SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Marin Orlić

PREDVIĐANJE UPORABE RESURSA U SUSTAVIMA TEMELJENIM NA PROGRAMSKIM KOMPONENTAMA

DOKTORSKI RAD

Zagreb, 2010.

UNIVERSITY OF ZAGREB
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

Marin Orlić

RESOURCE USAGE PREDICTION IN COMPONENT-BASED SOFTWARE SYSTEMS

DOCTORAL DISSERTATION

Zagreb, 2010

Doktorski rad je izrađen na Sveučilištu u Zagrebu, Fakultetu elektrotehnike i računarstva, Zavodu za automatiku i računalno inženjerstvo

Mentor: Prof. dr. sc. Mario Žagar

Doktorski rad ima 142 stranice.

Doktorski rad br.

Doktorski rad ocijenilo je povjerenstvo u sastavu:

1. Dr. sc. Danko Basch, izvanredni profesor
   Sveučilište u Zagrebu Fakultet elektrotehnike i računarstva
2. Dr. sc. Mario Žagar, redoviti profesor
   Sveučilište u Zagrebu Fakultet elektrotehnike i računarstva
3. Dr. sc. Ivica Crnković, redoviti profesor
   Sveučilište Mälardalen, Švedska

Doktorski rad obranjen je pred povjerenstvom u sastavu:

1. Dr. sc. Danko Basch, izvanredni profesor
   Sveučilište u Zagrebu Fakultet elektrotehnike računarstva
2. Dr. sc. Mario Žagar, redoviti profesor
   Sveučilište u Zagrebu Fakultet elektrotehnike i računarstva
3. Dr. sc. Ivica Crnković, redoviti profesor
   Sveučilište Mälardalen, Švedska
4. Dr. sc. Igor Čavrak, docent
   Sveučilište u Zagrebu Fakultet elektrotehnike i računarstva
5. Dr. sc. Mario Kušek, docent
   Sveučilište u Zagrebu Fakultet elektrotehnike i računarstva

Doktorski rad obranjen je: 04. studenog 2010. godine

Zahvaljujem se svom mentoru, prof.dr.sc. Mariju Žagaru na strpljenju, podršci i vodstvu, te prof.dr.sc. Ivici Crnkoviću na energiji, savjetima i pruženim prilikama. Zahvalio bih se također i bivšim i sadašnjim suradnicima na Fakultetu elektrotehnike i računarstva na svim savjetima i idejama kojima su pomogli i dalje pomažu.

To all my friends at Mälardalen University and University of Paderborn, I feel proud to have met you and worked with you.

I na kraju, posebno mjesto zaslužuju moja strpljiva obitelj i Kristina koja me je bodrila i bockala kad je trebalo.

Sažetak

Ugradbeni sustavi kao posebna skupina računalnih sustava imaju sve širu primjenu. Složenost razvoja ugradbenih sustava postavlja mnoga pitanja – učinkovitost uporabe resursa samo je jedno od njih, no ključno za razvoj kvalitetne programske potpore. Razvoj u smjeru integracije funkcionalnosti više uređaja u jedan i potreba za bržim i jeftinijim razvojem samo će povećati potrebu za metodama i postupcima razvoja učinkovitih sustava. U razvoju sklopovlja uređaja uobičajena je analiza ponašanja sustava u ranoj fazi razvoja, što nije slučaj u razvoju programske potpore.

Disertacija daje pregled komponentnog modela i modela ponašanja, predlaže model izvedbenog okruženja programske potpore i mogućnosti integracije različitih modela programskih komponenata. Navedeni modeli nužni su kao pretpostavka za razvoj postupaka predviđanja promjene i uporabe resursa sustava temeljenih na programskim komponentama. Kao ogledni komponentni model uzet je komponentni model za ugradbene sustave ProCom razvijen na Sveučilištu Mälardalen, Švedska i pripadni jezik (model) za opis ponašanja REMES koji je još u razvoju. Postupci predviđanja promjene i uporabe resursa zasnovani su na simulacijskom modelu dobivenom integracijom različitih modela u cjeloviti model sustava. Ogledni alati dokazuju mogućnosti primjene postupaka za analizu ponašanja sustava u ranoj fazi razvoja, kada su dostupni samo modeli sustava bez podataka o konačnoj izvedbi sustava.

Ključne riječi: programsko inženjerstvo temeljeno na programskim komponentama, modeliranje ponašanja, analiza utroška resursa, opis ponašanja u vremenskoj domeni, komponentni modeli, modeli ponašanja, razvoj vođen modelom, ugradbeni sustavi.

Abstract

RESOURCE USAGE PREDICTION IN COMPONENT-BASED SOFTWARE SYSTEMS

Embedded systems, as a special class of computer systems have ever wider applications. Complexity of software development for embedded system raises many questions. Resource usage efficiency is just one of them, but nonetheless an essential one in the process of development of quality software. Development directions show future embedded systems will integrate functionality of many devices into one. These also indicate an increasing need for new methodologies and development processes resulting in development of efficient systems. Hardware development process is an example of system behavior analysis applied early on in development stages, a feat often missing in the process of software development.

The question this thesis attempts to answer is – is it possible to predict future system's resource usage while still in early design phase? Thesis gives an overview of component and behavior models, proposes a profile of an executive environment and discusses the possibilities of integration of various component models. Models are a necessity for the process of resource usage analysis of component-based software. ProCom component model and REMES behavior model developed at the Mälardalen University, Sweden, were chosen as reference models. Resource usage models used in this thesis allow specification of constraints and resource usage in real-time systems.

Resource usage prediction methods developed are based on simulation models resulting from the integration of various component models into integrated system model. Simulation was chosen as a method applicable in the early design phase when only partial design information is available, which hinders classic analysis, e.g. formal methods. Proposed simulation methods continues similar work with priced timed automata and hybrid system simulators, and is designed to adapt to REMES behavior model structure. As a part of a larger integrated development environment effort around ProCom component model, tooling was developed to demonstrate the possibilities of applying these methods in analysis of system resource-wise behavior in early design phase, with only limited system models and no available deployment models.

The contribution of this thesis is in the following: execution environment model for component-based systems, describing system resources and resource limitations; an integral system model, generated from partial structure and behavior models and execution environment models; methods to generate the integral system model from available models; behavior simulation methods that can be used to predict resource usage; tooling and a design environment that implement and automate model generation and analysis.

Proposed methodologies have been compared with state-of-the-art formal analysis tools and demonstrate comparable results. Future work should be directed towards improvement of end-user tools to enable further testing of proposed methods. Both underlying component and behavior models chosen as a basis for this work are still in development, and the simulation and model merge methods will likely need to adapt to changes with respect to changes in underlying models.

Keywords: component-based software engineering, behavior modeling, resource usage analysis, timing behavior, component models, behavior models, model-driven architecture, embedded systems.

Sadržaj

Popis slika

Popis tablica

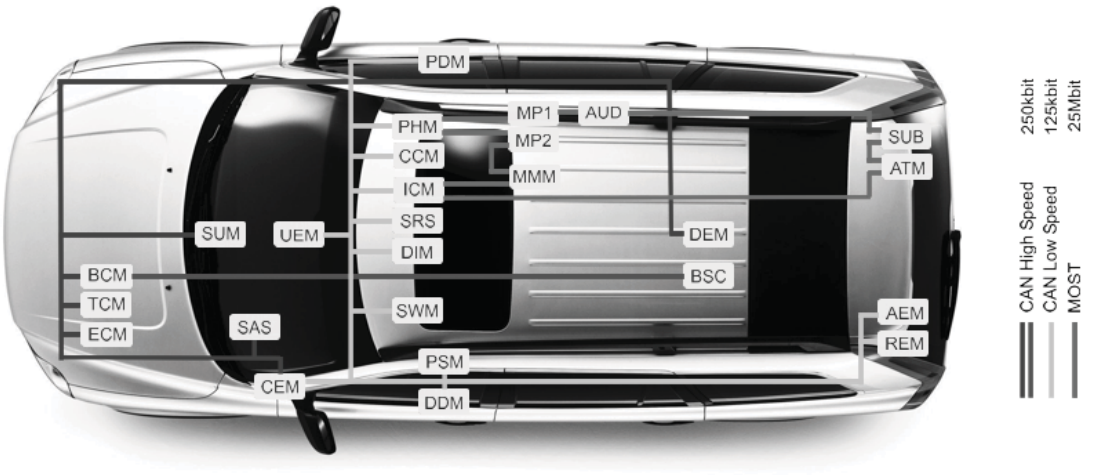
Popis algoritama

Poglavlje 1

Uvod

Širenje primjene ugradbenih računala i prenošenje sve većeg broja funkcionalnosti na programsku potporu ugradbenih računala uzrokovali su porast složenosti programske potpore i razvojnog procesa [94]. Neke situacije donose i dodatne zahtjeve kao što su mogućnost izdavanja serije relativno sličnih proizvoda nastalih na istoj osnovi, uz što je moguće kraći razvojni ciklus za svaki pojedini proizvod. Sve ovo zahtijeva prilagodbu postupaka razvoja i pripadnih alata posebnostima ugradbenih sustava – ograničenim resursima (izvedbene okoline – procesorske snage, memorije, energije, komunikacijskih mogućnosti) i radu u stvarnom vremenu (poštivanje rokova izvođenja operacija), uz mogućnost analize ponašanja sustava i detekcije nedozvoljenih stanja kao što je ulazak u potpuni zastoj (eng. *deadlock*).

Serije proizvoda zasnovane su na istoj osnovnoj platformi pri čemu svaki pojedini proizvod sadrži različitu konfiguraciju ugrađene programske potpore [25]. Razvojni proces takvih proizvoda zasniva se na višestrukim razvojnim ciklusima: dugom razvojnom ciklusu osnovne platforme i kratkom razvojnom ciklusu pojedinog proizvoda. Razvoj temeljen na komponentama obećava ponovnu iskoristivost jednom razvijene programske potpore na način koji štedi vrijeme za razvoj i testiranje, bilo kroz uporabu komercijalno dostupnih programskih komponenata ili kroz razvoj vlastitih. Ciljevi ovog pristupa su i ostvarenje predvidljivosti ponašanja i kvalitete krajnjeg proizvoda te izgradnja sustava od komponenata u različitim konfiguracijama za različite



Slika 1.1: Elektroničke upravljačke jedinice ugrađene u Volvo XC90 [47].

proizvode.

Automobilska industrija u literaturi se često izdvaja kao primjer primjene ugradbenih računala u režimu rada u stvarnom vremenu, s ograničenim resursima, sve većim sigurnosnim zahtjevima, naglim porastom funkcionalnosti i složenosti programske potpore uz istovremenu potrebu postojanja razvojnog procesa koji podržava serije proizvoda i uspostavlja arhitekturu koja podržava što brži razvoj konačnog proizvoda [32, 47, 34].

1.1 Motivacija

Prema istraživanju iz 2007. godine, u automobile tadašnje više klase (eng. *premium class*, njem. *Obere Mittelklasse*) ugrađuje se preko 70 upravljačkih jedinica (eng. *electronic control unit*, ECU), povezanih putem više od pet različitih sabirničkih podsustava (slika 1.1 prikazuje elektroničke upravljačke sustave ugrađene u Volvo XC90 [47, 34]). Od ukupnih proizvodnih troškova automobila, gotovo 40% otpada na elektroničke sustave i programsku potporu. Programska potpora koja se ugrađuje u automobile vrlo je široke namjene u rasponu od sustava kritičnih za sigurnost vozila i

putnika, upravljanje vozilom pa do zabave putnika i potpore uredskom radu. Ugrađenu programsku potporu možemo grupirati prema osnovnoj namjeni [80]:

- sučelje čovjek-stroj, multimedijalne i telematičke usluge,
- programska potpora za povećanje udobnosti u vožnji (eng. *body comfort software*),
- programska potpora elektroničke opreme za osiguranje sigurnosti,
- programska potpora upravljanja pogona i podvozja i
- infrastrukturna potpora.

Prema zahtjevima rada u stvarnom vremenu i obrade događaja koji se postavljaju pred ugrađenu programsku potporu, razlikujemo:

- rad u *mekom* stvarnom vremenu (eng. *soft real-time*), uglavnom obrada diskretnih događaja, potreba za povezivanjem sa sustavima izvan vozila: sučelje čovjek-stroj, multimedijalne i telematske usluge,
- rad u *mekom* stvarnom vremenu, obrada diskretnih događaja ima prednost nad upravljanjem: povećanje udobnosti u vožnji,
- rad u *tvrdom* (eng. *hard real-time*) i *mekom* stvarnom vremenu temeljen na obradi događaja: upravljanje *klasičnim* informacijskim sustavima vozila: dijagnostika i potpora osvježavanju programske potpore, infrastrukturna potpora,
- rad u *tvrdom* stvarnom vremenu, diskretni događaji, strogi zahtjevi sigurnosti: programska potpora opreme za osiguranje sigurnosti i
- rad u *tvrdom* stvarnom vremenu, algoritmi upravljanja imaju prednost nad obradom diskretnih događaja, strogi zahtjevi visoke dostupnosti: upravljanje pogonom i podvozjem.

Ukupno gledano, veličina i složenost strukture ugradbene programske potpore automobila iznimno je velika. Programska potpora izgrađena je nad operacijskim sustavima za rad u stvarnom vremenu i upravljačkim programima komunikacijskih sabirnica. Većina programske potpore mora odgovarati zahtjevima rada u stvarnom vremenu – kritičnom, *tvrdom* stvarnom vremenu, ili barem *mekom* stvarnom vremenu. Općenitost, složenost i širina zahtjeva je također osobita [32]:

- širok raspon korisnika – vozač, putnici, radnici na održavanju,
- širok raspon funkcija – od upravljanja u stvarnom vremenu do zabave, od funkcija udobnosti kao što je klimatizacija do aktivne pomoći vozaču, od upravljanja energijom do osvježavanja programske potpore,
- posebno održavanje.

Navedena raznolikost programske potpore i zahtjeva koji se pred nju postavljaju uzrokovala je nagli porast složenosti: u posljednjih trideset godina prosječan broj linija izvornog programskog kôda programske potpore koja se ugrađuje u automobile porasla je sa gotovo niti jedne na preko deset milijuna linija programskog kôda. U automobilima više klase, prema istraživanju iz 2007. godine, više od dvije tisuće raznih funkcija automobila ostvarene su programski ili su programski nadgledane. Obzirom na sve bržu smjenu generacija, spomenute karakteristike odgovaraju današnjim automobilima srednje klase. Za generaciju automobila više klase aktualnu u 2010. godini, predviđa se red veličine ugrađene programske potpore u vozilu od 1GB, dok vrijednost programske potpore iznosi 40% ukupne vrijednosti svih električkih, elektroničkih i programskih sustava u vozilima [55].

Istraživanje [80] predviđa daljnje povećanje uporabe i funkcionalnosti programske potpore ugrađene u automobile. Predviđeni trendovi daljnjeg razvoja su:

- potražnja za inovativnim novim funkcionalnostima i unaprijeđenjima postojećih,
- sve brže izmjene platforma i infrastrukture sustava,

- ubrzano povećanje troškova razvoja uz istovremenu potrebu konkurentnosti proizvoda,
- potreba za povećanjem kvalitete i pouzdanosti,
- sve kraće vrijeme razvoja (eng. *time-to-market*),
- povećane mogućnosti individualizacije proizvoda (sve veći broj *modela* istog proizvoda).

Slični trendovi razvoja mogu se predvidjeti i za druge tehničke proizvode, s različitim naglaskom na pojedine smjernice.

Automobili budućnosti umjesto velikog broja elektroničkih upravljačkih jedinica imati će manji broj centraliziranih višefunkcijskih jedinica, smanjen broj komunikacijskih kanala, senzora i aktuatora [32]. Umjesto današnjih 70 upravljačkih jedinica po automobilu, daljnji razvoj ponovno se kreće ka samo nekoliko namjenskih upravljačkih jedinica, prvenstveno za kritične sustave, te objedinjavanju ostalih funkcija u okviru manjeg broja općenamjenskih jedinica, sličnijih klasičnim računalima nego današnjim jedinicama. Takve promjene u organizaciji upravljačkih sustava zahtijevaju i nove tehnike i metode razvoja programskog inženjerstva, dok se utjecaj procesa i modela programskog inženjerstva širi na polje strojarstva i automatike. Klasična teorija upravljanja imala je velik utjecaj na razvoj automobilske industrije, no većina programske potpore koja se ugrađuje u automobile je temeljena na događajima, što zahtijeva postupke razvoja koji kombiniraju teoriju upravljanja sa sustavima diskretnih događaja (eng. *discrete event systems*).

U automobilskoj industriji, programska potpora postala je tako ključna tehnologija kojom se ostvaruje napredak funkcionalnosti i dok elektroničke upravljačke jedinice postaju sve jeftinije i mogu se smatrati robom opće namjene, upravo programska potpora određuje krajnju funkcionalnost i postaje dominantan, ali i ograničavajući faktor. I ne samo u automobilskoj industriji – programska potpora je ključni pokretač inovacija u tehničkim sustavima, programska potpora ostvaruju nove funkcije, stare funkcije implementiraju se na efikasniji način, uz povećanje kvalitete, smanjenje veličine, ci-

jene i potrošnje energije, a također se povezuju nekad odvojene funkcije u okviru novih višefunkcijskih uređaja. Smjer kojim se kreće razvoj, ka integraciji funkcija u manji broj uređaja donekle je u suprotnosti s očekivanjima većeg širenja masovno raspodijeljenih arhitektura za koje se vjerovalo da će složenim problemima pristupiti ugradnjom velikog broja procesnih jedinica u svaki element sustava [65, 74].

1.2 Postupci razvoja ugradbenih sustava

Računala kakva se danas pojavljuju u raznim sustavima, primjerice vozilima, su ugradbena računala – ugrađena računala posebne namjene, ugođena za određene zadatke uporabom odgovarajućeg sklopovlja i programske potpore. U usporedbi s klasičnim računalima opće namjene, jedna od određujućih karakteristika ugradbenih računala je da je njihov rad uvjetovan strogim ograničenjima, kako resursa (memorije, procesorske snage, energije napajanja, komunikacijskih mogućnosti), tako i radnih uvjeta u okolišu sa širokim rasponima radnih temperatura, jakim vibracijama i izloženom prašini. Osim toga, ugradbena računala koriste se u kritičnim primjenama s dodatnim zahtjevima na sigurnost, pouzdanost i osobine rada u stvarnom vremenu, pri čemu se od sustava očekuje odziv na događaje u poznatom vremenu. Primjena u kritičnim sustavima zahtijeva i iscrpno testiranje, pa i formalnu verifikaciju ispravnosti rada prema vremenskim i funkcionalnim zahtjevima.

Ograničenja dostupnih resursa i strogi zahtjevi sigurnosti, pouzdanosti i vremena odziva otežavaju razvoj ugradbenih sustava. Nadalje, kao što je navedeno, riječ je o razvoju umreženih, raspodijeljenih ugradbenih sustava – teškoće razvoja pojedinačnih sustava umnožavaju se kako raste njihov broj i broj međusobnih ovisnosti. Svaka upravljačka jedinica (ima ih tipično 30-70) izvodi programsku potporu reda veličine 1MB [47]. Klasičan pristup dekompozicije potrebnih funkcionalnosti u podsustave koji se implementiraju na zasebnim jedinicama postaje neprikladan s povećanjem funkcionalnosti i broja potrebnih jedinica – usko grlo postaje kako komunikacijski podsustav, tako i potreban fizički prostor, ali i računalni resursi. Kao rješenje, javlja se potreba smještaja više programskih podsustava na jednu upravljačku jedinicu, što

povećava složenost jer je potrebno dijeliti dostupne resurse [34].

U prošlosti se programska potpora ugradbenih sustava često razvijala na *ad hoc* način [45], no porast primjene i složenosti ugradbenih sustava zahtijeva promjenu pristupa razvoju ugradbenih sustava, od razmatranja pojedinih sastavnih dijelova prema razmatranju i analizi sustava kao cjeline [95]. Zato su posebno zanimljive mogućnosti primjene metoda dizajna vođenog modelom, koje nude mogućnosti formalne provjere pouzdanosti sustava i automatizacije procesa razvoja. Moguće rješenje vidi se u primjeni razvoja temeljenog na komponentama (eng. *component-based development*, CBD), koji izgradnju programske potpore vidi kao kombinaciju nezavisnih i dobro definiranih dijelova – komponenata. Razvoj temeljen na komponentama može olakšati upravljanje složenošću i daje mogućnost ponovne uporabe prethodno razvijenih i testiranih elemenata, te time povećanje pouzdanosti sustava uz istovremeno kraće vrijeme razvoja.

Razvoj temeljen na komponentama dokazao se u drugim poljima programskog inženjerstva – velikim poslovnim sustavima, sustavima temeljenim na uslugama, kao i klasičnim, svakodnevnim, računalnim sustavima [38]. Primjenjivost razvoja temeljenog na komponentama u domeni ugradbenih sustava ovisi o prilagodbi posebnostima ugradbenih sustava – vezama sa sklopovljem, raspodijeljenom radu, uvjetima sigurnosti i pouzdanosti te radu u stvarnom vremenu.

Osim navedenog komponente kao temelj razvoja ugradbenih računala prikladne su za primjenu u automobilskoj industriji iz dodatnih razloga. Automobilska industrija je tradicionalno vertikalno (modularno) podijeljena [32]. Mehanički dijelovi automobila su slijedom stotinjak godina kontinuiranog usavršavanja pretvoreni u zasebne podsustave koji se neovisno razvijaju i proizvode. Nezavisan razvoj i proizvodnja dijelova potiču široku raspodjelu posla u kojoj dobavljači preuzimaju osjetan dio razvoja, a proizvodnja se prepušta neovisnim tvrtkama (eng. *outsourcing*). Dijelove automobila tako proizvodi niz dobavljača, a proizvođači automobila ih u idealnom slučaju samo sastavljaju, te time raspodjeljuju rizik razvoja i smanjuju troškove.

Sve veći udio programske potpore u inovativnosti i vrijednosti automobila to mije-

nja:

- tradicionalno neovisne funkcije (npr. upravljanje vozilom, kočenje, nadzor motora) koje su dosad bile pod kontrolom vozača postaju međusobno povezane, automobil se iz sastavljenog pretvara u integrirani sustav – neočekivane i nenamjerne interakcije u takvom sustavu postaju problem,
- sastavljanje zasebnih dijelova pretvara se u integraciju sustava,
- ponašanje automobila postaje programirljivo – osobine kao što je ugodnost vožnje, upravljivost ili ovjes više nisu određene samo mehaničkim karakteristikama, već su i pod programskom kontrolom,
- troškovi izrade pod sve većim su utjecajem troškova razvoja programske potpore – tradicionalan model sume troška sastavnih dijelova više ne vrijedi.

Komponentizacija programske potpore trebala bi olakšati prijenos funkcionalnosti na programsku potporu i, barem djelomično, razvoj programske potpore uklopiti u okvire poslovnog procesa na kojega su proizvođači navikli – uporabu gotovih, prethodno (neovisno) razvijenih, testiranih i pouzdanih komponenata.

1.3 Ciljevi i metode istraživanja

Složenost razvoja ugradbenih sustava postavlja mnoga pitanja – učinkovitost uporabe resursa samo je jedno od njih, no ključno za razvoj kvalitetne programske potpore. Razvoj u smjeru integracije funkcionalnosti više uređaja u jedan i potreba za bržim i jeftinijim razvojem samo će povećati potrebu za metodama i postupcima razvoja učinkovitih sustava. U razvoju sklopovlja uređaja uobičajena je analiza ponašanja sustava u ranoj fazi razvoja, što nije slučaj u razvoju programske potpore. Pitanje na koje ova disertacija pokušava odgovoriti glasi: je li moguće predvidjeti uporabu resursa u ranoj fazi razvoja programske potpore?

Disertacija daje pregled komponentnog modela i modela ponašanja, predlaže model izvedbenog okruženja programske potpore i mogućnosti integracije različitih modela programskih komponenata. Navedeni modeli nužni su kao pretpostavke razvoja postupaka predviđanja promjene i uporabe resursa sustava temeljenih na programskim komponentama. Kao ogledni komponentni model uzet je komponentni model za ugradbene sustave ProCom razvijen na Sveučilištu Mälardalen, Švedska i pripadni jezik (model) za opis ponašanja REMES koji je tek u razvoju.

Razvoj modela i postupaka odvijao se kroz nekoliko koraka:

- pregled komponentnog modela ProCom i jezika REMES
- sudjelovanje u razvoju metamodela, analitičkog modela i formalne semantike jezika REMES u suradnji s istraživačima sa Sveučilišta Mälardalen,
- prijedlog modela izvedbenog okružja programskih komponenata – profila platforme za opis resursa i ograničenja resursa u izvedbenom okružju,
- prijedlog cjelovitog modela sustava izvedenog iz strukturnog modela, modela ponašanja i modela izvedbenog okružja,
- razvoj postupaka praćenja utroška resursa temeljenih na cjelovitom modelu sustava za analizu promjena stanja resursa,
- prijedlog alata za analizu utroška resursa na osnovi razvijenih postupaka, s ciljem jednostavnije analize komponentnih modela i modela ponašanja,
- komentar razvijenih rješenja na referentnim primjerima.

Komponentni model ProCom predstavlja pokušaj cjelovitog pristupa razvoju ugradbenih sustava kroz uporabu programskih komponenata. Autori komponentnog modela kao jedan od ciljeva komponentnog modela postavljaju uspostavljanje metoda koje će podržati razvoj temeljen na komponentama u svim fazama oblikovanja ugradbenih sustava, što postojeći postupci još uvijek ne podržavaju.

1.4 Organizacija disertacije

Disertacija je podijeljena u nekoliko cjelina. Uvodna poglavlja 2, 3 i 4 daju pregled principa razvoja sustava temeljenih na programskim komponentama te jezika za opis ponašanja i opis uporabe resursa ugradbenih sustava. Poglavlje 5 opisuje razvijeni model za opis izvedbenog okružja ugradbenih sustava, a poglavlje 6 daje pregled cjelovitog modela sustava izvedenog iz komponentnog modela, jezika za opis ponašanja i modela izvedbenog okružja. Cjeloviti model sustava preduvjet je postupcima analize utroška resursa opisanih u poglavlju 7. Razvijeni postupci implementirani su u skupu alata za analizu utroška resursa opisanih u poglavlju 8. Poglavlje 9 uspoređuje rezultate analize sustava dobivene razvijenim postupcima sa onim dobivenim općeprihvaćenim alatima. Poglavlje 10 daje komentar ostvarenih rezultata i nekoliko smjerova budućeg razvoja.

Poglavlje 2

Razvoj sustava temeljenih na programskim komponentama

Programsko inženjerstvo temeljeno na programskim komponentama promatra komponente kao osnovne jedinice apstrakcije programske složenosti i nastoji ostvariti dva osnovna cilja: izgradnju sustava kroz kompoziciju programskih komponenata i ponovnu uporabu jednom izgrađenih komponenata [56, 38]. Principom ponovne uporabe (eng. *reusability*) jednom razvijene programske potpore, nastoji se izbjeći probleme koji opterećuju razvoj sustava – kašnjenje, probijanje predviđenog budžeta, nezadovoljavajuću kvalitetu i stalan rast troškova održavanja [38]. Ponovna uporaba ugrađeno je u razvojni proces programskog inženjerstva temeljenog na komponentama – komponente su osnovni sastavni elementi sustava, unaprijed pripremljeni za ponovnu uporabu i integraciju u različite krajnje sustave čime se stvara ušteda vremena za razvoj i testiranje.

Iako ne postoji konsenzus o tome što je to programska komponenta, opća definicija komponente je *ponovno iskoristiva jedinica za ugradnju i kompoziciju*, što je naizgled čini sličnom konceptu objektno-orijentiranog razvoja. Druga opće prihvaćena definicija programske komponente je *programska jedinica koja drugima* nudi *i od drugih* zahtijeva *određene usluge* [62, 63].

Pristup uslugama ili funkcijama komponente ostvaruje se kroz njeno *sučelje* (eng. *interface*) – koje opisuje javno ponuđene funkcionalnosti komponente. Osim sučelja, za opis komponente nužna je i specifikacija *ugovora* (eng. *contract*), pa se programska komponenta preciznije definicira kao *jedinica kompozicije s ugovorom određenim sučeljem i eksplicitno određenim ovisnostima, pripremljena za neovisnu uporabu* [88, 38]. Iz takvog opisa jasno je da je komponenta jasno odijeljena od svoje okoline s kojom komunicira sučeljem, dok je stvarna implementacija skrivena i nedostupna. Za uspješnu neovisnu uporabu od strane korisnika koji nisu ujedno i autori komponente, potreban je potpun opis komponente – sučelja, funkcionalnih svojstava (ugovora), nefunkcionalnih svojstava (kvalitete, performanse, zahtjeva za resursima), načina uporabe i sl. Trenutne tehnologije za razvoj sustava temeljenih na programskim komponentama tipično se fokusiraju na deklaraciju sučelja analognu opisu klase objektno-orijentiranih programskih jezika, kao popis operacija i svojstava (atributa), s detaljima semantike funkcionalnosti opisanima u dokumentaciji – prirodnim, a ne formalnim jezikom. Potpora za opis nefunkcionalnih svojstava komponenata na istoj je ili nižoj razini.

Takav pristup prikladan je za izgradnju klasične programske potpore široke namjene, no primjenjiv je i na druge vrste programskih proizvoda. U općoj teoriji računarstva, vrijeme izvođenja se ne smatra ključnim, a osnovni je cilj ostvariti svojstva izračunljivosti (eng. *computability*) i okončavanja (eng. *termination*), te je princip najboljih napora (eng. *best-effort*) sasvim zadovoljavajući. U slučaju programske potpore za ugradbene sustave, takav pristup nije primjenjiv. Ugradbeni sustavi direktno su ugrađeni u procese koji nas okružuju (televizore, automobile, bijelu tehniku, telefone, rashladne sustave i dr.) i određeni su vremenskim ograničenjima tih procesa [64]. Tradicionalni razvoj ugradbenih sustava kao sustava temeljenih na malim računalima, s ograničenim procesnim kapacitetom i resursima kao što su memorija i energija, daje odgovarajuće rezultate samo u smjeru u kojem se to i može očekivati – upravo u optimiranju utroška tih resursa. Osim dobrim iskorištavanjem resursa, u nekim se slučajevima kvaliteta ugradbenih sustava odražava i u poštivanju zahtjeva sigurnog (eng. *safe*) rada procesa u kojeg su ugrađeni. Za postizanje takvih zahtjeva nužno je poštivanje vremenskih ograničenja i rokova za izvođenje pojedinih akcija (npr. aktiviranje

sustava za sprječavanje proklizavanja automobila) i drugih osobina rada u stvarnom vremenu [32].

Za postizanje sigurnog rada takvih sustava nije dovoljno klasično testiranje u radu – potrebno je formalno provjeriti sva stanja u kojima se sustav može naći i njihovu prihvatljivost ispitivanjem modela sustava (eng. *model-checking*), te dodatno i uključivanjem vremenskih zahtjeva u formalnu analizu sustava. Primjenom principa razvoja temeljenog na komponentama na ugradbene sustave, potrebno je opis komponenata proširiti nefunkcionalnim svojstvima i vremenskim zahtjevima koji su jednako važni kao i principi ponovne uporabe. Krajnji cilj prilikom izgradnje programske potpore za ugradbene sustave je ostvariti predvidljivost njegovog ponašanja.

Opis strukture sustava izgrađenog od komponenata je njegov komponentni model, specijalizacija komponentnog meta-modela (oba termina, model i meta-model koriste se ravnopravno za komponentni meta-model [24]). Komponentni model važan je element u pristupu razvoja temeljenog na programskim komponentama, komponenta opisom i građom odgovara komponentnom modelu, a norma kompozicije opisana komponentnim modelom određuje načine kompozicije komponenata [56]. Komponentni model tako pruža okvir za opis komponenata, njihovu građu, kompoziciju i uporabu, te analizu uvjeta i posljedica svih operacija nad komponentama.

Strukturni model sustava može se proširiti modelom ponašanja sustava ili komponenata koje čine sustav, čime se stvaraju preduvjeti za formalnu analizu ponašanja sustava, uz preduvjet dovoljne razine detaljnosti opisa – primjerice, analiza uvjeta rada u stvarnom vremenu moguća je jedino ako su u modelu prisutne odgovarajuće informacije o trajanju izvođenja akcija. Daljnjim proširenjem opisa ponašanja informacijama o utrošku resursa moguće je na osnovu modela izvesti zaključke o ponašanju ukupnog sustava u odnosu na utrošak resursa.

2.1 Komponentni modeli

Komponentni model centralni je element procesa razvoja i podloga za primjenu principa razvoja temeljenog na komponentama, počevši od grube specifikacije sustava nastale na osnovi početnih zahtjeva pa sve do krajnje, detaljne specifikacije i implementacije. Struktura komponentnog modela prilagođena je ciljnoj primjeni i usmjerava ili ograničava razvoj u tom smjeru. Odabir neprikladnog komponentnog modela znatno otežava razvoj jer takav model ne poznaje koncepte i nema svojstva koja su potrebna u razvoju. Odabir komponentnog modela opće namjene za razvoj sustava zasnovanog na uslugama prikladan je sve dok nije potrebno npr. analizirati kvalitetu usluge, za što su potrebni atributi kvalitete usluge koje komponentni model opće namjene vjerojatno ne poznaje. Bolji odabir je komponenteni model koji je dodatno prilagođen takvim potrebama (npr. komponentni model Palladio [83]). Dobre strane široko prihvaćenog komponentnog modela, npr. potpora korisnicima i veliki skup alata zasnovanih na tom modelu, sukobljavaju se sa tada s njegovim manama – nedostatkom potpore za analizu i predviđanje specifičnih karakteristika sustava.

Opis komponente u komponentnom modelu slijedi ciljeve koji su pred model postavljeni. Svi komponentni modeli opisuju sučelje komponente na sličan način. Razlike se prvenstveno javljaju u metapodacima – od općenitih svojstava kao što je opis okoline potrebne za izvođenje komponente i međuovisnosti komponenata pa do opisa svojstava kvalitete komponente, utroška resursa, ponašanja ili vremenskih karakteristika komponente.

Kompozicija komponenata moguća je u dva smjera:

- horizontalna kompozicija – *sastavljanje* sustava od komponenata njihovim povezivanjem i
- vertikalna kompozicija – ostvarivanje *hijerarhije* složenim komponentama.

U nastavku je dan pregled komponentnog modela ProCom, podloge za razvoj i analizu komponenata za ugradbene sustave.

2.2 Pregled komponentnog modela ProCom

Komponentni model ProCom nastao je kao logičan nastavak svog prethodnika, komponentnog modela SaveCCM [2, 3, 35], a donekle i pod utjecajem komponentnog modela Rubus [54]. Razvojni proces izgrađen oko komponentnog modela ProCom promatra komponentu kao skup informacija potrebnih i određenih u različitim fazama razvoja sustava [33]. Informacije vezane uz pojedinu komponentu su zahtjevi, izvorni kôd i dokumentacija, ali i različiti modeli komponente (npr. opis sučelja, ponašanja i vremenskih karakteristika) te podaci dobiveni eksperimentalno ili analitički (podaci o performansama, utrošku resursa i dr.). Ključna osobina komponente koju razvojni proces nastoji ostvariti je predvidljivost. Ostvarenje predvidljivosti nastoji se postići potporom analizama koje bi trebale dati ocjene i garancije ostvarenja svojstava sustava. Analize u ranim fazama razvoja sustava mogu dati manje ili više točne ocjene svojstava, u ovisnosti o potpunosti i točnosti izgrađenih modela. Takve rane ocjene mogu pomoći u donošenju odluka o dizajnu sustava, dok se analizama izgrađenog sustava može potvrditi sukladnost sustava postavljenim zahtjevima. Skup planiranih analitičkih metoda temeljenih na komponentnom modelu ProCom uključuje, između ostalog, analizu funkcionalne sukladnosti (eng. *functional compliance*), predviđanje pouzdanosti, analizu najgoreg vremena izvođenja (eng. *worst-case execution time*) i analizu utroška resursa [34].

Složenost ugradbenih sustava opisana u prethodnim poglavljima razlikuje se na različitim razinama sustava – pojedini podsustavi razlikuju se od elemenata visoke razine prema informacijama koje su potrebne za njihovo oblikovanje i analizu, mogućim analizama, pa i u strukturi prikladnih modela. U općem slučaju sustavi se mogu razložiti na relativno neovisne podsustave složenih funkcionalnosti. I sami raspodijeljeni, takvi podsustavi komuniciraju razmjenom poruka zajedničkom sabirnicom ili mrežom, čime dijele podatke ili obavijesti o promjeni stanja. Na nižoj razini, unutar podsustava razlikuju se dijelovi jednostavnije funkcionalnosti, smješteni unutar jedne fizičke upravljačke jedinice, s jednostavnijim i manje naglašenim komunikacijskim potrebama, no strožih vremenskih i sinkronizacijskih zahtjeva te ograničenja resursa.

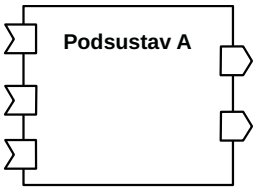
Iz ovog razloga komponentni model ProCom podijeljen je u dvije, međusobno povezane razine. Viša razina, **ProSys**, promatra sustav kao skup povezanih aktivnih podsustava (eng. *subsystems*) čije izvođenje se odvija paralelno, a međusobna komunikacija temelji na razmjeni poruka. Niža razina, **ProSave**, određuje internu strukturu podsustava kao skup međusobno povezanih primitivnih i složenih komponenata. Podsustavi razine ProSys aktivni su elementi, dok su komponente razine ProSave pasivne i reaktivne, a komunikacija je temeljena na paradigmi cjevovoda i filtara (eng. *pipes and filters*). Elementi obje razine opisani su nizom modela – strukturnim modelima ProCom, modelima ponašanja i dr. Sljedeća poglavlja daju pregled elemenata komponentnog modela ProCom, te njihovu grafičku (konkretnu) sintaksu i pripadni (meta)model (apstraktnu sintaksu).

2.2.1 Razina ProSys

Gradivni elementi sustava na razini ProSys su *podsustavi*. Podsustavi su aktivni, izvode se paralelno s ostalim podsustavima i komuniciraju porukama. Po građi podsustavi nisu uvijek atomički, već mogu biti sastavljeni od manjih podsustava u hijerarhijskoj strukturi. U okviru razvoja temeljenog na komponentama, možemo reći da je na razini ProSys *podsustav* analogan pojmu *komponente* i može se zasebno razvijati, promatrati, analizirati i ponovno koristiti. Neovisno i paralelno izvođenje podsustava podrazumijeva mogućnost njihove fizičke izvedbe na različitim upravljačkim jedinicama u sustavu, pa i raspodjeljivanju jednog podsustava na više upravljačkih jedinica, što nije predmet ovog razmatranja.

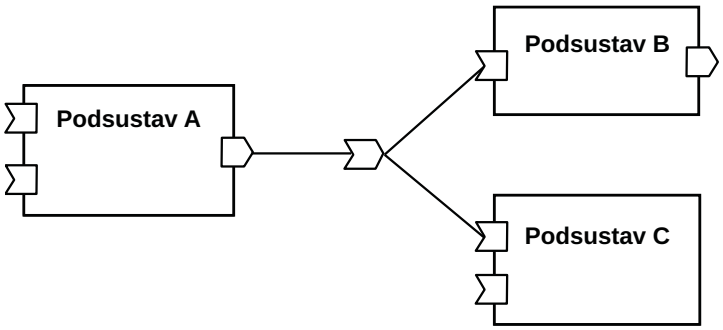
Slika 2.1 pokazuje grafički prikaz podsustava. Sučelje podsustava određeno je ulaznim i izlaznim *vratima* za poruke (eng. *input message ports, output message ports*) koja opisuju poruke što ih podsustav prima odnosno šalje. Vratima je pridijeljen *tip* poruke, no mogući tipovi i njihova interpretacija nisu određeni komponentnim modelom, već se je to ostavljeno korisnicima i alatima za analizu. Već spomenuto svojstvo podsustava da su aktivni odražava se u mogućnosti da aktivnosti koje se izvode ne ovise o vanjskoj aktivaciji, već mogu biti rezultat interne obrade, internog događaja, doga-

đati se periodički i slično. Poruke koje podsustav prima obrađuju se i također mogu izazvati određene (reaktivne) akcije. Veze među podsustavima ostvaruju se preko vrata



Slika 2.1: Grafički prikaz podsustava s ulaznim i izlaznim vratima za poruke.

za poruke. Nije dopušteno direktno povezivanje vrata za poruke, već se veza ostvaruje preko komunikacijskih *kanala* za poruke (eng. *message channel*). Kanali dopuštaju višestruke, N:N veze među podsustavima. Kanali ostvaruju povezivanje informacija s komunikacijskim kanalom umjesto s izvorom ili odredištem informacije, te je tako moguće odrediti informacije potrebne za rad sustava prije nego što postane poznat njihov izvor ili odredište (slika 2.2). Komunikacijski kanali su asinkroni i nemaju ograniče-

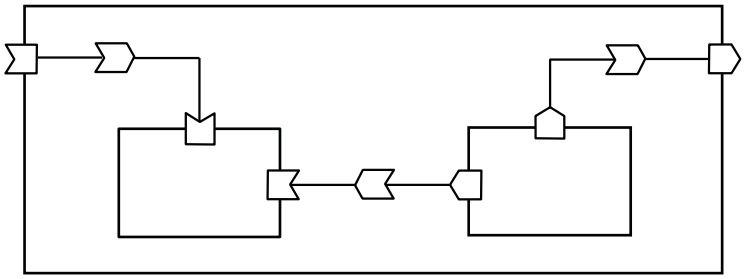


Slika 2.2: Povezivanje podsustava kanalom za poruke.

nje međuspremnika za poruke – slanje poruke ne blokira pošiljatelja, dok primatelji mogu sami odabrati strategiju primanja poruke, primjerice periodički provjeravati ima li novih poruka, odmah reagirati na događaj primitka poruke, pročitati samo najnoviju poruku i odbaciti starije i sl. Karakteristika komunikacijskih kanala da sadrže neograničen međuspremnik je ujedno i jedna od manjkavosti komponentnog modela.

Podsustav može interno biti primitivan ili složen (hijerarhijsko modeliranje). Pri-

mitivni podsustav gradi se od komponenata razine ProSave, dok je složeni podsustav građen od drugih podsustava razine ProSys i komunikacijskih kanala. Unutrašnjost i vanjština složenog podsustava povezuju se preko ulaznih i izlaznih vrata složenog podsustava – vezama između vrata složenog podsustava, kanala i vrata podsustava koji su sadržani u složenom podsustavu (slika 2.3). Uloga ulaznih i izlaznih vrata složenog podsustava kao sučelja promatranog izvana i iznutra podsustava je suprotna: vrata koja su s vanjske strane ulazna i primaju poruke, za unutarnju stranu složenog podsustava su izlazna i stvaraju poruke (tj. prosljeđuju poruke primljene izvana).



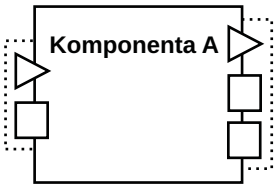
Slika 2.3: Složeni podsustav.

2.2.2 Razina ProSave

Građevni elementi podsustava na razini ProSave su *komponente*. Komponente su pasivne, reagiraju na aktivaciju izvana (reaktivne su) i općenito se ne izvode samostalno, već isključivo u kontekstu podsustava. Jednom aktivirana, komponenta izvodi svoju funkciju i zatim se vraća u neaktivno stanje, te čeka sljedeću aktivaciju. Aktivacija komponente odvija se u okviru podsustava, kao rezultat događaja (poruke) ili periodički. Komponente se mogu zasebno razvijati, analizirati i ponovno koristiti kao jedinice jednostavnih funkcija niske razine.

Slika 2.4 prikazuje grafički prikaz jednostavne komponente. Sučelje komponente određeno je ulaznim i izlaznim *vratima* za upravljanje (eng. *trigger ports*) i podatke (eng. *data ports*). Različiti tipovi vrata ostvaruju odvojen tok podataka i upravljačkih

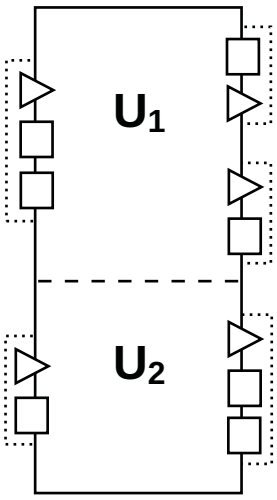
signala. Vrata predstavljena trokutom su upravljačka i određuju upravljački signal koji aktivira komponentu, dok su vrata predstavljena četverokutom podatkovna i prenose podatke. Vrata su grupirana u ulazne i izlazne *grupe* (eng. *input port groups*, *output port groups*), označene iscrtkanom linijom. Komponenta je općenito složenije građe



Slika 2.4: Grafički prikaz komponente sa ulaznim i izlaznim vratima.

od jednostavne komponente prikazane na slici 2.4. Opća komponenta sastoji se od skupa *usluga* (eng. *services*), kako je prikazano na slici 2.5. Svaka usluga predstavlja odvojenu funkcionalnost komponente koja se može aktivirati i izvesti neovisno od ostalih usluga komponente. Usluga se aktivira i komunicira s okolinom pomoću vrata, grupiranim u ulazne i izlazne grupe. Svaka grupa sastoji se od jednih upravljačkih i jednih ili više podatkovnih vrata. Usluga može imati jednu ulaznu grupu i jednu ili više izlaznih grupa vrata. Aktiviranje upravljačkih vrata ulazne grupe aktivira uslugu. Usluga zatim čita podatke s podatkovnih vrata ulazne grupe. Podaci se obrađuju i rezultat se potom vraća izlaznom grupom – aktiviranjem upravljačkih vrata prosljeđuje se upravljanje na sljedeću komponentu u lancu, dok se rezultati prosljeđuju izlaznim podatkovnim vratima grupe. Usluga koja ima više izlaznih grupa vrata može ih aktivirati neovisno i u različitim vremenskim trenutcima.

Izvođenje komponente prati ciklus *čitanje-izvođenje-pisanje-izvođenje-pisanje*. U stanju mirovanja komponenta je neaktivna i sve njezine usluge su neaktivne. Usluga miruje sve dok se ne aktivira signalom na ulaznim upravljačkim vratima, nakon čega čita vrijednosti svih ulaznih podatkovnih vrata odjednom (atomički) i prelazi u aktivno stanje. U aktivnom stanju usluga izvodi svoju funkcionalnost nakon čega postavlja rezultate na izlazne grupe vrata. Podaci i upravljački signal se na vratima izlazne grupe pojavljuju istovremeno. Ima li usluga više izlaznih grupa vrata, svaka od njih mora se aktivirati točno jednom prije nego što se usluga vrati u stanje mirovanja. Usluga koja



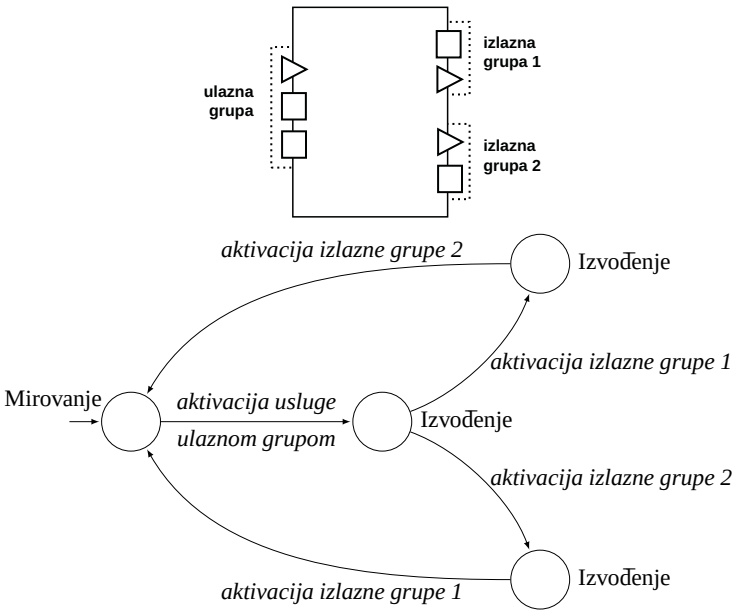
Slika 2.5: Grafički prikaz komponente sa dvije usluge.

je već u aktivnom stanju ne može biti ponovno aktivirana - aktivacija usluge moguća je isključivo iz stanja mirovanja. Postoji li u komponenti više usluga, one mogu biti aktivirane neovisno jedna o drugoj.

Slika 2.6 prikazuje stanja kroz koja usluga sa dvije izlazne grupe vrata prolazi. Ciklus započinje aktivacijom usluge putem upravljačkih vrata ulazne grupe. Usluga prelazi u stanje izvođenja funkcionalnosti i za povratak mora jednom aktivirati svaku od izlaznih grupa vrata, što je prikazano dodatnim stanjem izvođenja. Nakon aktivacije obje izlazne grupe vrata, usluga se vraća u stanje mirovanja i čeka sljedeću aktivaciju. Detaljna formalna semantika usluga i ostalih elemenata komponentnog modela ProCom opisana je u [92].

Složene komponente

Slično kao i podsustavi, i komponente mogu biti interno primitivne ili složene, tj. sastavljene od drugih komponenata. Primitivne komponente ostvaruju se s po jednom funkcijom programskog jezika C za svaku uslugu koju komponenta nudi. Složene komponente u sebi povezuju više komponenata i tako ostvaruju složene funkcionalnosti. Veze

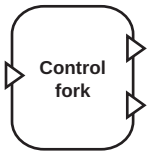


Slika 2.6: Skica dijagrama stanja (b) usluge s dvije izlazne grupe vrata (a).

između komponenata ostvaruju se elementima *veze* (eng. *connections*), koje samo prenose upravljačke signale ili podatke, i poveznicima (eng. *connectors*), koji ostvaruju logičke funkcije nad njima. Veze su usmjerene i povezuju dvoja vrata, upravljačka s upravljačkim ili podatkovna s podatkovnim, pri čemu jedna vrata mogu biti povezana samo s jednom vezom.

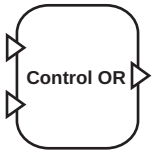
Poveznici su pasivni elementi koji mogu mijenjati upravljački ili podatkovni tok unutar komponente. Komponentni model ProCom definira niz poveznika, a spomenuti ćemo samo najvažnije [33].

Poveznik račvanja upravljačkog signala (eng. *control fork*) dijeli upravljački signal na nekoliko upravljačkih tokova. Poveznik ima jedna ulazna upravljačka vrata i dvoja ili više izlaznih upravljačkih vrata. Ulazni upravljački signal primljen na ulaznim upravljačkim vratima prenosi se na sva izlazna upravljačka vrata.

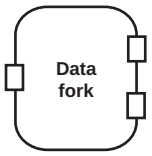


Poveznik spajanja upravljačkog signala (eng. *control join*) spaja upravljačke signale nekoliko upravljačkih tokova. Poveznik ima dvoja ili više ulaznih upravljačkih vrata i jedna izlazna upravljačka vrata. Izlazna vrata aktivirat će se samo kad su *sva* ulazna vrata aktivirana (logičko I) – suprotno od poveznika račvanja upravljačkog signala.

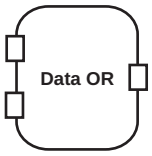
Poveznik disjunkcije upravljačkog signala (eng. *control or*) koristi se za spajanje alternativnih (disjunktnih) upravljačkih tokova. Poveznik ima dvoja ili više ulaznih upravljačkih vrata i jedna izlazna upravljačka vrata. Izlazna vrata aktivirat će se kad su *bilo koja* ulazna vrata aktivirana (logičko ILI).



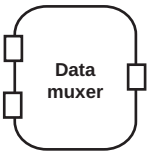
Poveznik račvanja podataka (eng. *data fork*) dijeli jednu podatkovnu vezu na njih nekoliko. Poveznik ima jedna ulazna podatkovna vrata i dvoja ili više izlaznih podatkovnih vrata. Ulazni podatak primljen na ulaznim podatkovnim vratima prenosi se na sva izlazna podatkovna vrata.



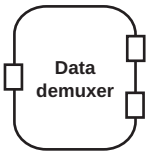
Poveznik disjunkcije podataka (eng. *data or*) koristi se za spajanje više podatkovnih tokova u jedan. Poveznik ima dvoja ili više ulaznih podatkovnih vrata i jedna izlazna podatkovna vrata. Ulazni podatak primljen na *bilo kojim* ulaznim podatkovnim vratima prenosi se na izlazna podatkovna vrata.



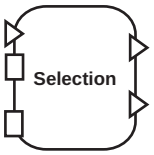
Ostali poveznici koje nisu detaljno spomenuti su:



multiplekser podataka,



demultiplekser podataka,



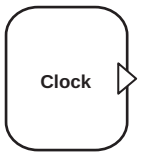
odabir.

Složene komponente sadrže podkomponente, veze i poveznike. Unutrašnjost i

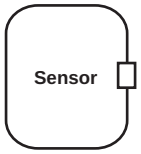
vanjština složene komponente povezuju se preko ulaznih i izlaznih vrata složene komponente. Uloga ulaznih i izlaznih vrata složene komponente kao vanjskog i unutarnjeg sučelja je upravo suprotna: vrata koja su s vanjske strane ulazna, s unutarnje strane složene komponente su izlazna. Vrijednosti upisane na podatkovna vrata izlazne grupe složene komponente *iznutra* nisu istog trena vidljive i *izvana* – podaci se proslje đuju van komponente tek kad se aktiviraju upravljačka vrata grupe. Isto vrijedi i za ulazne grupe vrata – vrijednost postavljena *izvana* na podatkovna vrata grupe postaje vidljiva *iznutra* tek kad se aktiviraju upravljačka vrata grupe (i time usluga postane aktivna).

U poglavlju 2.2.1 spomenuto je da podsustav razine ProSys može sadržavati komponente razine ProSave. Aktivni podsustav se tako može realizirati od pasivnih (reaktivnih) komponenata, za što su nužni elementi koji mogu takve pasivne komponente aktivirati. Takvi elementi grafički su predstavljeni slično poveznicima razine ProSave, iako to u načelu nisu – riječ je o elementima koji logički pripadaju razini ProSys jer su aktivni, ali se pojavljuju na razini ProSave kao veza između ovih dviju razina. Takvi elementi su *sat*, *senzori*, *aktuatori* i *vrata za poruke*. Senzori i aktuatori nisu definirani u opisu komponentnog modela ProCom [33], pa njihove karakteristike opisane ovdje donosimo na osnovi njihove strukture kakva je predložena u metamodelu komponentnog modela.

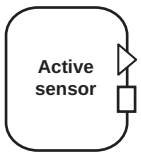
Element *sata* (eng. *clock*) izvor je signala vremenskog vođenja na svojim izlaznim upravljačkim vratima. Period signala razlikuje se među satovima u sustavu, ali svi satovi imaju stalan vremenski odmak – svim satovima vrijeme jednako teče, bez zaostajanja pojedinih satova za ostalima. Element sata može biti grafički prikazan i kao maleni sat.



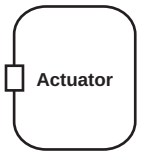
Element *senzora* (eng. *sensor*) predstavlja sklop senzora kao pasivan izvor podataka (rezultata mjerenja fizikalne veličine) za ostatak sustava. Senzor je pasivan i čitanje podatka s njegovih izlaznih podatkovnih vrata odgovara očitanju vrijednosti sklopovskog senzora. Očitanje vrijednosti senzora ne sinkronizira se sa stanjem senzora već se podrazumijeva da je senzor uvijek spreman i podatak je valjan.



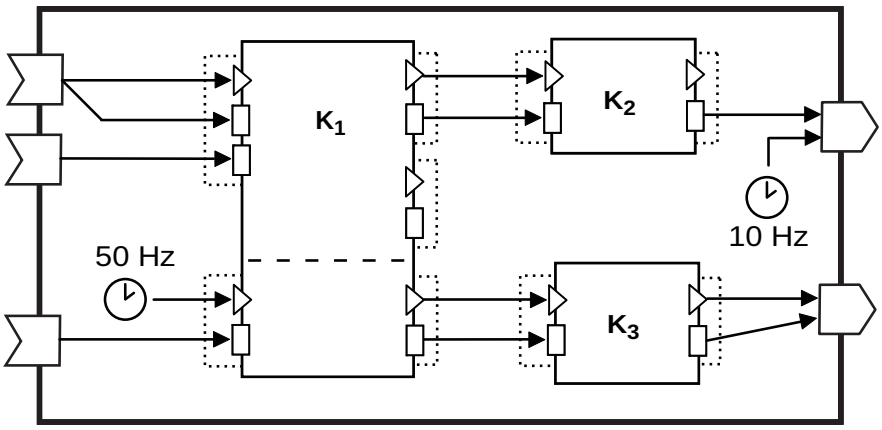
Element *aktivnog senzora* (eng. *active sensor*) predstavlja sklop senzora kao aktivan izvor podataka za ostatak sustava. Senzor svoju spremnost dojavljuje signalom na izlaznim upravljačkim vratima, te u isto vrijeme postavlja podatak (rezultat mjerenja fizikalne veličine) na izlazna podatkovna vrata.



Element *aktuatora* (eng. *actuator*) predstavlja sklop aktuatora kao elektromehaničku izlaznu jedinicu koja podatak koji joj je poslan pretvara u neku fizikalnu veličinu. Postavljanje podatkovne vrijednosti na ulazna podatkovna vrata aktuatora rezultira trenutačnom pretvorbom te vrijednosti u odgovarajuću fizikalnu veličinu.



Primjer uporabe elemenata razine ProSave za modeliranje interne strukture podsustava razine ProSys prikazan je na slici 2.7. Pretvorbu poruka razine ProSys u odvojene podatkovne i upravljačke tokove razine ProSave obavljaju vrata za poruke. Ulazna vrata za poruke primljenu poruku pretvaraju u par signala – upravljački i podatkovni signal – koji se mogu proslijediti elementima razine ProSave. Izlazna vrata za poruke primljeni par signala (upravljački i podatkovni) pretvaraju u poruku koju dalje prihvaćaju elementi razine ProSys.



Slika 2.7: Primjer uporabe elemenata razine ProSave za opis interne strukture podsustava: tri komponente, K_1, K_2 i K_3 i dva sata ostvaruju funkcionalnost podsustava [33].

2.3 Metamodel ProCom-a

Dok grafički prikaz elemenata predstavlja konkretnu sintaksu, metamodel opisuje apstraktnu sintaksu komponentnog modela ProCom. Ovo poglavlje dat će kratki pregled metamodela komponentnog modela ProCom, u dijelovima koji su potrebni za ostvarenje postupaka praćenja utroška resursa sustava. Razine komponentnog modela prikazane su kao odvojeni paketi *ProSys* i *ProSave* u metamodelu. Kasnije će se vidjeti da metamodel slijedi prethodni opis modela te tako to odvajanje nije potpuno i neki elementi su vezani preko granica paketa metamodela. Metamodel komponentnog modela opisat ćemo od najviše razine apstrakcije prema najmanjoj (od vrha prema dnu), počevši s paketom ProSys.

2.3.1 Paket ProSys

Paket *ProSys* prikazuje višu razinu komponentnog modela, te je znatno jednostavniji od niže razine, paketa *ProSave*. Slika 2.8 prikazuje metamodel podsustava (podsustav je grafički prikazan na slici 2.1) u paketu *ProSys*. Podsustav je predstavljen elementom tipa *Subsystem* i sadrži vrata za poruke apstraktnog tipa *MessagePort* kojeg nasljeđuju elementi *InputMessagePort* i *OutputMessagePort*. Ovi elementi predstavljaju sučelje

podsustava koje može biti i prazno (bez ulaznih odnosno izlaznih vrata). Postojanje takvog podsustava je dozvoljeno jer se sustav u cjelini promatra upravo kao takav podsustav bez sučelja (postojanje podsustava bez sučelja kao podelementa sustava naravno nema smisla).

Metamodel interne strukture podsustava prikazan je na slici 2.9. Apstraktna klasa *SubsystemRealization* odražava mogućnost da interna struktura podsustava može biti jedna od nekoliko ponuđenih i da se može mijenjati za vrijeme razvoja sustava. Tako se interna struktura složenog podsustava sastavljenog od drugih podsustava predstavlja elementom tipa *CompositeSubsystem*, dok se elementi sa razine ProSave mogu upotrijebiti za izvedbu podsustava u okviru elementa tipa *ProSaveSubsystem* (nalazi se u paketu ProSave).

Složeni podsustav *CompositeSubsystem* sadrži instance podsustava (*SubsystemInstance*), komunikacijske kanale za razmjenu poruka (*MessageChannel*) i veze (*Connection*). Veza *Connection* povezuje komunikacijski kanal *MessagePort* s vratima za poruke *MessagePort* (određena u kontekstu podsustava *Subsystem*) i konkretnom instancom podsustava *SubsystemInstance* kome ta vrata pripadaju. Time je ostvarena mogućnost da se isti podsustav uporabi više puta kroz više objekata tipa *SubsystemInstance*.

2.3.2 Paket ProSave

Paket *ProSave* prikazuje nižu, složeniju razinu komponentnog modela. Krovni element, koji objedinjava sve druge elemente paketa ProSave u cjelinu, element je tipa *ProSaveSubsystem*. Slika 2.10 prikazuje metamodel podsustava ProSave, kao jedne od mogućih izvedbi podsustava.

ProSaveSubsystem predstavlja vezu između razina ProSys i ProSave, te sadrži instance komponenata *SubcomponentInstance*, poveznike *Connector* i veze *Connections*. Konkretna instanca komponente predstavljena je elementom *SubcomponentInstance*, čime je (na isti način kao elementima *SubsystemInstance* u razini ProSave) omogu-

ćeno višestruko pozivanje na jednom definiranu komponentu.

Prijenos podataka s razine ProSys na razinu ProSave ostvaren je na razini ProSave vratima za poruke *InputMessagePort* i *OutputMessagePort*. Vrata nasljeđuju apstraktnu klasu poveznika *Connector*, za razliku od istovjetnog elementa na razini ProSys koji nasljeđuje klasu *MessagePort* i predstavlja *vrata*, a ne *poveznik*. Vrata preslikavaju poruke u kombinaciju upravljačkih i podatkovnih signala, te sadrže upravljačka i podatkovna vrata, što je vidljivo na slici 2.11.

Model komponente i njenog sučelja prikazan je slikom 2.12. Komponenta *Component* sadrži usluge tipa *Service* koje imaju jednu ulaznu grupu vrata *InputPortGroup* i proizvoljan broj izlaznih grupa vrata *OutputPortGroup*. Grupe se sastoje od jednih upravljačkih vrata izvedenih iz apstraktne klase *TriggerPort – InputTriggerPort*, odnosno *OutputTriggerPort*. Svaka grupa može imati i proizvoljan broj podatkovnih vrata izvedenih iz klase *DataPort – InputDataPort*, odnosno *OutputTriggerPort*. Sve vrste vrata nasljeđuju apstraktnu klasu *Port*.

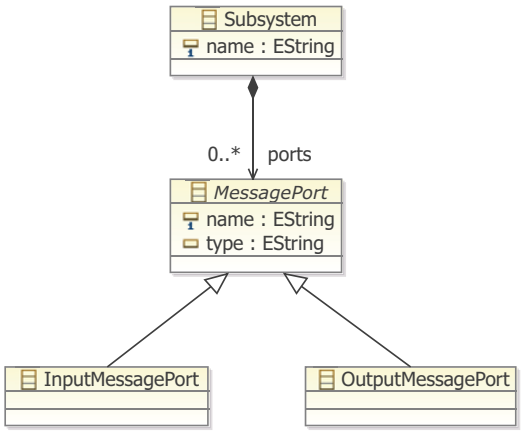
Poveznici opisani u prethodnim poglavljima dijele se u dvije grupe – poveznike koji se mogu upotrijebiti unutar elementa *ProSaveSubsystem* i poveznike koji se mogu upotrijebiti samo unutar složene komponente *CompositeComponent* (slika 2.13).

Složena komponenta *CompositeComponent* slično kao *ProSaveSubsystem* sadrži instance podkomponenata *ProSaveSubsystem*, poveznike (samo one koji nasljeđuju klasu *ConnectorInsideComponent*) i veze *Connections*. Elementi kojima se opisuje unutarnja građa komponente prikazani su na slici 2.14.

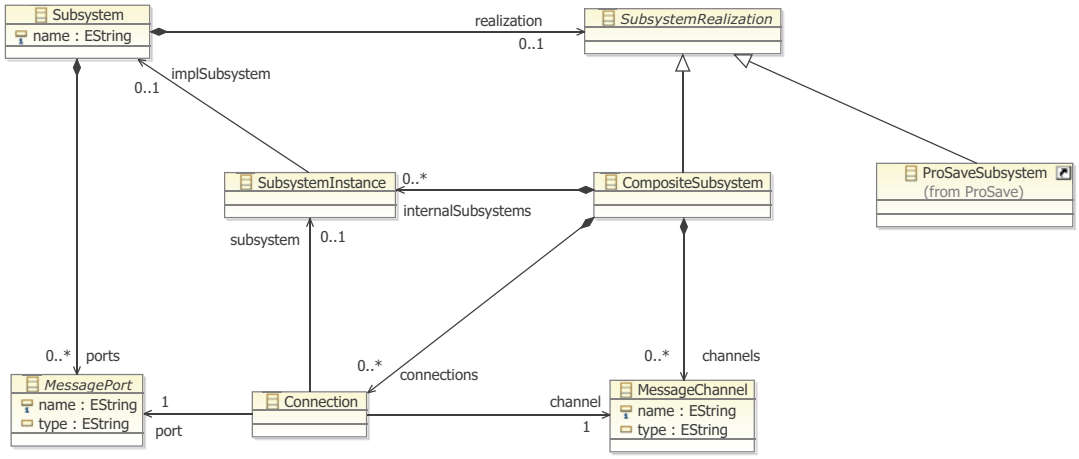
2.4 Zaključak

Komponentni modeli centralni su dio procesa razvoja temeljenog na komponentama. Komponentni model stvara okvir za opis sučelja i građe komponente, pravila kompozicije i uporabe, te analizu uvjeta i posljedica svih operacija nad komponentama. U ovom poglavlju dan je pregled komponentnog modela ProCom namijenjenog razvoju

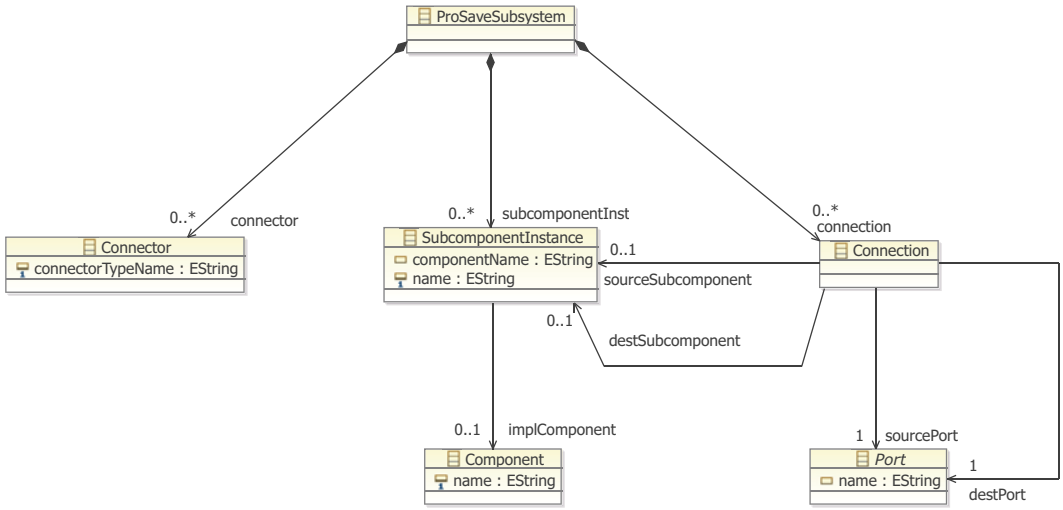
ugradbenih sustava temeljenih na komponentama. Komponentni model sastoji se od dvije razine, ProSys i ProSave, koje na različit način pristupaju problemu modeliranja sustava. Razina ProSys zasniva se na aktivnim elementima koji razmjenjuju poruke, dok je razina ProSave fokusirana na pasivne, reaktivne elemente koji se povezuju po načelu cjevovoda i filtra.



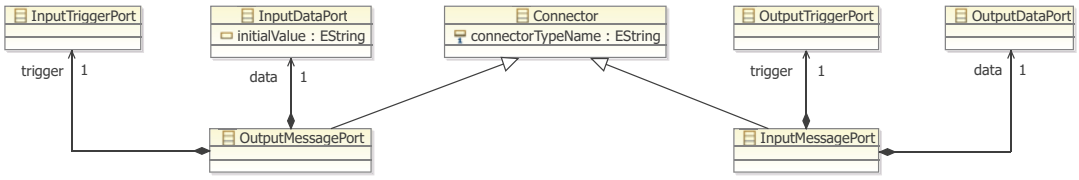
Slika 2.8: Elementi metamodela ProCom za opis podsustava i sučelja podsustava.



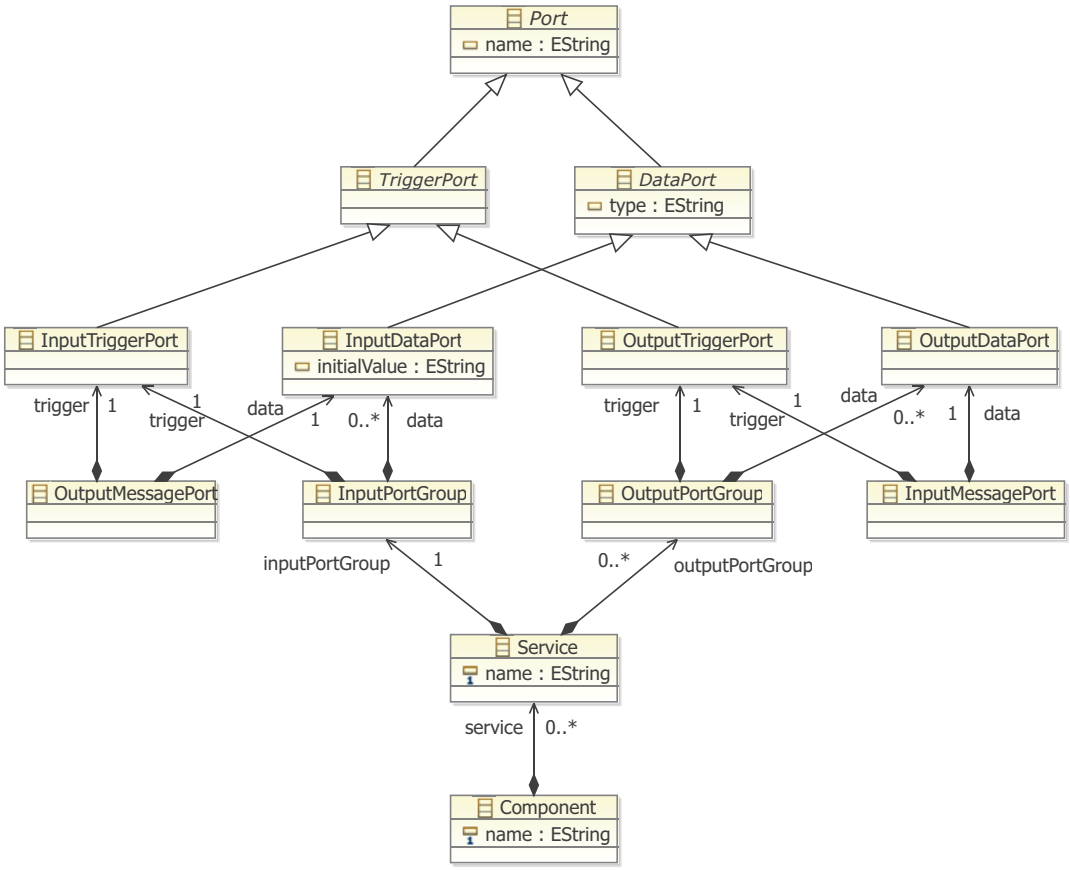
Slika 2.9: Elementi metamodela ProCom za opis unutarnje građe složenog podsustava.



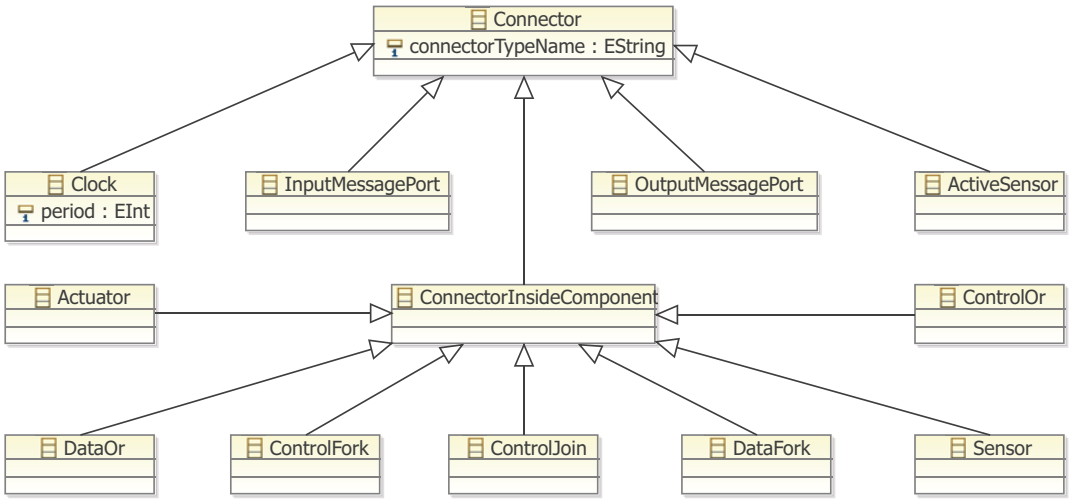
Slika 2.10: Elementi metamodela ProCom za opis građe podsustava ProSave.



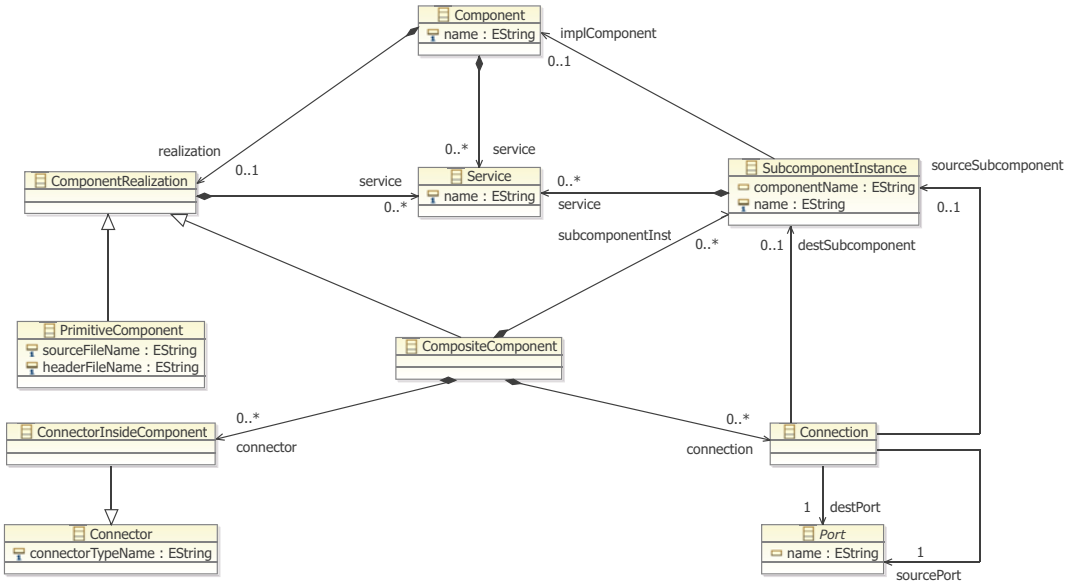
Slika 2.11: Vrata za poruke na razini ProSave kao veza s razinom ProSys.



Slika 2.12: Elementi metamodela ProCom za opis sučelja komponente razine ProSave.



Slika 2.13: Elementi poveznika razine ProSave.



Slika 2.14: Elementi metamodela ProCom za opis unutarnje građe komponente.

Poglavlje 3

Opis ponašanja

Jezici za opis ponašanja sustava nastoje ponuditi takvu razinu detalja da je iz opisa ponašanja moguće generirati programski kôd dijelova ili cijelih sustava. Najpoznatiji jezici za opis ponašanja su dijagrami jezika UML [73]. U kontekstu ovog rada takvi jezici nisu interesantni, ne samo zbog složenosti jezika UML već i zbog činjenice da jezik za opis ponašanja REMES koji čini osnovicu ovog istraživanja nije izveden iz jezika UML. Umjesto toga, koncentrirat ćemo se na jezike temeljene na automatima.

3.1 Jezici temeljeni na automatima

Jezici temeljeni na automatima strogo su formalni u opisu ponašanja i time pogodni za formalnu analizu. Velik broj alata za provjeru modela (eng. *model checking*) temelji analizu upravo na nekom obliku automata [4, 31, 36]

3.1.1 Vremenski automati

Vremenske automate kao model ponašanja sustava za rad u stvarnom vremenu predložili su Rajeev Alur i David Dill [6]. Vremenski automati proširuju konačne automate

s ograničenjima ponašanja u vremenskoj domeni. Za potrebe iskazivanja vremenskih ograničenja, konačni automati prošireni su skupom varijabli koje određuju stvarno vrijeme.

Model vremena uporabljen u konstrukciji vremenskih automata je model linearnog vremena – izvođenje operacije može se potpuno opisati kao niz stanja ili događaja nazvan *tragom* (eng. *trace*). Ponašanje sustava u ovom modelu je skup tragova, odnosno nizova stanja, što prirodno vodi do uporabe automata za opis i verifikaciju sustava. U postupku analize nastoji se konstruirati automat koji kao jezik prihvaća skup ispravnih tragova izvođenja i iz njegovih svojstava izvesti zaključke o ponašanju sustava. Događaji su pridijeljeni vremenskim trenucima *gustog vremena* (eng. *dense time*). Gusto vrijeme skup je trenutaka predstavljenih realnim brojevima (interval realnih brojeva) koji monotono rastu bez ograničenja [6].

Skup $\mathcal{X} = \{x_1, \ldots, x_{n_\mathcal{X}}\}$ je konačan skup realnih varijabli *sata* (eng. *c*locks). Dodavanjem sata $x_0, x_0 \notin \mathcal{X}$ koji predstavlja konstantu 0, dobijamo prošireni skup $\mathcal{X}^+ = \mathcal{X} \cup x_0$. Skup *ograničenja sata* preko $\mathcal{X}$ opisan je gramatikom [6, 77]

$$cc ::= true \mid x_i \sim c \mid x_i - x_j \sim c \mid cc \wedge cc$$

gdje su $x_i, x_j \in \mathcal{X}$, $c \in \mathbb{N}$, a $\sim$ označava relaciju, $\sim \in \{\leq, <, =, >, \geq\}$. Ograničenja oblika *true*, $x_i \sim c$ i $x_i - x_j \sim c$ nazivaju se atomičkim. Skup svih ograničenja preko $\mathcal{X}$ označen je s $C^{\ominus}_{\mathcal{X}}$, dok je njegov podskup ograničenja u kojima nejednakosti koje uključuju razliku satova $x_i - x_j \sim c$ nisu dozvoljena označen s $C_{\mathcal{X}}$.

Normirana ograničenja sata odgovaraju gramatici

$$cc ::= x_i - x_j \sim c \mid x_i < \infty \mid x_i - x_j < -\infty \mid cc \wedge cc$$

gdje su $x_i, x_j \in \mathcal{X}^+$, $c \in \mathbb{Z}$, a $\sim$ označava relaciju, $\sim \in \{\leq, <\}$. Skup normiranih ograničenja preko $\mathcal{X}^+$ označen je s $C^{\ominus}_{\mathcal{X}^+}$. Svako ograničenje sata može se pretvoriti u normalni oblik sljedećim postupkom:

- ograničenja oblika $x_i \sim c$, gdje je $x_i \in \mathcal{X}, \sim \in \{<, \leq, \geq, >\}$ i $c \in \mathbb{N}$ pretvaraju se u oblik $x - x_0 \sim c$,

- ograničenja oblika $x_i = c$ ili $x_i - x_j = c$ zamjenjuju se izrazom $x_i - x_0 \leq c \ \wedge \ x_0 - x_i \leq -c$, odnosno $x_i - x_j \leq c \ \wedge \ x_j - x_i \leq -c$,
- *true* se zamjenjuje izrazom $x_0 - x_0 < \infty$,
- sva ograničenja oblika $x_i - x_j > c$, gdje su $x_i, x_j \in X^+$ i $c \in \mathbb{N}$ zamjenjuju se izrazom $x_j - x_i < -c$, a ona oblika $x_i - x_j \geq c$ izrazom $x_j - x_i \leq -c$.

Ograničenja oblika $x_i - x_j < -\infty$ uvode se u svrhu prikaza ograničenja koja nikad nisu zadovoljena.

Valuacija sata u $\mathcal{X}$ je $n_\mathcal{X}$-torka $v \in \mathbb{R}_{0+}^{n_\mathcal{X}}$, pri čemu svaki član v_i n-torke v označava vrijednost sata x_i u v, što se kraće može označiti i kao $v(x_i)$ ili $v(i)$.

Za valuaciju v i $\delta \in \mathbb{R}_{0+}$, $v + \delta$ označava valuaciju v' takvu da je za svaki $x \in X, v'(x) = v(x) + \delta$. Podskup skupa satova $X \subseteq \mathcal{X}, v[X := 0]$ označava valuaciju *poništavanja sata* (eng. *clock reset*) v' takvu da je

$$v'(x) = \begin{cases} 0, & x \in X \\ v(x), & x \in \mathcal{X} \backslash X. \end{cases} \tag{3.1}$$

Relacija zadovoljenja $\models$ za ograničenje sata $cc \in C_\mathcal{X}^\ominus$ definirana je sa

- $v \models true$,
- $v \models (x_i \sim c)$ akko $v(x_i) \sim c$,
- $v \models (x_i - x_j \sim c)$ akko $v(x_i) - v(x_j) \sim c$,
- $v \models (cc \wedge cc')$ akko $v \models cc$ i $v \models cc'$.

Definicija 3.1. Vremenski automat (eng. *timed automaton*) $\mathcal{A}$ je struktura (n-torka)

$$\mathcal{A} = (A, L, l^0, E, \mathcal{X}, \mathcal{I}),$$

gdje su

- A konačni skup akcija, gdje je $A \cap \mathbb{R}_{0+} = \emptyset$,
- L konačni skup lokacija (stanja),
- $l^0 \in L$ početna lokacija,
- $\mathcal{X}$ konačan skup satova,
- $E \subseteq L \times A \times C^{\ominus}_{\mathcal{X}} \times 2^{\mathcal{X}} \times L$ relacija prijelaza,
- $\mathcal{I} : L \to C^{\ominus}_{\mathcal{X}}$ svakoj lokaciji pridjeljuje ograničenje sata – invarijant.

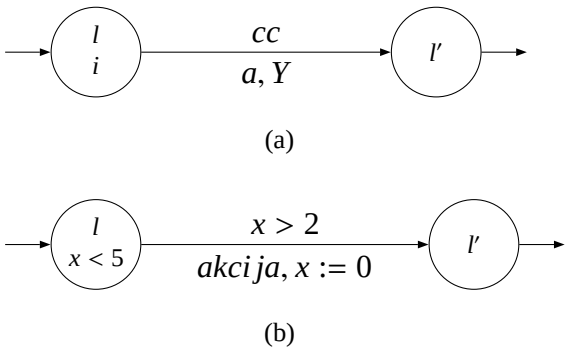
Brid $e \in E := (l, a, cc, Y, l')$ između lokacija l (izvorišne lokacije) i l' (odredišne lokacije) označen je ograničenjem sata cc, akcijom a i skupom Y satova koji će se poništiti. Brid e označava se sa $l \xrightarrow{cc,a,Y} l'$ čime se označava da je brid između lokacija l i l' omogućen zadovoljenjem ograničenja sata cc, izvodi akciju a i poništava satove iz skupa Y. Funkcija $\mathcal{I}$ svakoj lokaciji $l \in L$ pridjeljuje ograničenje sata i (invarijant) koje određuje uvjet pod kojim automat $\mathcal{A}$ smije ostati u l.

Ako su ograničenja sata i invarijanti lokacija strogo iz skupa C, za automat kažemo da je *bez dijagonala* (eng. *diagonal-free*). Za brid $e := (l, a, cc, Y, l')$ definiramo i funkcije izvor(e), odredište(e), akcija(e), uvjet(e), reset(e) za l, l', a, cc, Y. Uvjet brida ograničava aktiviranje brida (ali ne implicira da do prijelaza tim bridom mora doći). Invarijant lokacije dozvoljava automatu $\mathcal{A}$ ostanak u lokaciji l tako dugo dok je uvjet $\mathcal{I}(l)$ ispunjen.

Grafički se vremenski automati prikazuju kao automati čije su lokacije i bridovi obilježeni pripadnim uvjetima odnosno akcijama što je prikazano na slici 3.1.

3.1.2 Dijalekt vremenskih automata uporabljen u obitelji alata UPPAAL

Obitelj alata UPPAAL [61] koristi dijalekt opisanog jezika vremenskih automata proširen elementima jezika koji olakšavaju izračun dostupnih stanja ili smanjuju ukupan prostor stanja koji se analizira [18].



Slika 3.1: Primjer grafičkog prikaza lokacija i brida vremenskog automata: (a) značenje formalnih parametara i (b) primjer konkretnih vrijednosti formalnih parametara.

Jezik opisuje mrežu vremenskih automata predstavljenu kao skup predložaka (eng. *template*) pojedinih automata. Predlošku može biti pridružen skup parametara koji se zamjenjuju argumentima pridijeljenim predlošku u okviru deklaracije procesa. Parametri mogu biti bilo kojeg od podržanih tipova.

Uvedene su konstante (po definiciji nepromjenjive i s određenom cjelobrojnom vrijednosti), klasične varijable te omeđene cjelobrojne varijable (čija vrijednost može biti unutar intervala određenog donjom i gornjom granicom). Ovim se smanjuje prostor stanja jer se uvjeti, ograničenja sata i neke promjene varijable (koje se odnose na vrijednost varijable izvan dozvoljenog intervala) koji vode u nedozvoljena stanja mogu odbaciti.

Za sinkronizaciju dva automata predviđeni su binarni sinkronizacijski kanali. Uz svaki kanal *c* pridružen je par sinkronizacijskih operacija *c*! i *c*?. Brid koji sadrži uvjet označen kao *c*! (gdje je *c* ime sinkronizacijskog kanala) sinkronizira se s drugim bridom označenim s *c*?. Operacija ! može se predstaviti i kao *slanje* sinkronizacijske poruke, dok je operacija ? *primanje* poruke. U slučajevima kad je omogućeno više bridova koji čekaju na kanal, sinkronizacijski par odabire se nedeterministički. Operacija sinkronizacije je *blokirajuća* – ne postoji li primatelj koji čeka na sinkronizaciju, pošiljatelj neće moći ostvariti prijelaz i blokirati će.

Obavijesni sinkronizacijski kanali (eng. *broadcast synchronisation channels*) do-

puštaju jednom pošiljatelju *c*! da se sinkronizira s proizvoljnim brojem primatelja *c*? (pošiljatelj šalje obavijest primateljima). Za razliku od binarnih sinkronizacijskih kanala, obavijesni kanali neće blokirati pošiljatelja ako nema primatelja koji čekaju na sinkronizaciju.

Oba tipa kanala mogu biti dodatno označeni kao *hitni* (eng. *urgent*). Hitni kanali ne dopuštaju prolazak vremena (kašnjenje) ako je omogućen brid s operacijom sinkronizacije putem kanala i takvi bridovi ne smiju imati uvjete koji uključuju satove.

Osim sinkronizacijskih kanala, elementi koji mogu pojednostaviti opis i analizu ponašanja sustava su *hitne* i *sinkronizirane* lokacije. Hitne lokacije ne dopuštaju da vrijeme prolazi dok je sustav u stanju koje uključuje hitnu lokaciju. Isti efekt može se postići dodavanjem dodatnog sata *s* koji se briše na svim bridovima koji ulaze u hitnu lokaciju i na svim bridovima koji izlaze iz hitne lokacije, dok sama lokacija ima invarijant koji uključuje uvjet $s <= 0$.

Sinkronizirane lokacije (eng. *committed locations*) uvode dodatna ograničenja. Stanje S u kojem se sustav nalazi je sinkronizirano ako je bilo koja od lokacija uključena u stanju sustava označena kao sinkronizirana lokacija. Vrijeme ne prolazi u sinkroniziranom stanju, a prijelaz u sljedeće stanje (prijelaz kojim sustav napušta stanje S) mora sadržavati barem jedan brid koji vodi iz sinkronizirane lokacije. Tim uvjetom osigurava se da sustav mora napustiti sve sinkronizirane lokacije prije nego što vrijeme nastavi prolaziti.

Dijalekt jezika dopušta i deklaraciju polja te inicijalizaciju vrijednosti varijabli.

3.1.3 Izvođenje vremenskih automata

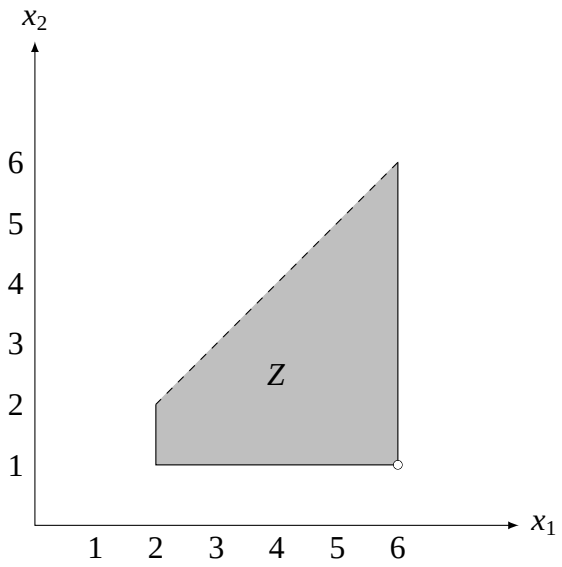
Uzmimo da je $C^{\ominus} = C_{X}^{\ominus} \cup C_{X^+}^{\ominus}$. Za ograničenje sata $cc \in C^{\ominus}$ označimo sa $[\![cc]\!]$ skup svih valuacija satova $x \in X$ koje zadovoljavaju ograničenje *cc*:

$$cc = \{v \in \mathbb{R}_{0+}^{n_X} \mid v \models cc\} \tag{3.2}$$

Vremenska zona u $\mathbb{R}^{n_X}_{0+}$ je svaki konveksni poliedar $Z \subseteq \mathbb{R}^{n_X}_{0+}$ određen ograničenjem sata:

$$Z = [\![cc]\!], cc \in C^{\ominus} \tag{3.3}$$

Radi jednostavnosti, pojam zone smatramo identičnim s ograničenjem sata koje ju određuje. Skup svih zona u $\mathcal{X}$ označavamo sa $Z(n_X)$.



Slika 3.2: Primjer vremenske zone $Z(2)$ za dva sata x_1 i x_2.

Slika 3.1.3 prikazuje primjer vremenske zone Z za dvije varijable sata x_1 i x_2. Zona je određena ograničenjem sata

$$Z = [\![x_1 \geq 2 \wedge x_1 \leq 6 \wedge x_2 \geq 1 \wedge x_2 \leq 6 \wedge x_1 - x_2 > 0 \wedge x_1 - x_2 < 5]\!]. \tag{3.4}$$

Isprekidana linija (npr. po pravcu $x_1 - x_2 = 0$) označava da je riječ o ograničenju *strogo manje,* dok puna linija (npr. po pravcu $x_1 = 2$) označava da je riječ o ograničenju *veće ili jednako*. Oznaka u kutu $(6, 1)$ dolazi od ograničenja $x_1 - x_2 < 5$ pa kutna točka nije u zoni.

Za $v, v' \in \mathbb{R}^{n_X}_{0+}$ te $Z, Z' \in Z(n_X)$ definirane su sljedeće operacije:

- $v \leq v'$ akko $\exists \delta \in \mathbb{R}_{0+}$ takav da je $v' = v + \delta$,

- $Z \backslash Z'$ je konačan skup disjunktnih zona takav da je $\{Z'\} \cup (Z \backslash Z')$ particija (familija disjunktnih podskupova) Z,
- $Z \nearrow := \{v' \in \mathbb{R}^{n_X}_{0+} \mid (\exists v \in Z) v \leq v'\}$,
- $Z[X := 0] = \{v[X := 0] \mid v \in Z\}$.

3.2 Zaključak

Vremenski automati efikasan su formalni jezik za opis ponašanja sustava, pogotovo sustava koji imaju izražene uvjete u vremenskoj domeni. Uporabom vremenskih automata možemo opisati ponašanje sustava slično dijagramima stanja, no uz dodatak vremenskih ograničenja. Vremenska ograničenja ponašanja opisuju se uporabom varijabli sata. Varijable sata opisuju prolazak vremena u prostoru realnih brojeva i mogu odrediti npr. trajanje pojedinog stanja ili uvjete prijelaza među stanjima. Stanje sustava određeno je s dvije komponente – diskretnom (lokacijom u kojoj se sustav nalazi) i kontinuiranom (vrijednošću varijabli sata). Formalna analiza sustava opisanog vremenskim automatom otkrit će mogućnost ulaska u nedozvoljeno stanje, čime se mogu otkriti skrivene vremenske ovisnosti. Za formalnu analizu sustava opisanih vremenskim automatima koriste se alati obitelji UPPAAL, čiji jezik vremenskih automata je ponešto prilagođen u odnosu na opisani. Modifikacije jezika ne smanjuju njegovu izražajnost, već ostvaruju efikasniju analizu modela sustava.

Poglavlje 4

Opis uporabe resursa

Ovo poglavlje daje kratak pregled nekoliko jezika za opis uporabe resursa. Predstavljeni su formalni jezici vremenskih automata sa cijenom ili troškom, te jezici Charon i REMES. Svi oni međusobno su povezani – vremenski automati predstavljaju analitički okvir za jezik REMES, a Charon je uvelike utjecao na gradnju jezika REMES.

Osim opisa jezika REMES, dan je i pregled algoritama za pretvorbu jezika REMES u analitički okvir vremenskih automata.

4.1 Vremenski automati s cijenom ili troškom

Vremenski automati s cijenom ili troškom (eng. *priced timed automata, weighted timed automata*) proširenje su vremenskih automata dobiveno pridjeljivanjem troška (cijene) lokacijama i bridovima [27, 26].

Definicija 4.1. Vremenski automat s cijenom (eng. *priced timed automaton*) je vremenski automat (prema definiciji 3.1) $\mathcal{A} = (A, L, l^0, E, \mathcal{X}, \mathcal{I}, C)$ proširen funkcijom cijene $C : L \cup E \to \mathbb{Z}$ koja pridjeljuje cjelobrojnu vrijednost cijene lokacijama i bridovima.

Značenje cijene lokacije i brida definirano je kroz diskretne i kontinuirane prijelaze [21]. Diskretni prijelazi (eng. *discrete transitions*) rezultat su prijelaza nekim od aktivnih bridova automata, čime odredišna lokacija postaje aktivna. Cijena prijelaza određena je cijenom brida.

Definicija 4.2. Prijelaz između dva stanja vremenskog automata s cijenom $(l, v) \xrightarrow[c]{a} (l', v')$ je diskretan prijelaz akko postoji brid $e = l \xrightarrow{cc,a,Y} l'$ takav da je ograničenje sata cc zadovoljeno u početnom stanju (l, v), v je valuacija satova koja je dobivena iz v brisanjem svih satova u skupu Y, a pritom je $c = C(e)$ cijena brida e.

Kontinuirani prijelazi (eng. *delay transitions*) ili kašnjenja rezultat su prolaska vremena, a ne promjene lokacije automata. Prijelaz je valjan samo ako je invarijant aktivne lokacije zadovoljen u svim međustanjima (vremenskim trenutcima kojima odgovaraju valuacije satova). Cijena kontinuiranog prijelaza određena je umnoškom trajanja prijelaza i cijene aktivne lokacije. Cijena lokacije u ovom slučaju ima značenje brzine promjene ukupne cijene u vremenu (eng. *cost rate*).

Definicija 4.3. Prijelaz između dva stanja vremenskog automata s cijenom $(l, v) \xrightarrow[c]{d} (l, v')$, gdje je d iznos prolaska vremena, je kontinuirani prijelaz akko je cijena prijelaza $c = d \cdot C(l)$, valuacija sata v' je rezultat prolaska vremena d, $v' = v + d$ i invarijant lokacije l zadovoljen je u izvorišnom stanju (l, v), odredišnom stanju (l, v') i svim stanjima između – za sve nenegativne $d' \leq d$ vrijedi $v + d' \models \mathcal{I}(l)$.

Za mreže vremenskih automata trenutno stanje određeno je n-torkom aktivnih lokacija. Tada je brzina promjena cijene n-torke određena sumom brzina promjena cijene pojedinih elemenata n-torke.

4.2 Jezik Charon

Charon [57] je namijenjen opisu strukture i ponašanja hibridnih sustava [5]. Charon sustav promatra kao skup agenata kojima se pridjeljuju opisi ponašanja, pri čemu se

koriste mehanizmi kao što su strukturna hijerarhija (dopuštena je kompozicija agenata za opis konkurentnog izvođenja, enkapsulacija podataka i stvaranje inačica agenata u cilju ponovne uporabe struktura), hijerarhija opisa ponašanja (hijerarhijski strojevi stanja za opis ponašanja), diskretne i kontinuirane promjene stanja te praćenje posljednje aktivacije varijablama povijesti. Diskretne promjene stanja ograničene su uvjetima (eng. *guard*), dok se za praćenje kontinuiranih promjena koriste varijable koje su označene kao *analogne* varijable te je njihova promjena ograničena diferencijalnim ili algebarskim jednadžbama, ili ograničenjima sata (invarijantima).

Zanimljiva osobina jezika Charon je iskorištavanje hijerarhije prilikom izračuna promjena stanja sustava. Prilikom određivanja sljedećeg stanja polazi se od pretpostavke da se modusi u nižoj razini hijerarhije brže izvode od onih u višoj razini hijerarhije. Za ubrzanje izračuna podrazumijeva se da su varijable vezane uz roditelja nepromjenjive dok se izvodi modus dijete, te se rezultati dobiveni proračunom diferencijalnih jednadžbi ponašanja djeteta mogu integrirati u rješenje roditelja.

4.3 Jezik REMES

Jezik REMES (eng. *REsource Model for Embedded Systems*) [85] jezik je za opis ponašanja (eng. *behavioral modeling language*) namijenjen opisu ponašanja sustava s ograničenim resursima koji moraju poštivati vremenska ograničenja rada u stvarnom vremenu, što je tipična karakteristika ugradbenih sustava. Takvi sustavi spadaju u klasu hibridnih sustava – sustava koji se modeliraju kombinacijom diskretnih modusa ponašanja (stanja) i kontinuiranih varijabli koje se mijenjaju prema zakonitostima određenim diferencijalnim jednadžbama. Diskretna stanja opisuju pojedina ponašanja modeliranog sustava uporabom skupa diferencijalnih i algebarskih jednadžbi (u praksi se umjesto diferencijalnih jednadžbi koriste i jednadžbe diferencija). Osim hibridnih sustava, jezik REMES može se primijeniti i drugdje ako postoji potreba za vremenskim ograničenjima i analizom utroška resursa [90]. Primjer dijagrama jezika REMES dan je na slici 4.1.

Jezik REMES strogo je namijenjen opisu *ponašanja* sustava, te time ne postoje elementi za opis strukture, odnosno arhitekture sustava. U tu svrhu koristi se komponentni model ProCom [33, 92, 91]. Za razliku od jezika Charon, koji je poslužio kao inspiracija pri oblikovanju jezika, REMES poznaje pojam *resursa* (prilagođenog za ugradbenim sustavima – ugrađeni su posebni tipovi resursa memorije, procesora, energije napajanja i dr.) koji su diskretne (npr. memorija) ili kontinuirane (npr. energija) prirode. Prvenstveno namijenjen za opis internog ponašanja komponente u ugradbenom sustavu prema komponentnom modelu ProCom, REMES opisuje ponašanje komponente kao *modus rada* (stanje, eng. *mode*) u dvije varijante – kao jednostavno (atomičko – ne sadrži podelemente) ili složeno (kompozitno – sadrži podelemente). Aktiviranje i upravljanje modusom izvodi se kroz *upravljačko sučelje* (eng. *control interface*) kojeg predstavljaju ulazne i izlazne točke (eng. *entry point*, eng. *exit point*) modusa. Prijenos podataka među modusima izvodi se kroz *podatkovno sučelje* (eng. *data interface*). Slijed aktivacije i prijenosa kontrole među modusima (upravljački tok, eng. *control flow*) određen je usmjerenim linijama koje moduse povezuju bridovima (eng. *edge*), pri čemu promatramo dijagram jezika REMES kao usmjereni graf. Bridovi predstavljaju diskretnu promjenu stanja (diskretan prijelaz) i označeni su izrazima koji određuju *uvjet* (eng. *guard*) pod kojim je omogućen prijelaz tim bridom. Uvjeti su logički izrazi nad skupom varijabli – cjelobrojnih, realnih te satova. Bridovima su pridijeljene i diskretne akcije (eng. *discrete actions*) koje određuju promjenu varijabli ili satova.

Varijable se deklariraju unutar modusa, što je ujedno i doseg varijable. Varijable se smatraju *lokalnim* za modus osim ako su označene kao *globalne*, kad imaju globalni doseg i vidljive su svima. Globalne varijable dijele se u varijable *za čitanje* (eng. *read*) i *za pisanje* (eng. *write*). Lokalne varijable dostupne su za čitanje i pisanje modusu unutar kojeg su deklarirane. Globalne varijable iz skupa *za čitanje* modus koji ih deklarira može *čitati*, dok ostali modusi mogu *pisati* vrijednost varijable. Globalnim varijablama iz skupa *za pisanje* modus koji ih deklarira može *pisati* vrijednost, dok ostali modusi mogu *čitati* vrijednost varijable. Varijabla može u isto vrijeme biti označena kao varijabla *za pisanje* i *za čitanje*. Tip varijable može biti cijeli broj (*integer*), prirodni broj (*natural*), realni broj (*float*), logička vrijednost (*boolean*), sat (*clock*) te

polje osnovnih tipova podataka.

Osim varijabli, element jezika su i konstante za koje vrijede iste napomene kao i za varijable, osim što pravila o pisanju i čitanju nisu primjenjiva.

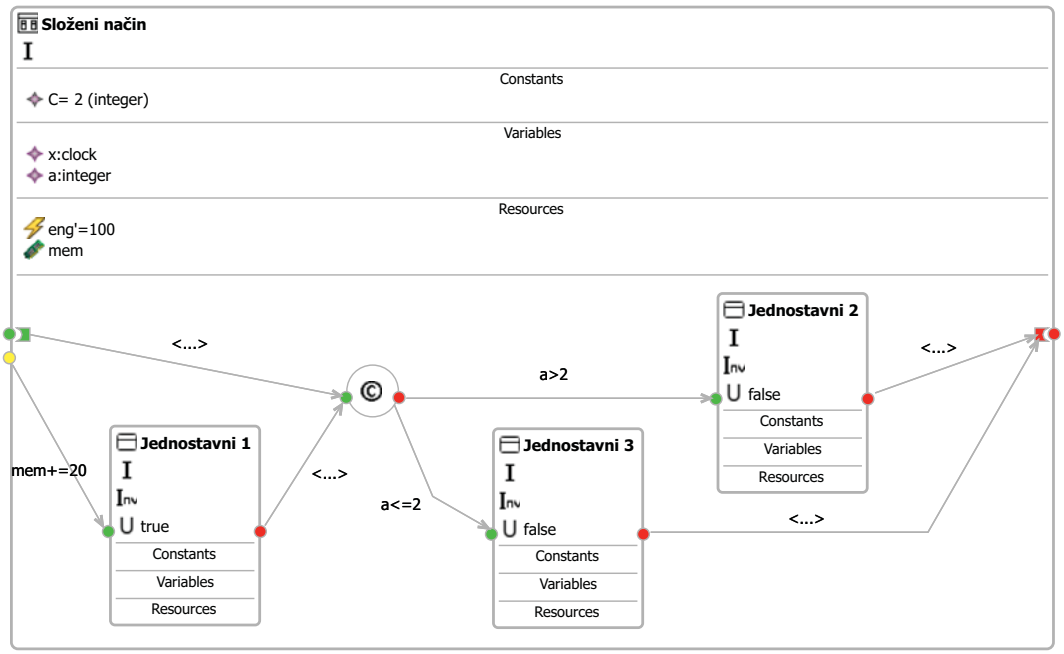
Resursi su poseban tip varijable i mogu biti tipa procesora (*cpu*), memorije (*memory*), napajanja (*power*), ulazno-izlaznih vrata (*port*) ili ocjene propusnosti komunikacije (*bandwidth*). Vrijednost resursa može se samo povećavati i smanjivati. Nije dozvoljeno čitanje vrijednosti resursa, te time nije dozvoljeno ni postavljanje uvjeta za prijelaz prema vrijednosti resursa. Ovo ograničenje prolazi iz prirode resursa – resursi mogu biti diskretni (memorija) ili kontinuirani (energija napajanja), a određivanje njihove točne vrijednosti u stvarnom sustavu može biti nemoguće ili vrlo skupo (npr. propusnost komunikacije ili procesora).

Kontinuirana promjena stanja (kašnjenje, prolazak vremena) predstavljena je *ostankom* u pojedinom modusu rada, što je moguće ograničiti pridjeljivanjem invarijanta (ograničenja sata) modusu. Dodatno, uvedeni su i *hitni* modusi (eng. *urgent modes*) koji ne dopuštaju kontinuiran prijelaz te možemo reći da njihov invarijant nikad ne vrijedi – invarijant je $\perp$. Promjena resursa može se opisati pridjeljivanjem diskretnih akcija bridovima ili pridjeljivanjem (prve) derivacije resursa modusu, čime se opisuje promjena resursa kad se izvodi kontinuirani prijelaz. Kontinuirana promjena resursa ograničena je time što promjena resursa mora biti cijeli broj.

Grafička sintaksa jezika REMES ilustrirana je dijagramom na slici 4.7. Složeni modus *HCController* može deklarirati varijable, konstante i resurse (na slici su skrivene zbog preglednosti), te mora sadržavati barem jedan jednostavni modus kao podelement (eng. *submode*). Upravljačko sučelje predstavljeno je ulaznim, izlaznim i inicijalizacijskim točkama. Ulazne točke predstavljene su zelenim kružićima, izlazne točke crvenim, a inicijalizacijske točke (prisutne samo kod složenih modusa) žutim. U unutrašnjosti složenog modusa upravljačke točke predstavljene su zastavicama iste boje. Inicijalizacijska točka nešto je jednostavnija i nema pripadnu zastavicu.

Bridovi povezuju izlaznu točku elementa prethodnika s ulaznom točkom elementa sljedbenika, čime određuju promjene diskretnog stanja modusa. Osim jednostavnih

modusa, unutar složenog modusa može se pojaviti i uvjetni poveznik (eng. *conditional connector*). Uvjetni poveznik ima više izlaznih bridova čiji uvjeti trebaju biti disjunktni te je tako u svakom trenutku barem jedan brid omogućen.



Slika 4.1: Grafički prikaz dijagrama jezika REMES.

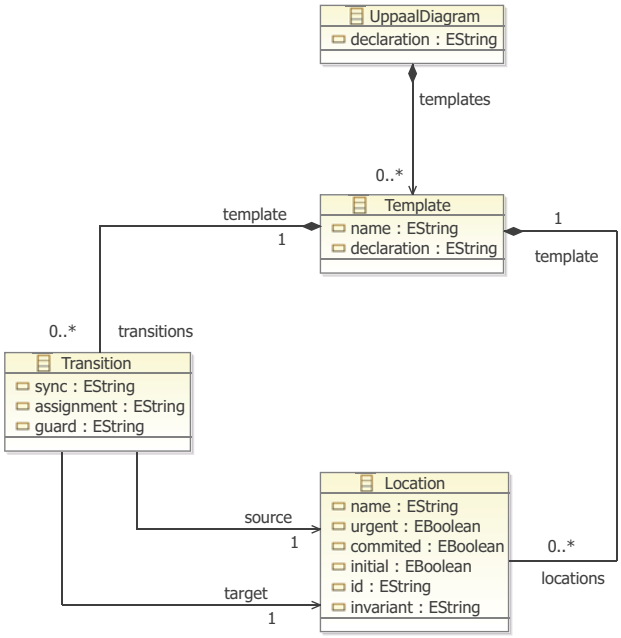
Prilikom izvođenja modela ponašanja, kontrola se prenosi slijednim aktiviranjem modusa kako je to određeno prijelazima. Jednostavni modus se aktivira prijenosom kontrole na njegovu ulaznu točku, aktivan je sve dok je pridijeljeni invarijant zadovoljen čime su određena vremenska ograničenja izvođenja. Modus se deaktivira kroz izlaznu točku i jedan od izlaznih bridova. Složeni modus ostaje aktivan sve dok su elementi koje sadrži aktivni. Prilikom prve aktivacije, kontrola se u složeni modus rada prenosi kroz inicijalizacijsku točku. Svaki sljedeći put, modus se aktivira kroz ulaznu točku. Izvođenje složenog modusa počinje aktivacijom nekog od elemenata koje sadrži i traje do izlaska kroz izlaznu točku, nakon čega se kontrola predaje drugom modusu. Prijenos kontrole do završetka izvođenja složenog modusa osiguran je uvjetom da svaka izlazna točka pod-modusa mora biti povezana s izlaznom točkom složenog modusa, ili s ulaznom točkom drugog pod-modusa, a naposlijetku i s izlaz-

nom točkom složenog modusa. Ciklusi u opisu ponašanja nisu eksplicitno zabranjeni, ali se njihova uporaba kosi sa idejom jezika, ciklička ponašanja izvode se izlaskom iz modusa i ponovnim ulaskom uz obnavljanje pohranjenog stanja.

4.3.1 Formalna semantika jezika REMES

Formalna semantika elemenata jezika REMES prikazana je uporabom vremenskih automata s cijenom. Formalna analiza promjene resursa modela opisanih jezikom REMES temeljena je na predstavljanju cijene vremenskih automata s cijenom kao težinske sume vrijednosti pojedinih resursa sustava [85]. Iako postoji teoretski okvir za analizu vremenskih automata s više troškova/cijena (eng. *multi-priced timed automata*) [59], zbog ograničenja alata za analizu, koristi se aproksimacija jedne cijene/troška u obliku težinske sume pojedinih resursa (vremenski automat s linearnom cijenom – eng. *Linear Priced Timed Automata*, LPTA).

Prije formalne analize modela ponašanja opisanog jezikom REMES potrebno je model pretvoriti u analitički model vremenskog automata. Za opis pretvorbe modela REMES u vremenski automat s cijenom predstavimo prvo metamodel zapisa mreže vremenskih automata. Metamodel *UL* prikazan na slici 4.2 pojednostavljen je prikaz strukture XML-dokumenta kojeg kao oblik zapisa koriste alati iz obitelji UPPAAL [18]. Svaki *UppaalDiagram* predstavlja mrežu vremenskih automata. Pojedini automat u mreži prikazan je jednim predloškom automata *Template*, kojeg se instancira u *proces* navođenjem u deklaraciji sustava (svojstvo *declaration* unutar *UppaalDiagram*). Predložak se sastoji od proizvoljnog broja lokacija *Location* i prijelaza *Transition* među njima. Lokacija ima pridijeljena svojstva imena (*name*), invarijanta (*invariant*), oznaku radi li se o početnoj lokaciji (*initial*). Oznake hitnosti (*urgent*) i sinkroniziranosti (*commited*) specifičnost su dijalekta jezika vremenskih automata izvedenog u alatima obitelji UPPAAL. Prijelaz je uvjetovan zadovoljenjem ograničenja sata *guard*, dok popis akcija *assignment* uključuje i akcije i brisanja satova (prema definiciji 3.1). Atribut *sync* također je specifičnost dijalekta obitelji UPPAAL. Metamodel *UL* primjeniv je za zapis vremenskih automata sa i bez cijene. U drugom slučaju, tro-



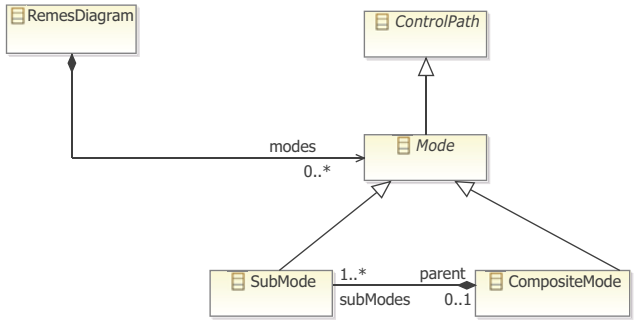
Slika 4.2: Metamodel *UL* mreže vremenskih automata.

šak kontinuiranog prijelaza zapisuje se u prošireni invarijant lokacije (atribut *invariant* klase *Location*), a trošak diskretnog prijelaza u popis akcija (atribut *assignment* klase *Transition*).

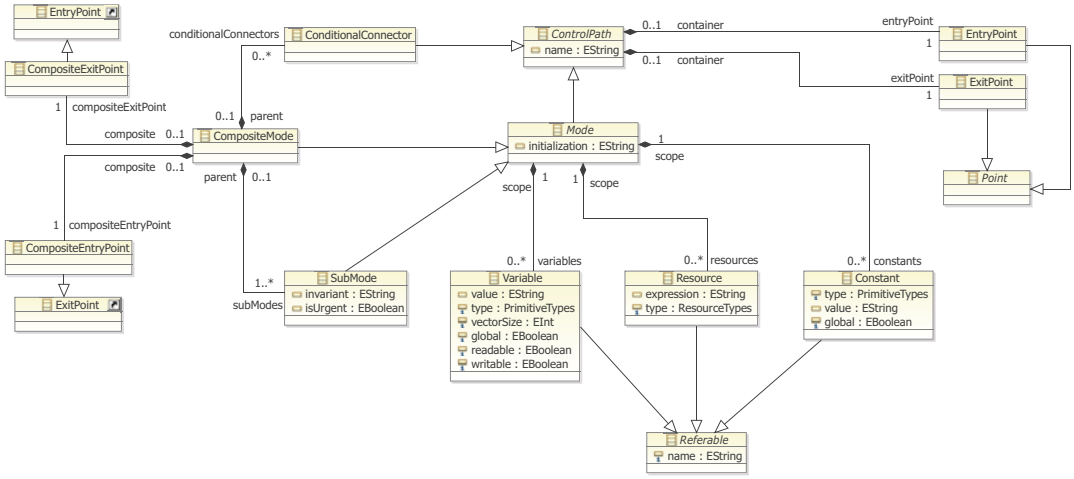
4.3.2 Struktura dijagrama jezika REMES

Slika 4.3 prikazuje pojednostavljeni metamodel elementa dijagrama jezika REMES. Dijagram prikazan s *RemesDiagram* opisuje ponašanje i sastoji se od jednog ili više elemenata tipa *Mode*. *Mode* je apstraktna klasa izvedena iz *ControlPath*, a koju dalje nasljeđuju *SubMode* i *CompositeMode*.

U okviru elementa *Mode* deklariraju se lokalne varijable, resursi i konstante prikazane klasama *Variable*, *Resource* odnosno *Constant*. *SubMode* je jednostavan modus rada koji ne sadrži druge elemente. *CompositeMode* predstavlja složeni modus rada koji se sastoji od više jednostavnih podelemenata tipa *SubMode* i uvjetnih poveznika



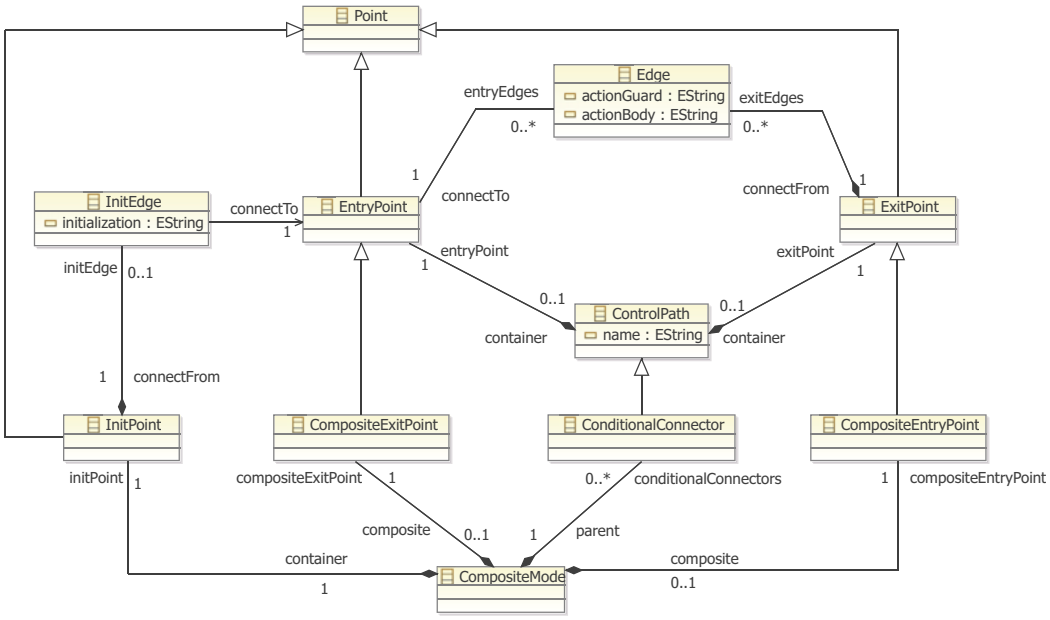
Slika 4.3: Elementi metamodela REMES-a za opis dijagrama *RemesDiagram*.



Slika 4.4: Prikaz modusa rada u metamodelu REMES-a.

tipa *ConditionalConnector* (slika 4.4). Aktivacija elemenata (složenih i jednostavnih modusa rada i uvjetnih poveznika) izvedena je kroz apstraktnu klasu *ControlPath* koja definira ulazne i izlazne točke za svaki aktivni element (*EntryPoint* i *ExitPoint*). Poseban slučaj je složeni modus rada *CompositeMode* koji dodatno uvodi dvije točke, *CompositeEntryPoint* i *CompositeExitPoint*. Ove točke predstavljaju vezu unutrašnjosti složenog modusa rada s vanjštinom pri čemu parovi *EntryPoint-CompositeEntryPoint* i *CompositeExitPoint-ExitPoint* konceptualno predstavljaju iste točke ulaza odnosno izlaza.

Prijelazi unutar dijagrama ostvareni su elementima tipa *Edge* odnosno *InitEdge*.



Slika 4.5: Prikaz bridova u metamodelu REMES-a.

Slika 4.5 prikazuje dio metamodela jezika REMES za modeliranje prijelaza u dijagramu. Veze tipa *Edge* povezuju elemente tipa *ControlPath*, čime se modelira put prijenosa kontrole. *ControlPath* nasljeđuju *Mode* (a time i *CompositeMode* i *SubMode*), te *ConditionalConnector*. *ControlPath* sadrži ulazne i izlazne točke *EntryPoint* i *ExitPoint*. Prijelazi su mogući samo između izlaznih i ulaznih točaka čime je ujedno osigurana i ispravnost strukture dijagrama. Složeni modus *CompositeMode* ostvaruje grupiranje elemenata koje sadrži. Na vanjštini elementa *CompositeMode* su ulazne i izlazne točke naslijeđene od *ControlPath*, dok se u unutrašnjosti te točke reflektiraju kao *CompositeEntryPoint* i *CompositeExitPoint*. Dodatna inicijalizacijska točka *InitPoint* posebnost je elementa *CompositeMode* i nije naslijeđena od razreda *ControlPath*.

Prijelazi tipa *Edge* prošireni su uvjetima i akcijama, dok je prijelaz *InitEdge* proširen akcijama koje se izvode prilikom inicijalizacije složenog modusa.

4.3.3 Pretvorba dijagrama jezika REMES u analitički model vremenskog automata

RemesDiagram sadrži elemente tipa *Mode* prve razine. Hijerarhija je ograničena na dvije razine – prvu, unutar elementa *RemesDiagram* i drugu, unutar elemenata *CompositeMode*. Prilikom pretvorbe u vremenski automat svaki element predstavlja vremenski automat u mreži tipa *Template* metamodela *UL*. Postupak pretvorbe opisan je pojednostavljeno u algoritmima 4.1, 4.2 i 4.3.

Algoritam 4.1: Pretvorba dijagrama jezika REMES u dijagram jezika UPPAAL

Input: rd : *RemesDiagram*
Output: ud : *UppalDiagram*
foreach $m \in rd.modes$ **do**
    $t \longleftarrow m$; // Pretvori *Mode* u *Template*
    $ud.templates \longleftarrow ud.templates \cup t$; // Dodaj *Template* u listu
end

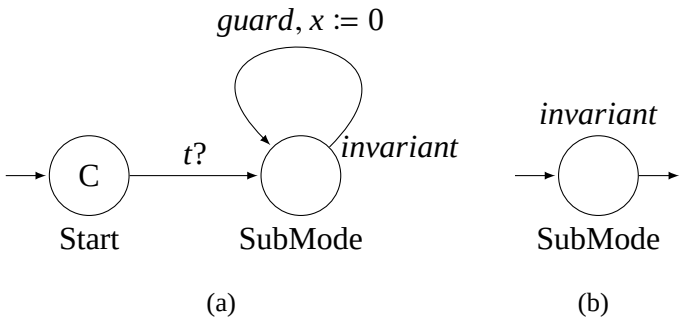
Postupak pretvorbe razlikuje se u ovisnosti je li riječ o elementima prve ili nižih razina hijerarhije dijagrama jezika REMES. Razliku pokažimo na primjeru jednostavnog modusa rada *SubMode* (slika 4.6). Kad je *SubMode* uporabljen za modeliranje jednostavnog modusa rada prve razine, rezultat pretvorbe je potpuni vremenski automat (slika 4.6(a)) upotpunjen dodatnom lokacijom *Start* s prijelazom koji se aktivira okidanjem kanala *t*. Okidanjem sinkronizacijskog kanala *t* predstavljen je trenutak pokretanja cijelog sustava.

Slika 4.6(b) prikazuje rezultat pretvorbe elementa *SubMode* kad je uporabljen unutar složenog modusa rada *CompositeMode*. Modus rada se zamjenjuje istoimenom lokacijom, a deklaracije varijabli, resursa, konstanti te ulazni i izlazni prijelazi prenose se u vremenski automat.

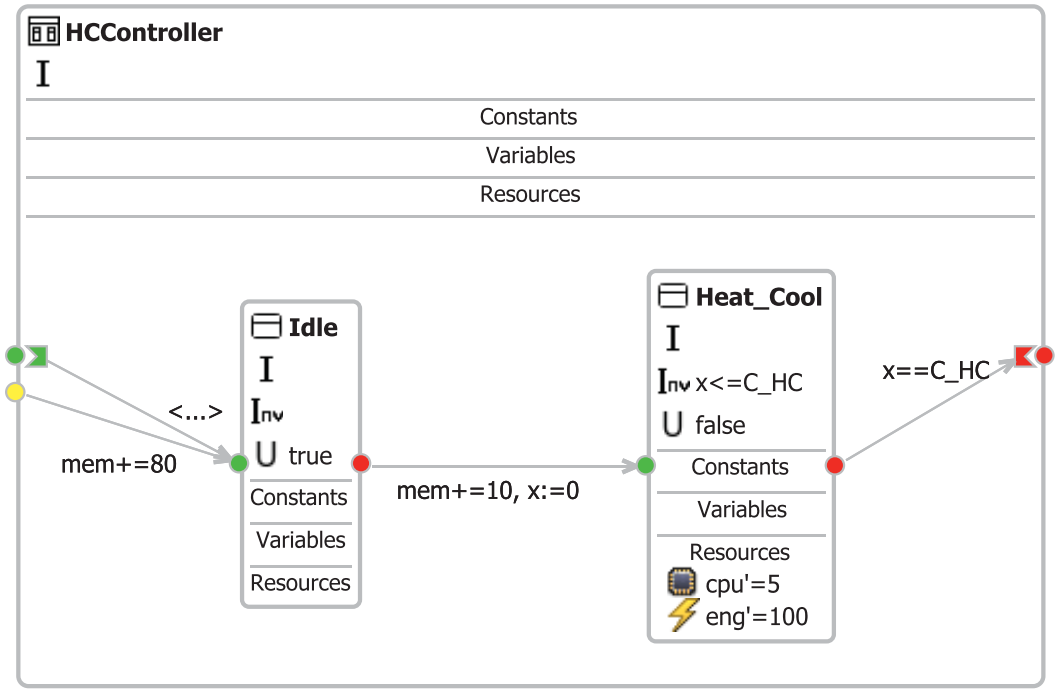
Pretvorba složenog modusa rada u pripadni vremenski automat nešto je složenija. Slika 4.7 prikazuje dijagram jezika REMES koji sadrži jedan složeni modus rada *HC-Controller* sa dva podelementa *Idle* i *HeatCool*.

Algoritam 4.2: Pretvorba elementa *SubMode* u predložak *Template*

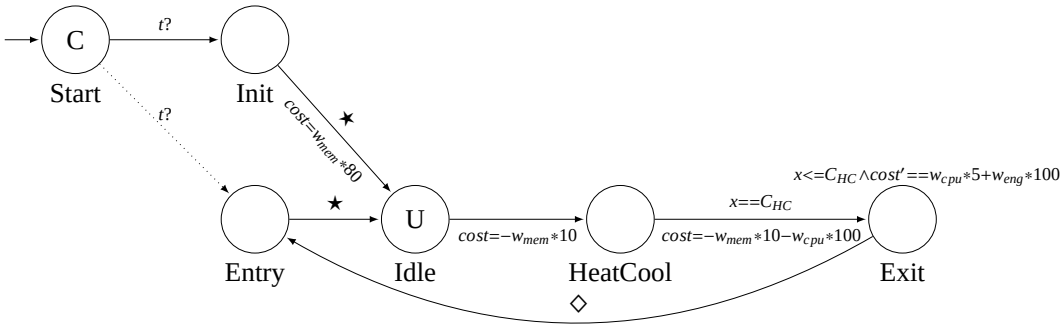
Input: rd : *RemesDiagram*, m : *SubMode*
InOut: t : *Template*
Data: et : *Transition*
if $m \in rd.modes$ **then** // Prva razina
  $t.name \longleftarrow m.name$;
  $t.declaration \longleftarrow m.variables \cup m.resources \cup m.constants$;
  $t.locations \longleftarrow start\ location \cup m$;
  $et \longleftarrow$ novi objekt *Transition* ;
  $et.source = start\ location, et.target = m$; // Prijelaz inicijalizacije
  $t.transitions \longleftarrow et \cup exit\ to\ entry\ transition$;
else // Unutar *CompositeMode*
  $t.locations \longleftarrow t.locations \cup m$; // Dodaj lokacije u predložak
end
foreach $e \in m.exitPoint.exitEdges$ **do**
  $et \longleftarrow$ novi objekt *Transition* ;
  $et.source = m, et.target = e.connectTo$; // Veze
  $et.guard = e.guard, et.assignment = e.action$; // Uvjet i akcije
  $t.transitions \longleftarrow t.transitions \cup et$; // Dodaj u listu prijelaza
end



Slika 4.6: Primjer pretvorbe elementa *SubMode* u slučaju: (a) kad je modus jednostavni modus prve razine i (b) modus je unutar složenog modusa rada *CompositeMode*.



Slika 4.7: Primjer dijagrama jezika REMES s složenim modusom *HCController* tipa *CompositeMode* koji sadrži jednostavne moduse *Idle* i *HeatCool* tipa *SubMode*.



Slika 4.8: Rezultat pretvorbe primjera na slici 4.7.

Rezultat pretvorbe prikazan je na slici 4.8. Slično kao i za pretvorbu jednostavnog modusa, dodane su dodatne lokacije *Start*, *Init*, *Entry* i *Exit*. Lokacija *Start* predstavlja čekanje na pokretanje cijelog sustava, što je određeno okidanjem sinkronizacijskog kanala *t*. Lokacije *Init*, *Entry* i *Exit* nastale su pretvorbom inicijalizacijskih, ulaznih i izlaznih točaka u zasebne lokacije.

Prijelaz iz lokacije *Init* u lokaciju *Idle* potiče od brida tipa *InitEdge* i osigurava pravilnu inicijalizaciju. Pretvorbom opisa ponašanja koja nemaju postavljen inicijalizacijski brid taj prijelaz neće postojati, već će automat iz lokacije *Start* prelaziti direktno u lokaciju *Entry*, kako je prikazano iscrtkanom linijom. Prijelaz između lokacija *Entry* i *Idle* određen je u dijagramu jezika REMES ulaznom vezom između ulazne točke *EntryPoint* i početnog modusa, u ovom slučaju pod-modusa *Idle*.

Oznaka $\star$ na prijelazu predstavlja dodatnu sinkronizaciju koja se na prijelaz dodaje u ovisnosti o povezanosti komponente ProCom čije ponašanje je opisano dijagramom. Aktivacija komponente okidanjem izvana (eng. *trigger*) kod pretvorbe u PTA izvedena je dodavanjem sinkronizacijskog kanala na ulazni i inicijalizacijski prijelaz. Podsjetimo se da se ponašanje prvi put aktivira kroz inicijalizacijsku točku, a svaki sljedeći put kroz ulaznu, te je time nužno imati istu sinkronizaciju na prijelazima koji izviru iz lokacija *Init* i *Entry*). Slično je i s povratnim prijelazom između lokacija *Exit* i *Entry* – oznaka $\diamond$ predstavlja okidanje po izlazu iz komponente (izlazni *trigger*). Povratni prijelaz osigurava cikličko ponašanje vremenskog automata, jer se nakon izlaska iz složenog modusa *HCController* automat se vraća u stanje čekanja na aktivaciju i lokaciju *Entry*.

Poseban slučaj povratnog prijelaza javlja se kad komponenta nema izlazno okidanje. Zbog nepostojanja uvjeta i sinkronizacije na povratnom bridu, moguće je da vremenski automat ostane u lokaciji *Exit* i da vrijeme prođe prije povratka u lokaciju *Entry*, što je u suprotnosti s pravilima jezika REMES koja kažu da će ponašanje odmah po izlasku kroz izlaznu točku biti spremno za sljedeću aktivaciju putem ulazne točke. Da bi se to osiguralo, lokacija *Exit* označava se kao sinkronizirana (eng. *committed*). Time se izbjegava mogućnost da automat u lokaciji *Exit* provede neko vrijeme, jer prijelazi iz sinkroniziranih lokacija imaju viši prioritet nad svim ostalim prijelazima i

moraju se izvesti odmah.

Pretvorba strukture dijagrama u vremenski automat nije potpuna bez uvažavanja *hitnih* modusa. Hitni modusi prevode se u hitne ili u sinkronizirane lokacije vremenskog automata. Time nije dopušten prolaz vremena u trenutnoj lokaciji. Hitni modus prevest će se u hitnu lokaciju ako je povezan s običnim (ne hitnim) modusom. Hitni modus prevest će se u sinkroniziranu lokaciju ako je povezan s hitnim modusom.

Razlika se uvodi zbog razlike u ponašanju hitnih i sinkroniziranih lokacija u dijalektu vremenskih automata izvedenih u obitelji alata UPPAAL. I hitni i sinkronizirani modus ne dopuštaju prolazak vremena, ali se njihovi izlazni prijelazi obavljaju različitim prioritetom. Prijelaz iz hitnog modusa može se odgoditi (bez povećavanja satova, odnosno bez prolaska vremena) i izvesti nakon prijelaza neke druge hitne lokacije (u drugom vremenskom automatu). Kad je hitni modus povezan s običnim modusom, takvo ponašanje je ispravno jer će u običnom modusu početi teći vrijeme pa se prijelaz iz hitnog modusa ne mora izvesti odmah.

Kad je hitnom modusu sljedbenik drugi hitni modus, pretvorba oba takva modusa u hitnu lokaciju mogla bi izazvati zastoj vremenskog automata TA_1 dok se ne obave hitni prijelazi u nekom drugom vremenskom automatu TA_2. Oba automata mogu biti povezani sinkronizacijskim kanalima i TA_2 može doći u sinkroniziranu lokaciju koja čeka na sinkronizaciju zajedničkim kanalom sa TA_1. Prijelazi iz sinkroniziranih kanala imaju viši prioritet od prijelaza iz hitnih kanala te u tom slučaju nastupa potpuni zastoj. Nasuprot tome, ako se takvi hitni modusi pretvore u sinkronizirane lokacije, njihovi izlazni prijelazi su višeg prioriteta i automat će sigurno doći u sigurno stanje u kojem može početi teći vrijeme.

Uvjetni poveznici kao elementi koji pojednostavljuju dijagram grupiranjem bridova, dakle kao elementi sintakse, a ne semantike, uklanjaju se prilikom pretvorbe u vremenski automat. Svaki brid iz skupa ulaznih bridova poveznika kombinira se sa svakim bridom iz skupa izlaznih poveznika na način da se za rezultantni brid uzima izvor ulaznog brida i odredište izlaznog brida, uvjet rezultantnog brida je operacija logičkog I na uvjetima ulaznog i izlaznog brida, a akcije rezultantnog brida su unija

Algoritam 4.3: Pretvorba elementa *CompositeMode* u predložak *Template*

Input: rd : *RemesDiagram*, m : *CompositeMode*

Output: t : *Template*

Data: et : *Transition*

$t.name \longleftarrow m.name$, $t.transitions \longleftarrow \{\}$;

$t.declaration \longleftarrow m.variables \cup m.resources \cup m.constants$;

$t.locations \longleftarrow start\ location \cup m.subModes \cup m.compositeEntryPoint \cup$
$m.compositeExitPoint \cup m.compositeInitPoint$;

foreach $sm \in m.subModes$ **do** // Prijelazi osim uvjetnih poveznika

  foreach $e \in m.exitPoint.exitEdges$ **do**

    if $\neg(e.connectTo$: ConditionalConnector$)$ **then**

      $et \longleftarrow$ novi objekt *Transition* ;

      $et.source = e.connectFrom, et.target = e.connectTo$; // Veze

      $et.guard = e.guard, et.assignment = e.action$; // Uvjet i akcije

      $t.transitions \longleftarrow t.transitions \cup et$; // Dodaj u listu prijelaza

foreach $cc \in m.conditionalConnectors$ **do** // Prijelazi iz uvjetnih poveznika

  foreach $in \in cc.entryPoint.entryEdges$ **do**

    foreach $out \in cc.exitPoint.exitEdges$ **do**

      $et \longleftarrow$ novi objekt *Transition* ;

      $et.source = in.connectFrom, et.target = out.connectTo$; // Veze

      // Prenesi uvjet i akcije ($\{in\} \times \{out\}$)

      $et.guard = in.guard \wedge out.guard, et.assignment = in.action \wedge out.action$;

      $t.transitions \longleftarrow t.transitions \cup et$; // Dodaj u listu prijelaza

foreach $e \in m.initPoint.exitEdges \cup m.entryPoint.exitEdges$ **do**

  if $\neg(e.connectTo$: ConditionalConnector$)$ **then**

    $et \longleftarrow$ novi objekt *Transition*

    , $et.source = e.connectFrom, et.target = e.connectTo$;

    $et.guard = e.guard, et.assignment = e.action$; // Uvjet i akcije

    $t.transitions \longleftarrow t.transitions \cup et$; // Dodaj u listu prijelaza

$t.transitions \longleftarrow t.transitions \cup start\ to\ entry\ transition \cup exit\ to\ entry\ transition$;

akcija ulaznog i izlaznog brida. Uvjetni poveznik je tim postupkom zamijenjen svim kombinacijama njegovih ulaznih i izlaznih bridova što osigurava istovjetnu funkcionalnost u vremenskom automatu.

Dijagrami jezika REMES ne dopuštaju mogućnost ulančanog povezivanja dva ili više uvjetnih poveznika. Uzmimo za primjer uzastopno povezane uvjetne poveznike CC_1 i CC_2. Pretpostavimo li da postoji ulazni brid CC_1 koji sadrži akcije mijenjanja vrijednosti varijabli koje se ispituju u uvjetima izlaznih bridova CC_2, opisani jednostavni algoritam uklanjanja uvjetnog poveznika dat će neispravne rezultate. Uvjeti izlaznih bridova od CC_2 bit će logičkom operacijom I pridodani uvjetima ulaznih bridova od CC_1, ali će se odnositi na varijable koje nisu promijenjene kao posljedica akcija ulaznog brida CC_1. Da bi se izbjegao takav slučaj, povezivanje više uvjetnih poveznika nije dopušteno. Naravno, moguće je da se takva kombinacija akcija koje utječu na varijable koje se ispituju u uvjetu pojavi i na bridovima **istog** uvjetnog poveznika. Takav rijedak i neuobičajen slučaj korisnik može sam jednostavno primijetiti i otkloniti, dok je kod više vezanih uvjetnih poveznika otkrivanje takve pogreške puno teže.

Poglavlje 5

Opis izvedbenog okružja

Ključ uspješne primjene postupaka procjene performansi tijekom razvoja sustava temeljenog na komponentama je postojanje jezika kojima se mogu izraziti atributi koji utječu na performanse sustava za vrijeme dizajna sustava.

5.1 Resursi u jeziku REMES

Izvedbeno okružje ugradbenog sustava ograničeno je raspoloživim resursima. Za potrebe analize promatrat ćemo resurse kao globalne, dijeljene veličine konačne vrijednosti. Za opći resurs r trenutna vrijednost resursa r određuje raspoloživu količinu jedinica tipa r u danom trenutku, utrošak resursa ukupnu potrošnju resursa od pokretanja sustava do danog trenutka, a sa $\dot{r} = \frac{\partial r}{\partial t}$ označena je promjena (derivacija r u vremenu) resursa r. Resursi mogu biti diskretne ili kontinuirane naravi – memorija je primjer diskretnog, dok je energija pohranjena u bateriji primjer kontinuiranog resursa. Za diskretne resurse promjena resursa definira se kao razlika trenutne vrijednosti i vrijednosti resursa u prethodnom vremenskom trenutku.

Jezik REMES dodatno uvodi podjelu resursa na adresirljive (eng. *referable*) i neadresirljive (eng. *non-referable*), te tako razlikuje tri klase resursa [85]:

- diskretne adresirljive resurse (npr. memorija)
- diskretne neadresirljive resurse (npr. procesorska snaga, propusnost mreže)
- kontinuirane (neadresirljive) resurse (npr. procesorska snaga, energija, propusnost mreže)

Problem ove klasifikacije resursa je što se resursi kao što je propusnost mreže mogu modelirati na više načina. Primjerice, vrijednost resursa može predstavljati kvalitetu signala koji određuje propusnost, ali i broj poruka/paketa poslanih u svakom trenutku – vrijednost može biti kontinuirane ili diskretne prirode. Zbog ove nejednoznačnosti, resursi ugrađeni u jezik REMES određeni su tipom (procesor, memorija, energija itd.), dok je značenje vrijednosti resursa prepušteno korisniku.

5.2 Profil platforme

Resursi kao globalno svojstvo sustava deklariraju se u okviru profila platforme. Profil opisuje karakteristike platforme na kojoj je zasnovan ugradbeni sustav, slijedom načela da je razvojni ciklus sustava odvojen od razvojnog ciklusa platforme [32]. Profil platforme određuje skupne karakteristike platforme i omogućuje platformski neovisan razvoj i testiranje sustava, barem u većem dijelu razvojnog ciklusa.

Profil ili specifikacija platforme određuje resurse koje platforma nudi – procesorsku snagu, dostupnu memoriju, energetske rezerve i dr. Deklarirani resursi koriste se (referenciraju) u opisu ponašanja komponenata. Resursi su modelirani kao varijable čija vrijednost se u okviru modela ponašanja ne može čitati, već samo povećati i smanjiti. Ograničenje pristupa vrijednosti varijabli izvire prvenstveno u činjenici da komponenta ne može sama doznati trenutno stanje resursa platforme (za resurse kao što su energija napajanja i propusnost komunikacije teško je točno odrediti trenutne vrijednosti). Pruža li platforma mogućnost ocjene stanja resursa, takve usluge mogu se modelirati kao zasebne komponente, čime primjenjivost ovog pristupa nije ograničena.

Resursi kao fizikalne veličine imaju određene granice. Profil opisuje ograničenja resursa kao funkcije minimuma, maksimuma i srednje vrijednosti konkretnih vrijednosti resursa ili promjena resursa. Ograničenja se ukratko mogu opisati gramatikom:

$$\begin{aligned} rc \quad ::= \quad & (\mathit{max} \mid \mathit{min} \mid \mathit{avg}) \\ & \text{'('} \quad (\mathit{resurs} \mid \mathit{resurs}') \quad \text{')'} \\ & (< \mid \leq \mid = \mid \geq \mid >) \quad \mathit{vrijednost} \end{aligned}$$

Kao primjer uzmimo resurse procesorske snage i dostupne memorije. Profil može postaviti ograničenja kao: $\max(cpu) < 200, \max(mem) < 16384$, i time odrediti veličinu memorije od 16 Ki lokacija i procesorsku snagu od 200 jedinica (npr. 200% iskoristivosti u slučaju sustava s dvije procesorske jezgre). Za resurs energije napajanja ograničenja mogu biti: $\max(eng') < 50, \max(eng) < 15000$, čime je ograničena vršna uporaba energije na 50 jedinica (maksimalna promjena resursa energije mora biti manja od 50), a ukupna maksimalna rezerva energije na 15000 jedinica. Uporaba operatora max, min i avg dopušta ograničavanje vršnih i srednjih vrijednosti i promjena resursa.

Osim resursa i ograničenja, profil može definirati i vrijednosti konstanti deklariranih u modelima ponašanja te time ostvariti jednostavno parametriziranje modela. Za konstante koje su deklarirane u modelu ponašanja u jeziku REMES, ali im vrijednosti nisu određene, pretpostavlja se da će biti određene kad se opisu sustava pridijeli profil platforme. Kao primjer navedimo konstante koje određuju uvećanje vrijednosti resursa u vrijeme inicijalizacije ponašanja komponente, dodatni utrošak resursa za potrebe platforme (eng. *overhead*). Navođenje konstanti definiranih u okviru profila platforme omogućuje platformski-neovisan opis ponašanja komponente.

Profil može pridijeliti i zadano ponašanje komponentama koje već nemaju određeno ponašanje ili osnovno ponašanje vezano uz platformu koje sve komponente nasljeđuju. Dizajner platforme može tako odrediti globalne karakteristike svih komponenata koje proizlaze iz karakteristika platforme, primjerice dodatno zauzeće memorije komponente zbog internih upravljačkih podatkovnih struktura platforme za svaku komponentu, vremenske karakteristike aktivacije ili deaktivacije komponenata i slično.

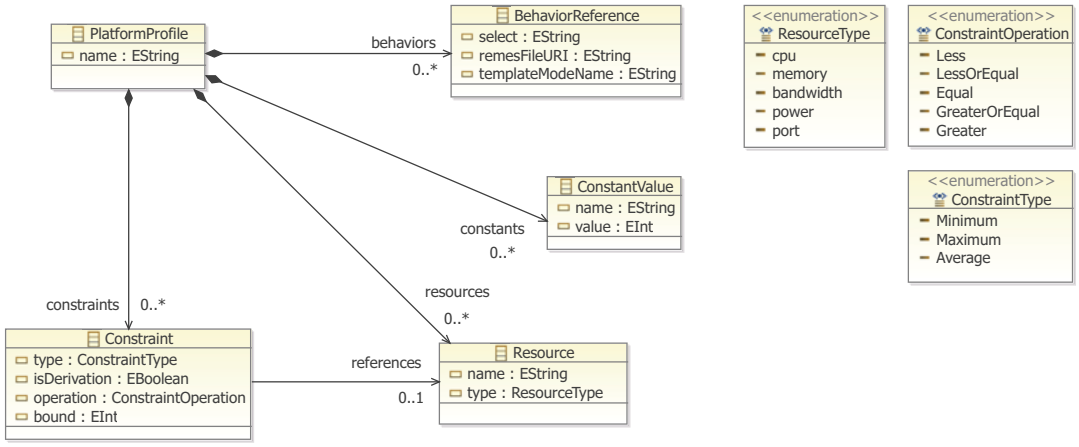
Odabir komponente na koju se takvo zadano ponašanje primjenjuje određuje se izrazom jezika OCL (*Object Constraint Language*).

Za vrijeme razvoja sustava, primijenjeni profil platforme može se u idealnom slučaju zamijeniti drugim. Primjena drugog profila omogućuje provjeru karakteristika dizajniranog sustava, poštivanje ograničenja drugog profila, druge konfiguracije platforme ili nove verzije platforme.

5.3 Model uporabe i ograničenja resursa

Slika 5.1 prikazuje pojednostavljeni model profila platforme s izdvojenim elementima za deklaraciju i ograničenja resursa te pridjeljivanje zadanih ponašanja i vrijednosti konstanti. Profil platforme predstavljen je elementom *PlatformProfile* kojem je pridijeljeno ime profila. Profil sadrži elemente deklaracije resursa *Resource*, te opisa ograničenja *Constraint*. Zadana ponašanja predstavljena su elementom *BehaviorReference*, a postavljene vrijednosti konstanti elementom *ConstantValue*.

Enumeracije *ResourceTypes*, *ConstraintType*, *ConstraintOperation* određuju tip resursa (procesorska snaga, memorija, energija napajanja, mreža, vrata), tip ograničenja resursa (funkcije max, min, avg) te relaciju ograničenja (strogo veće, veće ili jednako, jednako, manje ili jednako, strogo manje).



Slika 5.1: Model profila platforme.

Poglavlje 6

Cjeloviti model sustava

Dizajn ugradbenih sustava izveden je kroz nekoliko različitih modela i pogleda na sustav [89]. Prema postupcima razvoja opisanim u ovom radu, to su strukturni model, model ponašanja i dodatna sastavnica – profil platforme. Strukturni model često se naziva i specifikacijom arhitekture, (eng. *architecture specification*), a usmjeren je na opis statičke građe sustava, kompoziciju podsustava i komponenata, te detalje njihovih sučelja i veza. U okviru ovog rada strukturni model odgovara komponentnom (meta)modelu ProCom, dok je za opis ponašanja iskorišten (meta)model ponašanja REMES.

Komponentni model ProCom predviđa definiranje nekih elemenata kao što su primitivne komponente ili podsustavi kroz programski jezik C. Takav mehanizam, iako ga možemo smatrati *opisom* ponašanja (implementacija ponašanja – opis ponašanja), ipak ne razmatramo. Uporaba programskog jezika za opis ponašanja unosi dodatne elemente (npr. velika sloboda jezika, vanjske biblioteke) koji se teško mogu svrstati pod nazivnik modela ponašanja, a ujedno i sâm komponentni model smatra ove mehanizme potporom naslijeđenim (eng. *legacy*) komponentama uvedenim prvenstveno zbog kompatibilnosti unatrag. Ovo razmatranje se zato fokusira na kombinaciju komponentnog modela ProCom i jezika za opis ponašanja REMES.

Slika 6.1: Struktura cjelovitog modela sustava.

6.1 Građa cjelovitog modela

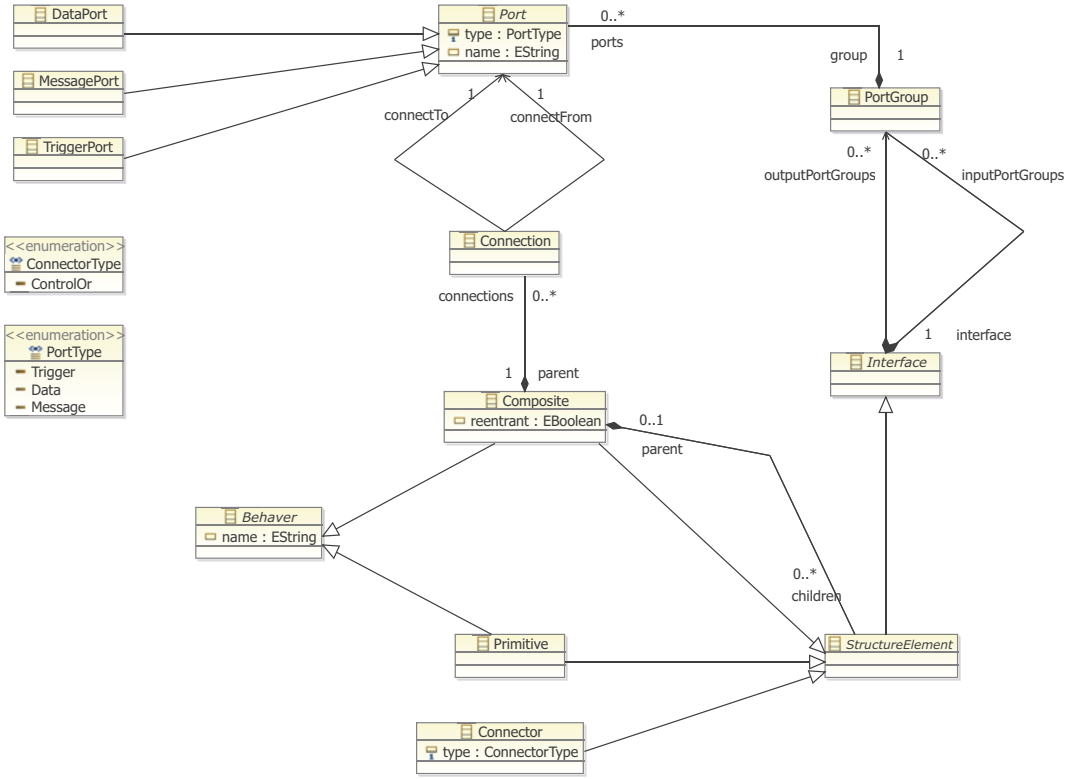
Povezivanje ova dva pogleda na sustav, strukturnog modela i modela ponašanja, ostvaruje se neformalno, konvencijom prema kojoj jednoj komponenti opisanom u modelu ProCom odgovara jedan opis ponašanja u jeziku REMES. Ova konvencija donekle smanjuje mogućnosti koje pružaju ova dva modela. Opis ponašanja načelno može biti pridijeljen bilo kojem od elemenata modela ProCom koji ostvaruju funkcionalnost – sustavu, komponenti ili usluzi.

Neformalna veza između strukture i ponašanja rješenje je koje je teško za održavanje prilikom promjena modela strukture ili ponašanja, podložno pogreškama i kao takvo vjerojatno privremeno. Cjeloviti model sustava predlaže vlastito rješenje za vezu strukture i ponašanja.

Slika 6.1 prikazuje građu paketa cjelovitog modela sustava. Model je podijeljen u nekoliko cjelina:

- paket *structure* predočuje strukturu sustava (komponente, vrata, veze) i analogan je komponentnom modelu ProCom,
- paket *behaviour* preslikava ponašanje elementa sustava i analogan je jeziku REMES,
- paket *profile* – preslika je modela izvršnog okružja – profila sustava,
- paket *mapping* – povezuje elemente modela ponašanja s elementima strukture sustava,
- paket *simulator* – određuje parametre simulatora utroška resursa,

- paket *expressions* – sadrži elemente sintakse izraza jezika REMES (apstraktna sintaksa),
- paket *core* – sadrži osnovne elemente tipova podataka i dr.



Slika 6.2: Dio cjelovitog modela za opis strukture sustava.

Paket *structure* (slika 6.2) izveden je iz komponentnog modela ProCom uz neka pojednostavljenja. Osnovna razlika u odnosu na ProCom je što za sintezu ukupnog ponašanja sustava i praćenje utroška resursa nisu nužni svi detalji koje ProCom sadrži. Tako je dvoslojna struktura razina ProSys i ProSave sa hijerarhijskim podsustavima, odnosno komponentama, pojednostavljena na dva osnovna elementa – jednostavni element *Primitive* i složeni *Composite*, koji pak može sadržavati jednostavne elemente, veze *Connection* i poveznike *Connector*. Ovo pojednostavnjenje moguće je iz razloga što dio elemenata modela ProCom postoji kako bi se izrazili strukturni odnosi (npr. usluga i komponenta), što je iz stajališta cjelovitog ponašanja opisanog jezikom

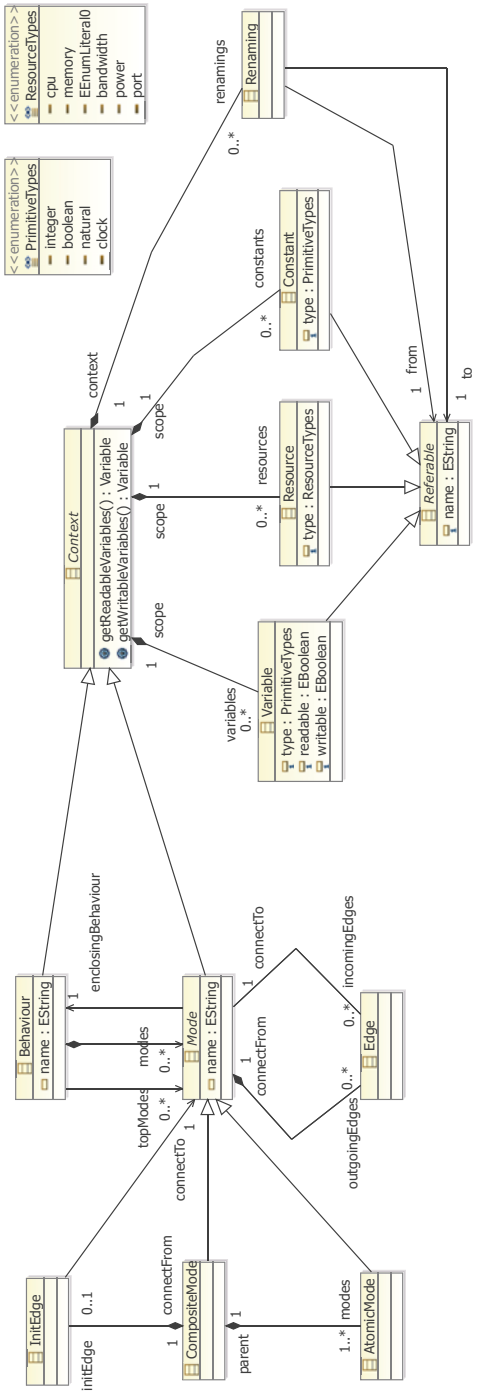
REMES nebitno, jer se modeli ponašanja jednako primjenjuju i na uslugu i na servis, u ovisnosti kojem kontekstu su pridijeljeni. Klasu *Behaver* nasljeđuju oni elementi kojima se može pridijeliti model ponašanja.

Razliku između aktivnih elemenata razine ProSys i pasivnih reaktivnih elemenata razine ProSave jezik REMES izražava pravilom da se u izvođenju ponašanja aktivnog elementa odmah nakon kraja izvođenja izlaskom putem izlazne točke ponovno aktivira ponašanje ulaskom kroz ulaznu točku. Ova osobina označena je u elementima *Primitive* i *Composite* atributom *active*. Oba elementa mogu imati pridruženo ponašanje jer nasljeđuju klasu *Behaver*.

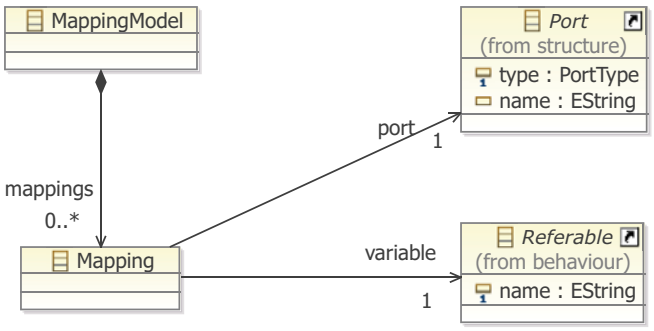
Paket *behaviour* prikazan na slici 6.3 izveden je iz metamodela jezika REMES kojemu uvelike sliči. U odnosu na REMES, umjesto elementa *RemesDiagram*, osnovni element koji sadrži sve ostale je *modus Behaviour*. Imenovani podaci (varijable, konstante, resursi) vezani su uz kontekst *Context* u kojem su definirani, što može biti globalno, na razini ponašanja *Behaver*, ili lokalno na razini modusa rada *Mode*. Dodan je i element preslikavanja varijabli *Renaming* koji eksplicitno određuje prijenos vrijednosti varijable jednog ponašanja jezika REMES u drugo. Podaci o prijenosu vrijednosti dolaze iz strukturnog modela – preslikavanje varijabli obavlja funkciju *veze* između vrata komponenata u strukturnom modelu.

Opisi strukture i ponašanja sustava dodirnu točku imaju u sučelju komponenata ili podsustava. Aktivacijom primitivne komponente ili podsustava aktivira se pridijeljeno ponašanje i u istom trenutku se vrijednosti podataka pristiglih na ulazno sučelje kopiraju u varijable ponašanja. Po završetku izvođenja pridijeljenog ponašanja, rezultati se s ostatkom sustava prenose izlaznim sučeljem na analogan način – kopiranjem vrijednosti varijabli ponašanja. Paket *mapping* (slika 6.4) sadrži elemente *Mapping* kojima se opisuje ova veza između varijabli (konstanti, resursa) definiranih u modelu ponašanja i vrata sučelja definiranih u strukturnom modelu.

Model profila u paketu *profile* ponešto se razlikuje od onog prikazanog u poglavlju 5.3. Profil platforme mora biti primjenjiv na sve modele sustava te zato ne sadrži veze s drugim modelima. U cjelovitog modelu sustava profil platforme povezan je s opisom

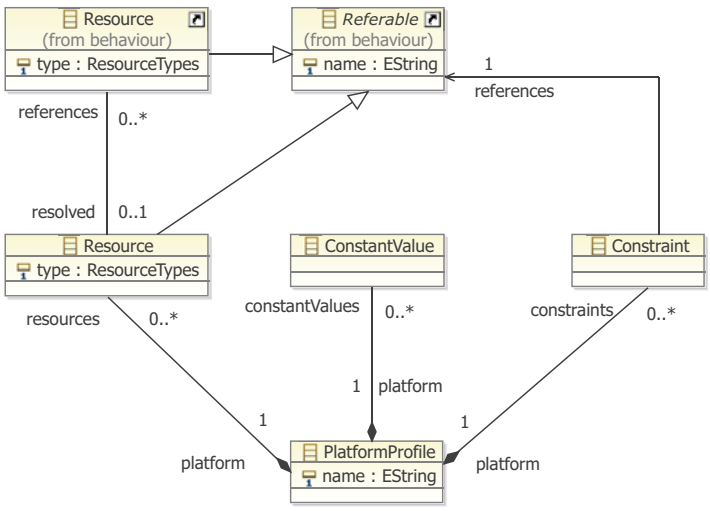


Slika 6.3: Dio cjelovitog modela za opis ponašanja.



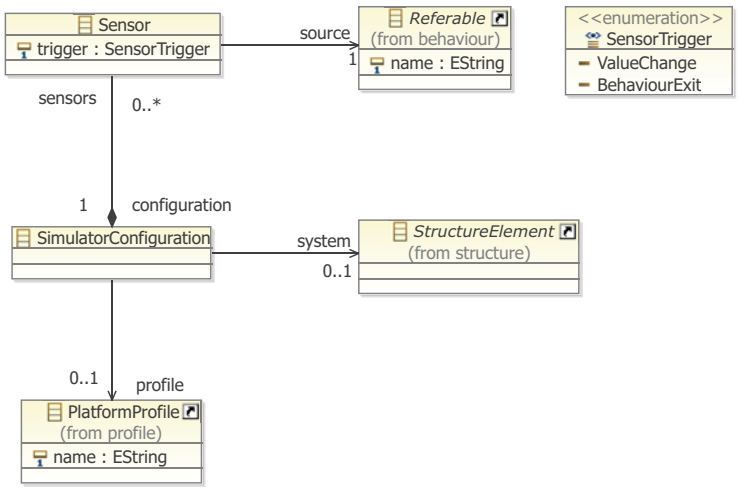
Slika 6.4: Dio cjelovitog modela za opis veze varijabli i strukturnih elemenata.

ponašanja i sadrži veze između resursa definiranih u profilu i uporabe tih resursa (varijable resursa) u ponašanju. Na slici 6.5 ta veza vidljiva je kroz veze resursa *Resource* prema istoimenoj klasi iz paketa *behaviour*. Isto vrijedi i za ograničenje *Constraint* koje ima veze prema klasi *Referable* kojom su predstavljene varijable, konstante i resursi.



Slika 6.5: Dio cjelovitog modela za opis profila platforme.

Posljednji dio cjelovitog modela je paket za konfiguraciju simulatora prikazan u paketu *simulator* (slika 6.6). Konfiguracija *SimulatorConfiguration* sadrži vezu s elementom koji opisuje sustav (*StructureElement*) i profilom platforme *PlatformProfile*. Osim

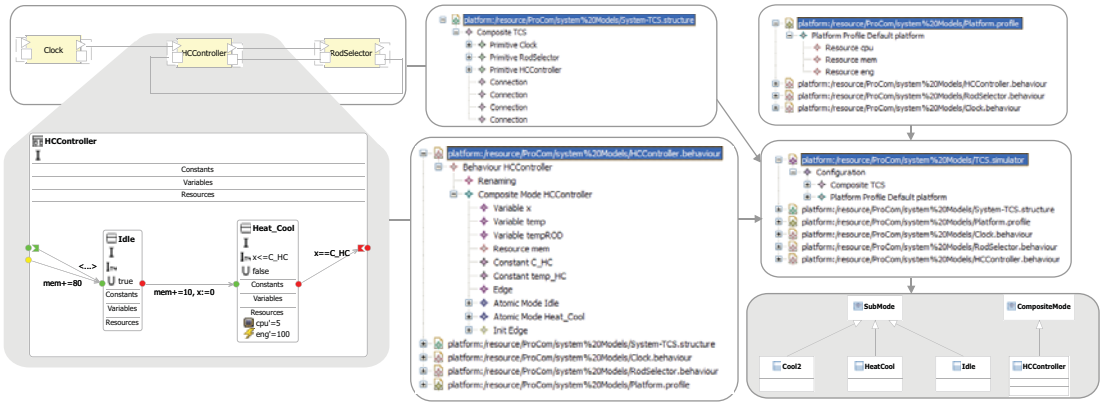


Slika 6.6: Dio cjelovitog modela za opis parametara simulatora.

ovih osnovnih elemenata, konfiguracija pruža mogućnost definiranja niza senzora *Sensor* (nisu vezani s poveznicima tipa senzora komponentnog modela ProCom). Senzori prate vrijednosti podatkovnih elemenata ponašanja (varijabli ili resursa) i pohranjuju ih za kasniju analizu. Senzori se mogu okidati na svaku promjenu vrijednosti varijable ili po završetku izvođenja ponašanja u kojem je varijabla definirana.

6.2 Generiranje cjelovitog modela

Kao priprema za proces simulacije, modeli strukture i ponašanja sustava povezuju se u cjeloviti prijelazni model sustava. Prijelazni model sustava sadrži podatke iz modela kojima je provjerena sintaksa i razrješene su reference varijabli ili resursa. Prijelazni model se generira iz modela ProCom i REMES u postupku koji obavlja promjene strukture modela, prevodi matematičke izraze u odgovarajuća sintaksna stabla i provjerava sukladnost tipova podataka upotrijebljenih u tim izrazima. Modeli strukture i ponašanja se prevode u isti model, kako bi se olakšalo njihovo povezivanje. U postupku stvaranja modela za simulaciju dodaje im se i profil platforme te se razrješavaju veze – povezuju se varijable iz modela ponašanja u jeziku REMES sa ulaznim i izlaznim



Slika 6.7: Postupak generiranja prijelaznog modela za simulaciju.

vratima komponenata modela ProCom.

Cjeloviti prijelazni model je ulazni model za simulaciju, pa zato mora biti potpun – reference na nepostojeće varijable ili neispravni izrazi nisu dozvoljeni. Proces generiranja prijelaznog modela sakriven je od korisnika. Svaka promjena strukturnih modela komponenata ili opisa ponašanja izaziva automatsko osvježavanje prijelaznog modela. Slika 6.7 prikazuje taj proces. Strukturni model sustava opisan komponentnim modelom ProCom (lijevo gore) i opisi ponašanja komponenata (lijevo dolje) prevode se u prijelazni oblik (sredina). Ta dva modela uz profil platforme (desno gore) čine osnovu za cjeloviti model sustava (desno sredina), koji je u kombinaciji s konfiguracijom simulatora podloga za postupak generiranja programskog kôda simulatora sustava.

Opisani model izvedbenog okružja može opisati resurse koje okružje nudi, kao i ograničenja njihove uporabe. Iako jednostavan, model je dostatan za apstraktan opis okružja i sukladan jeziku za opis ponašanja REMES. Cilj modela izvedbenog okružja je ostvariti jednostavne opise okruženja koji će biti od koristi u ranoj fazi razvoja sustava.

Poglavlje 7

Predviđanje utroška resursa

Za vrijeme oblikovanja arhitekture sustava moguće je analizirati svojstva sustava kvalitativnim i kvantitativnim pristupom. Kvalitativna analiza ispituje funkcionalna svojstva sustava prema zahtjevima (mogućnost ulaska sustava u nedopušteno stanje ili siguran rad sustava) i atributima kvalitete, dok je kvantitativna analiza zasnovana na mjerenjima ključnih parametara sustava. Metode ocjene performansi temeljene na simulaciji mogu obuhvatiti svu složenost ponašanja sustava na način prikladan za analizu, bilo direktnom analizom svojstava sustava, ili kao izvor kvantitativnih podataka za analizu [12].

Postupci ocjene utjecaja elemenata dizajna sustava na ponašanje sustava u ranim fazama razvoja tradicionalno se oslanjaju na izradu prototipa koji se zatim izvode na ciljnoj platformi, uz praćenje ponašanja i mjerenje performansi. Izrada prototipa je zahtjevna, a dobiveni rezultati ovise o stabilnosti dizajna i stupnju razvoja sustava. Mogućnost predviđanja i ocjene performansi sustava na osnovi modela dizajna, bez potrebe za poznavanjem detalja implementacije uvelike bi unaprijedili postupke razvoja sustava i smanjili potrebno vrijeme i trošak ocjene performansi [15]. Metode koje su razvijene temelje se na uporabi modela jezika UML, sintezi prototipa na osnovu modela ili raznim analitičkim modelima.

Becker i dr. u [15, 16] navode osnovna načela koje metode procjene trebaju po-

štivati (tablica 7.1). Ova načela ujedno su i zahtjevi koji se postavljaju pred postupke procjene performansi sustava, kao i postupaka praćenja utroška resursa sustava.

Tablica 7.1: Osnovna načela metoda procjene performansi.

Zahtjev	**Opis**
točnost	Procjena treba biti dovoljno točna da njeni rezultati budu korisni, uz odgovarajući kompromis između točnosti i truda potrebnog za efikasnu procjenu.
prilagodljivost	Postupak treba podržati mogućnost promjene strukture sustava u slučaju dodavanja ili promjene komponenata koje ga tvore.
isplativost	U odnosu na izradu prototipa i mjerenja ponašanja, metoda treba zahtijevati manje truda i vremena.
potpora kompoziciji	Postupak treba uvažiti uporabu načela kompozicije komponenata za izgradnju sustava i iskoristiti hijerarhijsku strukturu analiziranog sustava uporabom rezultata procjene niže razine za procjenu više razine.
skalabilnost	Sustav se tipično gradi uporabom velikog broja jednostavnih, ili malog broja složenih komponenata. Metoda treba podržati oba slučaja ravnopravno.
mogućnost analize	Osim otkrivanja uskih grla, postupak treba omogućiti otkrivanje elemenata arhitekture sustava koji su im uzrok.
općenitost	Odabrani pristup treba biti primjenjiv na različite komponentne tehnologije (modele) uz minimalne promjene, čime se omogućuje procjena performansi sustava izgrađenih uporabom više tehnologija.

Potpora analizi trebala bi biti ugrađena u komponentni model. Komponentni modeli opće namjene koji su danas u primjeni (COM/.Net [68], J2EE/EJB [87], Corba Component Model [72] i dr.) ne pružaju mehanizme za modeliranje atributa kvalitete sustava i predviđanje ponašanje sustava prema tim atributima (iako sadrže mehanizme

za nadogradnju komponentnog modela vlastitim atributima, pa tako i atributima kvalitete [38]). Komponentni modeli koji su nastali kao rezultat istraživanja u akademskoj zajednici i čiji je cilj analiza i predviđanje ponašanja sustava, izgrađeni su tako da podržavaju mogućnost oblikovanja svojstava kvalitete [62, 63].

Tri su glavna preduvjeta za predviđanje performansi sustava temeljenih na programskim komponentama [15]:

1. potpora specifikaciji performansi komponenata uz uvažavanje ovisnosti o okruženju (platformi) i različitim resursima,
2. odabir komponenata na osnovu performansi i karakteristika,
3. kombinacija metoda mjerenja i modeliranja ugrađena u automatizirani okvir (eng. *framework*).

7.1 Pregled postupaka analize i predviđanja utroška resursa

U preglednom radu Balsamo i dr. daju pregled tehnika predviđanja performansi temeljenih na modelu sustava [11]. Iz njihove studije vidljivo je nekoliko smjerova kojim se istraživanja kreću. Metode su prema analitičkom modelu podijeljene u:

- metode zasnovane na mrežama s čekanjem (eng. *queuing networks*),
- metode zasnovane na procesnoj algebri,
- metode zasnovane na Petrijevim mrežama,
- metode zasnovane na simulacijskim modelima,
- metode zasnovane na stohastičkim procesima.

Nekoliko predloženih pristupa zasniva se na uporabi simulacijskih modela izvedenih iz dijagrama jezika UML [46]. Proširi li se jezik UML profilima, može ga se uporabiti za opis vremenskih svojstava sustava direktno u modelu. Miguel i dr. predlažu skup proširenja jezika UML temeljenih na profilima, kojima se mogu opisati vremenski zahtjevi i utrošak resursa [40]. Tako prošireni dijagrami UML-a automatski se pretvaraju u modele raspoređivanja zadataka i simulacijske modele. Slično kao u postupcima predloženima u ovom radu, upotrijebljeni su generatori analitičkih i simulacijskih modela, koji modele elemenata sustava pretvaraju u podmodele te ih zatim integriraju u cjeloviti model sustava. Grassi i Mirandola [49] predlažu preslikavanje između elemenata jezika UML i vlastitog jezika s ciljem iskorištavanja metoda i algoritama analize performansi predloženih nad jezikom UML.

Sličan pristup predlažu Arief i Spiers koji koriste UML za opis sustava s detaljima potrebnim za simulaciju. Simulacija je procesno orijentirana i izgrađena je na temelju okvira prethodno razvijenog jezika Simulation Modeling Language (SimML) koji implementira ključne elemente procesno-orijentirane paradigme – komponente, procese, redove poruka i poruke [10].

Od metoda opisanih u preglednom radu, zanimljivo je da su jedine metode čije su glavno područje primjene bili ugradbeni sustavi navedene samo dvije: jedna metoda temeljena na slojevitim mrežama s čekanjem (eng. *layered queuing network*) i jedna metoda temeljena na simulaciji. Ta je metoda ujedno bila među rijetkima koje su nastojale korisniku dati povratnu informaciju o rezultatima analize. Metode zasnovane na simulaciji spadaju u skupinu metoda koje zahtijevaju visoku razinu detalja u modelu.

Autori studije [11], Balsamo i Marzolla i sami predlažu sličan alat za simulaciju performansi za procesno-orijentirane sustave [12, 67]. Dijagrami UML-a prošireni dodatnim informacijama iskorišteni su za opis parametara sustava i analizu performansi. Elementi korišteni u dijagramima bliski su elementima simulatora, a struktura i ponašanje simulatora slijede strukturu i ponašanje modelirano UML-om. Dijagrami su podloga za automatsko generiranje simulacijskog modela zasnovanog na diskretnim događajima, a zanimljivo je da su rezultati simulacije vidljivi u dijagramima kao oznake.

Razmatranje metoda analize u području ugradbenih sustava ne smije zaobići pristupe koji koriste alate Matlab i Simulink. Ovi alati postali su općeprihvaćeni alati za oblikovanje i analizu ugradbenih sustava. COMDES, okvir za analizu raspodijeljenih upravljačkih sustava sa zahtjevima tvrdog stvarnog vremena, koristi Matlab i Simulink za analizu. COMDES predstavlja podsustave i upravljačke signale dijagramima uloga (eng. *actor diagrams*) i koristi dijagrame stanja ili blok-dijagrame za opis ponašanja sustava, te zatim prevodi takav model sustava u oblik pogodan za simulaciju u Simulinku [8, 9, 66]. Kao kritiku ovih alata (pogotovo Simulinka) možemo izdvojiti slabu potporu za modularno oblikovanje sustava [7] i slabu povezanost kontinuiranog i diskretnog modela sustava [48].

Becker i dr. istražili su postupke predviđanja performansi sustava temeljenih na programskim komponentama s naglaskom na metode procjene performansi [15]. Postupke su podijelili u dvije osnovne grupe: kvalitativne i kvantitativne. Kvalitativni postupci predviđaju performanse sustava na osnovu grubog modela sustava, te su tako pogodni za ocjenu sustava u ranim fazama razvoja, prije izgradnje detaljnog analitičkog modela sustava.

Zaključak o ponašanju sustava može se izvesti uporabom atributa arhitekture sustava i komponenata sustava. Wallnau i dr. nastoje povezati te dvije razine apstrakcije sustava raspoznavanjem osobina arhitekture primijenjenih u izgradnji sustava u kombinaciji s atributima kvalitete pojedinih komponenata [93].

Metode temeljene na zaključivanju na visokoj razini rijetke su u odnosu na metode zasnovane na analizi modela sustava i za cilj uglavnom imaju izgradnju modela za analizu performansi, za što je ključna uporaba alata za automatiziranu pretvorbu modela sustava u analitički model i alata za automatsko predviđanje ponašanja sustava na osnovu neke analitičke metode [69]. Nedostatak ovakvog pristupa je manjak općenitosti – velik broj komponentnih modela i različitost atributa kvalitete onemogućavaju uspostavljanje općenito primjenjivog postupka. Kao rješenje neki istraživači predlažu privremene (među)modele (eng. *intermediate model*) koji se automatski generiraju iz modela sustava zadanog jezikom UML i predstavljaju osnovu za analizu [79]. Atributi kvalitete u modelu sustava opisani su u okviru standardnog profila UML-SPT [70].

Kvantitativni pristupi mogu se temeljiti na mjerenju performansi, analizi atributa modela i kombinaciji ova dva pristupa. Mjerenje performansi koristi se prvenstveno kad je sustav potpuno ili djelomično implementiran – kad postoji prototip sustava ili je moguće mjeriti performanse sustava uz poznavanje svojstava gotovih komercijalnih komponenata (eng. *COTS - commercial off the shelf*) i platforme na kojoj je sustav temeljen [37]. Pristupi temeljeni na modelu [12, 67] opisani su ranije.

7.2 Potpora postupcima analize u jeziku UML

Postupci predviđanja utroška resursa temeljeni na modelu sustava grade analitički okvir iz modela sustava. U slučaju da je model temeljen na jeziku UML, na raspolaganju je standardni profil jezika za modeliranje i analizu ugradbenih sustava za rad u stvarnom vremenu – UML MARTE [71]. Nastao kao nasljednik profila UML SPT [70], MARTE donosi proširenja od kojih je najvažniji novi model vremena koji proširuje jezik UML vremenskim elementima [78].

Profil UML MARTE sadrži između ostalog elemente za oblikovanje:

- vremena,
- nefunkcionalnih svojstava (eng. *non-functional properties*, NFP),
- generičkih resursa sustava.

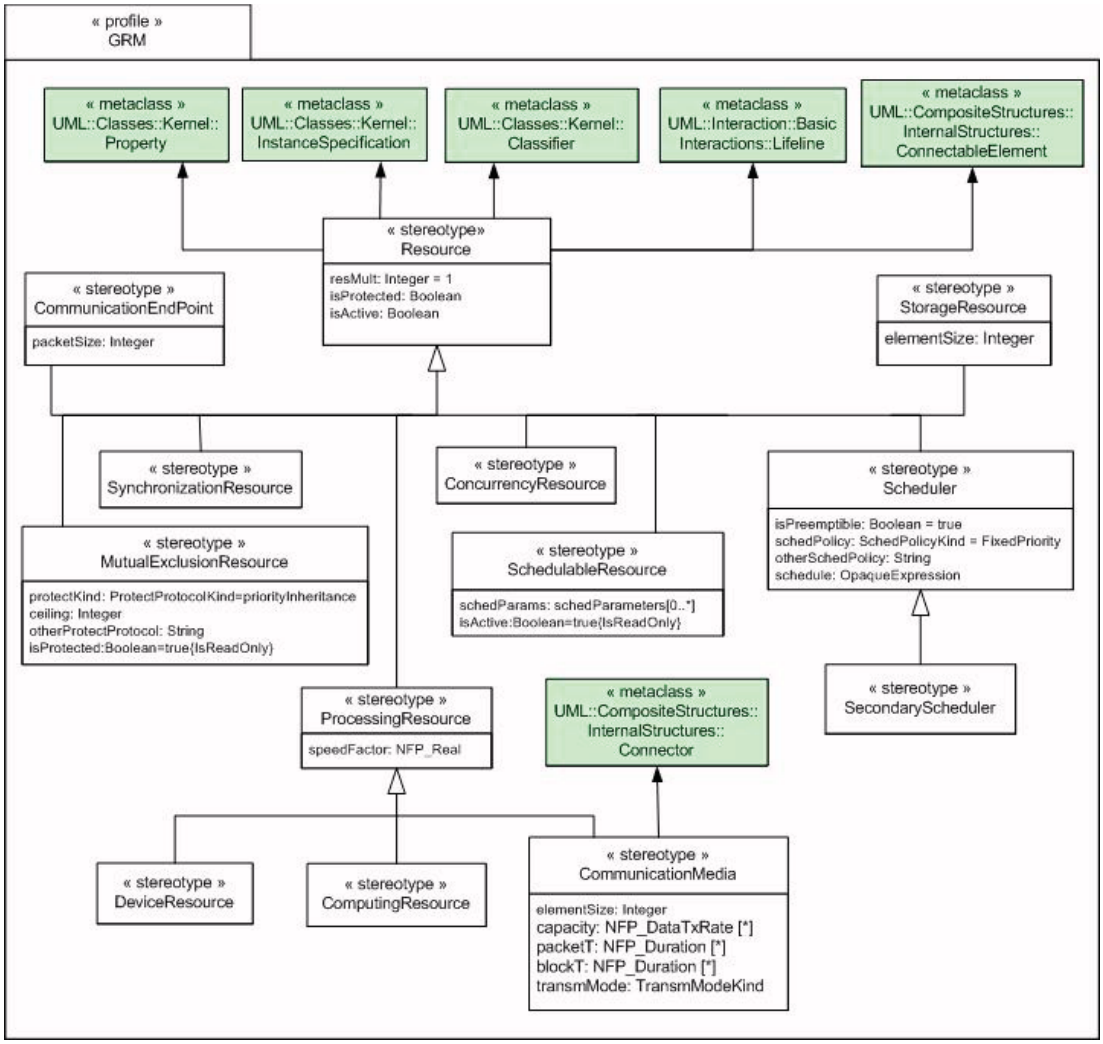
Profilom je tako moguće u model unijeti satove, ograničenja sata i resurse. Definirano je nekoliko tipova resursa.

- memorijski resurs (eng. *storage resource*) predstavlja memorijski element (memoriju, tvrdi disk, ...) zadanog kapaciteta i veličine pojedinog elementa (u bitovima). Brzina pristupa određuje se u odnosu na referentni sat kao broj ciklusa potrebnih za pristup jednom memorijskom elementu;

- vremenski resurs (eng. *timing resource*) predstavlja sklopovsku ili programsku jedinicu sposobnu pratiti prolazak vremena – sat ili vremenski sklop;
- sinkronizacijski resurs (eng. *synchronization resource*) predstavlja zaštićene resurse koji se koriste za sinkronizaciju paralelnih operacija, međusobno isključivanje i pristup dijeljenim resursima;
- računski resurs (eng. *computing resource*) predstavlja aktivni resurs koji izvodi programski kôd – procesni element, procesor;
- konkurentni resurs (eng. *concurrency resource*) predstavlja aktivni resurs koji može izvoditi operacije konkurentno s drugim takvim resursima – zadatak, nit izvođenja;
- resurs uređaja (eng. *device resource*) predstavlja vanjski uređaj koji zahtijeva poseban pristup ili upravljanje;
- komunikacijski resurs (eng. *communication resource*) predstavlja komunikacijske mogućnosti sustava – komunikacijski medij ili pristupnu točku (eng. *end point*).

Struktura elemenata za opis resursa prikazana je na slici 7.1. Profil ujedno sadrži i elemente za opis generičke kvantitativne analize sustava na osnovi statističke analize mjerenja ponašanja sustava, scenarija testiranja sustava i parametara generatora opterećenja.

Zbog svoje vezanosti uz složenu strukturu jezika UML i vlastite složenosti, MARTE nije prikladan za direktnu primjenu izvan jezika UML. Uzrok složenosti je općenitost pristupa koji nastoji stvoriti okvir za širok raspon primjena. Izrada modela sustava koji sadrži sve elemente profila MARTE vrlo je zahtjevna i isplativa samo u slučajevima kad je izrada prototipa i kasnija promjena dizajna vrlo skupa (npr. avionska industrija).



Slika 7.1: Dio profila MARTE za opis općenitih resursa [71].

7.3 Postupci temeljeni na vremenskim automatima

Jezik za opis ponašanja REMES predviđa uporabu vremenskih automata s cijenom kao analitičkog modela i alata UPPAAL Cora za provođenje postupka analize. Alat za formalnu analizu vremenskih automata s cijenom, UPPAAL Cora [60], omogućuje pronalaženje minimalne cijene prihvatljivog stanja sustava opisanog mrežom vremenskih automata s cijenom. U nastavku je dan pregled mogućnosti analize uporabom analitičkog modela vremenskih automata.

Osnovni algoritam implementiran u alatu UPPAAL Cora prikazan je algoritmom 7.1 [19]. Algoritam održava nekoliko listi simboličkih stanja (l, c) gdje je l lokacija, a c trenutna cijena. Lista *Passed* sadrži sva stanja koja su obiđena, dok je *Waiting* lista čekanja koja sadrži stanja što ih tek treba obići. Inicijalno lista čekanja sadrži samo početno stanje S_0. Varijabla cijene *Cost* u svakom trenutku sadrži najmanju pronađenu cijenu dolaska do ciljne lokacije (inicijalno je beskonačno velika). Algoritam pretražuje prostor stanja u petlji sve dok ne obiđe sva stanja. U svakom prolasku petlje iz liste čekanja odabire se stanje S. Ako je stanje S takvo da je njegova cijena veća od cijene nekog od obiđenih stanja ($Passed \leqslant_d omS$) ili do ciljnog stanja nije moguće doći s cijenom manjom od *Cost* ($C + remain(S) \geq Cost$), stanje S se odbacuje. U suprotnom, S se dodaje u listu obiđenih stanja. Ako je S ciljno stanje, osvježava se varijabla cijene *Cost*, inače se sva stanja sljedbenici S dodaju u listu čekanja *Waiting* i postupak se nastavlja [21].

Opisani algoritam pronalazi minimum jedne varijable cijene. U slučaju više varijabli cijene javlja se problem optimizacije više varijabli te se gubi jednoznačnost rješenja. Bouyer i dr. pokazali su u [29] da je za automat s dvije varijable cijene moguće odrediti optimalnu strategiju dolaska do ciljnog stanja u automatu s ciklusom pretvorbom problema u problem pronalaženja minimalnih ciklusa u usmjerenom grafu [58, 39]. Pronalaženje optimalne strategije vremenskih automata s cijenom ograničeno je na automate s jednim ili dva sata [30], dok za automate s tri sata nije moguće odrediti optimalnu strategiju [28]. U slučaju višestrukih varijabli cijene, do rješenja je moguće doći uz odabir jedne primarne varijable cijene za optimiranje i ograničavanje gornje

Algoritam 7.1: Algoritam pretraživanja prostora dostupnih stanja uz traženje najmanje cijene implementiran u alatu UPPAAL Cora

Input: *Goal* // skup ciljnih stanja
Data: $Cost := \infty$
Data: $Passed := \emptyset$
Data: $Waiting := \{S_0\}$
while $Waiting \neq \emptyset$ **do**
  select $S \in Waiting$; // //ovisno o strategiji prolaska kroz graf
  $C \leftarrow infimum(S)$;
  if $Passed \not\leq_{dom} S \wedge C + remain(S) < Cost$ **then**
    $Passed \leftarrow Passed \cup \{S\}$;
    if $S \in Goal$ **then**
      $Cost \leftarrow C$;
    else
      $Waiting \leftarrow \{S' \mid S' \in Waiting \vee S \rightarrow S'\}$;
    end
  end
end
return *Cost*

granice ostalih varijabli [20], a bez uvođenja ograničenja samo uz odabir pravilne granulacije vremena, što je i dalje otvoreni problem [59].

7.3.1 Komentar svojstava

Spomenuti postupci udovoljavaju dijelu zahtjeva koji se postavljaju pred metode procjene performansi sustava. Točnost ovih postupaka je potpuna i oni garantiraju pronalaženje optimalnog rješenja u slučaju jedne varijable cijene. U realnim sustavima rijetko se javlja situacija da je prilikom dizajna sustava dovoljno pratiti samo jedan resurs sustava, te se kao varijabla cijene koristi težinska sume troška pojedinih resursa sustava. Time je točnost ograničena jer ukupna cijena ovisi o odabranim težinskim faktorima, a dobivena ocjena perfomansi može dati samo grubi pregled ponašanja

sustava. Analiza se temelji na pretvorbi modela u vremenski automat, što zahtijeva detaljan model sustava kako bi se dobila ispravna ocjena. Uz pretpostavku automatiziranog postupka pretvorbe modela u vremenski automat (što je realna pretpostavka), postupak je prilagodljiv i isplativ – promjena strukture sustava zahtijeva ponovno generiranje vremenskog automata, a automatizirana pretvorba nije vremenski zahtjevna. Automatizirana pretvorba temelji se na transformaciji grafa što nije računski složena transformacija [13], a većina složenosti je u algoritmu analize. Načelo potpore kompozicije poštuje se utoliko što se analiza provodi na mreži vremenskih automata (horizontalna kompozicija), dok je vertikalna kompozicija komponenata slabo iskorištena. Rješenja koja nastoje iskoristiti hijerarhiju komponenata (vezanu uz pojedini komponentni model) za poboljšanje postupka analize vremenskih automata nisu prilagođena analizi vremenskih automata s cijenom te nisu općenita [53, 52, 51]. Skalabilnost kod ovih metoda dolazi u pitanje kod složenih sustava gdje je računski postupak vrlo složen zbog potrebe za obilaskom grafa svih dostupnih stanja. Opisani formalni postupci kao rezultat analize mogu otkriti ograničenja sustava, poštivanje postavljenih ograničenja i prikazati slijed koraka (stanja) koji je doveo do kršenja ograničenja (eng. *counter-example*). Još nije riješen problem interpretacije slijeda koraka u kontekstu komponentnog modela i modela ponašanja.

7.4 Postupci temeljeni na jeziku Charon

Jezik Charon opisan u poglavlju 4.2 poslužio je kao inspiracija za izgradnju jezika REMES. Sustav opisan jezikom Charon sastoji se od hijerarhijskih agenata s pridruženim ponašanjima građenim od modusa (eng. *mode*). Nakon uklanjanja hijerarhije, model se simulira kao hibridni automat, pa slično vremenskim automatima i Charon razlikuje diskretne i kontinuirane korake. Izvođenje modela odvija se u ciklusima (eng. *round*) – razlikujemo cikluse osvježavanja (eng. *update rounds*) i cikluse prolaska vremena (eng. *time rounds*) koji se međusobno izmjenjuju.

U svakom ciklusu osvježavanja nedeterministički se odabire jedan modus i osvježava njegovo stanje. Osvježavanje stanja sastoji se od izvođenja niza prijelaza počevši

od zadane ulazne točke modusa (eng. *default entry point*) pa sve do zadane izlazne točke modusa (eng. *default exit point*).

Ulaskom kroz zadanu ulaznu točku potrebno je osvježiti stanje modusa. Kontekst modusa pohranjen je u globalnu varijablu povijesti i vraća se tako što se redom izvode prijelazi sve dok se ne aktivira podelement koji je pohranjen u kontekst. Pohranjeni element je onaj u kojem daljnje napredovanje (diskretno i kontinuirano) nije bilo moguće jer uvjet za sljedeći prijelaz nije bio ispunjen. U nastavku, izvode se svi mogući prijelazi.

Vremenski ciklusi ostvaruju prolazak vremena – za stanje s dopušta se prolazak vremena koji stvara novo stanje s' takvo da zadovoljava ograničenja sata i poštuje pravila promjene diferencijalnih varijabli. Novo stanje je rješenje skupa jednadžbi diferencija s početnim uvjetom s.

Potporu hijerarhijskom simuliranju Charon ostvaruje načelom *modularnog simuliranja*. Svako ponašanje može se simulirati odvojeno od ostatka sustava i zatim dobivene rezultate diferencijalnih jednadžbi uključiti u cjelokupno rješenje postupkom numeričke integracije. Modularna simulacija koju predlažu autori jezika Charon aproksimacija je idealnog rješenja.

Modularnim simuliranjem ponašanja se ciklusi osvježavanja i prolaska vremena svakog elementa sastavljaju od istoimenih ciklusa podelemenata. Prednost ovog pristupa je što se hijerarhijska struktura ne uklanja, već ostaje sačuvana. Osnovni ciklus osvježavanja prikazan je algoritmom 7.2.

Vremenski ciklusi u modularnoj simulaciji temelje se na pretpostavci da modusi koji su više u hijerarhiji rade *sporije* od svojih podmodusa. Tako podmodus podrazumijeva da su varijable njegovog roditelja nepromjenjive za vrijeme izvođenja vlastitog vremenskog ciklusa. Vremenski ciklus podmodusa odvija se u okviru vremena vremenskog ciklusa roditelja. Po završetku izvođenja podmodusa, roditelj integrira rješenje diferencijalnih jednadžbi podmodusa u vlastito rješenje. Opisani postupak prikazan je algoritmom 7.3.

Algoritam 7.2: Ciklus osvježavanja modularne simulacije jezika Charon

```
Input: m : Nacin                                          // trenutni modus
Input: p : Tocka                                          // ulazna točka
Input: s : Stanje
Result: (Tocka, Stanje)
Data: Nacin md := m
Data: Tocka pt := p
Data: Stanje st := s
repeat
    if pt = zadana ulazna točka then
        izvedi zadani ulazni prijelaz iz pt ;
    else
        odaberi neki prijelaz t od omogućenih prijelaza za (md, pt, st) ;
        izvedi prijelaz t ;
    end
    if md = m ∧ pt je izlazna točka then
        return (pt, st)
    else ;                                        // došli smo do podmodusa

        (pt, st) =osvježi(md, pt, st) ;          // nastavi rekurzivno u podmodus
    end
until ima omogućenih prijelaza za (md, pt, st);
povijest = (md, pt) ;
return ( zadana izlazna točka, st )
```

Procjena parametra intervala vremenskog ciklusa modularne integracije zahtijeva da vrijeme u kojem su bilo koja dva modusa nesinkronizirana (zbog računanja jednadžbi podmodusa) nikad nije veće od nekog zadanog intervala ε. Pronalaženjem dva modusa s minimalnim lokalnim vremenima t_1 i t_2 i odabirom intervala $t_2 - t_1 + \varepsilon$ dobiva se interval vremenskog ciklusa.

7.4.1 Komentar svojstava

Analitički model jezika Charon je upravo sâm jezik. U ovisnosti koja se varijanta postupaka analize koristi, klasična ili modularna, za analizu se koristi model s uklonjenom hijerarhijom ili početni model. Primjena modularne analize u postupku pruža kompromis točnosti i brzine – ako je pretpostavka veće brzine integracije podmodusa nego roditelja ispunjena, modularni algoritam dobro iskorištava hijerarhiju modela. Greška zbog nesinkronizacije ograničena je minimalnim intervalom ε. U slučaju da je taj interval vrlo mali, gubi se prednost bržeg izvođenja modularnog algoritma.

Algoritam 7.3: Vremenski ciklus modularne simulacije jezika Charon

Input: m: *Nacin* // trenutni modus
Input: i: *Invarijant* // invarijant roditelja
Input: t: *Vrijeme* // vrijeme integracije roditelja
Input: s: *Stanje*
Result: (*Stanje*, *Vrijeme*)
Data: *Nacin md*
Data: *Vrijeme* $d, dt, tm := 0$
Data: *Stanje st*
Data: *Invarijant* $inv = m.invarijant \cup i$ // ukupni invarijant
$(md, pt) = povijest$;
while $tm < t$ **do** // ponavljaj u okviru vremena t
  $inv = pojednostavi(s, inv)$; // pojednostavi invarijant za varijable koje se ne mijenjaju
  $dt = procijeni(inv, s, m.ogranicenja, tm)$; // odredi dt
  $(st, d) = vremenskiCiklus(md, s, inv, dt)$; // izvedi podmodus
  $st = integriraj(st, m.ogranicenja, d)$; // integriraj rezultate podmodusa
  if $d < dt \vee prekrsen(inv, st, tm + d)$ **then**
    return $(st, tm + d)$; // izađi s trenutnim vremenom
  end
  $tm = tm + dt$;
end
return (st, tm) // iscrpljen cijeli interval

7.5 Postupci analize temeljeni na jeziku REMES

Postupci analize temeljeni na jeziku REMES razvijeni su uz uvažavanje načela predstavljenih u uvodu ovog poglavlja. Uvažavanje posebnosti platforme ostvareno je modelom izvedbenog okruženja, koji na visokoj razini opisuje platformu na kojoj se temelji sustav. Činjenica da jezik REMES trenutno ne podržava hijerarhiju na više od jedne razine ne olakšava bitno analizu, jer je potrebno uvažiti postojanje hijerarhije u komponentnom modelu.

Kao pristup analizi odabrana je simulacija. Simulacijski model izgrađen je iz cjelovitog modela sustava koji se kombinira s modelom izvedbenog okružja. Osnovni ciklus simulacije predstavljen je algoritmom 7.4. Simulacijski model izgrađen je iz svih ponašanja unutar cjelovitog modela sustava. Početno aktivni modusi su oni izvedeni iz ponašanja komponenata koje nemaju ulazne aktivacijske signale (pod pojmom komponente podrazumijevaju se komponente obje razine komponentnog modela ProCom – ProSys i ProSave). Osnovni ciklus simulatora ponavlja se sve dok ima aktivnih modusa. U svakom ciklusu pretražuje se lista aktivnih modusa i traži diskretni događaj s najranijim vremenom (redak 1). Pretraživanje događaja uzima u obzir:

- invarijante modusa – računa se vremenski interval u kojem je invarijant zadovoljen,
- uvjete bridova – računa se vremenski interval u kojem će uvjet postati ispunjen,
- prijelaze granica komponenata – prijenos kontrole između komponenata,
- prijelaze hijerarhije – aktivaciju podelemenata,
- odluke korisnika – korisnik ima mogućnost mijenjanja liste prijelaza,
- prioritet prijelaza – prijelazi višeg prioriteta izvode se odmah.

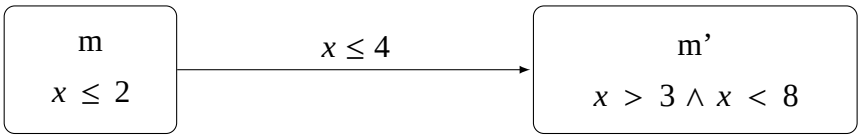
Pronalaskom događaja s najranijim vremenskim odmakom (koji može biti nula) poznat je vremenski interval u kojem sustav ne može promijeniti stanje diskretnim prijelazom.

U tom intervalu sustav mijenja stanje isključivo kontinuiranim prijelazom, te je u sljedećem koraku moguće osvježiti stanje uz uvažavanje uvjeta kontinuiranih promjena varijabli resursa (redak 2). Nakon kontinuiranih prijelaza dopušta se prolazak vremena i osvježavaju varijable sata (redak 3). Posljednji korak je filtiranje diskretnih prijelaza (redak 4) i obavljanje diskretnih prijelaza (redak 5).

U ovom trenutku treba naglasiti da varijable satova i resursa (sve varijable koje se osvježavaju kontinuiranim prijelazima) nisu skalarne, već su predstavljene intervalima. Tijekom izvođenja simulacije te varijable poprimaju vrijednost iz skupa predstavljenog intervalom. Za primjer, uzmimo da je prvi sljedeći diskretni događaj moguć u vremenskom intervalu $t = \langle 1, 3]$, odnosno $t > 1 \wedge t \leq 3$. Neka je sustav u stanju koje sadrži modus m sa troškom kontinuiranog prijelaza za resurs procesora $cpu' = 10$. Poslije izvođenja kontinuiranog prijelaza, vrijednost varijable resursa cpu biti će uvećana za $\langle 1, 3] \cdot 10 = \langle 10, 30]$.

Ključni element postupka je određivanje najranijeg diskretnog događaja u sustavu. Za svaki modus izračunava se vremenski interval u kojem njegovi izlazni diskretni prijelazi postaju aktivni, uz istovremeno uvažavanje invarijanta izvorišnog i odredišnog modusa za svaki prijelaz. Za primjer uzmimo ponašanje sa slike 7.2. Modus m ima invarijant $x \leq 2$, što znači da sustav mora napustiti taj modus najkasnije u vremenskom trenutku $x = 2$ ili ranije, što se može iskazati suprotnim uvjetom, $x > 2$. Diskretan prijelaz $m \xrightarrow{x \leq 4} m'$ omogućen je u vremenskom intervalu $x > 2 \wedge x \leq 4$, odnosno $x \in \langle 2, 4]$. Invarijant modusa m' je $x > 3 \wedge x < 8$, što znači da je ulazak u m' moguć u vremenskom intervalu $x \in \langle 3, 8\rangle$. Interval u kojem je moguće prijeći iz m u m' je presjek ova dva intervala, $x \in \langle 3, 4]$. Ako je trenutna vrijednost sata $x = \langle 0, 1\rangle$, prijelaz $m \xrightarrow{x \leq 4} m'$ moguć je po isteku vremena $t \in \langle 3, 3]$, odnosno $t = 3$.

Uporabom intervala moguće je izračunati vremena diskretnih prijelaza u sustavu. Postojanje više satova u istom kontekstu dopušteno je, jer se intervalima računa vremenski odmak u odnosu na trenutnu vrijednost sata. Problem se javlja kod definiranja uvjeta sa dva sata, oblika $x_1 - x_2 < c$. Takvi uvjeti nazivaju se i dijagonalnim uvjetima (s istim značenjem kao u vremenskim automatima, poglavlje 3.1.1). Ovakvi uvjeti ne mogu se prikazati intervalima. Uporaba intervala oblikuje uvjete koji odgovaraju vre-



Slika 7.2: Primjer izračuna intervala diskretnog prijelaza.

menskim zonama koje su strogo pravokutne (nemaju dijagonala). Za prevladavanje ovih ograničenja potrebno je primijeniti matrice ograničenih razlika (eng. *difference bounds matrix*) [22, 23].

7.6 Zaključak

U ovom poglavlju predstavljene su karakteristike postupaka za analizu i predviđanje performansi sustava. Predstavljeni postupci temeljeni na vremenskim automatima i jeziku Charon poslužili su u konstrukciji vlastitih postupaka analize temeljenih na simulaciji.

Osvrnimo se na opisane glavne preduvjete za uspješno predviđanje performansi. Uporabom komponentnog modela ProCom ostvarena je potpora specifikacije komponenata, jezik REMES osigurava ponašanja komponenata za ocjenu performansi uz uvažavanje ovisnosti o resursima platforme, a eventualni rezultati mogu se pridružiti opisu komponente pomoću atributa komponentnog modela. Predloženi postupak ostvaren je kombinacijom modeliranja jezikom REMES i mjerenja ponašanja simuliranog modela sustava kroz automatizirane alate opisane u poglavlju 8. Preduvjet mogućnosti odabira komponenata na osnovu performansi predmet je razvojnog procesa vezanog uz komponentni model ProCom.

Pristup simulacije omogućuje praćenje kretanja svakog resursa pojedinačno, što je u slučaju formalne verifikacije otežano. Trenutna implementacija alata UPPAAL Cora je u ovom pogledu ograničena – provjera modela, analiza i simulacija izvode se osvježavanjem jedne monotono rastuće varijable cijene. Proširenje postupka moguće je u smjeru analize cijelog prostora stanja (simulacija zahtijeva odabir jednog puta kroz

Algoritam 7.4: Osnovni ciklus simulacije jezika REMES

```
   Data: Stanje s
   Data: Vrijeme t := 0
   Data: ListaPrijelaza lp := ∅
   Data: ListaNacina tm := {...}                    // početno aktivni modusi
   s.aktivniNacini = tm ;
   while s.aktivniNacini ≠ ∅ do                     // dok ima aktivnih modusa
1      (t, p) = najranijiDogadjaj(s.aktivniNacini) ;
       if t ≠ ∅ then
          forall the Nacin m: s.aktivniNacini do
2            kontinuiraniKorak(m, t) ;
             forall the Sat x: m.varijableSata do
3               x = x + t ;
             end
4            filtrirajPrijelaze(lp) ;
             forall the Prijelaz p: lp do
5               s.izvediPrijelaz(p) ;        // redoslijed odabira diskretnih
                prijelaza nije garantiran
             end
          end
       end
   end
```

moguća stanja), no ugradbeni sustavi koji se modeliraju tipično su cikličkog ponašanja te je tako ukupni prostor stanja uz uvažavanje resursa – beskonačan. Predloženi postupak zadržava informacije o arhitekturi sustava i u procesu simulacije poštuje hijerarhijske odnose modeliranog sustava.

Poglavlje 8

Alati za analizu utroška resursa

Postupci opisani u prethodnom poglavlju predviđaju automatizaciju u svim fazama, uporabu automatske pretvorbe modela, generiranja programskog kôda iz modela i drugo. Ovo poglavlje daje pregled opće strukture alata za analizu koji su djelomično razvijeni u sklopu ovog rada.

8.1 Pregled platforme

Skup alata za analizu utroška resursa zasnovan je na platformi Eclipse [41] temeljenoj na komponentama (uslugama) prema specifikaciji OSGi (Open Services General Infrastructure) [76]. Inicijativa OSGi pokrenuta je s ciljem stvaranja otvorene specifikacije za gradnju sustava zasnovanih na uslugama uz osiguravanje horizontalne i vertikalne raslojenosti. Važna karakteristika platformi temeljenih na specifikaciji OSGi je dinamičnost – aplikacije se grade od komponenata koji se mogu preuzeti s mreže, instalirati, pokrenuti, zaustaviti, nadograditi ili ukloniti bez zaustavljanja sustava.

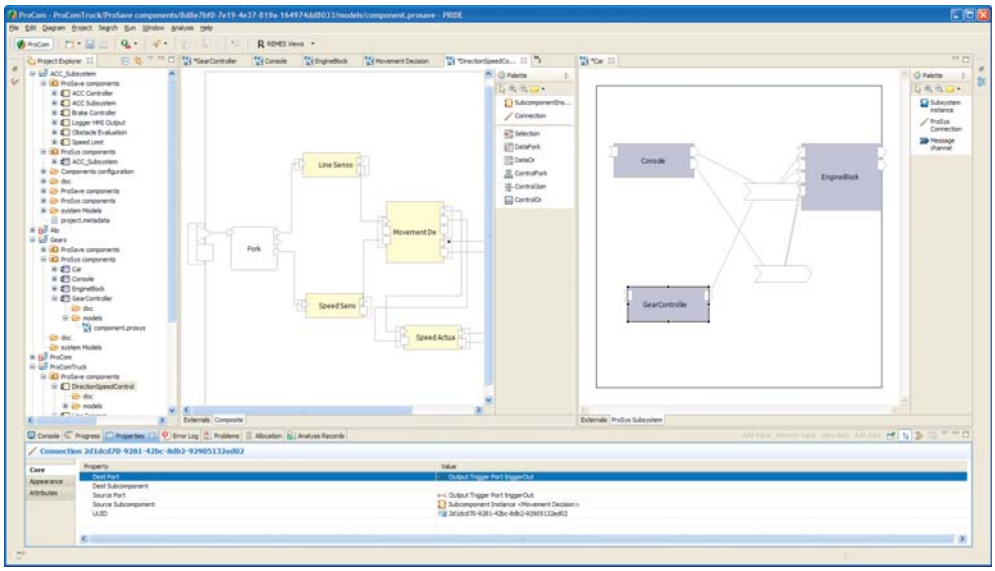
Jezgra platforme Eclipse je Equinox [44], implementacija specifikacije OSGi i okvir za izgradnju aplikacija koji ostvaruje životni ciklus aplikacije i registar dostupnih usluga. Registar usluga omogućava paketima da prate događaje kao što su pojavljiva-

nje novih usluga ili nestajanje usluga iz sustava i reagiraju shodno tome. OSGi raspoznaje dva temeljna koncepta koji odgovaraju pojmu komponente – paket (eng. *bundle*) i usluga (eng. *service*). Aplikacija se gradi od paketa koji sadrže usluge. Razlika između ova dva pojma je u tome što je paket samostalna jedinica koja se može instalirati i pokrenuti u sustavu, dok usluga može biti aktivna samo u sklopu paketa. U terminologiji koju koristi Eclipse, paket je uglavnom jednak pojmu dodatka (eng. *plugin*). Dodatak platforme Eclipse može proširiti funkcionalnost drugih dodataka koristeći točke proširenja (eng. *extension points*), definirati vlastite točke proširenja, nuditi usluge ili pakete jezika Java (eng. *packages*). Zadnja mogućnost je uključena u platformu kako bi se podržala upotreba kôda koji nije prilagođen platformi Eclipse, već koristi mehanizme jezika Jave za povezivanje. Dodatci manifestom oglašavaju svoje zahtjeve – postojanje i odgovarajuće verzije drugih dodataka, paketa jezika Jave, usluga ili točaka proširenja. Manifest se sastoji od jedne ili više datoteka koje opisuju dodatak – ime, verziju, zahtjeve i funkcionalnosti koje nudi. Equinox omogućava da paralelno bude aktivno više verzija istog dodatka.

Osim kroz dodatke, platforma Eclipse omogućava i nadogradnju funkcionalnosti kroz proširenja u obliku fragmenata (eng. *fragment*) i značajki (eng. *feature*). Iako se najčešće spominje kao razvojno okruženje za izradu aplikacija u programskom jeziku Javi i srodnim platformama, Eclipse je generičko okruženje i može poslužiti kao osnova za bilo koji tip aplikacije. Na osnovi platforme Eclipse nastalo je i okruženje za rad sa komponentama prema komponentnom modelu i razvojnom okviru ProCom – PRIDE (Progress IDE) [86]. Okruženje PRIDE je još u razvoju i sadrži dodatke koji omogućavaju definiranje i rad s komponentama razina ProSave (slika 8.1) i ProSys, te osnovne analize dizajniranog sustava.

8.2 Integrirano razvojno okruženje unutar PRIDE

Platforma izgrađena kroz PRIDE predviđa proširenja mogućnosti razvoja, modeliranja i analize sustava temeljenih na komponentama. Kao jedna od sastavnica budućeg razvojnog okruženja, u razvoju je i integrirano okruženje za razvoj i analizu utroška



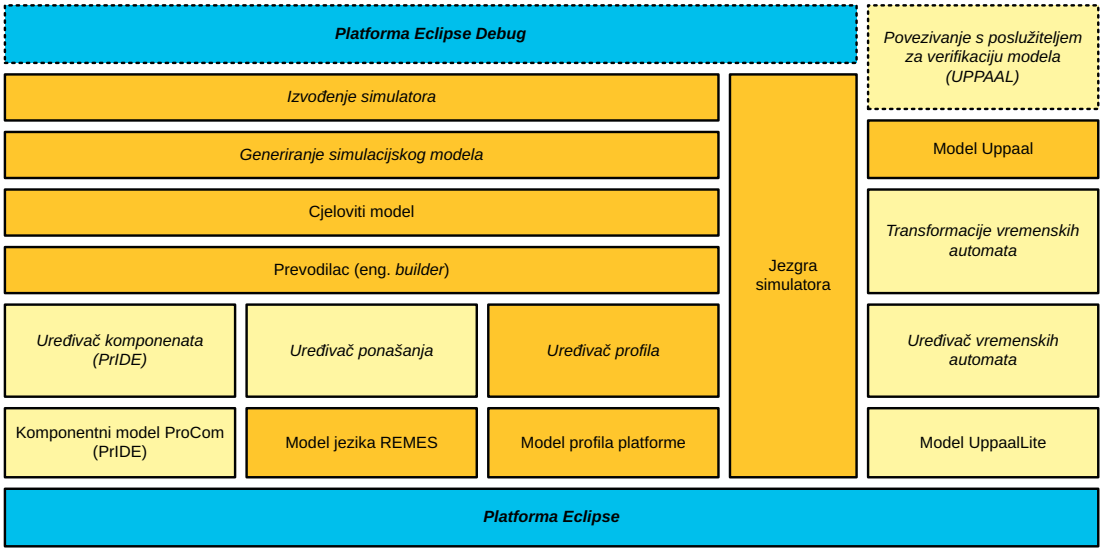
Slika 8.1: Uređivanje komponenata razina ProSave i ProSys u okruženju PRIDE.

resursa i ponašanja sustava opisanih u jeziku REMES [75]. Osnovna struktura skupa alata prikazana je na slici 8.2.

Razvojno okruženje temeljeno je na platformi Eclipse. Uporabom okvira za razvoj modela EMF [43] razvijeni su (meta)modeli za REMES i profil platforme te UppaalLite, pomoćni model za pretvorbu dijagrama jezika REMES u vremenske automate pogodne za analizu u alatima obitelji UPPAAL. Komponentni model ProCom preuzet je iz okruženja PRIDE. Za uređivanje modela ponašanja i profila platforme pripremljeni su odgovarajući uređivači.

Pripremljeni modeli prevode se u pozadini prevodiocem (eng. *builder* u terminologiji platforme Eclipse) koji nakon svake promjene datoteke s modelom strukture, ponašanja ili platforme, prevodi ta tri modela u cjeloviti model sustava. Generirani cjeloviti model dalje se može iskoristiti za generiranje simulacijskog modela, odnosno klasa programskog jezika Jave koje preslikavaju strukturu i ponašanje sustava u jezik Javu. Takav programski kôd podloga je za izvođenje simulatora.

Izvođenje simulatora nastoji se približiti korisnicima. Korisnici platforme Eclipse,



Slika 8.2: Pregled strukture platforme razvojnog okruženja.

navikli na moderna integrirana razvojna okruženja, upoznati su s konceptom alata za pronalaženje pogrešaka (eng. *debugger*). Alati za pronalaženje pogrešaka daju uvid u strukturu programa koji se izvodi i omogućavaju manipulaciju objekata programskog procesa kao što su procesi, niti izvođenja, okviri stoga i varijable, a moderne razvojne okoline prilagođene su za prikaz tih objekata.

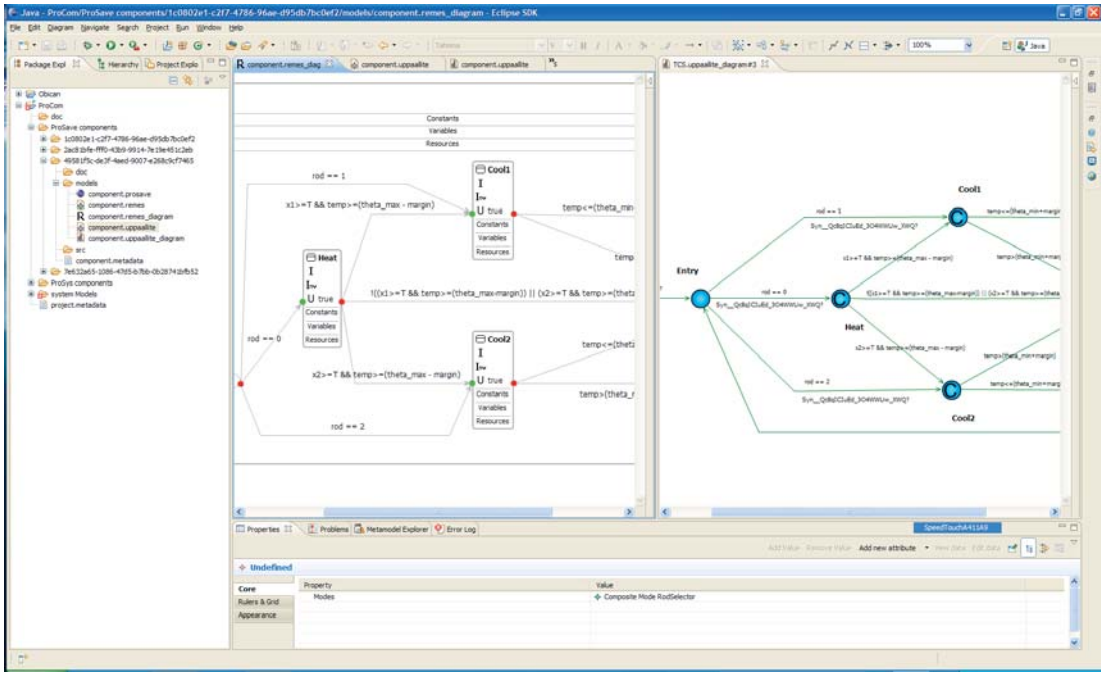
U sustavu kojeg modeliramo možemo razlikovati aktivne elemente (podsustave, komponente i moduse rada) koji dijele neke sličnosti sa spomenutim objektima strukture programa. Primjer odnosa te dvije klase objekata dan je u tablici 8.1.

U vrijeme simulacije, uporabom platforme Eclipse Debug [42], možemo manipulirati navedenim objektima na isti način kao što to činimo s objektima programskog procesa za vrijeme praćenja rada programa. Tako možemo zaustaviti izvođenje, mijenjati trenutno aktivni element, promatrati ili mijenjati trenutno stanje programa i slično. Na ovaj način koriste se metafore koje korisnik poznaje (nit, stog, trenutna naredba) za praćenje aktivnih elemenata modela (podsustav, komponenta, modus) i navigaciju kompleksnim i višeslojnim modelom sustava poznatim alatima, jednako kao što to čini s vlastitim programskim kôdom.

Tablica 8.1: Odnos klasičnih elemenata strukture programskog procesa i elemenata modela ProCom-REMES.

Objekt procesa	**Objekt modela**	**Komentar**
Proces	Sustav	Sadrži sve ostale objekte
Nit izvođenja	Podsustav (ProSys)	Osnova paralelnog izvođenja
Okvir stoga (metoda)	Komponenta (u hijerarhiji komponenata)	Jedinica (hijerarhijskog) izvođenja
Trenutna naredba	Aktivni modus	Najmanja jedinica izvođenja
Varijabla	Varijabla modusa	Varijable i resursi

Slika 8.3 prikazuje uređivače jezika REMES i vremenskih automata UppaalLite u okviru okruženja PRIDE.



Slika 8.3: Uređivači ponašanja i vremenskih automata u okruženju PRIDE.

Poglavlje 9

Rezultati

U ovom poglavlju usporedit ćemo rezultate dobivene simulacijom sustava opisanog komponentnim modelom ProCom i jezikom REMES s rezultatima dobivenim analitičkim metodama te rezultatima dobivenim alatima za formalnu analizu. Na primjeru raspoređivanja zrakoplova prilikom slijetanja prikazat ćemo gradnju modela ponašanja sustava u jeziku REMES i usporediti rezultate dobivene simulatorom ponašanja sa onim dobivenim analizom modela vremenskih automata. Primjer upravljanja temperaturom reaktora poslužit će za prikaz praćenja kretanja utroška različitih resursa u sustavu uporabom simulatora ponašanja.

9.1 Primjer: raspoređivanje slijetanja zrakoplova

Problem upravljanja slijetanjem zrakoplova (eng. *aircraft landing problem*) [1, 14] ilustracija je postupka raspoređivanja zadataka u sustavu, uz praćenje troška zadatka. Problem se sastoji u pronalaženju redoslijeda slijetanja zadanog skupa zrakoplova na ograničeni broj sletnih pista. Za svaki zrakoplov poznato je predviđeno vrijeme slijetanja (eng. *target landing time*) T koje odgovara najmanjem utrošku goriva. Zrakoplov može sletjeti ranije i kasnije, no ne ranije od najranijeg vremena slijetanja E (eng. *earliest landing time*) i ne kasnije od najkasnijeg vremena slijetanja L (eng. *latest landing*

time). Ranije slijetanje linearno povećava utrošak goriva prema koeficijentu e (eng. *early cost rate*). Kašnjenje linearno povećava utrošak goriva prema koeficijentu l (eng. *late cost rate*), a ograničene zalihe goriva dovoljne su za kontrolirano slijetanje najkasnije u trenutku L. Iz svega navedenog, jasno je da slijetanje zrakoplova ograničeno na interval $[E, L]$, te da je slijetanje u trenutku $T \in [E, L]$ povezano uz minimalan trošak.

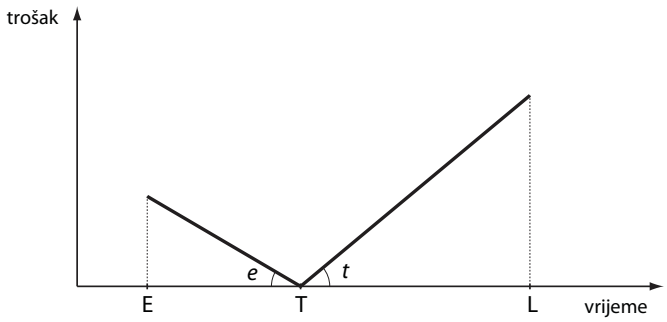
Ograničenje sletnih pista je potreba za vremenskim razdvajanjem zrakoplova kod slijetanja. Zbog turbulencija koje nastaju iza zrakoplova, nakon slijetanja zrakoplova potrebno je određeno vrijeme w (eng. *separation delay*) dok turbulencija ne nestane, prije nego se dozvoli slijetanje sljedećeg zrakoplova. Vremensko razdvajanje ovisi o kategoriji zrakoplova – veći zrakoplovi stvaraju više turbulencije, ali su i manje osjetljivi na turbulenciju od malih zrakoplova. Vremensko razdvajanje određuje se za svaki par kategorija zrakoplova.

Problem slijetanja zrakoplova zadan je sljedećim: poznato je ukupno n zrakoplova, svaki je zrakoplov predstavljen kao $A_i = (E_i, T_i, L_i, e_i, l_i, t_i, z_i), i \in [0, n]$, gdje su:

- E_i najranije vrijeme slijetanja,
- T_i predviđeno vrijeme slijetanja,
- L_i najkasnije vrijeme slijetanja,
- e_i koeficijent troška ranijeg slijetanja,
- l_i koeficijent troška kasnijeg slijetanja,
- t_i kategorija zrakoplova,
- z_i oznaka zrakoplova.

Vrijeme razdvajanja zrakoplova na istoj sletnoj pisti zadano je matricom $W = \left|w_{ij}\right|$, gdje je w_{ij} vrijeme razdvajanja zrakoplova kategorija t_i i t_j.

Uz linearnu promjenu troška u ovisnosti u vremenu slijetanja određenu koeficijentima e i l, utrošak goriva će se kretati prema slici 9.1. Utrošak goriva zrakoplova ovisi



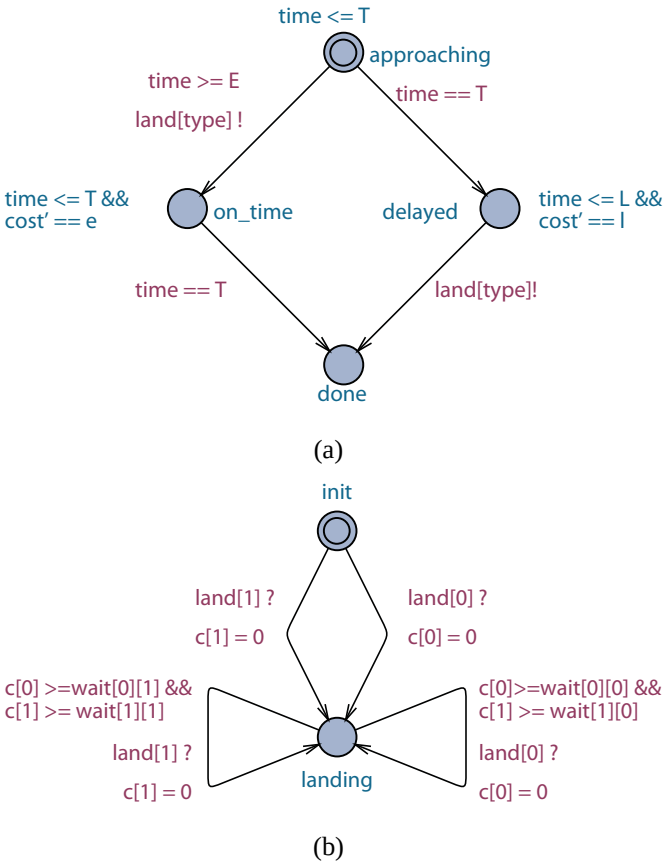
Slika 9.1: Kretanje utroška goriva zrakoplova iznad predviđenog.

o vremenu slijetanja – u intervalu od najranijeg mogućeg E, do najkasnijeg mogućeg L. Utrošak goriva za predviđeno vrijeme slijetanja T normiran je na nulu.

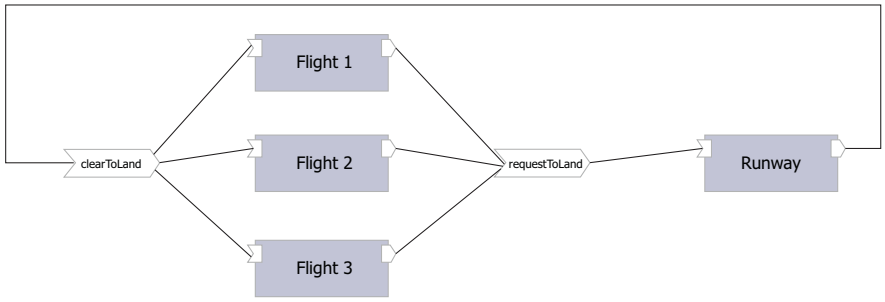
Rješenje problema slijetanja zrakoplova uporabom vremenskih automata s troškom opisano je u [20]. Vremenski automati koji opisuju ponašanje zrakoplova i sletne piste prikazani su slikom 9.2. Zrakoplov u dolasku predstavljen je lokacijom *approaching* iz koje može slijetanjem prijeći u lokaciju $on_t ime$. U slučaju ranijeg slijetanja, za lokaciju $on_t ime$ određena je promjena troška s koeficijentom e. Ranije slijetanje zrakoplova završava u trenutku T, kad model zrakoplova prelazi u lokaciju *done*. Iz početne lokacije *approaching* zrakoplov može po isteku vremena T prijeći i u lokaciju *delayed* koja predstavlja kašnjenje zrakoplova. Lokacija *delayed* ima pridijeljen koeficijent promjene troška l. Kasnije slijetanje zrakoplova moguće je sve do trenutka L, kad zrakoplov mora prijeći u lokaciju *done*.

Za sinkronizaciju zrakoplova sa sletnom pistom u trenutku slijetanja upotrijebljeni su sinkronizacijski kanali *land*[0] i *land*[1] kojima se zrakoplovi kategorija 0, odnosno 1 sinkroniziraju s pistom. U ovom primjeru definirane su samo dvije kategorije zrakoplova, no to ne umanjuje općenitost rješenja.

Sletna pista modelirana je s dvije lokacije. Iz početne lokacija *init* pista slijetanjem zrakoplova (sinkronizacijski kanali *land*[0], *land*[1]) prelazi u lokaciju *wait*. Slijetanje zrakoplova briše sat $c[0]$ ili $c[1]$ u ovisnosti o kategoriji zrakoplova. Prijelazom u lokaciju *wait* ne dopušta se slijetanje sljedećem zrakoplovu dok nije zadovoljen uvjet vremenskog razdvajanja određen konstantama u matrici *wait*.



Slika 9.2: Ponašanje zrakoplova (a) i sletne piste (b).

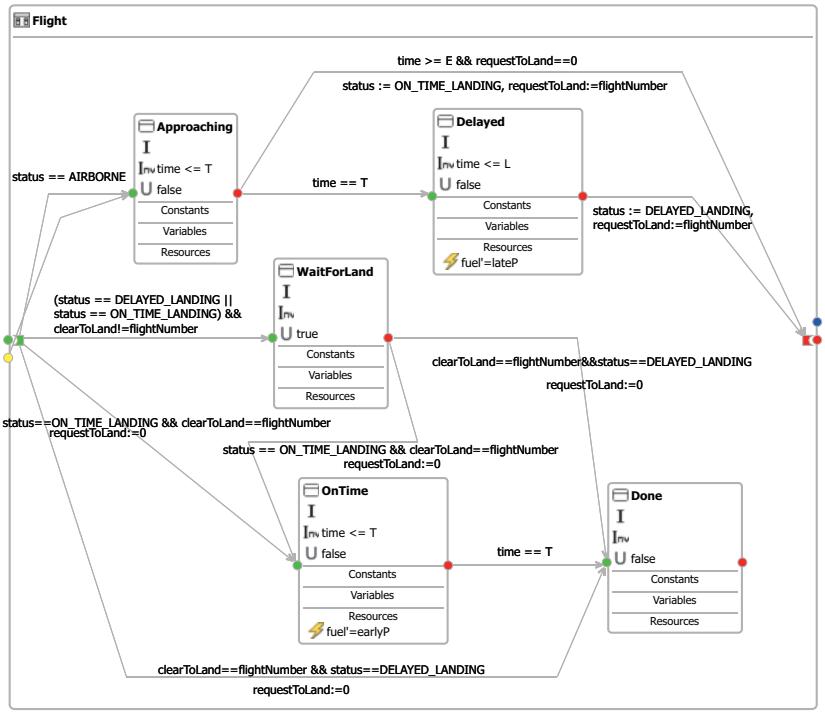


Slika 9.3: Sustav zrakoplova i sletne piste predstavljen komponentama.

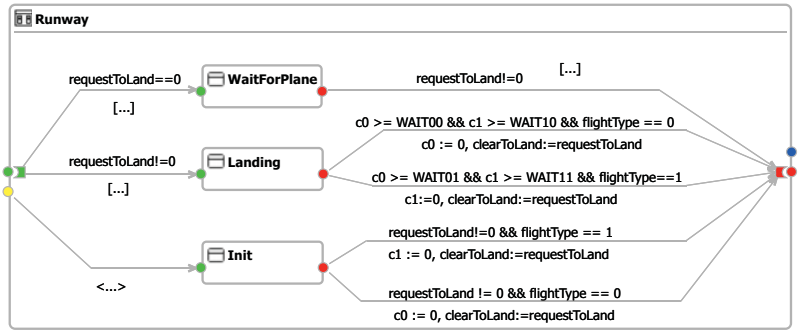
Model sustava izgrađen uporabom komponentnog modela ProCom prikazan je na slici 9.3. Kao primjer uzmimo tri zrakoplova predstavljena komponentama *Flight 1*, *Flight 2* i *Flight 3* razine ProSave. Umjesto sinkronizacijskih kanala uporabljenih za sinkronizaciju vremenskih automata, zrakoplovi komuniciraju sa pistom porukama. Zrakoplovi pisti šalju poruku zahtjeva za slijetanje *requestToLand* koja označava pisti da je zrakoplov pristigao i spreman je za slijetanje i sadrži oznaku zrakoplova z_i i tip zrakoplova t_i. Pista na zahtjev odgovara porukom *clearToLand* koja sadrži oznaku zrakoplova koji može sletjeti, dok ostali zrakoplovi ulaze u stanje čekanja. Nakon što je zrakoplov sletio, dojavit će to pisti porukom *requestToLand* bez oznake zrakoplova.

Ponašanje zrakoplova predstavljeno je složenim modusom *Flight* u jeziku REMES (slika 9.4). Modusi *Approaching*, *Delayed*, *OnTime*, *Done* odgovaraju istoimenim lokacijama iz vremenskog automata. Dodatni modus *WaitForLand* uveden je zbog potrebe čekanja na primitak poruke *clearToLand* (što odgovara sinkronizacijskom kanalu u modelu vremenskih automata). Prijelazi koji u modelu vremenskih automata sadrže sinkronizaciju kanalom *land* preneseni su u jezik REMES zamjenom sinkronizacijskih kanala porukama *requestToLand* i *clearToLand*. Poruke *requestToLand* i *clearToLand* preslikane su u istoimene varijable.

Ponašanje sletne piste opisano je ponašanjem *Runway* (slika 9.5). Modusi *Init* i *Landing* poznati su iz vremenskog automata. Za sinkronizaciju koristi se modus *WaitForPlane* koji čeka poruku zahtjeva za slijetanjem *requestToLand*. Prijelazi koji u vremenskom automatu sadrže sinkronizaciju preneseni su zamjenom sinkronizacijskih kanala porukama, na isti način kao u modelu ponašanja komponente *Flight*. Kons-



Slika 9.4: Opis ponašanja komponente *Flight*.



Slika 9.5: Opis ponašanja komponente *Runway*.

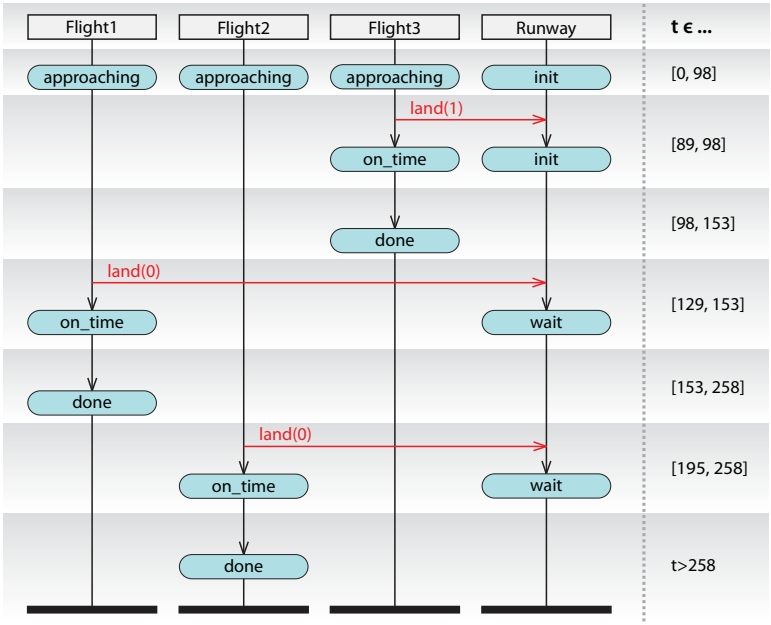
tante *WAIT00*, *WAIT01*, *WAIT10*, *WAIT11* odgovaraju elementima matrice vremena razdvajanja *W*.

Simulacijom sustava uporabom alata UPPAAL ili UPPAAL Cora može se pratiti slijed događaja u sustavu opisanom vremenskim automatima – korisnik može utjecati na odabir sljedećeg prijelaza i istovremeno pratiti vrijednosti satova i varijabli. Simulacija ponašanja opisanog jezikom REMES uporabom alata razvijenih u sklopu ovog rada nudi slične mogućnosti, te možemo usporediti rezultate dobivene različitim alatima.

Simulirani model sadrži tri inačice zrakoplova sa parametrima opisanim u tablici 9.1. Vrijeme razdvajanja *w* određeno je kao $w_{10} = 8$ u slučaju da zrakoplov kategorije 0 slijeće nakon zrakoplova kategorije 1, $w_{00} = 3$ u slučaju da zrakoplov kategorije 0 slijeće nakon zrakoplova kategorije 0, te $w_{11} = w_{01} = 15$ u ostalim slučajevima.

Tablica 9.1: Parametri modela slijetanja zrakoplova.

Komponenta	**E**	**T**	**L**	**rano slijetanje**	**e**	**l**	**tip**
Flight 1	129	153	559	[129, 153]	10	10	0
Flight 2	195	258	744	[195, 258]	10	10	0
Flight 3	89	98	510	[89, 98]	30	30	1



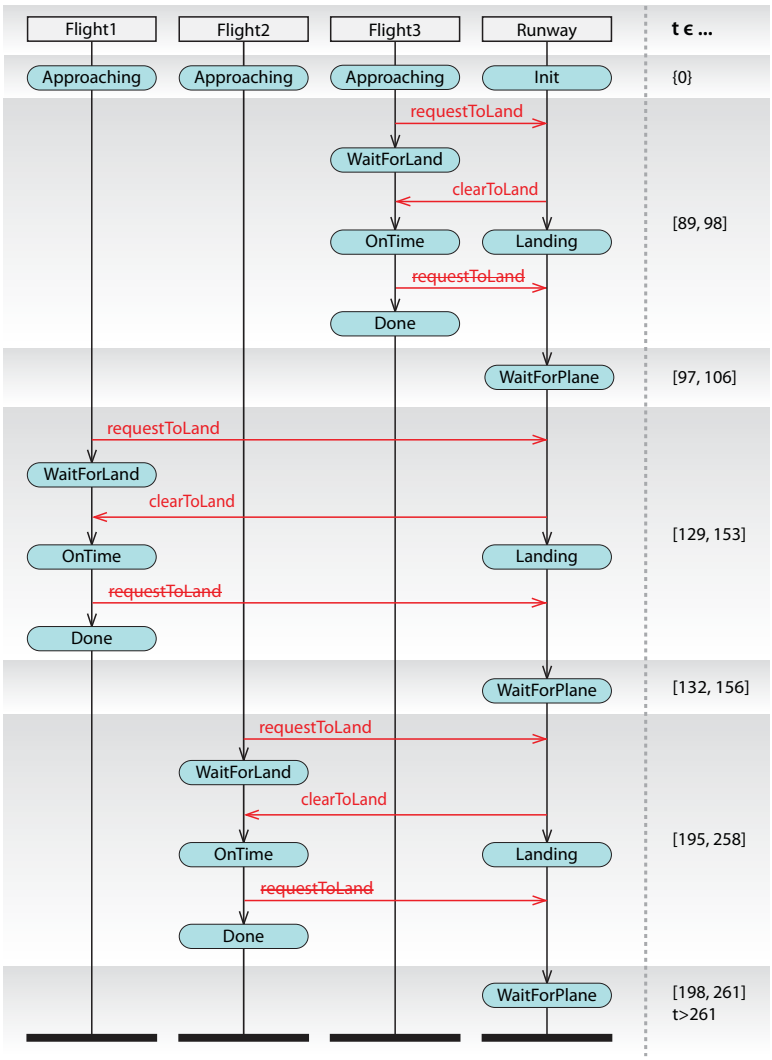
Slika 9.6: Slijed događaja u simulaciji alatom UPPAL (idealni slučaj).

Slika 9.6 prikazuje slijed događaja dobiven simulacijom modela u alatu UPPAAL Cora. Odabrani slijed događaja predstavlja idealan slučaj u kojem svi zrakoplovi slijeću u predviđenim vremenskim trenucima T, a ukupni trošak je nula.

Trenutak slijetanja (slanje poruke sinkronizacijskim kanalom *land*[*i*]) za sva tri zrakoplova pada u predviđeno vrijeme T na samom kraju vremenskog intervala ranog slijetanja, te niti jedan zrakoplov ne ulazi u lokaciju *delayed* i stanje kašnjenja.

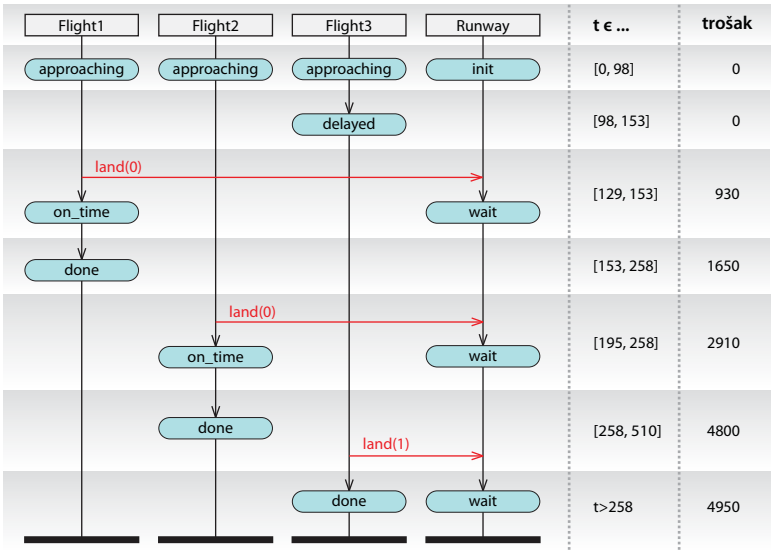
Slika 9.7 prikazuje isti slijed događaja, no dobiven simulacijom modela opisanog komponentama ProCom-a i modelom ponašanja REMES. Trenutak slijetanja zrakoplova također pada u vrijeme T, a slijed događaja je nešto složeniji zbog potrebe za komunikacijom zahtjeva za slijetanje. Za razliku od slijeda događaja dobivenog simulatorom alata UPPAAL, u slijedu je eksplicitno vidljivo čekanje na istek vremena razdvajanja zrakoplova na pisti, što je također posljedica različitog mehanizma sinkronizacije.

Na primjeru kašnjenja jednog leta prikazat ćemo izračun troška prilikom simulacije. Slika 9.8 prikazuje slijed događaja dobiven UPPAAL-ovim simulatorom u slučaju



Slika 9.7: Slijed događaja u simulaciji simulatorom ponašanja (idealni slučaj).

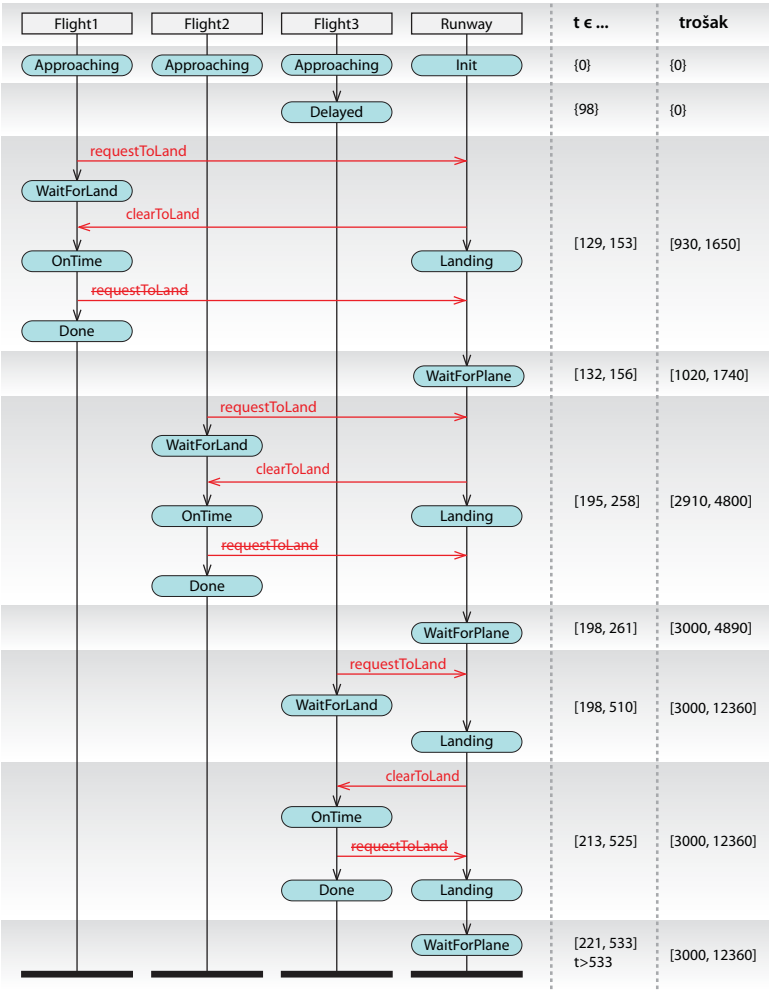
kašnjenja zrakoplova *Flight 3*. Zrakoplov *Flight 3* prelazi u stanje kašnjenja *Delayed* u trenutku $T_3 = 98$ i čeka slijedeću priliku za slijetanje. Sve dok je u stanju kašnjenja, obračunava se trošak određen parametrom $l_3 = 30$. UPPAAL trošak prikazuje kao cjelobrojnu vrijednost koja predstavlja minimalan trošak, tj. trošak koji odgovara donjoj granici vremenskog intervala trenutnog stanja.



Slika 9.8: Događaji i trošak u simulaciji alatom UPPAAL (jedan zrakoplov kasni).

Na slici 9.9 prikazan je slijed događaja dobiven simulatorom ponašanja. Za razliku od UPPAAL-a, simulator prikazuje trošak kao interval vrijednosti, a ne kao minimum te je tako moguće pratiti granice kretanja troška.

Prikazani rezultati pokazuju sukladnost događaja i praćenja utroška resursa u oba simulatora. Razlike prvenstveno proizlaze iz razlika ulaznih jezika. REMES ne sadrži mehanizme sinkronizacije slične sinkronizacijskim kanalima vremenskih automata jer predviđa uporabu komponentnog modela u te svrhe.



Slika 9.9: Događaji i trošak u simulatoru ponašanja (jedan zrakoplov kasni).

9.2 Primjer: upravljanje temperaturom

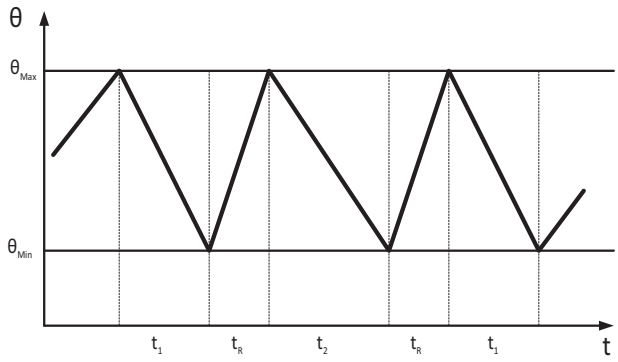
Primjer modela sustava za upravljanje temperaturom nuklearnog reaktora poznat je u literaturi, a analizirali su ga još Alur i dr. kao primjer hibridnog sustava [6]. Sustav za upravljanje temperaturom (eng. *temperature control system*, TCS) osigurava ispravan rad reaktora koji proizvodi toplinu, s temperaturom kao funkcijom vremena, $\theta(t)$.

Početno stanje sustava je temperatura θ_0 koja raste zadanom brzinom. Svaki put kad jezgra reaktora dosegne kritičnu temperaturu θ_{max}, treba je ohladiti uvođenjem jedne od dvije šipke za hlađenje. Šipka za hlađenje preuzima toplinu reaktora što se

manifestira povećanjem temperature šipke. Šipke su opisane varijablama x_1 i x_2 koje se također mijenjaju u vremenu i sadrže vrijeme koje je proteklo od zadnje upotrebe šipke.

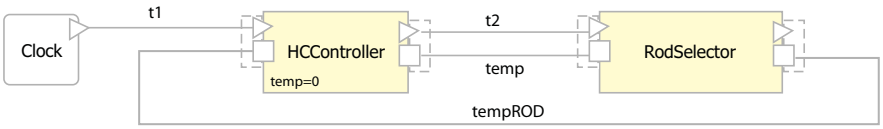
Hlađenje reaktora šipkama odvija se brzinom R_1 za prvu, odnosno R_2 za drugu šipku. Proces hlađenja prestaje kad temperatura reaktora padne ispod zadane minimalne temperature θ_{min}. Proces hlađenja prekida se izvlačenjem šipke. Šipka tada postaje nedostupna na unaprijed određeno vrijeme T, nakon čega se može ponovno koristiti.

Cilj analize ovog sustava je potvrditi zahtjev da je u trenutku kad reaktor postigne kritičnu temperaturu θ_{max} barem jedna šipka raspoloživa za hlađenje. Primjer promjene temperature reaktora dan je slikom 9.10 [84].



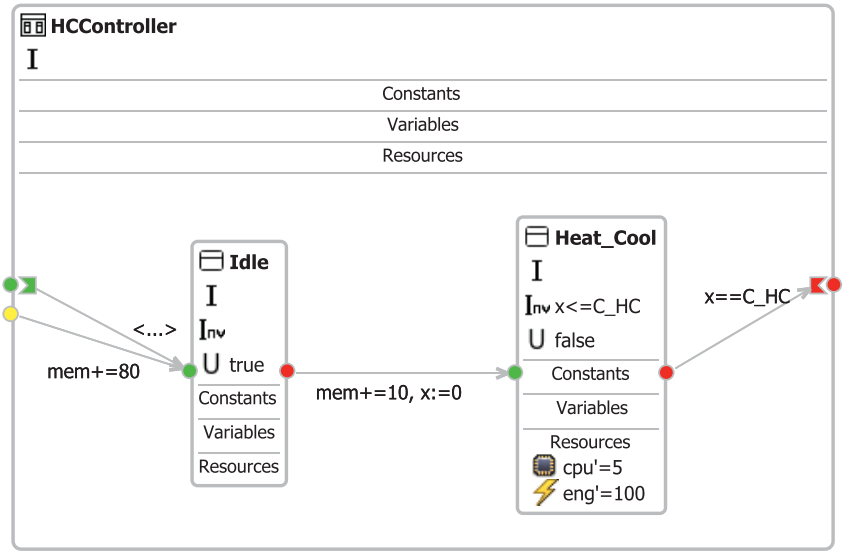
Slika 9.10: Promjena temperature u reaktoru.

Sustav upravljanja temperaturom modeliran komponentnim modelom ProCom prikazan je na slici 9.11 [85].



Slika 9.11: Sustav upravljanja temperaturom reaktora predstavljen komponentama.

Model sadrži dvije komponente, *HCController* i *RodSelector*, te sat *Clock*. Komponenta *HCController* mjeri temperaturu i prenosi podatke o temperaturi reaktora po-



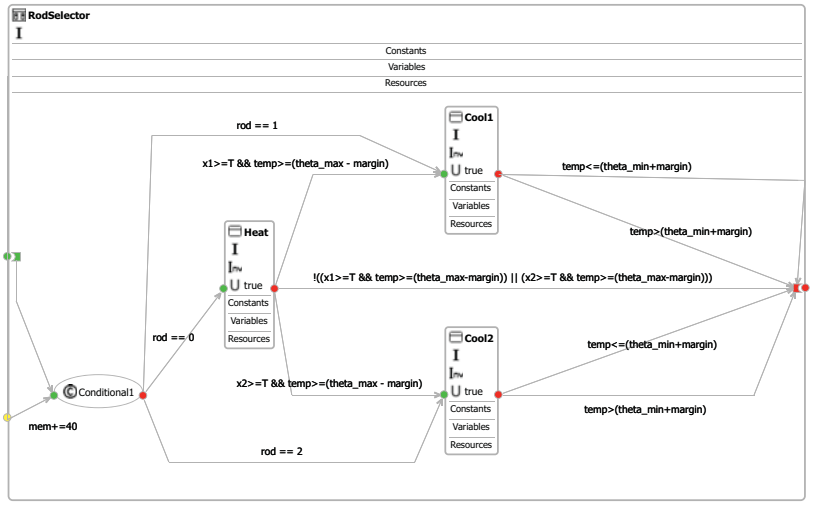
Slika 9.12: Opis ponašanja komponente *HCController*.

datkovnim vratima *temp*. Komponenta *Rod selector* koristi podatak o temperaturi za odluku treba li se reaktor nastaviti grijati ili će početi hlađenje reaktora uvođenjem šipke. Uporaba šipki podliježe ranije navedenim ograničenjima. Rezultat rada komponente *Rod selector* je podatak *tempROD* kojim je određena temperatura šipke za vrijeme hlađenja, a time i brzina hlađenja. Opisi ponašanja u jeziku REMES prikazani su slikama 9.12 i 9.13.

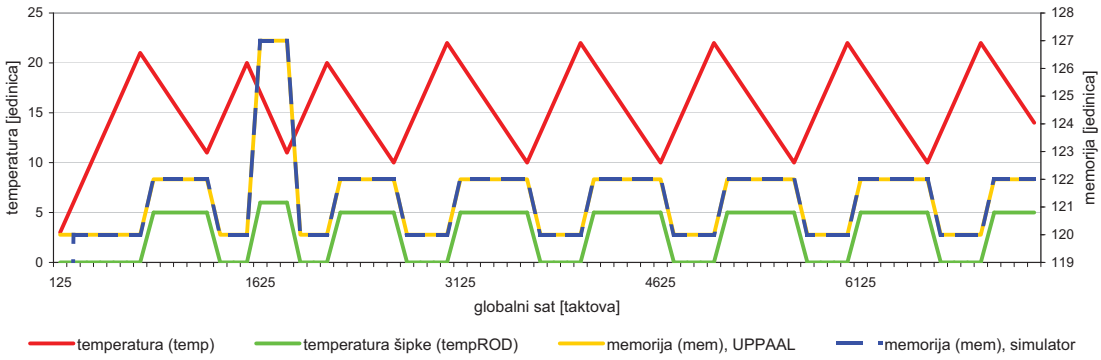
9.2.1 Rezultati analize

Analiza je izvedena usporedbom s alatima UPPAAL i UPPAAL Cora. Zbog nemogućnosti alata UPPAAL Cora da prati stanje više resursa odjednom, trošak je predstavljen težinskom sumom vrijednosti resursa. Oba simulatora pratila su isti put izvođenja prilikom odabira prijelaza. Rezultati dobiveni iz oba simulatora (trošak iz alata UPPAAL Cora i težinska suma preračunata iz podataka vlastitog simulatora) bili su jednaki.

Slika 9.14 prikazuje rezultate za resurs memorije dobivene simulatorom i alatom UPPAAL. UPPAAL ne podržava kontinuirane promjene varijabli, te su promjene re-



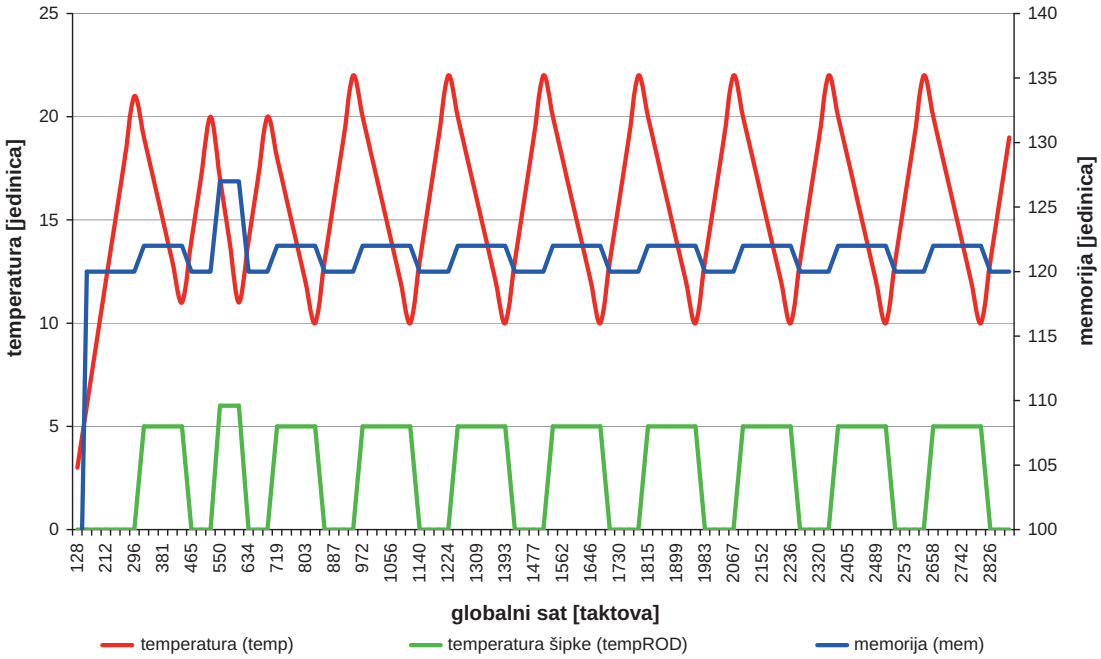
Slika 9.13: Opis ponašanja komponente *RodSelector*.



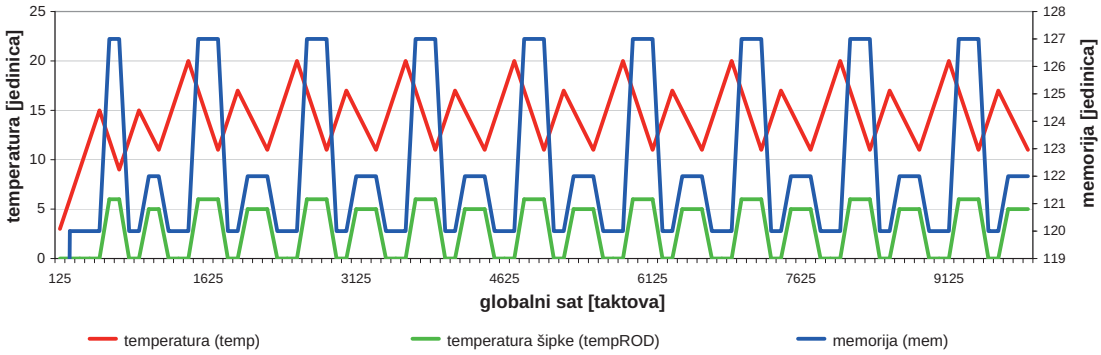
Slika 9.14: Usporedba rezultata izvođenja simulatora i alata UPPAAL.

sursa memorije izvedene na diskretnim prijelazima. Razlike koje se mogu primijetiti u dijagramu potiču od različitih strategija inicijalizacije komponenata – model komponente u vremenskom automatu inicijalizira sve komponente u sustavu odjednom (početnim prijelazom vremenskog automata za inicijalizaciju, slika 9.18(d)), dok simulator inicijalizira komponentu prilikom prvog pokretanja, u skladu sa semantikom jezika REMES.

Slika 9.17 prikazuje zastoj sustava zbog odabira parametara sustava koji ne omogućuju dovoljno brzo hlađenje šipki – brz rast temperature reaktora uz predugo vrijeme reakcije komponente *HCController*.



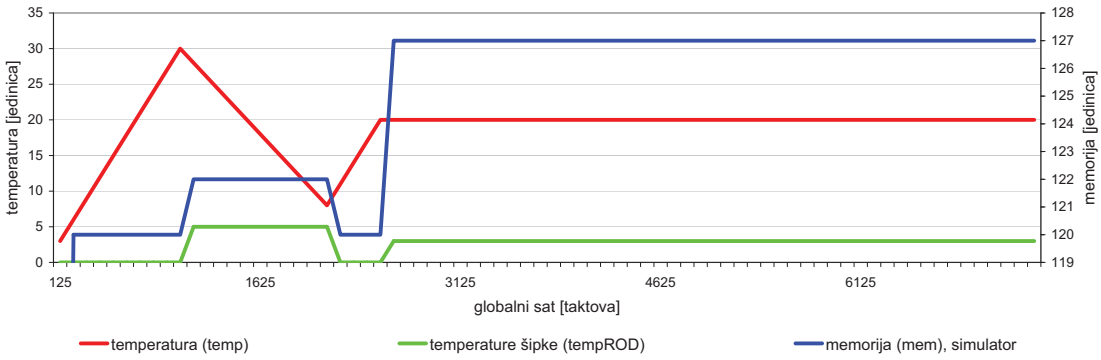
Slika 9.15: Prikaz kretanja parametara modela kroz vrijeme.



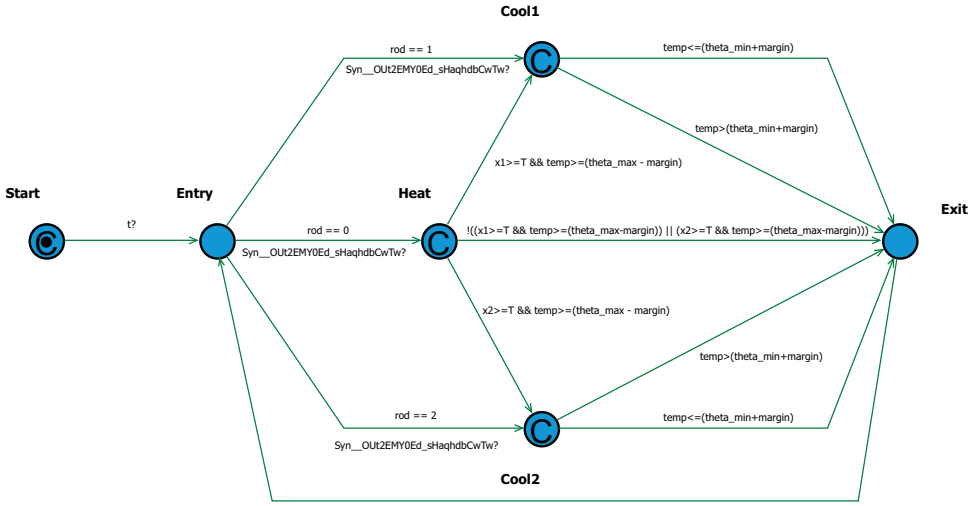
Slika 9.16: Prikaz kretanja parametara modela kroz vrijeme.

Graf kretanja parametara modela (memorija, temperatura reaktora i šipki) prikazan je na slici 9.15. Sustav ulazi u stacionarno stanje nakon druge izmjene šipki i nadalje se uvijek koristi ista šipka.

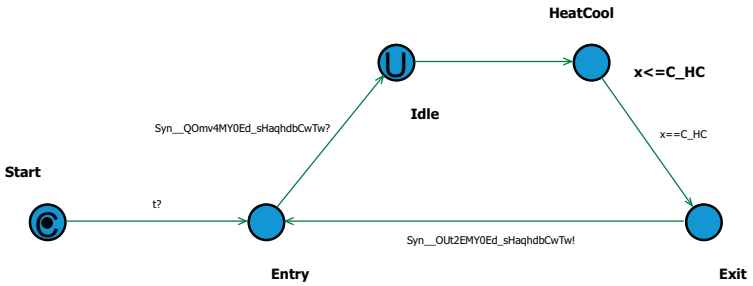
Rezultate promjene parametara modela, smanjene brzine hlađenja R_1 i R_2, vidimo na slici 9.16. Zbog nemogućnosti da se šipka ohladi prije nego što reaktor dosegne graničnu temperaturu θ_{max}, šipke se počinju izmjenjivati.



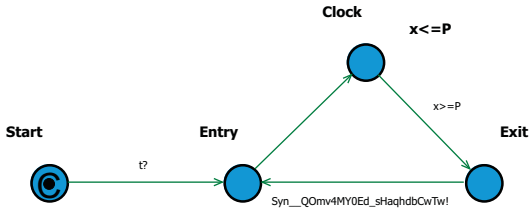
Slika 9.17: Rezultat izvođenja simulatora u slučaju parametara sustava koji dovode do zastoja.



(a)



(b)



(c)

(d)

Slika 9.18: Elementi sustava za upravljanje temperaturom pretvoreni u vremenski automat: (a) komponenta *Rod selector*, (b) komponenta *HC controller*, (c) sat, (d) dodatni automat za inicijalizaciju sustava, dodan automatski prilikom pretvorbe modela.

Poglavlje 10

Zaključak i komentar

Postupak predviđanja utroška resursa predložen u ovom radu ostvaren je kombinacijom modeliranja jezikom REMES i mjerenja ponašanja simuliranog modela sustava kroz automatizirane alate opisane u poglavlju 8. Simulacija je odabrana iz više razloga: prvo, simulacija se može usmjeriti na pojedine elemente sustava uz zanemarivanje ostalih, što je nužno u ranim fazama razvoja kad velik dio sustava nije definiran; drugo, simulacija omogućuje praćenje kretanja svakog resursa pojedinačno, što je u slučaju formalne verifikacije otežano zbog raznih dodatnih ograničenja kako bi se osiguralo pronalaženje rješenje. Trenutna implementacija alata za analizu vremenskih automata s troškom UPPAAL Cora je u ovom pogledu posebno ograničena – provjera modela, analiza i simulacija izvode se osvježavanjem jedne monotono rastuće varijable cijene.

Prikazani postupak predviđanja kretanja resursa sustava simulacijom pokazao se točnim u usporedbi s rezultatima dobivenim simulatorom alata UPPAAL. Uz potporu alata opisanih u poglavlju 8 praćenje kretanja utroška resursa ne zahtijeva velik dodatni trud kod razvijatelja programske potpore. Priroda simulacije, analiziranje samo jednog puta kroz prostor stanja u modelu čini ovaj postupak pogodnim za upotrebu umjesto klasičnih alata za pronalaženje pogrešaka (eng. *debugger*), kao i za vizualizaciju i analizu putova koji vode do nedozvoljenih stanja dobivenih metodama formalne analize. Slični alati za simulaciju izgrađeni su oko komponentnog modela Palladio [16, 83, 17].

Postupci predviđanja utroška resursa sustava temeljeni na jeziku REMES uključuju postupak pripreme cjelovitog modela sustava kombiniranjem strukturnog modela, modela ponašanja i profila platforme, postupak pretvaranja modela ponašanja iz jezika REMES ili cjelovitog modela sustava u vremenske automate s cijenom, te simulaciju sustava za analizu utroška resursa. Predloženi postupak zadržava informacije o arhitekturi sustava i u procesu simulacije poštuje hijerarhijske odnose modeliranog sustava. Razvojni alati razvijeni u sklopu okruženja PRIDE automatiziraju postupak pretvorbe modela iz jezika REMES u skup vremenskih automata, kao i pripremu podataka za simulaciju.

Primjena simulacijskog modela omogućila je neovisno praćenje pojedinih resursa sustava. Ogledni alati pokazuju mogućnosti primjene postupaka za analizu ponašanja sustava u ranoj fazi razvoja, kada su dostupni samo modeli sustava bez podataka o konačnoj izvedbi sustava. Usporedbom točnosti razvijenih postupaka s rezultatima koje daju alati obitelji UPPAAL pokazalo se da postupci simuliraju ponašanje sustava vjerno modelu. Oslanjanje na automatizaciju postupka pripreme modela za simulaciju ispunjava zahtjeve isplativosti i prilagodljivosti, jer se pripremne radnje odvijaju bez intervencije korisnika, što vrijedi i za slučaj promjene strukture sustava odabirom novih komponenata. Simulacijski model zadržava hijerarhijsku prirodu sustava, te je u slučaju proširenja jezika REMES hijerarhijskim modelom ponašanja moguće proširiti postupak da u potpunosti iskorištava svojstva hijerarhije. U opisu postupaka simulacije spomenuti su postupci modularne simulacije koji iskorištavaju hijerarhiju za smanjenje složenosti simulacije. Iako vezan uz cjeloviti model sustava izveden iz komponentnog modela ProCom i jezika za opis ponašanja REMES, postupak je načelno općenit i primjenjiv na sve sustave čije se ponašanje može opisati jezicima temeljenim na vremenskim automatima. Promjena baznog komponentnog modela zahtijevala bi promjenu postupka pripreme cjelovitog modela sustava, no sâm postupak se ne bi znatno promijenio.

Predloženi alati integrirani su s razvojnim okruženjem PRIDE za razvoj sustava temeljem komponentnog modela ProCom, iako mogu djelovati i kao samostalna razvojna okolina. Okruženje nudi alate za automatsku pripremu cjelovitog modela sus-

tava, uređivače raznih modela u uporabi i transformacije iz modela koji se koriste interno u modele prikladne za analizu u drugim alatima.

Doprinos ove disertacije je u sljedećem:

- predloženom modelu izvedbenog okruženja za sustave temeljene na programskim komponentama,
- cjelovitom modelu sustava u ranoj fazi razvoja,
- postupku generiranja cjelovitog modela sustava iz dostupnih modela
- postupku predviđanja utroška resursa kao unaprjeđenju postojećih postupaka primijenjenih na odabrani model ponašanja i
- izvedbi modela i postupaka u razvojnim alatima uz usporedbu s postojećim alatima.

Model izvedbenog okruženja sustava opisuje resurse koje sustav nudi i ograničenja njihovog korištenja. Iako jednostavan, modelom je moguće pripremiti profil platforme sa popisom resursa i tipičnih ograničenja njihovog korištenja – raspon dostupnih resursa (minimum i maksimum) te vremenske karakteristike njihove uporabe (promjena, srednja vrijednost). Cjeloviti model sustava ostvaruje veze između komponentnog modela koji opisuje građu sustava, modela ponašanja pojedinih komponenata, te profila platforme koji opisuje jedno od niza mogućih izvedbenih okruženja. Cilj cjelovitog modela je ostvariti homogenu cjelinu od raznorodnih modela kako bi se ostvarila podloga za simulaciju ponašanja sustava. Postupak generiranja cjelovitog modela automatiziran je i zajedno s simulatorom ponašanja sustava ugrađen u alat razvijen u sklopu okruženja PRIDE [81], ali se mogu koristiti i samostalno, kao REMES-IDE [82].

10.1 Budući pravci razvoja

Razvojno okruženje PRIDE za modeliranje sustava temeljenih na komponentnom modelu ProCom i jeziku za opis ponašanja REMES tek je u razvoju. Uporaba platforme

Eclipse osigurava dobru podlogu za buduće nadogradnje, veliku slobodu u razvoju alata ali i veliku složenost. Bogata potpora razvoju alata za jezike posebne namjene (eng. *domain-specific language*) [50] osigurava ugodno radno okruženje budućim korisnicima alata. Primjena ovdje opisanih postupaka u praksi uvelike ovisi upravo o kvaliteti dostupnih alata, budući da je ručna primjena postupaka praktički nemoguća. Budući razvoj alata kretat će se tako prema implementaciji svih elemenata prikazanih na slici 8.2, pogotovo u smjeru daljnje integracije s funkcionalnostima koje pruža platforma Eclipse Debug. Ugradnja simulatora u okviru platforme Eclipse Debug je djelomično ostvarena.

Drugi zanimljiv smjer razvoja je poboljšanje algoritma simulatora. Trenutno postavljena ograničenja rada s varijablama sata uvedena su zbog pojednostavljenja rada sa opisom invarijanti. Uklanjanje tih ograničenja uporabom matrica ograničenih razlika bitno ne mijenja algoritam simulatora, ali dodatno približava postupak alatima za formalnu analizu. Osim dinamičke analize ponašanja sustava izvođenjem u simulatoru, osnovni algoritam simulatora primjenjiv je i na proračun utroška resursa pojedine komponente. Takva analiza omogućila bi procjenu granica u kojima se kreće utrošak za komponentu usporedo s njenim dizajnom, pogotovo ako je analiza implementirana izravno u razvojno okruženje za dizajn sustava. Procjena utroška može dati okvire (minimum, maksimum) u kojima se kreću pojedini resursi opisani modelom ponašanja sustava. Zbog uporabe ulaznih i izlaznih varijabli u modusima jezika REMES rezultat takve analize trebao bi biti izražen simbolički.

Na poslijetku, opisani postupci predviđanja samo su jedan od niza elemenata koji trebaju ostvariti krajnji cilj predvidljivog ponašanja sustava od ranih faza razvoja pa sve do puštanja u pogon. Proces razvoja programske potpore, metode i alati koji će to omogućiti zanimljiv su smjer istraživanja.

Životopis

Marin Orlić rođen je 27. studenog 1978. godine u Rijeci, Republika Hrvatska, gdje je pohađao osnovnu i srednju školu, Gimnaziju Andrije Mohorovičića. Na Fakultetu elektrotehnike i računarstva Sveučilišta u Zagrebu 1996. godine upisao je studij računarstva te diplomirao 2001. godine.

Nakon završetka studija zapošljava se kao znanstveni novak na radnom mjestu asistenta na Zavodu za automatiku i računalno inženjerstvo Fakulteta elektrotehnike i računarstva Sveučilišta u Zagrebu. Sudjeluje u održavanju nastave na predmetima Osnove digitalnih računala, Arhitektura računala I, Otvoreno računarstvo, Računala i procesi, Odabrana poglavlja iz programskog inženjerstva, te predmetu Raspodijeljeni razvoj programske potpore (Distributed software development) koji se održava u suradnji sa Sveučilištem Mälardalen iz Švedske.

Sudjeluje na projektima Računala i procesi (2002.–2006.), Programsko inženjerstvo u sveprisutnom računarstvu (od 2006.), a također i na međunarodnim projektima Tempus KISEK i DICES. U sklopu projekta Tempus KISEK (Kolaborativna internacionalizacija programskog inženjerstva u Hrvatskoj) u suradnji sa Sveučilištem Mälardalen, Švedska, Sveučilištem Paderborn, Njemačka, Sveučilištem u Splitu i Sveučilištem J. J. Strossmayera u Osijeku sudjeluje u stvaranju Kompetencijskog centra za programsko inženjerstvo u otvorenim sustavima. U okviru projekta DICES surađuje u istraživanju raspodijeljenih ugradbenih sustava temeljenih na programskim komponentama.

Na Fakultetu elektrotehnike i računarstva upisuje poslijediplomski sveučilišni stu-

dij za stjecanje akademskog stupnja magistra znanosti, kojeg završava 2006. godine s radom na temu *Modeliranje interakcija u višeagentskim sustavima*. Od 2003 . godine pa do danas objavio je niz znanstvenih radova u području modeliranja, ugradbenih sustava, razvoja sustava temeljenih na komponentama i e-učenja. Istraživački interesi uključuju modeliranje strukture i ponašanja sustava te alate za analizu modela.

Biography

Marin Orlić was born on November 27^{th}, 1978 in Rijeka, Croatia, where he attended primary and secondary school, Gimnazija Andrije Mohorovičića. He continued his education at the University of Zagreb, Faculty of Electrical Engineering and Computing, where he graduated computing in 2001.

Since 2001, he has been working as a research assistant at the Department of Control and Computer Engineering, Faculty of Electrical Engineering and Computing, University of Zagreb. He is involved in educational activities on various courses such as Digital Computers, Computer Architecture I, Open Computing, Computers and Processes, Selected Topics in Software Engineering, and Distributed Software Development, an international course held jointly with Mälardalen University, Sweden.

He participated in scientific projects Computers and Processes (2002–2006), Software Engineering in Ubiquitous Computing (2006–present), and on international projects Tempus KISEK and DICES. As a part of Tempus KISEK (Collaborative Internationalization of Software Engineering in Croatia) project in cooperation with Mälardalen University, Sweden, University of Paderborn, Germany, University of Split, Croatia, and J. J. Strossmayer University of Osijek, Croatia, he was involved in establishing of Competence center for software engineering in open systems. Within the DICES project, he participated in research of distributed component-based embedded systems.

He has enrolled in the postgraduate university study at advanced Master of Science level on Faculty of electrical engineering and computing, and completed it in 2006 with a thesis on *Modeling interaction scenarios in multi-agent systems*. Since 2003 he

has published a number of papers on modeling, embedded systems, component-based development and e-learning. His research interests include structure and behavior modeling and tools for model analysis.

Bibliografija

[1] J. Abela, David Abramson, M. Krishnamoorthy i A. De Silva. Computing optimal schedules for landing aircraft. *Proceedings of the 12th National Conference of the Australian Society for Operations Research*, Adelaide, Australija, 1993.

[2] Mikael Åkerholm, Jan Carlson, John Håkansson, Hans Hansson, Mikael Nolin, Thomas Nolte i Paul Pettersson. The SaveCCM Language Reference Manual. Tehnički izvještaj, MRTC, Mälardalen University, Västerås, Sweden, 2007.

[3] Mikael Åkerholm, Johan Fredriksson, Hans Hansson, John Håkansson, Anders Möller, Paul Pettersson i Massimo Tivoli. The SAVE approach to component-based development of vehicular systems. *Journal of Systems and Software*, 80(5):655–667, svibanj 2007.

[4] Rajeev Alur, Costas Courcoubetis i David L. Dill. Model-Checking in Dense Real-Time. *Information and Computation*, 104(1):2–34, svibanj 1993.

[5] Rajeev Alur, Costas Courcoubetis, Thomas A. Henzinger i Pei-Hsin Ho. *Hybrid Automata: An Algorithmic Approach to the Specification and Verification of Hybrid Automata*, pp. 209–229. Computer Science. 1993.

[6] Rajeev Alur i David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[7] Rajeev Alur, Radu Grosu, Yerang Hur, Vijay Kumar i Insup Lee. Modular Specification of Hybrid Systems in Charon. *Hybrid Systems: Computation and Control, Third International Workshop*, LNCS 1790:6–19, 2000.

[8] Christo Angelov, Krzysztof Sierszecki i Nicolae Marian. *Component-Based Design of Embedded Software: an Analysis of Design Issues*, vol. 3409, pp. 1–11. Springer Berlin / Heidelberg, 2005.

[9] Christo Angelov, Krzysztof Sierszecki, Nicolae Marian i Jinpeng Ma. *A Formal Component Framework for Distributed Embedded Systems*, pp. 206–221. Springer Berlin / Heidelberg, 2006.

[10] L. B. Arief i N. A. Speirs. *A UML tool for an automatic generation of simulation programs*, vol. 21. ACM Press, New York, New York, USA, 2000.

[11] Simonetta Balsamo, Antinisca Di Marco, Paola Inverardi i Marta Simeoni. Model-based performance prediction in software development: a survey. *IEEE Transactions on Software Engineering*, 30(5):295–310, svibanj 2004.

[12] Simonetta Balsamo i Moreno Marzolla. A simulation-based approach to software performance modeling. *ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, pp. 363–366, New York, NY, USA, 2003. ACM.

[13] Luciano Baresi i Reiko Heckel. *Tutorial Introduction to Graph Transformation: A Software Engineering Perspective*, pp. 402–429. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2002.

[14] J. E. Beasley, M. Krishnamoorthy, Y. M. Sharaiha i David Abramson. Scheduling Aircraft Landings?The Static Case. *Transportation Science*, 34(2):180–197, May 2000.

[15] Steffen Becker, Lars Grunske, Raffaela Mirandola i Sven Overhage. *Performance Prediction of Component-Based Systems A Survey from an Engineering Perspective*, pp. 169–192. Springer Berlin / Heidelberg, 2006.

[16] Steffen Becker, Heiko Koziolek i Ralf Reussner. Model-based performance prediction with the palladio component model. *Proceedings of the 6th international workshop on Software and performance*, p. 65. ACM, 2007.

[17] Steffen Becker, Heiko Koziolek i Ralf Reussner. The Palladio component model for model-driven performance prediction. *Journal of Systems and Software*, 82(1):3–22, 2009.

[18] Gerd Behrmann, Alexandre David i Kim G Larsen. A Tutorial on Uppaal. *LNCS: Formal Methods for the Design of Real-Time Systems*, vol. 3185, pp. 200–236. Springer Berlin / Heidelberg, 2004.

[19] Gerd Behrmann, Ansgar Fehnker, Thomas Hune, Kim G. Larsen, Paul Pettersson i Judi Romijn. Efficient Guiding Towards Cost-Optimality in UPPAAL. T. Margaria i W. Yi, ur., *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, number 2031 u Lecture Notes in Computer Science, pp. 174–188. Springer–Verlag, 2001.

[20] Gerd Behrmann, Kim G. Larsen i Jacob I. Rasmussen. Optimal scheduling using priced timed automata. *ACM SIGMETRICS Performance Evaluation Review*, 32(4):34–40, ožujak 2005.

[21] Gerd Behrmann, Kim G. Larsen i Jacob I. Rasmussen. *Priced timed automata: Algorithms and applications*, pp. 162–182. Springer Berlin / Heidelberg, 2005.

[22] Johan Bengtsson. *Clocks, DBM, and States in Timed Systems*. Doktorat, Uppsala University, 2002.

[23] Johan Bengtsson i Wang Yi. Timed automata: Semantics, algorithms and tools. *Lectures on Concurrency and Petri Nets*, (316):87–124, 2003.

[24] Jean Bézivin. On the unification power of models. *Software and Systems Modeling*, 4(2):171–188, svibanj 2005.

[25] Jan Bosch i P.O. Bengtsson. Component Evolution in Product-Line Architectures. *Proceedings of International Workshop on Component Based Software Engineering*, pp. 92–97. Citeseer, 1999.

[26] Patricia Bouyer. Weighted timed automata: Model-checking and games. *Electronic Notes in Theoretical Computer Science*, 158:3–17, 2006.

[27] Patricia Bouyer, Thomas Brihaye, Véronique Bruyere i Jean-François Raskin. On the optimal reachability problem, 2006.

[28] Patricia Bouyer, Thomas Brihaye i Nicolas Markey. Improved undecidability results on weighted timed automata. *Information Processing Letters*, 98(5):188–194, 2006.

[29] Patricia Bouyer, Ed Brinksma i Kim G. Larsen. Optimal infinite scheduling for multi-priced timed automata. *Formal Methods in System Design*, 32(1):3–23, 2008.

[30] Patricia Bouyer, Kim G. Larsen i Nicolas Markey. Model-checking one-clock priced timed automata. *Foundations of Software Science and Computational Structures*, 4:108–122, 2008.

[31] Marius Bozga, Conrado Daws, Oded Maler, Alfredo Olivero, Stavros Tripakis i Sergio Yovine. *Kronos: A model-checking tool for real-time systems*, pp. 546–550. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 1998.

[32] Manfred Broy. Challenges in automotive software engineering. *Proceedings of the 28th international conference on Software engineering - ICSE '06*, p. 33, 2006.

[33] Tomáš Bureš, Jan Carlson, Ivica Crnković, Séverine Sentilles i Aneta Vulgarakis. ProCom - the Progress Component Model Reference Manual, 2008.

[34] Tomáš Bureš, Jan Carlson, Séverine Sentilles i Aneta Vulgarakis. A component model family for vehicular embedded systems. *Proceedings of the Third International Conference on Software Engineering Advances*, Sliema, Malta, 2008. IEEE.

[35] Jan Carlson, John Håkansson i Paul Pettersson. SaveCCM: An Analysable Component Model for Real-Time Systems. *Electronic Notes in Theoretical Computer Science*, 160:127–140, kolovoz 2006.

[36] Stefano Cattani i Marta Kwiatkowska. CSP + Clocks: a Process Algebra for Timed Automata. *AVOCS'03*, 2003.

[37] Shiping Chen, Ian Gorton, Anna Liu i Yan Liu. Performance prediction of COTS component-based enterprise applications. *Proceedings of 5th ICSE workshop on Component-Based Software Engineering (CBSE 2002)*, 2002.

[38] Ivica Crnković i Magnus Larsson. *Building reliable component-based software systems*. Artech House Publishers, 2002.

[39] Ali Dasdan i Rajesh K. Gupta. Faster maximum and minimum mean cycle algorithms for system-performance analysis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(10):889–899, 1998.

[40] Miguel de Miguel, Thomas Lambolais, Mehdi Hannouz, Stéphane Betgé-Brezetz i Sophie Piekarec. UML extensions for the specification and evaluation of latency constraints in architectural models. *Proceedings of the second international workshop on Software and performance - WOSP '00*, pp. 83–88, New York, New York, USA, 2000. ACM Press.

[41] Eclipse. `http://www.eclipse.org/` [2010-04-20].

[42] Eclipse. Eclipse Debug Project. `http://www.eclipse.org/eclipse/debug/` [2010-04-20].

[43] Eclipse. Eclipse Modeling Framework. `http://www.eclipse.org/modeling/emf/` [2010-04-20].

[44] Eclipse. Equinox. `http://www.eclipse.org/equinox/` [2010-04-20].

[45] S. Edwards, L. Lavagno, E.A. Lee i A. Sangiovanni-Vincentelli. Design of embedded systems: formal models, validation, and synthesis. *Proceedings of the IEEE*, 85(3):366–390, ožujak 1997.

[46] Alexander Egyed i Dave Wile. Statechart simulator for modeling architectural dynamics. *Proceedings Working IEEE/IFIP Conference on Software Architecture*, pp. 87–96, 2001.

[47] Ulrik Eklund i Carl Magnus Olsson. A case study of the Architecture Business Cycle for an in-vehicle software architecture. *Joint Working IEEE/IFIP Conference on Software Architecture 2009 & European Conference on Software Architecture WICSA/ECSA 2009*, pp. 91–100. IEEE, rujan 2009.

[48] Holger Giese, Sven Burmester, Wilhelm Schäfer i Oliver Oberschelp. Modular design and verification of component-based mechatronic systems with online-reconfiguration. *Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering - SIGSOFT '04/FSE-12*, p. 179, New York, New York, USA, 2004. ACM Press.

[49] Vincenzo Grassi i Raffaela Mirandola. Towards automatic compositional performance analysis of component-based systems. *ACM SIGSOFT Software Engineering Notes*, 29(1):59, siječanj 2004.

[50] Richard C. Gronback. *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*. Addison-Wesley, 2009.

[51] John Håkansson. *Design and Verification of Component Based Real-Time Systems*. Doktorat, Uppsala University, 2009.

[52] John Håkansson, Jan Carlson, Aurelien Monot i Paul Pettersson and. Component-based design and analysis of embedded systems with uppaal port. Sungdeok Cha, Jin-Young Choi, Moonzoo Kim, Insup Lee i Mahesh Viswanathan, ur., *6th International Symposium on Automated Technology for Verification and Analysis*, pp. 252–257. Springer Berlin / Heidelberg, October 2008.

[53] John Håkansson i Paul Pettersson. *Partial Order Reduction for Verification of Real-Time Components*, pp. 211–226. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2007.

[54] Kaj Hanninen, Jukka Maki-Turja, Mikael Nolin, Mats Lindberg, John Lundback i Kurt-Lennart Lundback. The Rubus component model for resource constrained real-time systems. *2008 International Symposium on Industrial Embedded Systems*, pp. 177–183, Le Grande Motte, lipanj 2008. IEEE.

[55] Bernd Hardung, Thorsten Kölzow i Andreas Krüger. Reuse of software in distributed embedded automotive systems. *International Conference On Embedded Software*, 2004.

[56] George T. Heineman i William T. Council, ur. *Component-Based Software Engineering: Putting the Pieces Together*. Addison-Wesley, Reading, MA, 2001.

[57] Franjo Ivančić. *Modeling and analysis of hybrid systems*. Doktorat, University of Pennsylvania, 2003.

[58] Richard M. Karp. A characterization of the minimum mean-cycle in a digraph. *Discrete Mathematics*, 23(3):309–311, 1978.

[59] Kim G. Larsen. Optimal reachability for multi-priced timed automata. *Theoretical Computer Science*, 390(2-3):197–213, 2008.

[60] Kim G. Larsen, Gerd Behrmann, Ed Brinksma, Ansgar Fehnker, Thomas Hune, Paul Pettersson i Judi Romijn. *Computer Aided Verification*, vol. 2102/2001 od *Lecture Notes in Computer Science*, pp. 493–505. Springer Berlin Heidelberg, Berlin, Heidelberg, srpanj 2001.

[61] Kim G. Larsen, Paul Pettersson i Wang Yi. UPPAAL in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, (1(1–2)):134–152, 1997.

[62] Kung-kiu Lau i Zheng Wang. A Taxonomy of Software Component Models. *31st EUROMICRO Conference on Software Engineering and Advanced Applications*, pp. 88–95. IEEE, 2005.

[63] Kung-kiu Lau i Zheng Wang. Software Component Models. *IEEE Transactions on Software Engineering*, 33(10):709–724, listopad 2007.

[64] Edward A. Lee. Absolutely positively on time: what would it take? *Computer*, 38(7):85–87, 2005.

[65] Philip Levis, Nelson Lee, Matt Welsh i David Culler. TOSSIM : Accurate and Scalable Simulation of Entire TinyOS Applications. *Proceedings of the 1st in-*

ternational conference on Embedded networked sensor systems, p. 137. ACM, 2003.

[66] Nicolae Marian i Yue Ma. Translation of Simulink Models to Component-based Software Models. *Proc. of the 8th International Workshop on Research and Education in Mechatronics REM'2007*, pp. 262–267, Talinn, Estonia, 2007.

[67] Moreno Marzolla i Simonetta Balsamo. UML-PSI: the UML performance simulator. *First International Conference on the Quantitative Evaluation of Systems, 2004. QEST 2004. Proceedings.*, pp. 340–341. IEEE, 2004.

[68] Microsoft. .NET Framework. `http://www.microsoft.com/net/` [2010-04-20].

[69] Gabrielo Moreno i Paulo Merson. Model-driven performance analysis. *Quality of Software Architectures. Models and Architectures*, 82(6):135–151, rujan 2003.

[70] Object Management Group. UML Profile for Schedulability, Performance, and Time Specification, 2002.

[71] Object Management Group. UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems, 2008.

[72] OMG. CORBA Component Model. `http://www.omg.org/technology/documents/formal/components.htm` [2010-04-20].

[73] Marin Orlić, Branko Mihaljević i Mario Žagar. Modelling Interaction Scenarios in Multi-Agent Systems. Vesna Lužar-Stiffler i Vesna Hljuz Dobrić, ur., *roceedings of the 28th International Conference on Information Technology Interfaces – ITI 2006*, pp. 373–378, Zagreb, 2006. SRCE.

[74] Marin Orlić, Igor Čavrak i Mario Žagar. Adapting Paintable Architecture Concepts to Wireless Sensor Networks. Vesna Lužar-Stiffler i Vesna Hljuz Dobrić, ur., *Proceedings of the 27th International Conference on Information Technology Interfaces – ITI2005*, pp. 567–572, Zagreb, 2005. SRCE.

[75] Marin Orlić, Aneta Vulgarakis i Mario Žagar. Towards Simulative Environment for Early Development of Component-Based Embedded Systems. *Fifteenth International Workshop on Component-Oriented Programming, WCOP2010*, Prague, Češka republika, 2010.

[76] OSGi Alliance. OSGi – The Dynamic Module System for Java. `http://www.osgi.org/` [2010-04-20].

[77] Wojciech Penczek i Agata Półrola. *Advances in verification of time petri nets and timed automata: a temporal logic approach*. Springer-Verlag New York Inc, 2006.

[78] Marie-Agnés Peraldi-Frati i Yves Sorel. From high-level modelling of time in MARTE to real-time scheduling analysis. *Proceedings of First International Workshop on Model Based Architecting and Construction of Embedded Systems ACES-MB 2008, with MoDELS'08 11th International Conference on Model Driven Engineering Languages and Systems*, pp. 129–144, Toulouse, France, 2008.

[79] Dorin B. Petriu i Murray Woodside. *A metamodel for generating performance models from UML designs*, pp. 41–53. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2004.

[80] Alexander Pretschner, Manfred Broy, Ingolf H Kruger i Thomas Stauner. Software Engineering for Automotive Systems: A Roadmap. *Future of Software Engineering (FOSE '07)*, pp. 55–71, Washington, DC, USA, svibanj 2007. IEEE.

[81] PROGRESS, DICES. PrIDE, PROGRESS Integrated Development Environment. `http://www.idt.mdh.se/pride/` [2010-04-20].

[82] PROGRESS, DICES. REMES-IDE, Behavior Modeling and Analysis of Embedded Systems. `http://www.fer.hr/dices/remes-ide` [2010-04-20].

[83] Ralf Reussner, Steffen Becker, Jens Happe, Heiko Koziolek, Klaus Krogmann i Michael Kuperberg. The Palladio component model. Tehnički izvještaj, Universität Karlsruhe - Fakultät für Informatik, Karlsruhe, 2007.

[84] Cristina Seceleanu. *A Methodology for Constructing Correct Reactive Systems*. Doktorat, Åbo Akademi University, Turku, Finland, 2005.

[85] Cristina Seceleanu, Aneta Vulgarakis i Paul Pettersson. REMES: A Resource Model for Embedded Systems. *2009 14th IEEE International Conference on Engineering of Complex Computer Systems*, pp. 84–94, lipanj 2009.

[86] Séverine Sentilles, Petr Štěpán, Jan Carlson i Ivica Crnković. Integration of Extra-Functional Properties in Component Models. Christine Hofmeister, Grace A. Lewis i Iman Poernomo, ur., *12th International Symposium on Component Based Software Engineering (CBSE 2009), LNCS 5582*. Springer Berlin, LNCS 5582, 2009.

[87] Sun Microsystems. Enterprise JavaBeans technology. `http://java.sun.com/products/ejb/` [2010-04-20].

[88] C. Szyperski. *Component Software - Beyond Object-Oriented Programming*. Addison-Wesley, Reading, MA, 1998.

[89] Sebastian Uchitel i Marsha Chechik. Merging partial behavioural models. *ACM SIGSOFT Software Engineering Notes*, 29(6):43, studeni 2004.

[90] Aida Čaušević i Aneta Vulgarakis. Towards a Unified Behavioral Model for Component-Based and Service-Oriented Systems. *2009 33rd Annual IEEE International Computer Software and Applications Conference*, pp. 497–503. IEEE Computer Society Press, srpanj 2009.

[91] Aneta Vulgarakis. Towards a resource-aware component model for embedded systems. *Doctoral Symposium of 33rd Annual IEEE International Computer Software and Applications Conference (COMPSAC 2009)*. IEEE Computer Society Press, July 2009.

[92] Aneta Vulgarakis, Jagadish Suryadevara, Jan Carlson, Cristina Seceleanu i Paul Pettersson. Formal Semantics of the ProCom Real-Time Component Model. *2009 35th Euromicro Conference on Software Engineering and Advanced Applications*, pp. 478–485, kolovoz 2009.

[93] Kurt Wallnau, Judith Stafford, Scott Hissam i Mark Klein. On the relationship of software architecture to software component technology. *Proceedings of the Sixth International Workshop on Component-Oriented Programming (WCOP'01)*, Budapest, Hungary, 2001. Citeseer.

[94] Wayne Wolf. What is embedded computing? *Computer*, 35(1):136–137, 2002.

[95] Wayne Wolf i Jan Madsen. Embedded systems education for the future. *Proceedings of the IEEE*, 88(1):23–30, siječanj 2000.

Prilozi

10.2 Gramatika tekstualnog prikaza jezika REMES

Kao alternativa grafičkom prikazu dijagrama jezika REMES s pripadnim grafičkim uređivačem (u trenutku pisanja ovog teksta bio je još u razvoju), razvijen je tekstualni uređivač jezika. Tekstovna sintaksa jezika REMES dana je sljedećom gramatikom:

Programski isječak 10.1: Gramatika konkretne tekstovne sintakse jezika REMES.

```
1  RemesDiagram::=
2    'remes' '{' (CompositeMode | SubMode)* '}'  ;
3
4  CompositeMode::=
5    'composite' name=ID '{'
6      ( Variable | Resource | Constant )*
7      ( SubMode | ConditionalConnector | InitPoint
8        | CompositeEntryPoint )* '}' ;
9
10 SubMode::=
11   'atomic' (isUrgent?='urgent')? name=ID '{'
12     ( Variable | Resource | Constant )*
13     ( 'invariant' '[' invariant ']' )?
14     ( ExitPoint )? '}' ;
15
16 ConditionalConnector::=
17   'conditional' name=ID '{'
18     ExitPoint '}' ;
```

```
19
20 InitPoint::=
21   'init' EnitEdge ;
22
23 ExitPoint::=
24   'edges' '{'
25     ( ExitEdge )* '}' ;
26
27 CompositeEntryPoint::=
28   'entry' ExitEdge ;
29
30 Edge::=
31   'edge' ( '(' actionGuard ')' )? ( '[' actionBody ']' )?
32     'to' connectTo=ID ;
33
34 InitEdge::=
35   'edge' ( '[' initialization ']' )? 'to' connectTo=ID ;
36
37 Variable::=
38   'var' ('global')? ( 'readable' )? ('writable')?
39   Type ('[' vectorSize ']')? name=ID ( '=' INT )? ;
40
41 Resource::=
42   'resource' Type name=ID ( ':' '(' expression ')' )? ;
43
44 Constant::=
45   'const' ('global')? Type name=ID ('=' INT)? ;
46
47 Type::=
48   'integer' | 'natural' | 'boolean' | 'clock' | 'float' ;
```

10.3 Gramatika tekstualnog opisa profila platforme

Programski isječak 10.2: Gramatika konkretne sintakse opisa profila platforme.

```
1  PlatformProfile::=
2    'profile' name=ID '{'
3    ( 'resources' '{'
4        Resource ( ',' Resource)*
5      '}' )?
6    ( 'constraints' '{'
7        Constraint ( ',' Constraint)*
8      '}' )?
9    ( 'defaults' '{'
10       ( Constant | Behavior ) ( ',' ( Constant | Behavior ))*
11     '}' )?
12   '}' ;
13
14 Resource::=
15   name=ID ':' ResourceType ;
16
17 Constraint::=
18   ConstraintType '(' Resource ('\'')? ')' ConstraintOp INT ;
19
20 Constant::=
21   'constant' name=FQN '=' INT;
22
23 Behavior::=
24   'behavior' 'for' select=OCLExpression '=' remesFile=URI
25   ':' templateModeName=ID ;
26
27 ResourceType::=
28   'cpu' | 'memory' | 'bandwidth' | 'power' | 'port' ;
29
30 ConstraintType::=
31   'min' | 'max' | 'avg' ;
32
33 ConstraintOp::=
34   '<' | '<=' | '==' | '>=' | '>' ;
```