq 11
218 #define L_DataInd 12
219 #define V76frame 13
220 #define DLCa__DLCstopped 14
221 #define DLCb__DLCstopped 15
222
223 /*     INCLUDE  DEFINIRANIH CASOVNIKOV  */
224 /*****************************************************************************/
225 /*                                                                           */
226 /*                          Sdl2Promela translator 1.0                       */
227 /*                                                                           */
228 /* AUTHOR:      A. Vreze, B. Vlaovic, University of Maribor, Slovenia        */
229 /* FILENAME:      timers.ph                                                  */
230 /*                                                                           */
231 /*                                                                           */
232 /*****************************************************************************/
233
234 #define DLCa__dispatch__T_401 16
235 #define DLCa__dispatch__T_402 17
236 #define DLCa__DLC__T_401 18
237 #define DLCa__DLC__T320 19
```

```
238 #define DLCb__dispatch__T_401 20
239 #define DLCb__dispatch__T_402 21
240 #define DLCb__DLC__T320 22
241 #define DLCb__DLC__T_401 23
242
243 /*     INCLUDE  DEFINIRANIH KANALOV  */
244 /*****************************************************************************/
245 /*                                                                           */
246 /*                          Sdl2Promela translator 1.0                       */
247 /*                                                                           */
248 /* AUTHOR:      A. Vreze, B. Vlaovic, University of Maribor, Slovenia        */
249 /* FILENAME:      channels.ph                                                */
250 /*                                                                           */
251 /*                                                                           */
252 /*****************************************************************************/
253
254 chan chan__env = [cv__buff] of {pt__sig,pt__pid,DLCident,int}
255 #define number_of_process 13
256 #define number_of_channels 1
257
258
259 #define table__dataLink__AtoB__max 1
260 hidden byte table__dataLink__AtoB__free;
261 #define max__table__dataLink__AtoB 1
262 #define chan__dataLink__AtoB__select 1
263 chan chan__dataLink__AtoB[table__dataLink__AtoB__max] =
264 [cv__buff] of {pt__sig,pt__pid,V76paramTyp};
265
266 #define table__dataLink__BtoA__max 1
267 hidden byte table__dataLink__BtoA__free;
268 #define max__table__dataLink__BtoA 1
269 #define chan__dataLink__BtoA__select 2
270 chan chan__dataLink__BtoA[table__dataLink__BtoA__max] =
271 [cv__buff] of {pt__sig,pt__pid,V76paramTyp};
272
273 #define table__DLCa__dispatch__max 1
274 hidden byte table__DLCa__dispatch__free;
275 #define max__table__DLCa__dispatch 1
276 #define chan__DLCa__dispatch__select 3
277 chan chan__DLCa__dispatch[table__DLCa__dispatch__max] =
278 [cv__buff] of {pt__sig,pt__pid,DLCident,V76paramTyp,int};
279
280 #define table__DLCa__DLC__max 2
281 hidden byte table__DLCa__DLC__free;
282 #define max__table__DLCa__DLC 2
283 #define chan__DLCa__DLC__select 4
284 chan chan__DLCa__DLC[table__DLCa__DLC__max] =
285 [cv__buff] of {pt__sig,pt__pid,DLCident,V76paramTyp,int};
286
287 #define table__DLCb__dispatch__max 1
288 hidden byte table__DLCb__dispatch__free;
289 #define max__table__DLCb__dispatch 1
290 #define chan__DLCb__dispatch__select 6
291 chan chan__DLCb__dispatch[table__DLCb__dispatch__max] =
292 [cv__buff] of {pt__sig,pt__pid,DLCident,V76paramTyp,int};
293
294 #define table__DLCb__DLC__max 2
295 hidden byte table__DLCb__DLC__free;
296 #define max__table__DLCb__DLC 2
297 #define chan__DLCb__DLC__select 7
298 chan chan__DLCb__DLC[table__DLCb__DLC__max] =
299 [cv__buff] of {pt__sig,pt__pid,DLCident,V76paramTyp,int};
300
301 #define table__environment__SUa__max 1
302 hidden byte table__environment__SUa__free;
303 #define max__table__environment__SUa 1
304 #define chan__environment__SUa__select 9
305 chan chan__environment__SUa[table__environment__SUa__max] =
306 [cv__buff] of {pt__sig,pt__pid,DLCident,int};
307
308 #define table__environment__SUb__max 1
309 hidden byte table__environment__SUb__free;
310 #define max__table__environment__SUb 1
```

```
311 #define chan__environment__SUb__select 10
312 chan chan__environment__SUb[table__environment__SUb__max] =
313 [cv__buff] of {pt__sig,pt__pid,DLCident,int};
314
315
316
317 /***** Tables for the channel monitoring BEGIN *****/
318 typedef pt__chandat{pt__sig name; bool save; bool prio} ;
319 typedef pt__chandata{pt__chandat data[cv__buff]};
320 pt__chan table_channame_channum[cv__procnum];
321 pt__chan table_pid_channum[cv__procnum];
322 pt__sig table_pid_channame[cv__procnum];
323 byte table_channum_ptr[cv__procnum];
324 pt__chandata table_channum_nsp[cv__procnum];
325 bool table_channum_prio[cv__procnum];
326
327 /***** Tables for the channel monitoring END *****/
328
329 /**** Update of the channel monitoring tables BEGIN ****/
330 inline update_chan_tab(){
331 /* popravki vhodne tabele */
332 atomic{
333 do
334 ::(pv__ptr < cv__buff-1) ->
335   table_channum_nsp[input].data[pv__ptr].name=
336     table_channum_nsp[input].data[pv__ptr+1].name;
337   table_channum_nsp[input].data[pv__ptr].save=
338     table_channum_nsp[input].data[pv__ptr+1].save;
339   table_channum_nsp[input].data[pv__ptr].prio=
340     table_channum_nsp[input].data[pv__ptr+1].prio;
341   pv__ptr++
342 :: else ->
343   table_channum_nsp[input].data[pv__ptr].name=pcv__none;
344   table_channum_nsp[input].data[pv__ptr].save=false;
345   table_channum_nsp[input].data[pv__ptr].prio=false;
346   break
347 od;
348 table_channum_ptr[input]--;
349 pv__cur=0;
350 }
351 }
352 inline set__prio(sig){
353   table_channum_nsp[input].data[pv__ptr].name==sig ->
354   atomic{
355     table_channum_nsp[input].data[pv__ptr].save=false;
356     table_channum_nsp[input].data[pv__ptr].prio=true;
357     table_channum_prio[input]=true
358   }
359 }
360 inline set__save(sig){
361   table_channum_nsp[input].data[pv__ptr].name==sig ->
362   atomic{
363     table_channum_nsp[input].data[pv__ptr].save=true;
364     table_channum_nsp[input].data[pv__ptr].prio=false;
365   }
366 }
367
368 inline set__clear(){
369   atomic{
370     table_channum_nsp[input].data[pv__ptr].save=false;
371     table_channum_nsp[input].data[pv__ptr].prio=false;
372   }
373 }
374 #define recv__sig(pog,sig)\
375  ((pog) && table_channum_nsp[input].data[pv__ptr].name==sig)
376
377 inline skip__save(){
378   table_channum_nsp[input].data[pv__ptr].save==true -> pv__ptr++
379 }
380
381 /**** Update of the channel monitoring tables END ****/
```

C.2.2.2 Model uporabnika SUa

```
1 /*****************************************************************************/
2 /*                                                                           */
3 /*                          Process environment__SUa                         */
4 /*                                                                           */
5 /*                                                                           */
6 /*****************************************************************************/
7 /* F. PAR:                                                                   */
8 /* KOMENTAR:                                                                 */
9 /* LOKACIJA:      environment                                                */
10 /* MAX:      1                                                               */
11 /* MIN:      1                                                               */
12 /*                                                                           */
13 /*****************************************************************************/
14 proctype environment__SUa(pt__chan input; pt__pid parent){
15 pt__pid offspring, sender;
16 byte pv__ptr, pv__cur;
17 xr input;
18
19 DLCident V_DLCid = 0;
20 DLCident V_DLCid_tmp;
21 environment__SUa__T_DATA V_Data = 0;
22
23 /* Output signal : L_SetparmReq */
24 if
25 :: true ->
26   d_step{
27     pfv_tmp=1;
28     do
29     :: pfv_tmp < table__DLCa__dispatch__free ->
30        pfv_tmp++ ;
31     :: break;
32     od;
33     pfv_tmp--;
34   };
35   atomic{
36   chan__DLCa__dispatch[pfv_tmp]!L_SetparmReq(_pid,pcv__null,\
37   undefined__V76paramTyp,pcv__null);
38   table_channum_nsp[table_channame_channum[chan__DLCa__dispatch__select +
39   pfv_tmp]].data[table_channum_ptr[table_channame_channum[\
40   chan__DLCa__dispatch__select + pfv_tmp]]].name=  L_SetparmReq;
41   table_channum_ptr[table_channame_channum[chan__DLCa__dispatch__select +
42   pfv_tmp]]++;
43   }
44 fi;
45 goto waitParmResp;
46 waitParmResp:
47 end_1:
48 do
49 :: table_channum_ptr[input] > pv__cur ->
50    table_channum_prio[input]=false;
51    pv__cur++;
52    pv__ptr=0;
53    atomic{
54    do
55    :: pv__ptr <= cv__buff-1 ->
56       if
57       :: else -> set__clear();
58       fi;
59       pv__ptr++ ;
60    :: else -> goto waitParmResp_start;
61    od;
62    }
63 waitParmResp_start:
64    if
65    :: table_channum_prio[input]==true ->
66       pv__ptr=0;
67       do
68       :: (pv__ptr <= cv__buff-1)
69       && (table_channum_nsp[input].data[pv__ptr].prio==true) ->
70          if
71          /* PRIORITY INPUT */
```

```
72          :: else  -> skip;
73          fi;
74       :: (pv__ptr == cv__buff) -> break;
75       :: else -> pv__ptr++
76       od;
77    :: else ->
78       pv__ptr=0;
79       do
80       :: (pv__ptr <= cv__buff-1) ->
81          if
82          :: skip__save()
83          :: else ->
84             if
85             /* INPUT SIGNALS */
86             ::atomic{
87                atomic{recv__sig(true, L_SetparmConf) ->
88                input??L_SetparmConf(sender,_,_);
89                update_chan_tab();
90                }
91                /* === Decision start === */
92                if
93                :: true ->
94                  V_DLCid.val = 0;
95                :: true ->
96                  V_DLCid.val = 1;
97                :: else -> pv__runtime_error = true;
98                fi;
99                /* ==== Decision end === */
100               /* Output signal : L_EstabReq */
101               if
102               :: true ->
103                 d_step{
104                   pfv_tmp=1;
105                   do
106                   :: pfv_tmp < table__DLCa__dispatch__free ->
107                      pfv_tmp++ ;
108                   :: break;
109                   od;
110                   pfv_tmp--;
111                 };
112                 atomic{
113                 chan__DLCa__dispatch[pfv_tmp]!L_EstabReq(_pid,V_DLCid,\
114                 undefined__V76paramTyp,pcv__null);
115                 table_channum_nsp[table_channame_channum[\
116                 chan__DLCa__dispatch__select + pfv_tmp]].\
117                 data[table_channum_ptr[table_channame_channum[\
118                 chan__DLCa__dispatch__select + pfv_tmp]]].name=
119                 L_EstabReq;
120                 table_channum_ptr[table_channame_channum[\
121                 chan__DLCa__dispatch__select + pfv_tmp]]++;
122                 }
123               fi;
124               goto waitEstabConf;
125            } /* === end atomic transition ===*/
126            :: else ->
127               if
128               /* IMPLICITNI PREHOD */
129               ::atomic{recv__sig(true,L_DataInd) ->
130               input??L_DataInd(sender,_,_);
131               update_chan_tab();
132               goto waitParmResp;
133               }
134               ::atomic{recv__sig(true,L_DataReq) ->
135               input??L_DataReq(sender,_,_);
136               update_chan_tab();
137               goto waitParmResp;
138               }
139               ::atomic{recv__sig(true,L_EstabConf) ->
140               input??L_EstabConf(sender,_,_);
141               update_chan_tab();
142               goto waitParmResp;
143               }
144               ::atomic{recv__sig(true,L_EstabInd) ->
```

```
145                 input??L_EstabInd(sender,_,_);
146                 update_chan_tab();
147                 goto waitParmResp;
148                 }
149                 ::atomic{recv__sig(true,L_EstabReq) ->
150                 input??L_EstabReq(sender,_,_);
151                 update_chan_tab();
152                 goto waitParmResp;
153                 }
154                 ::atomic{recv__sig(true,L_EstabResp) ->
155                 input??L_EstabResp(sender,_,_);
156                 update_chan_tab();
157                 goto waitParmResp;
158                 }
159                 ::atomic{recv__sig(true,L_ReleaseInd) ->
160                 input??L_ReleaseInd(sender,_,_);
161                 update_chan_tab();
162                 goto waitParmResp;
163                 }
164                 ::atomic{recv__sig(true,L_ReleaseReq) ->
165                 input??L_ReleaseReq(sender,_,_);
166                 update_chan_tab();
167                 goto waitParmResp;
168                 }
169                 ::atomic{recv__sig(true,L_SetparmInd) ->
170                 input??L_SetparmInd(sender,_,_);
171                 update_chan_tab();
172                 goto waitParmResp;
173                 }
174                 ::atomic{recv__sig(true,L_SetparmReq) ->
175                 input??L_SetparmReq(sender,_,_);
176                 update_chan_tab();
177                 goto waitParmResp;
178                 }
179                 ::atomic{recv__sig(true,L_SetparmResp) ->
180                 input??L_SetparmResp(sender,_,_);
181                 update_chan_tab();
182                 goto waitParmResp;
183                 }
184                 :: else -> goto waitParmResp;
185                 fi;
186             fi;
187          fi;
188       od;
189    fi;
190    goto waitParmResp;
191    /* SPONTANEOUS TRANSITION */
192    /* TIMERS */
193    /* TRAJEN SIGNAL */
194 od;
195 waitEstabConf:
196 end_2:
197 do
198 :: table_channum_ptr[input] > pv__cur ->
199    table_channum_prio[input]=false;
200    pv__cur++;
201    pv__ptr=0;
202    atomic{
203    do
204    :: pv__ptr <= cv__buff-1 ->
205       if
206       :: else -> set__clear();
207       fi;
208       pv__ptr++ ;
209    :: else -> goto waitEstabConf_start;
210    od;
211    }
212 waitEstabConf_start:
213    if
214    :: table_channum_prio[input]==true ->
215       pv__ptr=0;
216       do
217       :: (pv__ptr <= cv__buff-1)
```

```
218        && (table_channum_nsp[input].data[pv__ptr].prio==true) ->
219           if
220           /* PRIORITY INPUT */
221           :: else  -> skip;
222           fi;
223        :: (pv__ptr == cv__buff) -> break;
224        :: else -> pv__ptr++
225        od;
226     :: else ->
227        pv__ptr=0;
228        do
229        :: (pv__ptr <= cv__buff-1) ->
230           if
231           :: skip__save()
232           :: else ->
233              if
234              /* INPUT SIGNALS */
235              ::atomic{
236                 atomic{recv__sig(true, L_EstabConf) ->
237                 input??L_EstabConf(sender,V_DLCid_tmp,_);
238                 update_chan_tab();
239                 }
240                 /* === Decision start === */
241                 if
242                 :: ((V_DLCid.val==V_DLCid_tmp.val) == true) ->
243                   /* === Decision start === */
244                   if
245                   :: true ->
246                     /* === Decision start === */
247                     if
248                     :: true ->
249                       V_Data.val = 39;
250                     :: true ->
251                       V_Data.val = 86;
252                     :: else -> pv__runtime_error = true;
253                     fi;
254                     /* ==== Decision end === */
255                     /* Output signal : L_DataReq */
256                     if
257                     :: true ->
258                       d_step{
259                         pfv_tmp=1;
260                         do
261                         :: pfv_tmp < table__DLCa__dispatch__free ->
262                            pfv_tmp++ ;
263                         :: break;
264                         od;
265                         pfv_tmp--;
266                       };
267                       atomic{
268                       chan__DLCa__dispatch[pfv_tmp]!L_DataReq(_pid,\
269                       V_DLCid,undefined__V76paramTyp,V_Data);
270                       table_channum_nsp[table_channame_channum[\
271                       chan__DLCa__dispatch__select + pfv_tmp]].data[\
272                       table_channum_ptr[table_channame_channum[\
273                       chan__DLCa__dispatch__select + pfv_tmp]]].name=
274                       L_DataReq;
275                       table_channum_ptr[table_channame_channum[\
276                       chan__DLCa__dispatch__select + pfv_tmp]]++;
277                       }
278                     fi;
279                   :: true ->
280                     /* Output signal : L_ReleaseReq */
281                     if
282                     :: true ->
283                       d_step{
284                         pfv_tmp=1;
285                         do
286                         :: pfv_tmp < table__DLCa__dispatch__free ->
287                            pfv_tmp++ ;
288                         :: break;
289                         od;
290                         pfv_tmp--;
```

```
291                     };
292                     atomic{
293                     chan__DLCa__dispatch[pfv_tmp]!L_ReleaseReq(_pid,V_DLCid,\
294                     undefined__V76paramTyp,pcv__null);
295                     table_channum_nsp[table_channame_channum[\
296                     chan__DLCa__dispatch__select + pfv_tmp]].data[\
297                     table_channum_ptr[table_channame_channum[\
298                     chan__DLCa__dispatch__select + pfv_tmp]]].name=
299                     L_ReleaseReq;
300                     table_channum_ptr[table_channame_channum[\
301                     chan__DLCa__dispatch__select + pfv_tmp]]++;
302                     }
303                   fi;
304                 :: else -> pv__runtime_error = true;
305                 fi;
306                 /* ==== Decision end === */
307                 goto SUa_endstate;
308               :: else ->
309                 /* Output signal : L_EstabReq */
310                 if
311                 :: true ->
312                   d_step{
313                     pfv_tmp=1;
314                     do
315                     :: pfv_tmp < table__DLCa__dispatch__free ->
316                        pfv_tmp++ ;
317                     :: break;
318                     od;
319                     pfv_tmp--;
320                   };
321                   atomic{
322                   chan__DLCa__dispatch[pfv_tmp]!L_EstabReq(_pid,V_DLCid,\
323                   undefined__V76paramTyp,pcv__null);
324                   table_channum_nsp[table_channame_channum[\
325                   chan__DLCa__dispatch__select + pfv_tmp]].data[\
326                   table_channum_ptr[table_channame_channum[\
327                   chan__DLCa__dispatch__select + pfv_tmp]]].name=
328                   L_EstabReq;
329                   table_channum_ptr[table_channame_channum[\
330                   chan__DLCa__dispatch__select + pfv_tmp]]++;
331                   }
332                 fi;
333                 goto waitEstabConf;
334               fi;
335               /* ==== Decision end === */
336             } /* === end atomic transition ===*/
337             :: else ->
338               if
339               /* IMPLICITNI PREHOD */
340               ::atomic{recv__sig(true,L_DataInd) ->
341               input??L_DataInd(sender,_,_);
342               update_chan_tab();
343               goto waitEstabConf;
344               }
345               ::atomic{recv__sig(true,L_DataReq) ->
346               input??L_DataReq(sender,_,_);
347               update_chan_tab();
348               goto waitEstabConf;
349               }
350               ::atomic{recv__sig(true,L_EstabInd) ->
351               input??L_EstabInd(sender,_,_);
352               update_chan_tab();
353               goto waitEstabConf;
354               }
355               ::atomic{recv__sig(true,L_EstabReq) ->
356               input??L_EstabReq(sender,_,_);
357               update_chan_tab();
358               goto waitEstabConf;
359               }
360               ::atomic{recv__sig(true,L_EstabResp) ->
361               input??L_EstabResp(sender,_,_);
362               update_chan_tab();
363               goto waitEstabConf;
```

```
364                 }
365                 ::atomic{recv__sig(true,L_ReleaseInd) ->
366                 input??L_ReleaseInd(sender,_,_);
367                 update_chan_tab();
368                 goto waitEstabConf;
369                 }
370                 ::atomic{recv__sig(true,L_ReleaseReq) ->
371                 input??L_ReleaseReq(sender,_,_);
372                 update_chan_tab();
373                 goto waitEstabConf;
374                 }
375                 ::atomic{recv__sig(true,L_SetparmConf) ->
376                 input??L_SetparmConf(sender,_,_);
377                 update_chan_tab();
378                 goto waitEstabConf;
379                 }
380                 ::atomic{recv__sig(true,L_SetparmInd) ->
381                 input??L_SetparmInd(sender,_,_);
382                 update_chan_tab();
383                 goto waitEstabConf;
384                 }
385                 ::atomic{recv__sig(true,L_SetparmReq) ->
386                 input??L_SetparmReq(sender,_,_);
387                 update_chan_tab();
388                 goto waitEstabConf;
389                 }
390                 ::atomic{recv__sig(true,L_SetparmResp) ->
391                 input??L_SetparmResp(sender,_,_);
392                 update_chan_tab();
393                 goto waitEstabConf;
394                 }
395                 :: else -> goto waitEstabConf;
396                 fi;
397             fi;
398          fi;
399       od;
400    fi;
401    goto waitEstabConf;
402    /* SPONTANEOUS TRANSITION */
403    /* TIMERS */
404    /* TRAJEN SIGNAL */
405 od;
406 SUa_endstate:
407 progress_1:
408 do
409 :: table_channum_ptr[input] > pv__cur ->
410    table_channum_prio[input]=false;
411    pv__cur++;
412    pv__ptr=0;
413    atomic{
414    do
415    :: pv__ptr <= cv__buff-1 ->
416       if
417       :: else -> set__clear();
418       fi;
419       pv__ptr++ ;
420    :: else -> goto SUa_endstate_start;
421    od;
422    }
423 SUa_endstate_start:
424    if
425    :: table_channum_prio[input]==true ->
426       pv__ptr=0;
427       do
428       :: (pv__ptr <= cv__buff-1)
429       && (table_channum_nsp[input].data[pv__ptr].prio==true) ->
430          if
431          /* PRIORITY INPUT */
432          :: else  -> skip;
433          fi;
434       :: (pv__ptr == cv__buff) -> break;
435       :: else -> pv__ptr++
436       od;
```

```
437     :: else ->
438        pv__ptr=0;
439        do
440        :: (pv__ptr <= cv__buff-1) ->
441           if
442           :: skip__save()
443           :: else ->
444              if
445              /* INPUT SIGNALS */
446              ::atomic{
447                 atomic{recv__sig(true, L_DataInd) ->
448                 input??L_DataInd(sender,_,_);
449                 update_chan_tab();
450                 }
451                 goto SUa_endstate__asteriks;
452              } /* === end atomic transition ===*/
453              ::atomic{
454                 atomic{recv__sig(true, L_DataReq) ->
455                 input??L_DataReq(sender,_,_);
456                 update_chan_tab();
457                 }
458                 goto SUa_endstate__asteriks;
459              } /* === end atomic transition ===*/
460              ::atomic{
461                 atomic{recv__sig(true, L_EstabConf) ->
462                 input??L_EstabConf(sender,_,_);
463                 update_chan_tab();
464                 }
465                 goto SUa_endstate__asteriks;
466              } /* === end atomic transition ===*/
467              ::atomic{
468                 atomic{recv__sig(true, L_EstabInd) ->
469                 input??L_EstabInd(sender,_,_);
470                 update_chan_tab();
471                 }
472                 goto SUa_endstate__asteriks;
473              } /* === end atomic transition ===*/
474              ::atomic{
475                 atomic{recv__sig(true, L_EstabReq) ->
476                 input??L_EstabReq(sender,_,_);
477                 update_chan_tab();
478                 }
479                 goto SUa_endstate__asteriks;
480              } /* === end atomic transition ===*/
481              ::atomic{
482                 atomic{recv__sig(true, L_EstabResp) ->
483                 input??L_EstabResp(sender,_,_);
484                 update_chan_tab();
485                 }
486                 goto SUa_endstate__asteriks;
487              } /* === end atomic transition ===*/
488              ::atomic{
489                 atomic{recv__sig(true, L_ReleaseInd) ->
490                 input??L_ReleaseInd(sender,_,_);
491                 update_chan_tab();
492                 }
493                 goto SUa_endstate__asteriks;
494              } /* === end atomic transition ===*/
495              ::atomic{
496                 atomic{recv__sig(true, L_ReleaseReq) ->
497                 input??L_ReleaseReq(sender,_,_);
498                 update_chan_tab();
499                 }
500                 goto SUa_endstate__asteriks;
501              } /* === end atomic transition ===*/
502              ::atomic{
503                 atomic{recv__sig(true, L_SetparmConf) ->
504                 input??L_SetparmConf(sender,_,_);
505                 update_chan_tab();
506                 }
507                 goto SUa_endstate__asteriks;
508              } /* === end atomic transition ===*/
509              ::atomic{
```

```
510                atomic{recv__sig(true, L_SetparmInd) ->
511                input??L_SetparmInd(sender,_,_);
512                update_chan_tab();
513                }
514                goto SUa_endstate__asteriks;
515             } /* === end atomic transition ===*/
516             ::atomic{
517                atomic{recv__sig(true, L_SetparmReq) ->
518                input??L_SetparmReq(sender,_,_);
519                update_chan_tab();
520                }
521                goto SUa_endstate__asteriks;
522             } /* === end atomic transition ===*/
523             ::atomic{
524                atomic{recv__sig(true, L_SetparmResp) ->
525                input??L_SetparmResp(sender,_,_);
526                update_chan_tab();
527                }
528                goto SUa_endstate__asteriks;
529             } /* === end atomic transition ===*/
530 SUa_endstate__asteriks:
531 atomic{
532                /* Output signal : L_ReleaseReq */
533                if
534                :: true ->
535                  d_step{
536                    pfv_tmp=1;
537                    do
538                    :: pfv_tmp < table__DLCa__dispatch__free ->
539                       pfv_tmp++ ;
540                    :: break;
541                    od;
542                    pfv_tmp--;
543                  };
544                  atomic{
545                  chan__DLCa__dispatch[pfv_tmp]!L_ReleaseReq(_pid,V_DLCid,\
546                  undefined__V76paramTyp,pcv__null);
547                  table_channum_nsp[table_channame_channum[\
548                  chan__DLCa__dispatch__select + pfv_tmp]].data[\
549                  table_channum_ptr[table_channame_channum[\
550                  chan__DLCa__dispatch__select + pfv_tmp]]].name=
551                  L_ReleaseReq;
552                  table_channum_ptr[table_channame_channum[\
553                  chan__DLCa__dispatch__select + pfv_tmp]]++;
554                  }
555                fi;
556                goto SUa_endstate;
557             } /* === end atomic transition ===*/
558             :: else ->
559                if
560                /* IMPLICITNI PREHOD */
561                :: else -> goto SUa_endstate;
562                fi;
563             fi;
564          fi;
565       od;
566    fi;
567    goto SUa_endstate;
568    /* SPONTANEOUS TRANSITION */
569    /* TIMERS */
570    /* TRAJEN SIGNAL */
571 od;
572
573 end: skip
574 }
```

C.2.2.3 Model procesa `dispatch` v bloku `DLCa`

```
1 /*************************************************************************/
2 /*                                                                       */
3 /*                          Process DLCa__dispatch                       */
4 /*                                                                       */
```

```
 5 /*                                                                          */
 6 /****************************************************************************/
 7 /* F. PAR:                                                                  */
 8 /* KOMENTAR:                                                                */
 9 /* LOKACIJA:      DLCa                                                      */
10 /* MAX:      1                                                              */
11 /* MIN:      1                                                              */
12 /*                                                                          */
13 /****************************************************************************/
14 proctype DLCa__dispatch(pt__chan input; pt__pid parent){
15 pt__pid offspring, sender;
16 byte pv__ptr, pv__cur;
17 xr input;
18 timer tmr__DLCa__dispatch__T_401;
19 timer tmr__DLCa__dispatch__T_402;
20
21 DLCident DLCnum;
22 DLCident DLCpeer;
23 int uData = int_undefined__value;
24 V76paramTyp V76para;
25 DLCa__dispatch__DLCsArray DLCs;
26
27 goto ready;
28 ready:
29 end_1:
30 do
31 :: table_channum_ptr[input] > pv__cur ->
32    table_channum_prio[input]=false;
33    pv__cur++;
34    pv__ptr=0;
35    atomic{
36    do
37    :: pv__ptr <= cv__buff-1 ->
38       if
39       :: set__prio(L_ReleaseReq)
40       :: else -> set__clear();
41       fi;
42       pv__ptr++ ;
43    :: else -> goto ready_start;
44    od;
45    }
46 ready_start:
47    if
48    :: table_channum_prio[input]==true ->
49       pv__ptr=0;
50       do
51       :: (pv__ptr <= cv__buff-1)
52       && (table_channum_nsp[input].data[pv__ptr].prio==true) ->
53          if
54          /* PRIORITY INPUT */
55          ::atomic{
56             atomic{recv__sig(true,L_ReleaseReq) ->
57             input??L_ReleaseReq(sender,DLCnum,undefined__V76paramTyp,_);
58             update_chan_tab();
59             }
60             /* === Decision start === */
61             if
62             :: ((DLCs.val[DLCnum.val]) == (pid_undefined__value)) ->
63             :: else ->
64               /* Output signal : L_ReleaseReq */
65               if
66               :: table_pid_channame[DLCs.val[DLCnum.val]]==
67                  chan__DLCa__DLC__select ->
68               table_pid_channum[DLCs.val[DLCnum.val]]!
69               L_ReleaseReq(_pid,DLCnum,undefined__V76paramTyp,pcv__null);
70               table_channum_nsp[table_pid_channum[DLCs.val[DLCnum.val]]]. \
71               data[table_channum_ptr[table_pid_channum[DLCs.val[DLCnum.val]]]].\
72               name=L_ReleaseReq;
73               table_channum_ptr[table_pid_channum[DLCs.val[DLCnum.val]]]++;
74
75               :: table_pid_channame[DLCs.val[DLCnum.val]]==
76               chan__DLCa__DLC__select + 1 ->
77               table_pid_channum[DLCs.val[DLCnum.val]]!L_ReleaseReq(_pid,
```

```
78              DLCnum,undefined__V76paramTyp,pcv__null);
79              table_channum_nsp[table_pid_channum[DLCs.val[DLCnum.val]]]. \
80              data[table_channum_ptr[table_pid_channum[DLCs.val[DLCnum.val]]]].\
81              name=L_ReleaseReq;
82              table_channum_ptr[table_pid_channum[DLCs.val[DLCnum.val]]]++;
83
84              :: table_pid_channame[DLCs.val[DLCnum.val]]==
85              chan__environment__SUa__select ->
86              table_pid_channum[DLCs.val[DLCnum.val]]!\
87              L_ReleaseReq(_pid,DLCnum,pcv__null);
88              table_channum_nsp[table_pid_channum[DLCs.val[DLCnum.val]]]. \
89              data[table_channum_ptr[table_pid_channum[DLCs.val[DLCnum.val]]]].\
90              name=L_ReleaseReq;
91              table_channum_ptr[table_pid_channum[DLCs.val[DLCnum.val]]]++;
92
93              :: else -> pv__runtime_error = true
94              fi;
95           fi;
96           /* ==== Decision end === */
97           goto ready;
98        } /* === end atomic transition ===*/
99        :: else  -> skip;
100       fi;
101    :: (pv__ptr == cv__buff) -> break;
102    :: else -> pv__ptr++
103    od;
104 :: else ->
105    pv__ptr=0;
106    do
107    :: (pv__ptr <= cv__buff-1) ->
108       if
109       :: skip__save()
110       :: else ->
111          if
112          /* INPUT SIGNALS */
113          ::atomic{
114             atomic{recv__sig(true, DLCa__DLCstopped) ->
115             input??DLCa__DLCstopped(sender,DLCnum,undefined__V76paramTyp,_);
116             update_chan_tab();
117             }
118             /* Output signal : L_ReleaseInd */
119             if
120             :: true ->
121               d_step{
122                 pfv_tmp=1;
123                 do
124                 :: pfv_tmp < table__environment__SUa__free ->
125                    pfv_tmp++ ;
126                 :: break;
127                 od;
128                 pfv_tmp--;
129               };
130               atomic{
131               chan__environment__SUa[pfv_tmp]!L_ReleaseInd(_pid,DLCnum,\
132               pcv__null);
133               table_channum_nsp[table_channame_channum[\
134               chan__environment__SUa__select + pfv_tmp]].\
135               data[table_channum_ptr[table_channame_channum[\
136               chan__environment__SUa__select + pfv_tmp]]].name= L_ReleaseInd;
137               table_channum_ptr[table_channame_channum[\
138               chan__environment__SUa__select + pfv_tmp]]++;
139               }
140             fi;
141             DLCs.val[DLCnum.val] = pid_undefined__value;
142             goto ready;
143          } /* === end atomic transition ===*/
144          ::atomic{
145             atomic{recv__sig(true, L_DataReq) ->
146             input??L_DataReq(sender,DLCnum,undefined__V76paramTyp,uData);
147             update_chan_tab();
148             }
149             /* === Decision start === */
150             if
```

```
151                 :: ((DLCs.val[DLCnum.val]) == (pid_undefined__value)) ->
152                 :: else ->
153                   /* Output signal : L_DataReq */
154                   if
155                   :: table_pid_channame[DLCs.val[DLCnum.val]]==
156                   chan__DLCa__DLC__select ->
157                   table_pid_channum[DLCs.val[DLCnum.val]]!L_DataReq(_pid,\
158                   DLCnum,undefined__V76paramTyp,uData);
159                   table_channum_nsp[table_pid_channum[DLCs.val[DLCnum.val]]]. \
160                   data[table_channum_ptr[table_pid_channum[DLCs.val[DLCnum.\
161                   val]]]].name=L_DataReq;
162                   table_channum_ptr[table_pid_channum[DLCs.val[DLCnum.val]]]++;
163
164                   :: table_pid_channame[DLCs.val[DLCnum.val]]==
165                   chan__DLCa__DLC__select + 1 ->
166                   table_pid_channum[DLCs.val[DLCnum.val]]!L_DataReq(_pid,\
167                   DLCnum,undefined__V76paramTyp,uData);
168                   table_channum_nsp[table_pid_channum[DLCs.val[DLCnum.val]]]. \
169                   data[table_channum_ptr[table_pid_channum[DLCs.val[DLCnum.\
170                   val]]]].name=L_DataReq;
171                   table_channum_ptr[table_pid_channum[DLCs.val[DLCnum.val]]]++;
172
173                   :: table_pid_channame[DLCs.val[DLCnum.val]]==
174                   chan__environment__SUa__select ->
175                   table_pid_channum[DLCs.val[DLCnum.val]]!L_DataReq(_pid,\
176                   DLCnum,uData);
177                   table_channum_nsp[table_pid_channum[DLCs.val[DLCnum.val]]]. \
178                   data[table_channum_ptr[table_pid_channum[DLCs.val[DLCnum.\
179                   val]]]].name=L_DataReq;
180                   table_channum_ptr[table_pid_channum[DLCs.val[DLCnum.val]]]++;
181
182                   :: else -> pv__runtime_error = true
183                   fi;
184                 fi;
185                 /* ==== Decision end === */
186                 goto ready;
187             } /* === end atomic transition ===*/
188             ::atomic{
189                 atomic{recv__sig(true, L_EstabReq) ->
190                 input??L_EstabReq(sender,DLCnum,undefined__V76paramTyp,_);
191                 update_chan_tab();
192                 }
193                 /* === Decision start === */
194                 if
195                 :: ((DLCs.val[DLCnum.val]) == (pid_undefined__value)) ->
196                   atomic{
197                       if
198                       :: table__DLCa__DLC__free < table__DLCa__DLC__max ->
199                       offspring = run
200                       DLCa__DLC(chan__DLCa__DLC[table__DLCa__DLC__free],_pid,\
201                       DLCnum,true);
202                       table_pid_channum[offspring] =
203                       chan__DLCa__DLC[table__DLCa__DLC__free];
204                       table_pid_channame[offspring] =
205                       chan__DLCa__DLC__select + table__DLCa__DLC__free;
206                       table_channame_channum[chan__DLCa__DLC__select +
207                       table__DLCa__DLC__free] =
208                       chan__DLCa__DLC[table__DLCa__DLC__free];
209                       if
210                          ::(offspring==0) -> pv__runtime_error = true;
211                          ::(offspring!=0) -> table__DLCa__DLC__free++;
212                       fi;
213                       :: else  -> pv__runtime_error = true;
214                       fi;
215                       }
216                   DLCs.val[DLCnum.val] = offspring;
217                 :: else ->
218                   /* Output signal : L_ReleaseInd */
219                   if
220                   :: true ->
221                     d_step{
222                       pfv_tmp=1;
223                       do
```

```
224                     :: pfv_tmp < table__environment__SUa__free ->
225                        pfv_tmp++ ;
226                     :: break;
227                     od;
228                     pfv_tmp--;
229                   };
230                   atomic{
231                   chan__environment__SUa[pfv_tmp]!L_ReleaseInd(_pid,DLCnum,\
232                   pcv__null);
233                   table_channum_nsp[table_channame_channum[\
234                   chan__environment__SUa__select + pfv_tmp]].data[\
235                   table_channum_ptr[table_channame_channum[\
236                   chan__environment__SUa__select + pfv_tmp]]].name =
237                   L_ReleaseInd;
238                   table_channum_ptr[table_channame_channum[\
239                   chan__environment__SUa__select + pfv_tmp]]++;
240                   }
241                 fi;
242                 goto ready;
243               fi;
244               /* ==== Decision end === */
245               goto waitUA;
246             } /* === end atomic transition ===*/
247             ::atomic{
248               atomic{recv__sig(true, L_SetparmReq) ->
249               input??L_SetparmReq(sender,_,undefined__V76paramTyp,_);
250               update_chan_tab();
251               }
252               V76para.XIDcmd = 0;
253               V76para.presen.val = (T_CHOICE__XIDcmd);
254               /* Output signal : V76frame */
255               if
256               :: true ->
257                 d_step{
258                   pfv_tmp=1;
259                   do
260                   :: pfv_tmp < table__dataLink__AtoB__free ->
261                      pfv_tmp++ ;
262                   :: break;
263                   od;
264                   pfv_tmp--;
265                 };
266                 atomic{
267                 chan__dataLink__AtoB[pfv_tmp]!V76frame(_pid,V76para);
268                 table_channum_nsp[table_channame_channum[\
269                 chan__dataLink__AtoB__select + pfv_tmp]].data[\
270                 table_channum_ptr[table_channame_channum[\
271                 chan__dataLink__AtoB__select + pfv_tmp]]].name=
272                 V76frame;
273                 table_channum_ptr[table_channame_channum[\
274                 chan__dataLink__AtoB__select + pfv_tmp]]++;
275                 }
276               fi;
277               goto ready;
278             } /* === end atomic transition ===*/
279             ::atomic{
280               atomic{recv__sig(true, V76frame) ->
281               input??V76frame(sender,_,V76para,_);
282               update_chan_tab();
283               }
284               /* === Decision start === */
285               if
286               :: ((V76para.presen.val) == (T_CHOICE__SABME)) ->
287                 DLCpeer.val = V76para.SABME.DLCi.val;
288                 /* === Decision start === */
289                 if
290                 :: ((DLCs.val[DLCpeer.val]) == (pid_undefined__value)) ->
291                   /* Output signal : L_EstabInd */
292                   if
293                   :: true ->
294                     d_step{
295                       pfv_tmp=1;
296                       do
```

```
297                     :: pfv_tmp < table__environment__SUa__free ->
298                        pfv_tmp++ ;
299                     :: break;
300                     od;
301                     pfv_tmp--;
302                   };
303                   atomic{
304                   chan__environment__SUa[pfv_tmp]!L_EstabInd(_pid,DLCpeer,\
305                   pcv__null);
306                   table_channum_nsp[table_channame_channum[\
307                   chan__environment__SUa__select + pfv_tmp]].\
308                   data[table_channum_ptr[table_channame_channum[\
309                   chan__environment__SUa__select + pfv_tmp]]].name=
310                   L_EstabInd;
311                   table_channum_ptr[table_channame_channum[\
312                   chan__environment__SUa__select + pfv_tmp]]++;
313                   }
314                 fi;
315                  atomic{
316                  set(tmr__DLCa__dispatch__T_402,70);
317                 }
318                 goto waitEstabResp;
319               :: else ->
320                 /* Output signal : L_ReleaseInd */
321                 if
322                 :: true ->
323                   d_step{
324                     pfv_tmp=1;
325                     do
326                     :: pfv_tmp < table__environment__SUa__free ->
327                        pfv_tmp++ ;
328                     :: break;
329                     od;
330                     pfv_tmp--;
331                   };
332                   atomic{
333                   chan__environment__SUa[pfv_tmp]!L_ReleaseInd(_pid,\
334                   DLCpeer,pcv__null);
335                   table_channum_nsp[table_channame_channum[\
336                   chan__environment__SUa__select + pfv_tmp]].data[\
337                   table_channum_ptr[table_channame_channum[\
338                   chan__environment__SUa__select + pfv_tmp]]].name=
339                   L_ReleaseInd;
340                   table_channum_ptr[table_channame_channum[\
341                   chan__environment__SUa__select + pfv_tmp]]++;
342                   }
343                 fi;
344                 V76para.DM.DLCi.val = DLCpeer.val;
345                 V76para.presen.val = (T_CHOICE__DM);
346                 /* Output signal : V76frame */
347                 if
348                 :: true ->
349                   d_step{
350                     pfv_tmp=1;
351                     do
352                     :: pfv_tmp < table__dataLink__AtoB__free ->
353                        pfv_tmp++ ;
354                     :: break;
355                     od;
356                     pfv_tmp--;
357                   };
358                   atomic{
359                   chan__dataLink__AtoB[pfv_tmp]!V76frame(_pid,V76para);
360                   table_channum_nsp[table_channame_channum[\
361                   chan__dataLink__AtoB__select + pfv_tmp]].data[\
362                   table_channum_ptr[table_channame_channum[\
363                   chan__dataLink__AtoB__select + pfv_tmp]]].name=
364                   V76frame;
365                   table_channum_ptr[table_channame_channum[\
366                   chan__dataLink__AtoB__select + pfv_tmp]]++;
367                   }
368                 fi;
369                 goto ready;
```

```
370                 fi;
371                 /* ==== Decision end === */
372               :: ((V76para.presen.val) == (T_CHOICE__UA)) ->
373                 V76para.presen.val = (T_CHOICE__UA);
374                 /* === Decision start === */
375                 if
376                 :: ((DLCs.val[V76para.UA.DLCi.val]) == (pid_undefined__value)) ->
377                   goto ready;
378                 :: else ->
379                 fi;
380                 /* ==== Decision end === */
381                 /* Output signal : V76frame */
382                 if
383                 :: table_pid_channame[DLCs.val[V76para.UA.DLCi.val]]==
384                 chan__DLCa__DLC__select ->
385                 table_pid_channum[DLCs.val[V76para.UA.DLCi.val]]!\
386                 V76frame(_pid,pcv__null,V76para,pcv__null);
387                 table_channum_nsp[table_pid_channum[DLCs.val[V76para.UA.DLCi.\
388                 val]]].data[table_channum_ptr[table_pid_channum[DLCs.val[\
389                 V76para.UA.DLCi.val]]]].name=V76frame;
390                 table_channum_ptr[table_pid_channum[DLCs.val[V76para.UA.DLCi.\
391                 val]]]++;
392
393                 :: table_pid_channame[DLCs.val[V76para.UA.DLCi.val]]==
394                 chan__DLCa__DLC__select + 1 ->
395                 table_pid_channum[DLCs.val[V76para.UA.DLCi.val]]!V76frame(\
396                 _pid,pcv__null,V76para,pcv__null);
397                 table_channum_nsp[table_pid_channum[DLCs.val[V76para.UA.DLCi.\
398                 val]]]. \
399                 data[table_channum_ptr[table_pid_channum[DLCs.val[V76para.UA.\
400                 DLCi.val]]]].name=V76frame;
401                 table_channum_ptr[table_pid_channum[DLCs.val[V76para.UA.DLCi.\
402                 val]]]++;
403
404                 :: table_pid_channame[DLCs.val[V76para.UA.DLCi.val]]==
405                 chan__dataLink__AtoB__select ->
406                 table_pid_channum[DLCs.val[V76para.UA.DLCi.val]]!V76frame(\
407                 _pid,V76para);
408                 table_channum_nsp[table_pid_channum[DLCs.val[V76para.UA.DLCi.\
409                 val]]]. \
410                 data[table_channum_ptr[table_pid_channum[DLCs.val[V76para.UA.\
411                 DLCi.val]]]].name=V76frame;
412                 table_channum_ptr[table_pid_channum[DLCs.val[V76para.UA.DLCi.\
413                 val]]]++;
414
415                 :: else -> pv__runtime_error = true
416                 fi;
417                 goto ready;
418               :: ((V76para.presen.val) == (T_CHOICE__I)) ->
419                 V76para.presen.val = (T_CHOICE__I);
420                 /* === Decision start === */
421                 if
422                 :: ((DLCs.val[V76para.I.DLCi.val]) == (pid_undefined__value)) ->
423                   goto ready;
424                 :: else ->
425                 fi;
426                 /* ==== Decision end === */
427                 /* Output signal : V76frame */
428                 if
429                 :: table_pid_channame[DLCs.val[V76para.I.DLCi.val]]==
430                 chan__DLCa__DLC__select ->
431                 table_pid_channum[DLCs.val[V76para.I.DLCi.val]]!V76frame(\
432                 _pid,pcv__null,V76para,pcv__null);
433                 table_channum_nsp[table_pid_channum[DLCs.val[V76para.I.DLCi.\
434                 val]]]. \
435                 data[table_channum_ptr[table_pid_channum[DLCs.val[V76para.I.\
436                 DLCi.val]]]].name=V76frame;
437                 table_channum_ptr[table_pid_channum[DLCs.val[V76para.I.DLCi.\
438                 val]]]++;
439
440                 :: table_pid_channame[DLCs.val[V76para.I.DLCi.val]]==
441                 chan__DLCa__DLC__select + 1 ->
442                 table_pid_channum[DLCs.val[V76para.I.DLCi.val]]!V76frame(\
```

```
443                 _pid,pcv__null,V76para,pcv__null);
444                 table_channum_nsp[table_pid_channum[DLCs.val[V76para.I.DLCi.\
445                 val]]]. \
446                 data[table_channum_ptr[table_pid_channum[DLCs.val[V76para.I.\
447                 DLCi.val]]]].name=V76frame;
448                 table_channum_ptr[table_pid_channum[DLCs.val[V76para.I.DLCi.\
449                 val]]]++;
450
451                 :: table_pid_channame[DLCs.val[V76para.I.DLCi.val]]==
452                 chan__dataLink__AtoB__select ->
453                 table_pid_channum[DLCs.val[V76para.I.DLCi.val]]!V76frame(_pid,\
454                 V76para);
455                 table_channum_nsp[table_pid_channum[DLCs.val[V76para.I.DLCi.\
456                 val]]]. \
457                 data[table_channum_ptr[table_pid_channum[DLCs.val[V76para.I.\
458                 DLCi.val]]]].name=V76frame;
459                 table_channum_ptr[table_pid_channum[DLCs.val[V76para.I.DLCi.\
460                 val]]]++;
461
462                 :: else -> pv__runtime_error = true
463                 fi;
464                 goto ready;
465               :: ((V76para.presen.val) == (T_CHOICE__DM)) ->
466                 V76para.presen.val = (T_CHOICE__DM);
467                 /* === Decision start === */
468                 if
469                 :: ((DLCs.val[V76para.DM.DLCi.val]) == (pid_undefined__value)) ->
470                   goto ready;
471                 :: else ->
472                 fi;
473                 /* ==== Decision end === */
474                 /* Output signal : V76frame */
475                 if
476                 :: table_pid_channame[DLCs.val[V76para.DM.DLCi.val]]==
477                 chan__DLCa__DLC__select ->
478                 table_pid_channum[DLCs.val[V76para.DM.DLCi.val]]!V76frame(\
479                 _pid,pcv__null,V76para,pcv__null);
480                 table_channum_nsp[table_pid_channum[DLCs.val[V76para.DM.DLCi.\
481                 val]]]. \
482                 data[table_channum_ptr[table_pid_channum[DLCs.val[V76para.DM.\
483                 DLCi.val]]]].name=V76frame;
484                 table_channum_ptr[table_pid_channum[DLCs.val[V76para.DM.DLCi.\
485                 val]]]++;
486
487                 :: table_pid_channame[DLCs.val[V76para.DM.DLCi.val]]==
488                 chan__DLCa__DLC__select + 1 ->
489                 table_pid_channum[DLCs.val[V76para.DM.DLCi.val]]!V76frame(\
490                 _pid,pcv__null,V76para,pcv__null);
491                 table_channum_nsp[table_pid_channum[DLCs.val[V76para.DM.DLCi.
492                 val]]]. \
493                 data[table_channum_ptr[table_pid_channum[DLCs.val[V76para.DM.\
494 DLCi.val]]]].name=V76frame;
495                 table_channum_ptr[table_pid_channum[DLCs.val[V76para.DM.DLCi.\
496                 val]]]++;
497
498                 :: table_pid_channame[DLCs.val[V76para.DM.DLCi.val]]==
499                 chan__dataLink__AtoB__select ->
500                 table_pid_channum[DLCs.val[V76para.DM.DLCi.val]]!V76frame(\
501                 _pid,V76para);
502                 table_channum_nsp[table_pid_channum[DLCs.val[V76para.DM.DLCi.\
503                 val]]]. \
504                 data[table_channum_ptr[table_pid_channum[DLCs.val[V76para.DM.\
505                 DLCi.val]]]].name=V76frame;
506                 table_channum_ptr[table_pid_channum[DLCs.val[V76para.DM.DLCi.\
507                 val]]]++;
508
509                 :: else -> pv__runtime_error = true
510                 fi;
511                 goto ready;
512               :: ((V76para.presen.val) == (T_CHOICE__DISC)) ->
513                 V76para.presen.val = (T_CHOICE__DISC);
514                 /* === Decision start === */
515                 if
```

```
516                 :: ((DLCs.val[V76para.DISC.DLCi.val]) ==
517                   (pid_undefined__value)) ->
518                   goto ready;
519                 :: else ->
520                 fi;
521                 /* ==== Decision end === */
522                 /* Output signal : V76frame */
523                 if
524                 :: table_pid_channame[DLCs.val[V76para.DISC.DLCi.val]]==
525                 chan__DLCa__DLC__select ->
526                 table_pid_channum[DLCs.val[V76para.DISC.DLCi.val]]!V76frame(\
527                 _pid,pcv__null,V76para,pcv__null);
528                 table_channum_nsp[table_pid_channum[DLCs.val[V76para.DISC.\
529                 DLCi.val]]].data[table_channum_ptr[table_pid_channum[DLCs.\
530                 val[V76para.\
531                 DISC.DLCi.val]]]].name=V76frame;
532                 table_channum_ptr[table_pid_channum[DLCs.val[V76para.DISC.\
533                 DLCi.val]]]++;
534
535                 :: table_pid_channame[DLCs.val[V76para.DISC.DLCi.val]]==
536                 chan__DLCa__DLC__select + 1 ->
537                 table_pid_channum[DLCs.val[V76para.DISC.DLCi.val]]!V76frame(\
538                 _pid,pcv__null,V76para,pcv__null);
539                 table_channum_nsp[table_pid_channum[DLCs.val[V76para.DISC.\
540                 DLCi.val]]]. \
541                 data[table_channum_ptr[table_pid_channum[DLCs.val[V76para.\
542                 DISC.DLCi.val]]]].name=V76frame;
543                 table_channum_ptr[table_pid_channum[DLCs.val[V76para.DISC.\
544                 DLCi.val]]]++;
545
546                 :: table_pid_channame[DLCs.val[V76para.DISC.DLCi.val]]==
547                 chan__dataLink__AtoB__select ->
548                 table_pid_channum[DLCs.val[V76para.DISC.DLCi.val]]!V76frame(\
549                 _pid,V76para);
550                 table_channum_nsp[table_pid_channum[DLCs.val[V76para.DISC.\
551                 DLCi.val]]].data[table_channum_ptr[table_pid_channum[DLCs.\
552                 val[V76para.DISC.DLCi.val]]]].name=V76frame;
553                 table_channum_ptr[table_pid_channum[DLCs.val[V76para.DISC.\
554                 DLCi.val]]]++;
555
556                 :: else -> pv__runtime_error = true
557                 fi;
558                 goto ready;
559               :: ((V76para.presen.val) == (T_CHOICE__XIDcmd)) ->
560                 V76para.presen.val = (T_CHOICE__XIDcmd);
561                 /* Output signal : L_SetparmInd */
562                 if
563                 :: true ->
564                   d_step{
565                     pfv_tmp=1;
566                     do
567                     :: pfv_tmp < table__environment__SUa__free ->
568                        pfv_tmp++ ;
569                     :: break;
570                     od;
571                     pfv_tmp--;
572                   };
573                   atomic{
574                   chan__environment__SUa[pfv_tmp]!L_SetparmInd(_pid,\
575                   pcv__null,pcv__null);
576                   table_channum_nsp[table_channame_channum[\
577                   chan__environment__SUa__select + pfv_tmp]].\
578                   data[table_channum_ptr[table_channame_channum[\
579                   chan__environment__SUa__select + pfv_tmp]]].name =
580                   L_SetparmInd;
581                   table_channum_ptr[table_channame_channum[\
582                   chan__environment__SUa__select + pfv_tmp]]++;
583                   }
584                 fi;
585                  atomic{
586                  set(tmr__DLCa__dispatch__T_401,70);
587                 }
588                 goto waitParmResp;
```

```
589                :: ((V76para.presen.val) == (T_CHOICE__XIDresp)) ->
590                  V76para.presen.val = (T_CHOICE__XIDresp);
591                  /* Output signal : L_SetparmConf */
592                  if
593                  :: true ->
594                    d_step{
595                      pfv_tmp=1;
596                      do
597                      :: pfv_tmp < table__environment__SUa__free ->
598                         pfv_tmp++ ;
599                      :: break;
600                      od;
601                      pfv_tmp--;
602                    };
603                    atomic{
604                    chan__environment__SUa[pfv_tmp]!L_SetparmConf(_pid,\
605                    pcv__null,pcv__null);
606                    table_channum_nsp[table_channame_channum[\
607                    chan__environment__SUa__select + pfv_tmp]].data[\
608                    table_channum_ptr[table_channame_channum[\
609                    chan__environment__SUa__select + pfv_tmp]]].name=
610                    L_SetparmConf;
611                    table_channum_ptr[table_channame_channum[\
612                    chan__environment__SUa__select + pfv_tmp]]++;
613                    }
614                  fi;
615                  goto ready;
616                :: else ->
617                  pv__runtime_error = true;
618                  goto ready;
619                fi;
620                /* ==== Decision end === */
621            } /* === end atomic transition ===*/
622            :: else ->
623                if
624                /* IMPLICITNI PREHOD */
625                ::atomic{recv__sig(true,L_EstabResp) ->
626                input??L_EstabResp(sender,_,undefined__V76paramTyp,_);
627                update_chan_tab();
628                goto ready;
629                }
630                ::atomic{recv__sig(true,L_SetparmResp) ->
631                input??L_SetparmResp(sender,_,undefined__V76paramTyp,_);
632                update_chan_tab();
633                goto ready;
634                }
635                :: else -> goto ready;
636                fi;
637            fi;
638        fi;
639      od;
640    fi;
641    goto ready;
642    /* SPONTANEOUS TRANSITION */
643    /* TIMERS */
644    /* TRAJEN SIGNAL */
645 od;
646 waitParmResp:
647 do
648 :: table_channum_ptr[input] > pv__cur ->
649    table_channum_prio[input]=false;
650    pv__cur++;
651    pv__ptr=0;
652    atomic{
653    do
654    :: pv__ptr <= cv__buff-1 ->
655       if
656       :: else -> set__clear();
657       fi;
658       pv__ptr++ ;
659    :: else -> goto waitParmResp_start;
660    od;
661    }
```

```
662 waitParmResp_start:
663    if
664    :: table_channum_prio[input]==true ->
665       pv__ptr=0;
666       do
667       :: (pv__ptr <= cv__buff-1)
668       && (table_channum_nsp[input].data[pv__ptr].prio==true) ->
669          if
670          /* PRIORITY INPUT */
671          :: else  -> skip;
672          fi;
673       :: (pv__ptr == cv__buff) -> break;
674       :: else -> pv__ptr++
675       od;
676    :: else ->
677       pv__ptr=0;
678       do
679       :: (pv__ptr <= cv__buff-1) ->
680          if
681          :: skip__save()
682          :: else ->
683             if
684             /* INPUT SIGNALS */
685             ::atomic{
686                atomic{recv__sig(true, L_SetparmResp) ->
687                input??L_SetparmResp(sender,_,undefined__V76paramTyp,_);
688                update_chan_tab();
689                }
690                if
691                :: atomic{input??[DLCa__dispatch__T_401(_,pcv__none,\
692                   undefined__V76paramTyp,pcv__none)] ->
693                   input??DLCa__dispatch__T_401(_,\
694                   pcv__none,undefined__V76paramTyp,pcv__none)
695                }
696                :: else -> skip
697                fi;
698                reset(tmr__DLCa__dispatch__T_401);
699                V76para.XIDresp = 0;
700                V76para.presen.val = (T_CHOICE__XIDresp);
701                /* Output signal : V76frame */
702                if
703                :: true ->
704                  d_step{
705                    pfv_tmp=1;
706                    do
707                    :: pfv_tmp < table__dataLink__AtoB__free ->
708                       pfv_tmp++ ;
709                    :: break;
710                    od;
711                    pfv_tmp--;
712                  };
713                  atomic{
714                  chan__dataLink__AtoB[pfv_tmp]!V76frame(_pid,V76para);
715                  table_channum_nsp[table_channame_channum[\
716                  chan__dataLink__AtoB__select + pfv_tmp]].data[\
717                  table_channum_ptr[table_channame_channum[\
718                  chan__dataLink__AtoB__select + pfv_tmp]]].name=
719                  V76frame;
720                  table_channum_ptr[table_channame_channum[\
721                  chan__dataLink__AtoB__select + pfv_tmp]]++;
722                  }
723                fi;
724                goto ready;
725             } /* === end atomic transition ===*/
726             ::atomic{
727                atomic{recv__sig(true, DLCa__dispatch__T_401) ->
728                input??DLCa__dispatch__T_401(sender,_,undefined__V76paramTyp,_);
729                update_chan_tab();
730                }
731                goto ready;
732             } /* === end atomic transition ===*/
733             :: else ->
734                if
```

```
735                 /* IMPLICITNI PREHOD */
736                 ::atomic{recv__sig(true,L_DataReq) ->
737                 input??L_DataReq(sender,_,undefined__V76paramTyp,_);
738                 update_chan_tab();
739                 goto waitParmResp;
740                 }
741                 ::atomic{recv__sig(true,L_EstabReq) ->
742                 input??L_EstabReq(sender,_,undefined__V76paramTyp,_);
743                 update_chan_tab();
744                 goto waitParmResp;
745                 }
746                 ::atomic{recv__sig(true,L_EstabResp) ->
747                 input??L_EstabResp(sender,_,undefined__V76paramTyp,_);
748                 update_chan_tab();
749                 goto waitParmResp;
750                 }
751                 ::atomic{recv__sig(true,L_ReleaseReq) ->
752                 input??L_ReleaseReq(sender,_,undefined__V76paramTyp,_);
753                 update_chan_tab();
754                 goto waitParmResp;
755                 }
756                 ::atomic{recv__sig(true,L_SetparmReq) ->
757                 input??L_SetparmReq(sender,_,undefined__V76paramTyp,_);
758                 update_chan_tab();
759                 goto waitParmResp;
760                 }
761                 :: else -> goto waitParmResp;
762                 fi;
763              fi;
764           fi;
765        od;
766     fi;
767     goto waitParmResp;
768     /* SPONTANEOUS TRANSITION */
769     /* TIMERS */
770      :: atomic{expire(tmr__DLCa__dispatch__T_401) ->
771     input!DLCa__dispatch__T_401(_pid,pcv__none,undefined__V76paramTyp,pcv__none);
772     table_channum_nsp[input].\
773     data[table_channum_ptr[input]].name = DLCa__dispatch__T_401;
774     table_channum_ptr[input]++;
775     reset(tmr__DLCa__dispatch__T_401);
776     }
777     /* TRAJEN SIGNAL */
778 od;
779 waitEstabResp:
780 do
781 :: table_channum_ptr[input] > pv__cur ->
782     table_channum_prio[input]=false;
783     pv__cur++;
784     pv__ptr=0;
785     atomic{
786     do
787     :: pv__ptr <= cv__buff-1 ->
788        if
789        :: set__save(V76frame)
790        :: set__prio(L_ReleaseReq)
791        :: else -> set__clear();
792        fi;
793        pv__ptr++ ;
794     :: else -> goto waitEstabResp_start;
795     od;
796     }
797 waitEstabResp_start:
798     if
799     :: table_channum_prio[input]==true ->
800        pv__ptr=0;
801        do
802        :: (pv__ptr <= cv__buff-1)
803        && (table_channum_nsp[input].data[pv__ptr].prio==true) ->
804           if
805           /* PRIORITY INPUT */
806           ::atomic{
807              atomic{recv__sig(true,L_ReleaseReq) ->
```

```
808             input??L_ReleaseReq(sender,DLCnum,undefined__V76paramTyp,_);
809             update_chan_tab();
810             }
811             if
812             :: atomic{input??[DLCa__dispatch__T_402(_,pcv__none,\
813                undefined__V76paramTyp,pcv__none)] ->
814                input??DLCa__dispatch__T_402(_,pcv__none,\
815                undefined__V76paramTyp,pcv__none)
816             }
817             :: else -> skip
818             fi;
819             reset(tmr__DLCa__dispatch__T_402);
820             /* === Decision start === */
821             if
822             :: ((DLCnum.val==DLCpeer.val) == true) ->
823               V76para.presen.val = (T_CHOICE__DM);
824               V76para.DM.DLCi.val = DLCpeer.val;
825               /* Output signal : V76frame */
826               if
827               :: true ->
828                 d_step{
829                   pfv_tmp=1;
830                   do
831                   :: pfv_tmp < table__dataLink__AtoB__free ->
832                      pfv_tmp++ ;
833                   :: break;
834                   od;
835                   pfv_tmp--;
836                 };
837                 atomic{
838                 chan__dataLink__AtoB[pfv_tmp]!V76frame(_pid,V76para);
839                 table_channum_nsp[table_channame_channum[\
840                 chan__dataLink__AtoB__select + pfv_tmp]].data[\
841                 table_channum_ptr[table_channame_channum[\
842                 chan__dataLink__AtoB__select + pfv_tmp]]].name=
843                 V76frame;
844                 table_channum_ptr[table_channame_channum[\
845                 chan__dataLink__AtoB__select + pfv_tmp]]++;
846                 }
847               fi;
848             :: ((DLCnum.val==DLCpeer.val) == false) ->
849             :: else -> pv__runtime_error = true;
850             fi;
851             /* ==== Decision end === */
852             goto ready;
853          } /* === end atomic transition ===*/
854          :: else  -> skip;
855          fi;
856       :: (pv__ptr == cv__buff) -> break;
857       :: else -> pv__ptr++
858       od;
859    :: else ->
860       pv__ptr=0;
861       do
862       :: (pv__ptr <= cv__buff-1) ->
863          if
864          :: skip__save()
865          :: else ->
866             if
867             /* INPUT SIGNALS */
868             ::atomic{
869                atomic{recv__sig(true, L_EstabResp) ->
870                input??L_EstabResp(sender,_,undefined__V76paramTyp,_);
871                update_chan_tab();
872                }
873                if
874                :: atomic{input??[DLCa__dispatch__T_402(_,pcv__none,\
875                   undefined__V76paramTyp,pcv__none)] ->
876                   input??DLCa__dispatch__T_402(_,\
877                   pcv__none,undefined__V76paramTyp,pcv__none)
878                }
879                :: else -> skip
880                fi;
```

```
881                reset(tmr__DLCa__dispatch__T_402);
882                atomic{
883                   if
884                   :: table__DLCa__DLC__free < table__DLCa__DLC__max ->
885                   offspring = run
886                   DLCa__DLC(chan__DLCa__DLC[table__DLCa__DLC__free],_pid,\
887                   DLCpeer,false);
888                   table_pid_channum[offspring] =
889                   chan__DLCa__DLC[table__DLCa__DLC__free];
890                   table_pid_channame[offspring] =
891                   chan__DLCa__DLC__select + \
892                   table__DLCa__DLC__free;
893                   table_channame_channum[chan__DLCa__DLC__select
894                   + table__DLCa__DLC__free] =
895                   chan__DLCa__DLC[table__DLCa__DLC__free];
896                   if
897                      ::(offspring==0) -> pv__runtime_error = true;
898                      ::(offspring!=0) -> table__DLCa__DLC__free++;
899                   fi;
900                   :: else  -> pv__runtime_error = true;
901                   fi;
902                   }
903                DLCs.val[DLCpeer.val] = offspring;
904                goto ready;
905             } /* === end atomic transition ===*/
906             ::atomic{
907                atomic{recv__sig(true, DLCa__dispatch__T_402) ->
908                input??DLCa__dispatch__T_402(sender,_,undefined__V76paramTyp,_);
909                update_chan_tab();
910                }
911                goto ready;
912             } /* === end atomic transition ===*/
913             :: else ->
914                if
915                /* IMPLICITNI PREHOD */
916                ::atomic{recv__sig(true,L_DataReq) ->
917                input??L_DataReq(sender,_,undefined__V76paramTyp,_);
918                update_chan_tab();
919                goto waitEstabResp;
920                }
921                ::atomic{recv__sig(true,L_EstabReq) ->
922                input??L_EstabReq(sender,_,undefined__V76paramTyp,_);
923                update_chan_tab();
924                goto waitEstabResp;
925                }
926                ::atomic{recv__sig(true,L_SetparmReq) ->
927                input??L_SetparmReq(sender,_,undefined__V76paramTyp,_);
928                update_chan_tab();
929                goto waitEstabResp;
930                }
931                ::atomic{recv__sig(true,L_SetparmResp) ->
932                input??L_SetparmResp(sender,_,undefined__V76paramTyp,_);
933                update_chan_tab();
934                goto waitEstabResp;
935                }
936                :: else -> goto waitEstabResp;
937                fi;
938             fi;
939          fi;
940       od;
941    fi;
942    goto waitEstabResp;
943    /* SPONTANEOUS TRANSITION */
944    /* TIMERS */
945     :: atomic{expire(tmr__DLCa__dispatch__T_402) ->
946    input!DLCa__dispatch__T_402(_pid,pcv__none,undefined__V76paramTyp,pcv__none);
947    table_channum_nsp[input].\
948    data[table_channum_ptr[input]].name = DLCa__dispatch__T_402;
949    table_channum_ptr[input]++;
950    reset(tmr__DLCa__dispatch__T_402);
951    }
952    /* TRAJEN SIGNAL */
953 od;
```

```
954 waitUA:
955 do
956 :: table_channum_ptr[input] > pv__cur ->
957    table_channum_prio[input]=false;
958    pv__cur++;
959    pv__ptr=0;
960    atomic{
961    do
962    :: pv__ptr <= cv__buff-1 ->
963       if
964       :: else -> set__clear();
965       fi;
966       pv__ptr++ ;
967    :: else -> goto waitUA_start;
968    od;
969    }
970 waitUA_start:
971    if
972    :: table_channum_prio[input]==true ->
973       pv__ptr=0;
974       do
975       :: (pv__ptr <= cv__buff-1)
976       && (table_channum_nsp[input].data[pv__ptr].prio==true) ->
977          if
978          /* PRIORITY INPUT */
979          :: else  -> skip;
980          fi;
981       :: (pv__ptr == cv__buff) -> break;
982       :: else -> pv__ptr++
983       od;
984    :: else ->
985       pv__ptr=0;
986       do
987       :: (pv__ptr <= cv__buff-1) ->
988          if
989          :: skip__save()
990          :: else ->
991             if
992             /* INPUT SIGNALS */
993             ::atomic{
994                atomic{recv__sig(true, DLCa__DLCstopped) ->
995                input??DLCa__DLCstopped(sender,DLCnum,undefined__V76paramTyp,_);
996                update_chan_tab();
997                }
998                /* Output signal : L_ReleaseInd */
999                if
1000               :: true ->
1001                 d_step{
1002                   pfv_tmp=1;
1003                   do
1004                   :: pfv_tmp < table__environment__SUa__free ->
1005                      pfv_tmp++ ;
1006                   :: break;
1007                   od;
1008                   pfv_tmp--;
1009                 };
1010                 atomic{
1011                 chan__environment__SUa[pfv_tmp]!L_ReleaseInd(_pid,DLCnum,\
1012                 pcv__null);
1013                 table_channum_nsp[table_channame_channum[\
1014                 chan__environment__SUa__select + pfv_tmp]].data[\
1015                 table_channum_ptr[table_channame_channum[\
1016                 chan__environment__SUa__select + pfv_tmp]]].name=
1017                 L_ReleaseInd;
1018                 table_channum_ptr[table_channame_channum[\
1019                 chan__environment__SUa__select + pfv_tmp]]++;
1020                 }
1021               fi;
1022               DLCs.val[DLCnum.val] = pid_undefined__value;
1023               goto ready;
1024            } /* === end atomic transition ===*/
1025            ::atomic{
1026               atomic{recv__sig(true, V76frame) ->
```

```
1027                input??V76frame(sender,_,V76para,_);
1028                update_chan_tab();
1029                }
1030                /* === Decision start === */
1031                if
1032                :: ((V76para.presen.val) == (T_CHOICE__UA)) ->
1033                  V76para.presen.val = (T_CHOICE__UA);
1034                  /* === Decision start === */
1035                  if
1036                  :: ((DLCs.val[V76para.UA.DLCi.val]) == (pid_undefined__value)) ->
1037                    goto waitUA;
1038                  :: else ->
1039                    /* Output signal : V76frame */
1040                    if
1041                    :: table_pid_channame[DLCs.val[V76para.UA.DLCi.val]]==
1042                    chan__DLCa__DLC__select ->
1043                    table_pid_channum[DLCs.val[V76para.UA.DLCi.val]]!V76frame\
1044                    (_pid,pcv__null,V76para,pcv__null);
1045                    table_channum_nsp[table_pid_channum[DLCs.val[V76para.UA.\
1046                    DLCi.val]]]. \
1047                    data[table_channum_ptr[table_pid_channum[DLCs.val[V76para.\
1048                    UA.DLCi.val]]]].name=V76frame;
1049                    table_channum_ptr[table_pid_channum[DLCs.val[V76para.UA.\
1050                    DLCi.val]]]++;
1051
1052                    :: table_pid_channame[DLCs.val[V76para.UA.DLCi.val]]==
1053                    chan__DLCa__DLC__select + 1 ->
1054                    table_pid_channum[DLCs.val[V76para.UA.DLCi.val]]!\
1055                    V76frame(_pid,pcv__null,V76para,pcv__null);
1056                    table_channum_nsp[table_pid_channum[DLCs.val[V76para.UA.\
1057                    DLCi.val]]]. \
1058                    data[table_channum_ptr[table_pid_channum[DLCs.val[V76para.\
1059                    UA.DLCi.val]]]].name=V76frame;
1060                    table_channum_ptr[table_pid_channum[DLCs.val[V76para.UA.\
1061                    DLCi.val]]]++;
1062
1063                    :: table_pid_channame[DLCs.val[V76para.UA.DLCi.val]]==
1064                    chan__dataLink__AtoB__select ->
1065                    table_pid_channum[DLCs.val[V76para.UA.DLCi.val]]!\
1066                    V76frame(_pid,V76para);
1067                    table_channum_nsp[table_pid_channum[DLCs.val[V76para.UA.\
1068                    DLCi.val]]]. \
1069                    data[table_channum_ptr[table_pid_channum[DLCs.val[V76para.\
1070                    UA.DLCi.val]]]].name=V76frame;
1071                    table_channum_ptr[table_pid_channum[DLCs.val[V76para.UA.\
1072                    DLCi.val]]]++;
1073
1074                    :: else -> pv__runtime_error = true
1075                    fi;
1076                    goto ready;
1077                  fi;
1078                  /* ==== Decision end === */
1079                :: ((V76para.presen.val) == (T_CHOICE__DM)) ->
1080                  V76para.presen.val = (T_CHOICE__DM);
1081                  /* === Decision start === */
1082                  if
1083                  :: ((DLCs.val[V76para.UA.DLCi.val]) == (pid_undefined__value)) ->
1084                    goto waitUA;
1085                  :: else ->
1086                    /* Output signal : V76frame */
1087                    if
1088                    :: table_pid_channame[DLCs.val[V76para.DM.DLCi.val]]==
1089                    chan__DLCa__DLC__select ->
1090                    table_pid_channum[DLCs.val[V76para.DM.DLCi.val]]!V76frame\
1091                    (_pid,pcv__null,V76para,pcv__null);
1092                    table_channum_nsp[table_pid_channum[DLCs.val[V76para.DM.\
1093                    DLCi.val]]]. \
1094                    data[table_channum_ptr[table_pid_channum[DLCs.val[V76para.\
1095                    DM.DLCi.val]]]].name=V76frame;
1096                    table_channum_ptr[table_pid_channum[DLCs.val[V76para.DM.\
1097                    DLCi.val]]]++;
1098
1099                    :: table_pid_channame[DLCs.val[V76para.DM.DLCi.val]]==
```

```
1100                    chan__DLCa__DLC__select + 1 ->
1101                    table_pid_channum[DLCs.val[V76para.DM.DLCi.val]]!V76frame\
1102                    (_pid,pcv__null,V76para,pcv__null);
1103                    table_channum_nsp[table_pid_channum[DLCs.val[V76para.DM.\
1104                    DLCi.val]]]. \
1105                    data[table_channum_ptr[table_pid_channum[DLCs.val[V76para.\
1106                    DM.DLCi.val]]]].name=V76frame;
1107                    table_channum_ptr[table_pid_channum[DLCs.val[V76para.DM.\
1108                    DLCi.val]]]++;
1109
1110                    :: table_pid_channame[DLCs.val[V76para.DM.DLCi.val]]==
1111                    chan__dataLink__AtoB__select ->
1112                    table_pid_channum[DLCs.val[V76para.DM.DLCi.val]]!V76frame\
1113                    (_pid,V76para);
1114                    table_channum_nsp[table_pid_channum[DLCs.val[V76para.DM.\
1115                    DLCi.val]]]. \
1116                    data[table_channum_ptr[table_pid_channum[DLCs.val[V76para.\
1117                    DM.DLCi.val]]]].name=V76frame;
1118                    table_channum_ptr[table_pid_channum[DLCs.val[V76para.DM.\
1119                    DLCi.val]]]++;
1120
1121                    :: else -> pv__runtime_error = true
1122                    fi;
1123                    goto ready;
1124                  fi;
1125                  /* ==== Decision end === */
1126                :: else ->
1127                  pv__runtime_error = true;
1128                  goto waitUA;
1129                fi;
1130                /* ==== Decision end === */
1131              } /* === end atomic transition ===*/
1132              :: else ->
1133                if
1134                /* IMPLICITNI PREHOD */
1135                ::atomic{recv__sig(true,L_DataReq) ->
1136                input??L_DataReq(sender,_,undefined__V76paramTyp,_);
1137                update_chan_tab();
1138                goto waitUA;
1139                }
1140                ::atomic{recv__sig(true,L_EstabReq) ->
1141                input??L_EstabReq(sender,_,undefined__V76paramTyp,_);
1142                update_chan_tab();
1143                goto waitUA;
1144                }
1145                ::atomic{recv__sig(true,L_EstabResp) ->
1146                input??L_EstabResp(sender,_,undefined__V76paramTyp,_);
1147                update_chan_tab();
1148                goto waitUA;
1149                }
1150                ::atomic{recv__sig(true,L_ReleaseReq) ->
1151                input??L_ReleaseReq(sender,_,undefined__V76paramTyp,_);
1152                update_chan_tab();
1153                goto waitUA;
1154                }
1155                ::atomic{recv__sig(true,L_SetparmReq) ->
1156                input??L_SetparmReq(sender,_,undefined__V76paramTyp,_);
1157                update_chan_tab();
1158                goto waitUA;
1159                }
1160                ::atomic{recv__sig(true,L_SetparmResp) ->
1161                input??L_SetparmResp(sender,_,undefined__V76paramTyp,_);
1162                update_chan_tab();
1163                goto waitUA;
1164                }
1165                :: else -> goto waitUA;
1166                fi;
1167              fi;
1168          fi;
1169      od;
1170  fi;
1171  goto waitUA;
1172  /* SPONTANEOUS TRANSITION */
```

```
1173    /* TIMERS */
1174    /* TRAJEN SIGNAL */
1175 od;
1176
1177 end: skip
1178 }
```

Biografija

Kandidat Boštjan Vlaovič se je rodil 24. januarja 1975 v Celju. Obiskoval je I. osnovno šolo v Celju, nato pa Srednjo tehniško šolo — elektrotehniški program v Celju. Leta 1993 se je vpisal na študij Gospodarskega inženirstva, smer elektrotehnika, na tedanji Tehniški fakulteti, VTO Elektrotehnika, računalništvo in informatika v Mariboru. Leta 1995 se je prepisal na študij elektrotehnike. Diplomiral je 7. 9. 1999 z zaključno oceno 9,29. Diplomsko delo *Generator klicev za telefonsko centralo MLB SI2000 V5* je napisal pod mentorstvom prof. dr. Zmaga Brezočnika. Od začetka srednješolske poti do zaključka študija je bil Zoisov štipendist. Obvezno študijsko prakso je izvajal v podjetju Iskratel v Kranju, kjer je šest mesecev delal na razvoju telefonske centrale SI2000 V5. Med študijem je aktivno sodeloval pri organizaciji strokovnih tečajev v okviru fakultete.

V študijskem letu 1999/2000 se je vpisal na podiplomski študij na Fakulteti za elektrotehniko, računalništvo in informatiko v Mariboru, program Elektrotehnika. Z znanstvenoraziskovalno dejavnostjo je na Inštitutu za elektroniko začel po končanem rednem študiju, kjer je med leti 1999 in 2003 bil zaposlen kot mladi raziskovalec pod mentorstvom prof. dr. Zmaga Brezočnika. Istega leta je bil izvoljen v znanstveno-raziskovalni naziv asistent raziskovalec. V letih 2000 do 2004 je sodeloval v programski skupini Napredne metode v telekomunikacijah. Področje njegovega dela je obsegalo omrežja naslednje generacije, protokole v telekomunikacijah in formalne metode. Leta 2001 mu je bil s sklepom senata fakultete odobren neposredni prehod na doktorski študij. Istega leta je bil izvoljen za asistenta za predmetno področje Elektronika. V letu 2002 je bil izvoljen v znanstveno-raziskovalni naziv samostojni raziskovalec. V zadnjem obdobju se je na znanstvenem področju posvečal predvsem formalnemu preverjanju sistemov s sočasnostjo z uporabo tehnike preverjanja modelov. Od leta 2004 je član programske skupine Napredne metode interakcij v telekomunikacijah.

Kandidatova bibliografija trenutno obsega 59 enot. Med njimi je članek v reviji druge kategorije (Elektrotehniški vestnik), po en strokovni in poljudni članek, vabljeno predavanje na mednarodni znanstveni konferenci, pet objavljenih znanstvenih prispevkov na mednarodnih konferencah (AI IAESTED 2000, WSEAS CSCC 2001, IEEE EUROCON 2001, TecNet 2002), šest objavljenih znanstvenih prispevkov na domačih konferencah (ERK), trije strokovni prispevki na konferencah, en objavljeni povzetek strokovnega prispevka na konferenci, en samostojni znanstveni sestavek v monografiji, en univerzitetni učbenik z recenzijo, eno predavanje na tuji univerzi ter 26 elaboratov in drugih zaključenih del, ki so nastala na osnovi večletnega sodelovanja z industrijo.

Že med dodiplomskim študijem je bil študentski član IEEE (od leta 1994). V študijskem letu 1999/2000 je vodil Študentsko vejo IEEE Univerze v Mariboru. V tem obdobju je organiziral simpozij “Extra Skills for Young Engineers” z mednarodno udeležbo. V študijskem letu 2001/2002 je aktivno sodeloval v skupini “Young Members Focus Group” kot eden izmed predstavnikov IEEE regije 8. Od leta 2001 je študentski član ACM, leta

2003 pa se je pridružil interesni skupini SIGPLAN (Special Interest Group on Programming Languages). Več let že aktivno sodeluje pri organizaciji Teleinfosa in dogodkov Vitel. Na štirinajsti delavnici Vitel je opravljal funkcijo predsednika organizacijskega odbora in bil urednik zbornika. Je tudi radioamater s klicnim znakom S56WBV. Še posebno ga zanimajo digitalne oblike komunikacije v okviru slovenskega radioamaterskega brezžičnega računalniškega omrežja AX.25, ki se kot prvo na svetu pripravlja za delovanje na frekvenčnih področjih nad 5 GHz.

Bibliografija

BOŠTJAN VLAOVIČ [20030]
Osebna bibliografija za obdobje 1975-2004

ČLANKI IN DRUGI SESTAVNI DELI

1.01 Izvirni znanstveni članek

VLAOVIČ, Boštjan
Analog subscriber call generator / Boštjan Vlaovič, Zmago Brezočnik.

V: Elektrotehniški vestnik. - ISSN 0013-5852. - Letn. 69, št. 5 (2002), str. 259-265.

COBISS.SI-ID 7689750 **1**

1.04 Strokovni članek

VLAOVIČ, Boštjan
Radioamatersko računalniško omrežje / [Boštjan Vlaovič, Robert Meolic].

V: Software power. - ISSN 1318-1750. - Št. 93 (2001), str. 10.

COBISS.SI-ID 6095894 **2**

1.05 Poljudni članek

ČASAR, Aleš
IEEE - ? / Aleš Časar in Boštjan Vlaovič.

V: Študentski utrip. - ISSN 1408-6077. - Leto 8, št. 4 (okt. 2000), str. 18.

COBISS.SI-ID 5753878 **3**

1.06 Objavljeni znanstveni prispevek na konferenci (vabljeno predavanje)

VLAOVIČ, Boštjan
IP telephony today and tomorrow / Boštjan Vlaovič, Zmago Brezočnik. - Bibliografija: str. 10. - Abstract.

V: Proceedings [Elektronski vir] / International Conference on Advances in Infrastructure for Electronic Business, Science, and Education on Internet, L'Aquila, July 31-August 6,2000. - [S. l.] : Scuola Superiore G. Reiss Romoli, 2000. - 10 str.

COBISS.SI-ID 5613078 **4**

1.08 Objavljeni znanstveni prispevek na konferenci

DEBEVC, Matjaž

Navidezna učilnica kot pripomoček pri študiju na daljavo / Matjaž Debevc, Boštjan Vlaovič, Rolando Radej.

V: Zbornik šeste Elektrotehniške in računalniške konference ERK '97, 25. - 27. september 1997, Portorož, Slovenija / uredil Baldomir Zajc. - Ljubljana : IEEE Region 8, Slovenska sekcija IEEE, 1997. - ISBN 961-6062-11-5. - Str. B/409-412.

COBISS.SI-ID 3105046 **5**

VLAOVIČ, Boštjan

Testing of switch node with Call generator software module / Boštjan Vlaovič, Zmago Brezočnik. - Bibliografija: str. 574. - Abstract.

V: Applied informatics [Elektronski vir] / editor M. H. Hamza. - Anaheim ; Calgary ; Zürich : IASTED/ACTA Press, 2000. - (Applied informatics, ISSN 1027-2666). - ISBN 0-88986-280-X. - Str. 568-574.

COBISS.SI-ID 5346582 **6**

VLAOVIČ, Boštjan

Uporaba jezika SDL pri predmetu Programska oprema v TK sistemih / Boštjan Vlaovič, Zmago Brezočnik. - Bibliografija: str. 354. - Abstract.

V: Zbornik devete Elektrotehniške in računalniške konference ERK 2000, 21. - 23. september 2000, Portorož, Slovenija / uredil Baldomir Zajc. - Ljubljana : IEEE Region 8, Slovenska sekcija IEEE, 2000. - ISBN 961-6062-19-0. - Zv. B, str. 351-354.

COBISS.SI-ID 5816342 **7**

FSK message indication service for analog telephone [Elektronski vir] / Simon Nedok ... [et al.]. - Soavtorji: Boštjan Vlaovič, Tatjana Kapus, Zmago Brezočnik. - Abstract.

V: Proceedings of the 5th WSES International Conference on Circuits, Systems, Communications and Computers (CSCC 2001) [and]Proceedings of the 3rd WSES International Conference on Mathematics and Computers in Physics (MCP 2001) [and] Proceedings of the 3rd WSES International Conference on Mathematics and Computers in Mechanical Engineering (MCME 2001) [računalniška datoteka] / editor N. E. Mastorakis. - [S.l.] : WSES : IEEE, 2001. - ISBN 960-8052-33-5. - [4] str.

COBISS.SI-ID 6522390 **8**

VLAOVIČ, Boštjan

Packet based telephony / Boštjan Vlaovič, Zmago Brezočnik. - Bibliografija: str. 213. -

Abstract.

V: Technical Program, Proceedings / International conference on trends in communications, Eurocon'2001, 4-7 July 2001, FEI STU Bratislava, Slovakia. - Bratislava : Institute of Electrical and Electronic Engineers, 2001. - ISBN 0-7803-6490-2. - Vol. 1, str. 210-213.

COBISS.SI-ID 6387734 **9**

PETROVIČ, Dragoslav
Orodja za uporabo omrežja MBone / Dragoslav Petrovič, Boštjan Vlaovič. - Bibliografija: str. 70. - Abstract.

V: Zbornik desete Elektrotehniške in računalniške konference ERK 2001, 24. - 26. september 2001, Portorož, Slovenija / uredil Baldomir Zajc. - Ljubljana : IEEE Region 8, Slovenska sekcija IEEE, 2001. - ISBN 961-6062-21-2. - Zv. B, str. 67-70.

COBISS.SI-ID 6541334 **10**

PETROVIČ, Dragoslav
MBone : ogrodno omrežje za oddajo več uporabnikom hkrati / Dragoslav Petrovič, Boštjan Vlaovič. - Bibliografija: str. 74. - Abstract.

V: Zbornik desete Elektrotehniške in računalniške konference ERK 2001, 24. - 26. september 2001, Portorož, Slovenija / uredil Baldomir Zajc. - Ljubljana : IEEE Region 8, Slovenska sekcija IEEE, 2001. - ISBN 961-6062-21-2. - Zv. B, str. 71-74.

COBISS.SI-ID 6541590 **11**

SIP-DSS1 Signalling gateway / Boštjan Vlaovič ... [et al.]. - Soavtorji: Simon Nedok, Peter Vicman, Zmago Brezočnik.

V: E-generation: technology for business momentum 2002 / Information technology and telecommunications in the Institute of Technology and Industry sectors (IT&T) annual conference, 30-31 October, Waterford, Ireland. - [Cork? : TecNet, 2002?]. - Str. 111-120.

COBISS.SI-ID 7495702 **12**

SIPIS - SIP Info Service / Peter Vicman ... [et al.]. - Soavtorji: Simon Nedok, Boštjan Vlaovič, Zmago Brezočnik. - Bibliografija: str. 125. - Abstract.

V: Proceedings of the IASTED International Conference Applied Informatics, Innsbruck, Austria, February 18-21, 2002 / editor: M. H. Hamza. - Anaheim ; Calgary ; Zurich : ACTA Press, cop. 2001. - (Applied informatics, ISSN 1027-2666). - ISBN 0-88986-321-0. - Str. 122-125.

COBISS.SI-ID 6914838 **13**

KREBS, Rudi
Prehod na konvergenčna omrežja z uporabo tuneliranja DSS1 v H.323 / Rudi Krebs, Boštjan Vlaovič, Zmago Brezočnik.

V: Zbornik dvanajste mednarodne Elektrotehniške in računalniške konference ERK 2003, 25. - 26. september 2003, Ljubljana, Slovenija / uredil Baldomir Zajc. - Ljubljana : IEEE Region 8, Slovenska sekcija IEEE, 2003. - (Zbornik ... Elektrotehniške in računalniške konference ERK ... ; 1581-4572). - Str. 131-134.

COBISS.SI-ID 8221462 **14**

ZORKO, Anton, 1974-
Digitalni prenos merilnih vrednosti v industrijskih sistemih / Anton Zorko, Boštjan Vlaovič, Zmago Brezočnik.

V: Zbornik dvanajste mednarodne Elektrotehniške in računalniške konference ERK 2003, 25. - 26. september 2003, Ljubljana, Slovenija / uredil Baldomir Zajc. - Ljubljana : IEEE Region 8, Slovenska sekcija IEEE, 2003. - (Zbornik ... Elektrotehniške in računalniške konference ERK ... ; 1581-4572). - Str. 311-314.

COBISS.SI-ID 8227350 **15**

1.09 Objavljeni strokovni prispevek na konferenci

DEBEVC, Matjaž
Interaktivna učna gradiva v omrežju Internet / Matjaž Debevc, Boštjan Vlaovič, Rajko Svečko.

V: Mednarodna izobraževalna računalniška konferenca - MIRK ’98, 20. maj - 22. maj 1998, OŠ Cirila Kosmača, Piran / [organizatorji Ministrstvo za šolstvo in šport ... [et al.]. - Ljubljana : Ministrstvo za šolstvo in šport : Zavod Republike Slovenije za šolstvo : Zavod za odprto družbo, 1998. - ISBN 961-234-122-2. - Str. 134-140.

COBISS.SI-ID 3674646 **16**

VLAOVIČ, Boštjan
Planet of visions / Boštjan Vlaovič. - Abstract.

V: Proceedings of the Symposium Extra Skills for Young Engineers, October 18-20, 2000, Maribor, Slovenia / edited by Boštjan Vlaovič ... [et al.]. - Maribor : Faculty of Electrical Engineering and Computer Science, 2000. - ISBN 86-435-0357-6. - Str. 51-53.

COBISS.SI-ID 5755158 **17**

VLAOVIČ, Boštjan
VoIP - protokola SIP in SDP / Boštjan Vlaovič, Zmago Brezočnik. - Ilustr.

V: TELEINFOS [Elektronski vir] / organizator Infos d.o.o. - Ljubljana : INFOS, 2001. - [32] str.

COBISS.SI-ID 6269462 **18**

1.13 Objavljeni povzetek strokovnega prispevka na konferenci

GLOBALNA virtualna šola - GVŠ / Matjaž Debevc ... [et al.]. - Soavtorji: Žarko Čučej, Boštjan Vlaovič, Dalibor Baškovč. - Seminar INFOS ’97, Cankarjev dom od 25. do 29. novembra 1997.

V: Seminar Infos ... [Elektronski vir]. - ISSN 1408-1393. - (1997).

COBISS.SI-ID 5397526 **19**

1.16 Samostojni znanstveni sestavek v monografiji

FSK message indication service for analog telephone / Simon Nedok ... [et al.]. - Soavtorji: Boštjan Vlaovič, Tatjana Kapus, Zmago Brezočnik. - Bibliografija: str. 118. - Abstract.

V: Communications World / edited by N. Mastorakis. - [S.l.] : WSES Press, cop. 2001. - (Electrical and computer engineering series). - ISBN 960-8052-38-6. - Str. 115-118.

COBISS.SI-ID 6386454 **20**

1.20 Predgovor, spremna beseda

VLAOVIČ, Boštjan
Foreword / Boštjan Vlaovič.

V: Proceedings of the Symposium Extra Skills for Young Engineers, October 18-20, 2000, Maribor, Slovenia / edited by Boštjan Vlaovič ... [et al.]. - Maribor : Faculty of Electrical Engineering and Computer Science, 2000. - ISBN 86-435-0357-6. - Str. VII-VIII.

COBISS.SI-ID 5873686 **21**

1.25 Drugi članki ali sestavki

VLAOVIČ, Boštjan
Maribor student chapter / Bostjan Vlaovic.

V: Region 8 news. - ISSN Y503-9398. - Vol. 3, no. 4 (2000), str. 13-14.

COBISS.SI-ID 6043926 **22**

VLAOVIČ, Boštjan
SB Maribor activities / Bostjan Vlaovic.

V: Region 8 news. - ISSN Y503-9398. - Vol. 4, no. 1 (2001), str. 14.

COBISS.SI-ID 6096150 **23**

MONOGRAFIJE IN DRUGA ZAKLJUČENA DELA

2.03 Univerzitetni ali visokošolski učbenik z recenzijo

DEBEVC, Matjaž
Oblikovanje dokumentov za svetovni splet / Matjaž Debevc, Boštjan Vlaovič. - Maribor : Fakulteta za elektrotehniko, računalništvo in informatiko, 2001. - 164 str.

1000 izv.
ISBN 86-435-0371-1

COBISS.SI-ID 45488641 **24**

2.05 Drugo učno gradivo

DEBEVC, Matjaž
Oblikovanje interaktivnega učnega gradiva za svetovni splet : zapiski predavanj / Matjaž Debevc, Boštjan Vlaovič, Aleš Živkovič. - Maribor : Fakulteta za elektrotehiko, računalništvo in informatiko, Center za razvoj študija na daljavo, 2000. - 1 zv. (loč. pag.)

COBISS.SI-ID 5397782 **25**

2.06 Priročnik, slovar, leksikon, atlas, zemljevid

CGI programiranje / Matjaž Debevc, Boris Lahovnik, Boštjan Vlaovič, David Vrtin. - Maribor : MAT design d.o.o., 1997. - 1-8 str. - (WIN.INI ; Let. 6. št. 2)

COBISS.SI-ID 2675990 **26**

2.11 Diplomsko delo

VLAOVIČ, Boštjan
Generator klicev za telefonsko centralo MLB SI2000 V5 : diplomsko delo univerzitetnega študija / Boštjan Vlaovič ; mentor Zmago Brezočnik ; somentor Monika Kapus-Kolar. - Maribor : [B. Vlaovič], 1999. - XX, 191 f. - (Fakulteta za elektrotehniko, računalništvo in informatiko)

COBISS.SI-ID 4828438 **27**

2.13 Elaborat, predštudija, študija

VLAOVIČ, Boštjan
SI 2000 / Boštjan Vlaovič. - Maribor : [Fakulteta za elektrotehniko, računalništvo in informatiko], 1999. - 91 str.

COBISS.SI-ID 6047254 **28**

ANALIZA dostopne programske kode za uporabo pri prehodu SIP-V5.2 / Skupina za telekomunikacije. - Maribor : Fakulteta za elektrotehniko, računalništvo in informatiko, 2001. - 35 str.

Avtorji: Boštjan Vlaovič, Peter Vicman, Simon Nedok, Zmago Brezočnik

COBISS.SI-ID 6133270 **29**

BREZOČNIK, Zmago
Analiza protokola MGCP za potrebe razpoznavalnika in konstruktorja / Skupina za telekomunikacije. - Maribor : Fakulteta za elektrotehniko, računalništvo in informatiko, 2001. - 33 str.

Nosilec projekta: Zmago Brezočnik. - Avtorji: Boštjan Vlaovič, Peter Vicman, Simon Nedok, Aleksander Vreže

COBISS.SI-ID 6759190 **30**

BREZOČNIK, Zmago

Funkcijska specifikacija krmilnika protokola za SIP telefon / Skupina za telekomunikacije. - Maribor : Fakulteta za elektrotehniko, računalništvo in informatiko, 2001. - 19 str.

Nosilec projekta: Zmago Brezočnik. - Avtorji: Boštjan Vlaovič, Peter Vicman, Simon Nedok, Aleksander Vreže

COBISS.SI-ID 6760214 **31**

FUNKCIJSKA specifikacija protokola MGCP za potrebe razpoznavalnika in konstruktorja / Skupina za telekomunikacije. - Maribor : Fakulteta za elektrotehniko, računalništvo in informatiko, 2001. - 29 str.

Avtorji: Boštjan Vlaovič, Peter Vicman, Simon Nedok, Aleksander Vreže. - Nosilec projekta: Zmago Brezočnik

COBISS.SI-ID 6760470 **32**

FUNKCIJSKA specifikacija protokolov SIP in SDP za prehod SIP-V5.2 / Skupina za telekomunikacije. - Maribor : Fakulteta za elektrotehniko, računalništvo in informatiko, 2001. - 22 str.

Avtorji: Boštjan Vlaovič, Peter Vicman, Simon Nedok, Zmago Brezočnik

COBISS.SI-ID 6293270 **33**

IZVEDBENA specifikacija krmilnika protokola za SIP telefon / Skupina za telekomunikacije. - Maribor : Fakulteta za elektrotehniko, računalništvo in informatiko, 2001. - 22 str.

Nosilec projekta: Zmago Brezočnik. - Avtorji: Boštjan Vlaovič, Peter Vicman, Simon Nedok, Aleksander Vreže

COBISS.SI-ID 6759702 **34**

PRIMERJALNA analiza tržno dobavljivih protokolnih skladov SIP in SDP / Skupina za telekomunikacije. - Maribor : Fakulteta za elektrotehniko, računalništvo in informatiko, 2001. - 91 str.

Avtorji: Boštjan Vlaovič, Simon Nedok, Peter Vicman, Zmago Brezočnik

COBISS.SI-ID 6046998 **35**

SDP (session description protocol) / Skupina za telekomunikacije. - Maribor : Fakulteta za elektrotehniko, računalništvo in informatiko, 2001. - 33 str.

Avtorji: Boštjan Vlaovič, Peter Vicman, Simon Nedok, Zmago Brezočnik

COBISS.SI-ID 6135830 **36**

SIP (Session Initiation Protocol) / Skupina za telekomunikacije. - Maribor : Fakulteta za elektrotehniko, računalništvo in informatiko, 2001. - 107 str.

Avtorji: Boštjan Vlaovič, Peter Vicman, Simon Nedok, Zmago Brezočnik

COBISS.SI-ID 6295062 **37**

TRŽNO dobavljivi skladi SIP in SDP / Skupina za telekomunikacije. - Maribor : Fakulteta za elektrotehniko, računalništvo in informatiko, 2001. - 49 f.

Avtorji: Boštjan Vlaovič, Simon Nedok, Peter Vicman, Zmago Brezočnik

COBISS.SI-ID 6046742 **38**

ANALIZA testnega okolja za protokol MGCP [**za Iskratel, Kranj**] / Skupina za telekomunikacije. - Maribor : Fakulteta za elektrotehniko, računalništvo in informatiko, 2002. - 7 f. - (SI2000)

Avtorji: Aleksander Vreže, Boštjan Vlaovič, Peter Vicman, Zmago Brezočnik

COBISS.SI-ID 7787030 **39**

FUNKCIJSKA specifikacija prehoda SIP-V5.2 / Skupina za telekomunikacije. - Maribor : Fakulteta za elektrotehniko, računalništvo in informatiko, 2002. - 61 str.

Nosilec projekta: Zmago Brezočnik. - Avtorji: Boštjan Vlaovič, Peter Vicman, Simon Nedok, Aleksander Vreže

COBISS.SI-ID 6909462 **40**

FUNKCIJSKA specifikacija sklada SIP MB-FERI / Skupina za telekomunikacije. - Maribor : Fakulteta za elektrotehniko, računalništvo in informatiko, 2002. - 18 f.

Avtorji: Boštjan Vlaovič, Peter Vicman, Simon Nedok, Aleksander Vreže, Zmago Brezočnik

COBISS.SI-ID 7072534 **41**

IZVEDBENA specifikacija protokola MGCP za potrebe razpoznavalnika in konstruktorja / Skupina za telekomunikacije. - Maribor : Fakulteta za elektrotehniko, računalništvo in informatiko, 2002. - 7 str.

Nosilec projekta: Zmago Brezočnik. - Avtorji: Boštjan Vlaovič, Peter Vicman, Simon Nedok, Aleksander Vreže

COBISS.SI-ID 6908694 **42**

IZVEDBENA specifikacija sklada SIP MB-FERI / Skupina za telekomunikacije. - Maribor : Fakulteta za elektrotehniko, računalništvo in informatiko, 2002. - 22 f.

Avtorji: Boštjan Vlaovič, Peter Vicman, Simon Nedok, Aleksander Vreže, Zmago Brezočnik

COBISS.SI-ID 7072790 **43**

POVEZLJIVOST CA in GW po protokolu MGCP [**za Iskratel, Kranj**] / Skupina za telekomunikacije. - Maribor : Fakulteta za elektrotehniko, računalništvo in informatiko, 2002. - 16 str. - (SI2000)

Avtorji: Peter Vicman, Boštjan Vlaovič, Aleksander Vreže, Zmago Brezočnik

COBISS.SI-ID 7787286 **44**

FUNKCIJSKA specifikacija testnega okolja za protokol MGCP [**za Iskratel, Kranj**] / Skupina za telekomunikacije. - Maribor : Fakulteta za elektrotehniko,

računalništvo in informatiko, 2003. - 18 f. - (SI2000)

Avtorji: Aleksander Vreže, Boštjan Vlaovič, Peter Vicman, Zmago Brezočnik

COBISS.SI-ID 7787542 **45**

IZVEDBENA specifikacija testnega okolja za protokol MGCP [**za Iskratel, Kranj**] / Skupina za telekomunikacije. - Maribor : Fakulteta za elektrotehniko, računalništvo in informatiko, 2003. - 20 f. - (SI2000)

Avtorji: Aleksander Vreže, Boštjan Vlaovič, Peter Vicman, Zmago Brezočnik

COBISS.SI-ID 7787798 **46**

SIGNALIZACIJI SIP in SIP-T [**za Iskratel**] : funkcijska specifikacija / avtor: Vidic Vlasta, Vilko Eržen, Fakutleta za elektrotehniko, računalništvo in informatiko Univerze v Mariboru. - Maribor : Fakulteta za elektrotehniko, računalništvo in informatiko, 2003. - 102 str. - (SI2000)

Avtorji FERI: Peter Vicman, Aleksander Vreže, Boštjan Vlaovič, Zmago Brezočnik

COBISS.SI-ID 8460310 **47**

2.25 Druge monografije in druga zaključena dela

VLAOVIČ, Boštjan

Protokol H.323 / Boštjan Vlaovič, Zmago Brezočnik. - Maribor : Fakulteta za elektrotehniko, računalništvo in informatiko, 2000. - 50 f.

COBISS.SI-ID 6048278 **48**

ORODJA za gradnjo razpoznavalnikov / Skupina za telekomunikacije - STEK. - Maribor : [Fakulteta za elektrotehniko, računalništvo in informatiko], 2001. - 37 f.

Avtorji: Boštjan Vlaovič, Peter Vicman, Simon Nedok, Zmago Brezočnik

COBISS.SI-ID 6297878 **49**

VLAOVIČ, Boštjan

Pregled protokola SIP / Boštjan Vlaovič. - Maribor : Fakulteta za elektrotehniko, računalništvo in informatiko, 2001. - 70 f.

COBISS.SI-ID 6298902 **50**

VLAOVIČ, Boštjan

Protokol SIP / Boštjan Vlaovič, Zmago Brezočnik. - Maribor : Fakulteta za elektrotehniko, računalništvo in informatiko, 2001. - 52 f.

COBISS.SI-ID 6048022 **51**

VLAOVIČ, Boštjan

Radioamater in paketna omrežja / Boštjan Vlaovič. - Maribor : Fakulteta za elektrotehniko, računalništvo in informatiko, 2001. - 82 f.

COBISS.SI-ID 6136854 **52**

PETROVIČ, Dragoslav
SDP / Dragoslav Petrovič, Boštjan Vlaovič, Zmago Brezočnik. - Maribor : Fakulteta za elektrotehniko, računalništvo in informatiko, 2001. - 49 f.

COBISS.SI-ID 6048534 **53**

IZVEDENA DELA (DOGODKI)

3.13 Organiziranje znanstvenih in strokovnih sestankov

SYMPOSIUM Extra skills for young engineers / organizing committee Boštjan Vlaovič ... [et al.]. - 2000

Soorganizatorji: Zmago Brezočnik, Filip Samo Balan, Aleš Časar, Bogdan Dugonik, Dejan Kastelic, Robert Meolic, Aleksander Vreže

COBISS.SI-ID 5753366 **54**

EXTRA skills for young engineers ESYE 2001 : second symposium, October 17-19, 2001, Maribor, Slovenia / program committee Zmago Brezočnik ... [et al.] ; organizing committee Saša Jevtić ... [et al.]. - 2001

COBISS.SI-ID 6650390 **55**

3.14 Predavanje na tuji univerzi

VLAOVIČ, Boštjan
IEEE engineer in the new millennium : invited paper at 4. kongres studenata elektrotehnike i računarstva Job fair 2001, Zagreb 26.3 - 30.3.2001 / Boštjan Vlaovič. - 2001

COBISS.SI-ID 6137366 **56**

SEKUNDARNO AVTORSTVO

Urednik

SYMPOSIUM Extra Skills for Young Engineers (2000 ; Maribor)
Proceedings of the Symposium Extra Skills for Young Engineers, October 18-20, 2000, Maribor, Slovenia / edited by Boštjan Vlaovič ... [et al.] ; organized by University of Maribor, IEEE Student Branch ... [et al.]. - Maribor : Faculty of Electrical Engineering and Computer Science, 2000. - X, 64 str.

150 izv.
ISBN 86-435-0357-6

COBISS.SI-ID 45324545 **57**

DELAVNICA o telekomunikacijah (14 ; 2003 ; Brdo pri Kranju)

Varnost v telekomunikacijskih sistemih : [zbornik referatov] / Štirinajsta delavnica o telekomunikacijah - VITEL, 19. in 20. maj 2003, Brdo pri Kranju, Slovenija ; [organizira] Elektrotehniška zveza Slovenije ; [uredili Boštjan Vlaovič ... et al.]. - Ljubljana : Elektrotehniška zveza Slovenije, 2003. - X, 114 str. - (Vitel)

COBISS.SI-ID 124426496 **58**

Komentor pri diplomskih delih

FERREIRA, Vitor

Voice communications over internet protocol networks : [diplomsko delo univerzitetnega študija] / Vitor Ferreira ; mentor Zmago Brezočnik ; komentor Boštjan Vlaovič. - Maribor : [V. Ferreiro], 2001. - XVI, 92 str. - (Fakulteta za elektrotehniko, računalništvo in informatiko)

COBISS.SI-ID 6442774 **59**