UNIVERSITATEA POLITEHNICA TIMIŞOARA
FACULTATEA DE AUTOMATICĂ ŞI CALCULATOARE

TEZĂ DE DOCTORAT

Conducător ştiinţific:
Prof.dr.ing Vladimir Creţu

Doctorand:
Ing. Ioana Maria ŞORA

BIBLIOTECA CENTRALĂ
UNIVERSITATEA "POLITEHNICA"
TIMIŞOARA

2003

BUPT

Model compoziţional pentru sisteme cu arhitecturi multi-flux bazate pe componente compozabile

Teză de doctorat

ing. Ioana Maria Şora

Conducător ştiinţific: Prof.dr.ing. Vladimir Creţu

Departamentul Calculatoare
Universitatea "Politehnica" Timişoara

2003

UNIVERSITATEA "POLITEHNICA"
TIMIŞOARA
BIBLIOTECA CENTRALĂ
Nr. volum
Dulap

BUPT

Cuprins

BUPT

BUPT

BUPT

Capitolul 1

Introducere

O caracteristică a software-ului în ultimii ani este creşterea continuă a dimensiunilor şi complexităţii sale, ca urmare a expansiunii majore a folosirii software-ului în domenii foarte variate. Pentru a putea face faţă acestor provocări, dezvoltarea software a trebuit să găsească modalităţile prin care complexitatea să poată fi controlată şi timpul de răspuns la noi cerinţe impuse să fie cât mai scurt. În acest context, reutilizarea unor componente software existente pentru construcţia de noi sisteme implică beneficii esenţiale din punct de vedere economic şi tehnic.

Dezvoltarea software bazată pe componente ([Szy97], [BW98]) conţine două aspecte, cel tehnic şi cel comercial. Din perspectiva acestor două aspecte, componentele pot fi văzute ca abstracţii de bază ale unei noi metode de proiectare, respectiv ca unităţi comerciale (*off-the-shelf*) de construcţie şi implementare a unui sistem.

Dezvoltarea software bazată pe componente are ca prim avantaj creşterea reutilizării software-ului. În contextul componentelor, spre deosebire de programarea orientată pe obiecte, reutilizarea este în general de tip *blackbox*, refolosirea implementării bazându-se exclusiv pe interfaţa şi specificarea ei. Alte două argumente în favoarea tehnologiei bazate pe componente sunt extensibilitatea şi capacitatea de a evolua (*evolvability*). Un sistem este extensibil dacă i se pot adăuga noi facilităţi şi noile facilităţi sunt capabile să interacţioneze cu vechile facilitaţî. De exemplu, un sistem clasic procedural nu este extensibil: facilităţile noi le pot apela pe cele vechi, dar nu şi invers. Un sistem are capacitatea de a evolua daca facilităţile vechi pot fi înlocuite de altele noi, pentru a contribui la îmbunătăţirea sistemului, în termeni de calitate, funcţionalitate, facilităţi, etc.

Dezvoltarea de software bazat pe componente promovează activitatea de compoziţie (construirea de aplicaţii prin compunerea unor abstracţii de nivel înalt) în defavoarea activităţii de programare (înţeleasă ca utilizare de primitive ale unui limbaj de programare pentru definirea de noi obiecte). Nierstrasz, în [SN99], argumentează

BUPT

în favoarea unor limbaje de compoziţie, care să suporte compunerea de aplicaţii din componente specifice domeniului. Scopul principal al unui asemenea limbaj ar fi să acţioneze ca un liant (*glue*) pentru compunerea de componente existente, deja implementate, posibil în diferite limbaje de programare şi platforme.

Pe lângă faptul că oferă o modalitate mai rapidă de a construi noi sisteme prin compoziţie, componentele ridică un număr de probleme. In mod evident, compoziţia fără restricţii a componentelor nu este posibilă, este nevoie de modele compoziţionale în cadrul cărora se stabilesc următoarele aspecte: cum se descriu şi identifică componentele, ce componente pot fi compuse, cum se realizează integrarea şi conectarea lor. Aceste probleme şi altele (adaptarea componentelor, evoluţia lor) formează domeniul unei noi discipline de cercetare, Ingineria Software Bazată pe Componente (*Component Based Software Engineering - CBSE*) ([BW98], [Crn01], [CL02], [JSS03], [CSSW01], [CSSW02], [CSSW03]).

Această teză abordează câteva din aceste probleme ale ingineriei software bazate pe componente. Lucrarea propune un model compoziţional pentru sisteme cu arhitectură multi-flux. Modelul propus abordează problema compoziţiei pentru o întreagă clasă de sisteme, aparţinând unor domenii de aplicaţii diferite şi care sunt caracterizate de stilul arhitectural multi-flux. Acest model conţine o schemă originală de descriere a componentelor şi un formalism de descriere propriu derivat din această schemă. Schema de descriere stabileşte, într-un mod independent de un anumit formalism de descriere, categoriile de informaţii care trebuie cunoscute şi specificate în vederea compoziţiei. În lucrarea de faţă se defineşte şi un formalism de descriere adecvat schemei propuse, şi anume limbajul CCDL (Composable Components Description Language). Acesta are avantajul de a avea o complexitate mai redusă decât alte formalisme de descriere, iar schema de descriere pe care se bazează permite mai multă flexibilitate în specificare decât abordări similare cunoscute din literatură. Totodată modelul compoziţional propus este suficient de riguros pentru a putea fi utilizat pentru realizarea de unelte automatizate care să sprijine compunerea sistemelor. Modelul propus este validat prin uneltele realizate pe baza lui pentru compunerea interactivă şi compunerea automată. În domeniul compoziţiei automate, s-a definit o strategie de compoziţie automată, a cărei implementare este utilizată la realizarea de protocoale de comunicaţie în reţea auto-configurante. În ceea ce priveşte compunerea interactivă, s-a realizat un mediu de compoziţie vizuală cu verificai automate ale sistemului compus interactiv.

BUPT

Structura lucrării

Această lucrare este structurată după cum urmează:

Capitolul 2 face o scurtă prezentare a problemelor actuale de cercetare în domeniul ingineriei software bazate pe componente. La început se prezintă noţiunile de componentă software, compoziţie, contracte între componente, insistând asupra caracteristicilor lor din punct de vedere arhitectural, acesta fiind nivelul de abstractizare utilizat în această lucrare. În continuare, se prezintă selectiv aspecte din domeniu care sunt relevante în contextul acestei teze. Se urmăresc probleme şi soluţii descrise în literatură referitoare la automatizarea lucrului cu componente şi referitoare la realizarea de arhitecturi dinamice şi autoadaptive. Se acordă o atenţie deosebită metodelor prezentate în literatură pentru managementul variabilităţii în procesele care implică decizii automate de compoziţie. Capitolul se încheie cu o analiză a realizărilor existente şi a problemelor din acest domeniu, analiză care serveşte ca bază pentru definirea obiectivelor acestei teze.

Capitolele 3 şi 4 reprezintă contribuţia teoretică a acestei teze, definirea modelului compoziţional pentru sisteme cu arhitecturi multi-flux bazate pe componente compozabile.

Capitolul 3 defineşte un o schemă şi un formalism de descriere a componentelor compozabile în arhitecturi multi-flux. Capitolul începe prin definirea conceptelor de bază ale modelului, introducând conceptul de *componente compozabile* şi mecanismul *constrângerilor structurale* ca o nouă metodă de management a varabilităţii. Se prezintă definiţia şi implementarea formalismului dezvoltat pentru specificarea componentelor compozabile, şi anume *limbajul declarativ CCDL (Composable Component Description Language).*

Capitolul 4 defineşte o strategie de compoziţie bazată pe mecanismul propagării cerinţelor. Această strategie poate fi utilizată pentru compoziţia automată a unui sistem cu arhitectură multi-flux, pornind de la descrierea proprietăţilor dorite ale sistemului. Se propune şi un model de utilizare a compoziţiei automate pentru realizarea de sisteme auto-adaptive.

Capitolul 5 prezintă situaţii din două domenii de aplicaţii diferite în care a fost validat modelul compoziţional definit ca şi contribuţie teoretică a aceastei teze. Subcapitolul 5.1 descrie utilizarea strategiei de compunere automată pentru realizarea de protocoale de reţea autoconfigurante. Subcapitolul 5.2.2 prezintă o metodă de compoziţie vizuală interactivă pentru arhitecturi multi-flux şi o aplicaţie (mediu integrat de măsurare bazat pe instrumente virtuale) dezvoltată în acest stil. In final este prezentată o aplicaţie de utilizare a strategiei de compoziţie prezentate la verificarea

BUPT

corectitudinii circuitelor compuse din instrumente virtuale.

Capitolul 6 prezintă concluziile şi principalele contribuţii ale acestei teze.

Mulţumiri

Elaborarea tezei s-a făcut sub îndrumarea competentă a prof.dr.ing. Vladimir Creţu, căruia îi sunt recunoscătoare şi îi mulţumesc pentru sprijinul acordat şi încurajările pentru finalizarea acestei teze.

Pentru perioada anilor 2000-2002 a studiilor doctorale pe care le-am efectuat în cadrul laboratorului DistriNet de la Departamentul de Calculatoare, Universitatea Catolică din Leuven, Belgia, mulţumesc profesorilor Yolande Berbers şi Pierre Verbaeten.

Mulţumesc de asemenea colectivului Laboratorului de Testarea Asistată de Calculator a Maşinilor Electrice (D109) de la Universitatea Politehnica Timişoara pentru o îndelungată colaborare.

Timişoara, 1 Noiembrie 2003

BUPT

Capitolul 2

Ingineria programării bazată pe componente: probleme actuale de cercetare

În acest capitol se prezintă şi se analizează probleme şi soluţii descrise în literatura din domeniul ingineriei programării bazate pe componente. Se urmăresc în special aspectele cele mai relevante pentru încadrarea în context şi realizarea contribuţiilor acestei teze.

Se începe cu prezentarea definiţiilor noţiunii de componentă şi ale altor concepte fundamentale legate de dezvoltarea bazată pe componente (subcapitolul 2.1).

O componentă importantă a modelului compoziţional propus în această teză este schema şi formalismul de descriere a componentelor şi compoziţiilor. Pentru definirea acestora, trebuie pornit de la analiza metodelor existente de descriere a utilizării componentelor în compoziţii, de specificare a contractelor componentelor prin diferite metode (clauze requires-provides, descrieri arhitecturale)(subcapitolul 2.2).

O altă parte a modelului compoziţional propus în lucrarea de faţă o constiuie strategia de compoziţie adecvată pentru implementare în unelte de verificare sau generare automată în sisteme autoconfigurante. De aceea, se investighează aspecte necesare automatizării lucrului cu componente şi realizării de arhitecturi dinamice autoadaptive (subcapitolul 2.3). În aceste situaţii, trebuie rezolvate probleme tehnice similare celor abordate în această lucrare, cum sunt exprimarea contractelor între componente, identificarea dependenţelor între componente, managementul variabilităţii în deciziile de compoziţie. Managementul variabilităţii se referă la modalităţile de exprimare şi impunere a unor restricţii sau presupuneri iniţiale privind structura unei compoziţii viitoare. Subcapitolul 2.4 discută stadiul actual al soluţiilor cunoscute pentru aceste probleme, constituind punctul de plecare pentru soluţia proprie descrisă în capitolele următoare.

BUPT

2.1 Ce este o componentă

Noţiunea de componentă software este utilizată foarte frecvent şi în diferite contexte. Ceea ce se prezintă sub denumirea de componente îmbracă forme, funcţionalităţi şi caracteristici diferite, de aici rezultând un mare număr de definiţii.

Definirea noţiunii de componentă trebuie să ia în considerare atât aspecte metodologice, cât şi tehnologice. O aplicaţie dezvoltată pe bază de componente cuprinde două aspecte, cel computaţional (nivelul tehnologic) si cel compoziţional (nivelul metodologic). O aplicaţie poate fi văzută simultan, după cum se prezintă în figura 2.1 ([ND95]), ca şi:

1. o entitate computaţională ce produce anumite rezultate,

2. o construcţie făcută din componente interconectate

Definiţiile componentelor din aceste două puncte de vedere sunt date în paragrafele 2.1.1 şi 2.1.2.

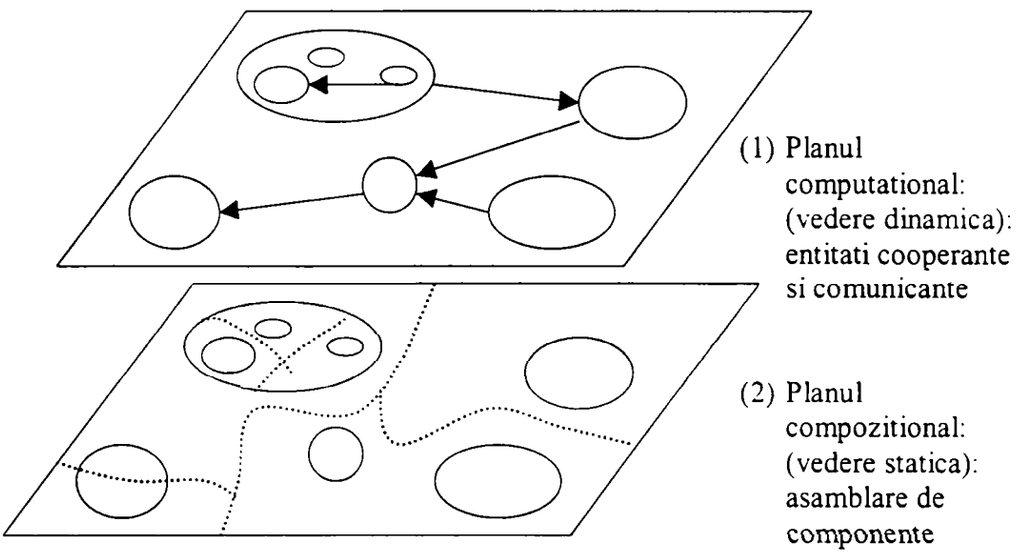


Figura 2.1: Aplicaţie văzută din punct de vedere computaţional respectiv compoziţional

În multe lucrări, definiţia noţiunii de componentă este puternic influenţată de tehnologia utilizată pentru realizarea componentelor şi de *nivelul de abstractizare* ales. În această lucrare se vor utiliza concepte legate de dezvoltarea bazată pe componente din punctul de vedere al unei abordări la nivelul de abstractizare corespunzător arhitecturii software, prezentate în paragraful 2.1.3.

BUPT

2.1.1 Componente la nivel metodologic

La nivel metodologic, o componentă este definită de faptul că este o entitate proiectată pentru compoziţie, adică proiectată să fie utilizată împreună cu alte componente ([Szy99], [Mey00b], [Dou00], [ND95]). În [Szy99], Clemens Szypersky defineşte o componentă în felul următor:

> O componentă software este o unitate de compoziţie care are interfeţe specificate prin contracte şi dependenţe de context explicite. O componentă poate fi utilizată în mod independent în compoziţii de către terţe părţi.

Această definiţie stabileşte următoarele proprietăţi caracteristice pentru o componentă:

- Este o *unitate* de compoziţie cu interfaţă şi dependenţe explicite. Fiind o unitate de utilizare/instanţiere, nu este posibil ca o componentă să fie utilizată "parţial". O componentă se utilizează întotdeauna ca un întreg.
- Este o unitate de utilizare *independentă* (*independent deployment*). Pentru ca o componentă să poată fi utilizată în mod independent, e necesar să fie bine separată de mediul înconjurător şi de alte componente. O componentă îşi încapsulează caracteristicile constituente.
- Orice componentă trebuie să fie autonomă şi completă (*self-contained*) pentru a permite ca o terţă parte să o utilizeze în compoziţii. O terţă parte nu poate fi încărcată cu responsabilitatea de a cunoaşte detalii constructive ale tuturor componentelor cu care interacţionează. Pentru aceasta, o componentă trebuie să aibă specificaţii clare a ceea ce furnizează şi a condiţiilor pe care le cere în interfeţele sale contractuale.

În [Mey00b], Bertrand Meyer defineşte un set de criterii pentru ca un element software să fie considerat componentă:

1. O componentă trebuie să fie utilizabilă de către alte elemente software (clienţi). Acest criteriu exclude un program în sensul tradiţional, care e destinat pentru a fi utilizat de către un operator uman, independent de alte evenimente software. Un program tradiţional destinat utilizării directe de către un utilizator uman nu constituie deci o componentă software.
2. O componentă trebuie să poată fi utilizată de clienţi – terţe părti – fără intervenţia autorului său. O componentă trebuie să prezinte interes pentru un

BUPT

spectru mai larg de clienţi independenţi de autorul original. Aceasta exclude cazul rutinelor, claselor şi altor elemente software utilizate de diferite părţi ale aceluiaşi software.

3. O componentă trebuie să includă specificaţii ale tuturor dependenţelor (platforma hardware şi software, versiuni, alte componente). Fără a avea la dispoziţie asemenea specificaţii, noii clienţi ar fi obligaţi sa contacteze autorul componentei, contrar criteriului 2.

4. O componentă trebuie să includă specificaţii ale functionalităţilor pe care le oferă, din aceleaşi motive.

5. O componentă trebuie să poată fi utilizată doar pe baza specificaţiilor sale. fără să fie nevoie de accesarea informaţiilor care nu sunt prezente în interfaţă. Chiar dacă codul sursă ar fi disponibil, utilizarea componentei trebuie să se facă doar pe baza specificaţiilor prezente în interfaţa publică.

6. O componentă trebuie sa fie compozabilă cu alte componente. În practică, aceasta se realizează prin faptul că o componentă e de obicei parte a unui *component framework*, care impune o arhitectură, un stil şi anumite tipare de utilizare după cum se va discuta în paragraful 2.1.3.

Nierstrasz [ND95] defineşte o componentă ca fiind o

abstracţie statică cu puncte de conectare (*plugs*).

Noţiunile utilizate în definiţia lui Nierstrasz au următoarele semnificaţii: Prin *static* se înţelege faptul că o componentă software este o entitate cu o durată de viaţa lungă ce poate fi stocată într-o bază de date, independent de aplicaţiile în care a fost şi de cele în care va fi utilizată. Prin *abstracţie* se înţelege faptul că o componentă impune o delimitare (mai mult sau mai puţin opacă) în jurul software-ului pe care îl încapsulează. Precizarea *cu puncte de conectare* se referă la faptul că există modalităţi bine definite de interacţiune şi comunicare cu componenta (parametri, porturi, mesaje). Punctele de interacţiune definesc intrări (servicii de care componenta depinde) şi ieşiri (servicii pe care componenta le exportă). Acestea sunt echivalente contractelor şi dependenţelor din formularea lui Szyperski. Văzută din exterior, o componentă apare ca o singură entitate ce poate fi mutată sau instanţiată într-un anumit context, unde e legată la alte componente prin intermediul punctelor de conectare.

BUPT

Se poate concluziona că definiţiile întâlnite în literatură sunt în esenţa lor similare, indiferent de terminologia sau modelul utilizat. Elementele de componentă ca şi entitate independentă de compoziţie, puncte de conectare, compatibilitate de conectare, dependenţe, se regăsesc în esenţa lor în toate variantele de definiţie. Denumirile utilizate, natura punctelor de conectare, mecanismele de legare şi regulile de compatibilitate pentru conectarea componentelor pot să difere de la o abordare la alta. Acestea sunt influenţate şi de aspectele tehnologice (paragraful 2.1.2) şi de nivelul de abstractizare ales pentru prezentare (paragraful 2.1.3). Unele detalii specifice modului de definire a componentelor în această lucrare vor fi adăugate în subcapitolul 3.1.1.

2.1.2 Componente la nivel tehnologic

La nivel tehnologic, ideea de dezvoltare software de componente este destul de veche, fiind prezentă încă din primele încercari de structurare şi modularizare a programării [McI69]. Ulterior ideea a găsit un sprijin în mecanismele furnizate de limbajele orientate pe obiecte, dar nu se identifică cu acestea.

Există diferenţe nete între conceptele de obiect şi de componentă. Componentele sunt abstracţii utilizate pentru a construi sisteme. O aplicaţie este, din punct de vedere static, o compoziţie de componente, iar din punct de vedere dinamic (la execuţie) este un sistem de obiecte care cooperează şi comunică. Deşi este uzual să se întâmple aşa, componentele nu trebuie neapărat să fie implementate cu ajutorul claselor. O componentă poate fi implementată şi procedural. Chiar dacă o clasă este compilată într-un format binar. în general nu este considerată automat componentă. O clasă nu este în general explicită asupra serviciilor pe care le oferă şi a condiţiilor pe care le cere. Dacă serviciile sunt totuşi specificate prin interfeţele furnizate, cerinţele în general nu sunt specificate deloc în cazul claselor. In plus, clasele sunt construite în general utilizând moştenirea implementării, făcând utilizarea separată a clasei de bază şi a subclaselor practic imposibilă, deoarece cele două nu pot evolua separat.

De multe ori o componentă conţine mai multe clase, între care există un grad ridicat de cuplare. O componentă construită prin utilizarea mai multor clase se va manifesta la execuţie prin obiecte care sunt instanţe ale acestor clase.

Problema tehnologică dacă o componentă trebuie sau nu sa fie o entitate sub formă binară a fost controversată. Szyperski defineşte componentele în mod explicit ca şi entitaţi binare ([Szy99] [Szy00]). Meyer ([Mey99]) combate necesitatea ca o componentă sa fie o formă binară, fiind de părere că într-adevar, forma binară favorizează "ascunderea" informaţiilor inutile şi forţează încapsularea, condiţii definitorii pentru

BUPT

o componentă (discutate în paragraful anterior), dar nu constituie în sine o condiţie.

Dezvoltarea de software pe bază de componente necesită un anumit suport tehnologic. Câteva din problemele care apar sunt:

- Care sunt paradigmele şi mecanismele pentru conectarea componentelor (mecanismele pentru compozitie) ?
- Care e structura unei componente ?
- Cum caracterizăm procesul de compoziţie din punct de vedere al momentului la care intervine decizia de compoziţie ?
- Cum modelăm componente şi compoziţii şi cum se poate verifica faptul că entităţile independente sunt compuse corect ?

Aceste întrebări constituie domenii active de studiu şi cercetare, având multiple răspunsuri posibile şi fiind încă în curs de explorare ([SC003], [Nie95]). În cadrul modelului compoziţional propus în capitolul 3 vor fi prezentate soluţii propri la unele din aceste probleme.

2.1.3 Componente arhitecturale

Arhitectura unui sistem software este legată de noţiunea de componente. În acest paragraf se vor prezenta componentele şi sistemele bazate pe componente din punctul de vedere al arhitecturii software.

În paragrafele 2.1.1 şi 2.1.2 s-au prezentat definiţii generale ale noţiunii de componentă din punct de vedere metodologic şi tehnologic. Noţiunea de componentă poate fi utilizată la diverse niveluri de abstractizare, diferiţi autori şi diferite colective de cercetare utilizând diferite moduri de prezentare a noţiunii de componentă în funcţie de aceasta. De asemenea, modurile de prezentare a noţiunii de componentă sunt deseori influenţate de aspectele tehnologice de realizare a componentelor. În această teză se va utiliza în continuare noţiunea de componentă descrisă în termeni corespunzători *nivelului arhitecturii software*.

Arhitectura software

Definiţia frecvent utilizată în prezent pentru arhitectura software, dată în [LCK99], este următoarea:

> Arhitectura software a unui program sau sistem software este structura sistemului, care cuprinde componentele software, proprietăţile vizibile din exterior ale acestor componente şi relaţiile între ele.

BUPT

Arhitectura sistemului este descrisă ca o colecţie de *componente* şi *conectori*. *Componentele* sunt elementele computaţionale şi de date ale sistemului. *Conectorii* realizează interacţiunile dintre componente şi reprezintă un liant (*glue*) pentru realizarea sistemului. Conectorii pot fi realizaţi prin diverse mecanisme, cum ar fi apel de procedură, *pipes*, evenimente etc.

Un sistem se poate descrie arhitectural în forma unui graf în care nodurile reprezintă componente, iar arcele reprezintă conectori. În general, sistemele pot fi ierarhice: componentele şi conectorii pot fi subsisteme care au propriile lor arhitecturi interne.

În decursul ultimei decade, cercetarea în domeniul arhitecturilor software s-a afirmat prin studii sistematice asupra structurii sistemelor, a relaţiilor între subsisteme şi componente. Disciplina a evoluat de la descrieri calitative ale organizării sistemelor, la dezvoltarea de notaţii, tehnici şi unelte de analiză sistematică ([PW92], [Sha01], [Gar01]).

Un *stil arhitectural* este dat de un set de caracteristici esenţiale care definesc un şablon al unei arhitecturi. Un stil arhitectural reprezintă o familie de sisteme înrudite, este o abstractizare a unei mulţimi de arhitecturi.

Garlan şi Shaw dau următoarele definiţii pentru stilul arhitectural al unui sistem software: Un stil arhitectural defineşte un vocabular de elemente de proiectare (componente şi conectori), împreună cu regulile pentru compunerea acestora ([Gar01]). Stilul arhitectural al unui sistem defineşte următoarele aspecte ([SC97]): ce tipuri de componente şi conectori pot fi utilizate; modul în care fluxul de control este transferat între componente (topologia fluxului de control, sincronicitatea, momentul legării componentelor); modul de comunicare a datelor (topologia fluxurilor de date, continuitatea fluxurilor de date, modul de transmitere a datelor).

Utilizarea componentelor trebuie făcută întotdeauna având în vedere aspectele arhitecturale, pentru a evita situaţii de tip *architectural mismatch* (nepotrivire arhitecturală) [GAO95]. La încercarea de a asambla împreună componente existente, pot să apară probleme datorate următoarelor cauze identificate în [GAO95]:

- *Natura componentelor.* Se materializează prin presupuneri eronate în ceea ce priveşte: infrastructura componentelor, modelul de control sau modelul de date al componentelor.

- *Natura conectorilor.* Presupuneri eronate asupra protocoalelor de comunicaţie sau a modelului de date care sunt comunicate.

- *Structura arhitecturală globală.* Presupuneri eronate asupra topologiei sistemului, prezenţei sau absenţei anumitor tipuri de componente sau conectori.

BUPT

Arhitecturile trebuie să fie compoziţionale, să asigure faptul că o proprietate stabilită la nivel de componentă este păstrată şi în cazul participării componentei într-o compoziţie de componente.

În concluzie, compoziţia componentelor trebuie tratată numai în contextul unei arhitecturi ([Ham02], [SB02]), pentru că aceasta permite specificarea şi verificarea respectării unor proprietăţi globale ale sistemului, cum sunt stilurile de interacţiune şi constrângerile referitoare la interacţiunea componentelor. Pentru fiecare componentă, pe lângă specificarea funcţionalităţii serviciilor sale şi a necesarului de resurse, este nevoie să se specifice şi stilul arhitectural utilizat sau presupus pentru interacţiunea componentei cu mediul.

Componente arhitecturale

În acest paragraf, se vor rezuma definiţiile principalelor concepte legate de dezvoltarea software bazată pe componente văzute la un nivel de abstractizare corespunzător nivelului arhitecturii software, aşa cum sunt prezentate acestea în lucrări de referinţă ca [BBB+00] şi [CL02] şi cum vor fi utilizate în modelul compoziţional propus în această teză.

Dezvoltarea software bazată pe componente are ca scop construirea de sisteme prin asamblarea unor piese independente (componentele). Pentru a realiza aceasta, este util un cadru (*framework*) care furnizează contextul în care aceste piese pot fi utilizate. În general, un *framework* poate fi văzut ca un proiect generic reutilizabil, sau ca şi un skeleton al unei aplicaţii, pe baza căruia se instanţiază aplicaţii particulare.

Conceptele dezvoltării bazate pe componente şi relaţiile dintre aceste noţiuni sunt prezentate în figura 2.2.

Un cadru de componente (*component framework*) însumează serviciile care oferă suportul necesar unui model de componente la momentul compoziţiei şi al execuţiei. *Component framework* are un rol similar cu o placă de cablaje în care pot fi inserate componentele.

Un *model de componente* defineşte ce este o componentă prin prisma a cum arată interfeţele componentei şi ce trebuie să conţină acestea. Modelul defineşte regulile de interacţiune între componente. Modelul de componente mai defineşte servicii auxiliare (*lookup*, *security*). Exemple de modele de componente sunt: EJB, COM, COM+.

Conceptele de model de componente şi cadru de componente (*component framework*) sunt foarte strâns legate, dar denotă totuşi lucruri diferite: *un model de componente defineşte standardele şi convenţiile utilizate de proiectantul de componente, în timp ce component framework constituie infrastructura ce sprijină în timpul execuţiei modelul de componente.*

BUPT

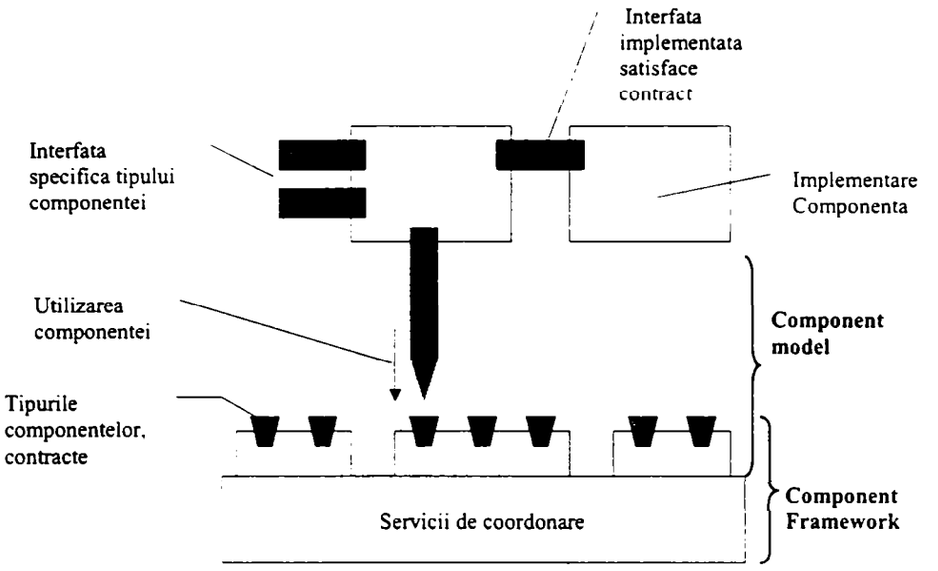


Figura 2.2: Conceptele dezvoltării bazate pe componente şi relaţiile între ele

O *interfaţă* a unei componente este o descriere a punctelor de interacţiune ale componentei. Interfaţa nu depinde de implementarea componentei. Aceasta face posibilă înlocuirea implementării unei componente fără a-i schimba interfaţa şi facilitează adăugarea de noi interfeţe şi implementări fără a le modifica pe cele existente. Interfeţele sunt de două tipuri, interfeţe de tip *export*, care descriu serviciile furnizate de componentă mediului şi interfeţe de tip *import* care specifică serviciile cerute de către componentă.

Interpretarea componentelor ca şi componente arhitecturale unifică două perspective asupra acestora: componente ca şi *implementări* şi componente ca şi *abstracţii arhitecturale*. Conceptul de componentă ca şi implementare se referă la majoritatea produselor de tip COTS (*components off-the-shelf*), care pot fi utilizate şi asamblate în sisteme. Aceste COTS implementează funcţionalitatea componentei şi realizează coordonarea interacţiunii componentei cu mediul într-un mod unic acelui produs. O componentă arhitecturală trebuie să respecte anumite reguli de proiectare (însumate în modelul de componente) care impun un standard pentru coordonare.

În acest context, [BBB+00] dă următoarea definiţie a unei componente arhitecturale:

> O componentă este o implementare opacă a unei funcţionalităţi care este capabilă să participe în compoziţii cu terţe părţi *în concordanţă cu un model de componente.*

BUPT

Ultimul criteriu, al concordanţei cu un model de componente, este cel care diferenţiază componentele arhitecturale de produse convenţionale de tip COTS. Un model de componente prescrie modul de interacţiune al componentelor între ele, exprimă constrângeri globale, arhitecturale, de proiectare. Spre deosebire de sistemele bazate pe COTS, care rezultă într-o integrare ad-hoc după scheme de interacţiune specifice produsului, sistemele bazate pe componente sunt realizate prin scheme de coordonare uniforme, standardizate de către modelul de componente utilizat. Respectarea standardelor impuse de un model de componente asigură o compoziţie uniformă, reducând riscurile de nepotriviri între componente care să le faca necompozabile.

2.2 Componente în compoziţii

2.2.1 Contracte

Pentru a permite o utilizare corectă a componentelor în compoziţii, e nevoie de specificaţii bune ale părtilor care se compun. Ori de câte ori componentele sunt asamblate, există un contract între acestea asupra termenilor colaborării. Pentru a putea verifica corectitudinea unei compoziţii, contractele trebuie să fie explicite. Aceste probleme de verificare a contractelor între componente sunt asemănătoare celor tratate de către sistemele de tipuri ale limbajelor de programare. Totuşi, sistemele de tipuri convenţionale au o expresivitate limitată, care nu cuprinde toate aspectele necesare unui contract între componente. O modalitate de a rezolva această problemă este de a îmbogăţi interfeţele componentelor cu constrângeri adiţionale exprimând cerinţele şi proprietăţile fiecărui partener (de ex, limbajul Eiffel [Mey88]). O parte dintre constrângeri vor fi verificate de către sistemul de tipuri, altă parte va fi verificată la execuţie, de fiecare dată când o colaborare între două componente are loc.

In literatură ([Mey00a], [LLH02], [BBB+00]) se face distincţie între mai multe niveluri de contractare între componente. O posibilă definire a acestor niveluri este:

1. Contracte sintactice
2. Contracte semantice
3. Contracte de performanţă şi de calitate a serviciului.

Contracte sintactice

Contractele sintactice se referă la posibilitatea de verificare a tipurilor în codul client. Acest lucru există în limbajele de programare tipizate. În limbajele de descriere a interfeţelor (*Interface Description Languages - IDL*) din majoritatea *component*

BUPT

framework-urilor (cum sunt de exemplu COM si CORBA cu IDL-urile lor) există specificări de tipuri pentru argumentele operaţiilor. Acest gen de contracte se referă la proprietăţile funcţionale ale componentei exprimate în termeni de sevicii pe care aceasta le exportă şi semnăturile acestor servicii (tipul şi ordinea parametrilor şi rezultatului).

Specificaţia unei componente ar trebui să includă şi specificaţia semantică. De exemplu, considerând specificaţia IDL a unei operaţii a unei componente responsabile pentru rezervarea de locuri de avion:

```
Boolean ReserveFlight (in Seats places)
                raises(InvalidSeat,SeatAlreadyBooked);
```

Specificaţia sintactică spune că respectiva componentă oferă o operaţie numită `ReserveFlight` care aşteaptă ca argument o listă de locuri care trebuie rezervate. Această specificaţie IDL se limitează însa la specificaţii structurale, fără a preciza nimic despre semantica operaţiei, în termeni comportamentali ai descrierii a ceea ce face aceasta.

Contracte semantice

Contractele semantice se referă la descrierea comportamentului operaţiilor unei componente.

O modalitate de a le exprima este cu ajutorul unor clauze contractuale constituind pre– şi postcondiţii ([Pah03]). Limbaje de programare cum este Eiffel ([Mey88]) permit îmbogăţirea codului cu asemenea forme de specificare comportamentală. O specificare semantică completă ar trebui să includă:

- *Invariantul* clasei, descriind constrângerile globale. Pentru exemplul componentei de rezervare a locurilor de avion, invariantul clasei ar putea fi propoziţia "numărul total al locurilor rezervate este întotdeauna pozitiv şi nu mai mare decât numărul fizic de locuri plus marja de overbooking"

- *Precondiţia* operaţiei, descriind constrângerile de apelare a operaţiei. De exemplu, "aplicabil doar pentru zboruri care nu au decolat încă"

- *Postcondiţia* operaţiei, descriind rezultatele aşteptate. De exemplu "dacă rezervarea s-a terminat cu succes, toate locurile cerute sunt rezervate"

Pentru specificarea comportamentului componentelor software mai pot fi utilizate şi metode formale definite înainte de apariţia programării orientate pe obiecte, precum VDM, Z, Larch ([FoC97]), dar utilizarea lor în contextul componentelor este redusă.

BUPT

Pentru descrierea comportării unei componente se pot utiliza automate finite ([SPR01], [Sch01]). Modelul CHAM ([Wer98], [IW95]) descrie o arhitectură ca şi o masină abstractă dupa modelul reacţiilor chimice. Un CHAM este specificat prin definirea de molecule (componentele), a soluţiilor lor şi a regulilor de transformare care descriu cum evolueaza soluţiile, date de modul de interacţiune a componentelor.

Contracte de performanţă şi calitate a serviciului

Contractele de performanţă şi calitate a serviciului descriu diverse atribute non-funcţionale, de performanţă şi calitate. De exemplu: pentru transmisii multimedia specificaţiile de performanţă mentionează parametri ca jitter, cerinţe de lăţime de bandă, etc, calculate pe baza calităţii dorite. Alte exemple de atribute de calitate sunt: conformitatea cu anumite modele de securitate sau standarde de testabilitate.

Specificarea atributelor de calitate este o problemă de cercetare mai nouă şi pentru care există mai puţine rezultate decât în domeniul specificării semantice. De exemplu, notaţia NoFun prezentată în [Fra98] permite definirea atributelor non-funcţionale ale software-ului (cum sunt eficienţa din punct de vedere al timpului şi memoriei, *reusability*, *maintainability*, *reliability*, *usability*).

Contracte pentru componente arhitecturale

O abordare care combate creşterea formalismului pentru creşterea acurateţii specificării unei componente îi aparţine lui Shaw ([Sha96]). Această abordare constă în propunerea de *credentials* pentru specificarea proprietăţilor interfeţelor. Abordarea porneşte de la premisa că nu orice proprietate poate fi demonstrată (sau demonstrată cu un efort rezonabil de calcul), şi deci utilizarea respectivei proprietăţi trebuie să se bazeze pe încredere, cunoscându-se gradul de credibilitate al evaluatorului proprietăţii. Un *credential* este un triplet (atribut, valoare, credibilitate), în care atributul se referă la numele unei proprietăţi a componentei, valoarea este o măsură a acelei proprietăţi şi credibilitatea este dată de modul în care s-a obţinut măsura respectivei proprietăţi. O asemenea abordare este foarte importantă în contextul în care este necesar să se facă anumite compromisuri în raportul cost/încredere în ceea ce priveşte certificarea anumitor proprietăţi greu sau imposibil de demonstrat prin metode mai formale.

Referitor la ce trebuie specificat despre componentele arhitecturale, Shaw defineşte în [Sha96] ce trebuie să fie o specificaţie arhitecturală de componentă. Shaw rezumă criteriile cerute de teoria clasică pentru specificaţii de componente. Conform accepţiunii clasice, o specificaţie de componentă trebuie să fie:

BUPT

- completă - specificaţia componentei spune tot ceea ce un utilizator ar trebui să ştie pentru a putea utiliza componenta
- statică - specificaţia este redactata o dată după care se consideră îngheţată
- omogenă - specificaţia e scrisă într-o singură notaţie

În cazul componentelor arhitecturale, Shaw arată însă că aceste criterii clasice nu pot fi satisfăcute. Descrierea unei componente arhitecturale trebuie să conţină mai mult decât descrierea funcţionalităţii computaţionale. Descrierea mai trebuie să se refere la *proprietăţile structurale* care specifică modul cum componenta poate fi compusă din alte componente, *proprietăţile non-funcţionale* care descriu performanţa, capacitatea etc şi *proprietăţile de familie* care stabilesc relaţii între componente înrudite.

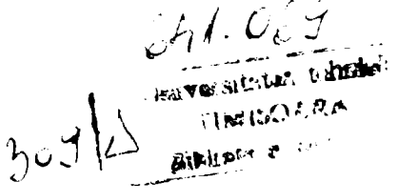
În consecinţa, specificaţiile componentelor arhitecturale ar trebui să fie caracterizate de următoarele proprietăţi:

- specificaţia este în cele mai multe cazuri *incompletă*: de multe ori proiectantul componentei nu poate anticipa toate aspectele care i-ar putea interesa pe utilizatorii componentei.
- specificaţia trebuie să fie *extensibilă*, deoarece în timp se pot descoperi noi tipuri de dependenţe în încercarea de a refolosi împreună componente dezvoltate independent.
- specificaţia este *eterogena*, trebuie să descrie toate categoriile de proprietăţi.

Din studiul literaturi rezultă că sunt puţine abordări în care se propun specificaţii care să cuprindă proprietăţile structurale, proprietăţile funcţionale şi non-funcţionale ale componentelor şi care să aibă proprietatea de a fi extensibile. În această teză se propune o soluţie care respectă aceste cerinţe, în forma schemei de descriere a componentelor compozabile care este definită în capitolul 3.

2.2.2 Clauze Requires-Provides

Contractele au loc *între* două sau mai multe componente şi specifică obligaţii *reciproce* între părţi. Chiar şi într-un sistem client-server, relaţia este mai mult decât faptul că modulul client depinde de modulul server spre a-i furniza un anumit tip de serviciu. Şi modulul server depinde de client ca acesta să acceseze şi să utilizeze serviciile într-un anumit fel. De aceea, specificarea unei interfeţe poate fi văzută în termenii *requires–provides*.



BUPT

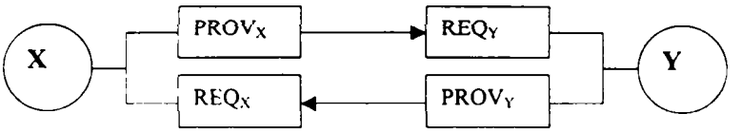


Figura 2.3: Contract requires-provides

Figura 2.3 ilustrează interacţiunea a două componente X şi Y din punct de vedere al relaţiilor lor *requires–provides*.

Componenta X garantează faptul că implementează partea ei de *provides* (notată PRO_X) dar în condiţiile în care are anumite cerinţe (notate REQ_X). Componenta Y cu care interacţionează are o interfaţă proprie, cuprinzând serviciile furnizate şi cele cerute, notate $PROV_Y$ şi respectiv REQ_Y. Pentru ca interacţiunea între X şi Y să aibâ loc corect, X trebuie să furnizeze ceea ce cere Y şi Y trebuie să furnizeze ceea ce cere X.

Contribuţiile lui Perry ([Per87], [Per89]) în domeniul clauzelor requires–provides, făcute în cadrul modelului de interconexiuni semantice pe care l-a propus au reprezentat deschideri de noi drumuri şi în contextul realizării compoziţiei corecte.

Modelul lui Perry se inspiră din tehnologiile utilizate de specificaţiile formale, cele bazate pe predicate de intrare/ieşire ([Hoa69]). Predicatele de intrare definesc precondiţii ce trebuie să fie satisfăcute pentru a se putea executa cu succes o anumită secvenţă de cod. Predicatele de ieşire definesc rezultatele, sau postcondiţii care sunt garantate să se întâmple (să fie adevărate) dacă predicatele de intrare au fost satisfăcute. Fiecare din aceste predicate, fie că sunt precondiţii sau postcondiţii, reprezintă un fapt ştiut asupra comportării sistemului sau a unei componente. Predicatele furnizează vocabularul de bază cu ajutorul căruia se poate descrie comportamentul cerut şi cel furnizat de către o componentă într-un sistem.

Perry defineşte modelul interconexiunilor semantice Inscape ([Per87]), prin analogie cu un chip hardware cu pini de intrare şi iesire, care poate fi echivalentul componentei software cu precondiţii şi postcondiţii. Specificaţiile interfeţelor sunt utilizate ca un mijloc practic pentru compoziţia programului. Perry extinde metoda de specificare a lui Hoare ([Hoa69]), introducând şi obligaţii pe lângă pre- şi postcondiţii. Obligaţiile sunt condiţii care trebuiesc să fie satisfăcute la un moment dat (de exemplu, obligaţia de a dealoca un buffer, dacă a fost alocată memorie pentru un buffer, sau obligaţia de a închide un fişier, daca operaţia a deschis anterior fişierul. Obligaţiile implică o acţiune la distanţă: ele sunt condiţii ce ar putea fi satisfăcute local de către componente adiacente, dar în general depind de proprietăţi globale ale sistemu-

BUPT

lui (proprietăţi ale unor componente neadiacente). Obligaţiile sunt propagate către modulele care le conţin unde în final trebuie să fie satisfăcute de nişte postcondiţii.

Perry introduce notiunea de **propagare** a condiţiilor: precondiţiile şi obligaţiile depind de postcondiţii sau sunt propagate către interfaţa care le cuprinde, unde vor fi în final satisfăcute de anumite postcondiţii. Postcondiţiile satisfac precondiţii şi obligaţii şi pot fi propagate spre interfaţa care le cuprinde, în măsura în care sunt potrivite pentru abstracţia realizată.

Un exemplu de utilizare a modelului de interconexiuni în construcţia unui program (din [Per87]) ilustrează utilizarea interconexiunilor în construcţia unei operaţii şi arată relaţia dintre implementarea operaţiei şi intrerfaţa sa propagată automat. Exemplul (prezentat în figura 2.4) consideră implementarea unei proceduri `ObtainRecord` în contextul unui sistem de fişiere, cu fişiere cu nume distincte, unde fiecare fişier constă din înregistrări identificate prin numărul inregistrării. Figura 2.4 ilustrează în mod grafic implementarea operaţiei `ObtainRecord` ca şi o secvenţă a trei apeluri de operaţii: `OpenFile`, `ReadRecord` şi `CloseFile`. Abrevierile din partea superioară a fiecărui modul reprezintă precondiţiile operaţiei, cele din partea inferioară postcondiţii iar cele din lateral obligaţiile.

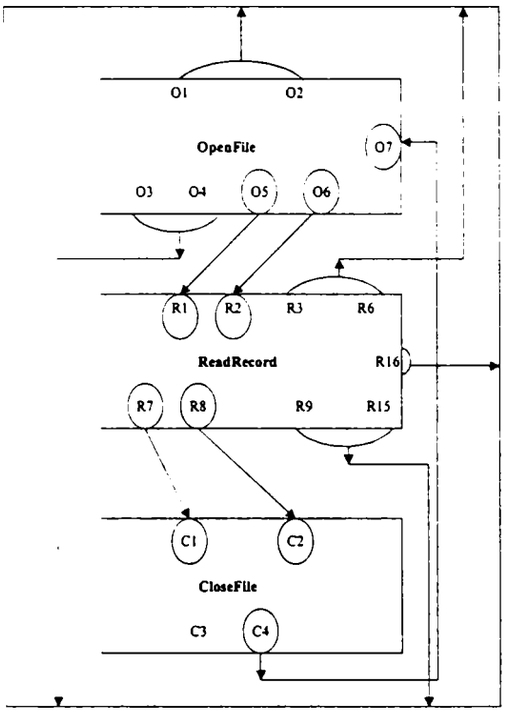


Figura 2.4: Exemplu de compunere a unei operaţii înInscape

Operaţia `ReadRecord` este cea care asigură în cea mai mare măsură funcţionalita-

BUPT

tea dorită pentru procedura `ObtainRecord`. Oricum, există un număr de precondiţii ($R1 - R6$) care trebuie satisfăcute înainte de a putea citi o înregistrare din fişier, un număr de postcondiţii care devin adevărate după citire ($R7 - R15$), precum şi o obligaţie ($R16$).

```
Operatia ReadRecord(FP, R, &L, &Bufptr):
Preconditii:
 R1  : ValidFilePtr(FP)
 R2  : FileOpen(FP)
 R3  : LegalRecordNr(R)
 R4  : RecordExists(R)
 R5  : RecordReadable(R)
 R6  : RecordConsistent(R)
Postconditii:
 R7  : ValidFilePtr(FP)
 R8  : FileOpen(FP)
 R9  : LegalRecordNr(R)
 R10 : RecordExists(R)
 R11 : Was(RecordReadable(R))
 R12 : Was(RecordConsistent(R))
 R13 : Allocated(*Bufptr)
 R14 : 0<=L<=Allocated(*Bufptr)
 R15 : RecordIn(*Bufptr)
Obligatii:
 R16 : Deallocated(*Bufptr)
```

Două dintre aceste precondiţii ($R1$ şi $R2$) sunt satisfăcute de postcondiţiile $O5$ şi $O6$ ale operaţiei `OpenFile`. Utilizarea procedurii `OpenFile` implică obligaţia ($O7$) de a închide fişierul, condiţie ce poate fi satisfăcuta de operaţia `CloseFile` prin postcondiţia sa $C4$. Precondiţiile operaţiei `CloseFile`, $C1$ şi $C2$, pot fi satisfăcute de postcondiţiile $R7$ şi $R8$ ale procedurii `ReadRecord`. Nu toate precondiţiile operaţiilor `OpenFile` sau `ReadRecord` au fost satisfăcute în cadrul implementării `ObtainRecord`. Acestea vor fi satisfăcute în afara acestei implementări şi deci exportate către interfaţa operaţiei care le conţine (O1 şi O2, R3-R6). Unele din postcondiţiile îndeplinite după execuţia procedurii `CloseFile` nu sunt exportate către interfaţa operaţiei care le cuprinde pentru că nu sunt corespunzătoare din punct de vedere semantic abstracţiei `ObtainRecord` (de exemplu $C3$ şi $C4$). Noţiunile de fişiere deschise şi închise ca şi pointer la fişier nu sunt utile pentru utilizatorul operaţiei implementate, `ObtainRecord`.

BUPT

În schimb, aspectele cunoscute despre fişier şi înregistrări sunt importante şi deci sunt propagate către interfaţa care le include (postcondiţiile $O3$ şi $O4$, $R9 - R14$). De asemenea şi obligaţiile neîndeplinite sunt propagate ($R16$).

Deşi obiectele care sunt supuse compoziţiei în modelul de interconexiuni al lui Perry sunt proceduri şi funcţii. modelul introdus de el, de clauze *requires-provides*, defineşte unul din punctele de plecare clasice în domeniul compoziţiei în general. Modelul compoziţional definit în capitolul 3 al acestei teze utilizează clauze de tip *requires-provides* pentru descrierea de componente. Mecanismul propagări cerinţelor introdus de Perry în forma discutată în acest paragraf constituie un punct de plecare pentru generalizarea propusă în această teză a acestui mecanism de propagare şi care stă la bază pentru definirea strategiei de compoziţie propuse în capitolul 4.

2.2.3 Descrieri arhitecturale

Modalităţi de specificare a contractelor între componente arhitecturale şi a modului în care pot interacţiona acestea sunt incluse şi în limbajele de descriere arhitecturală (*Architectural Description Languages - ADL*).

Limbajele din familia ADL reprezintă o metodă de a descrie arhitectura unui sistem utilizând o notaţie formală. Aceste descrieri formale pot fi procesate apoi de unelte automate pentru a analiza consistenţa sau pentru a genera cod.

ADL fac descrierea sistemelor la un nivel de abstractizare înalt, bazându-se pe grafuri de componente între care există interacţiuni. În general, sunt definite *componente*, ca şi punctele principale de procesare în sistem şi *conectori*, care definesc interacţiunile între componente. O componentă este elementul de bază în construcţia unui sistem. Se pot construi componente complexe prin compunerea mai multor componente elementare. Aceste componente elementare (sau primitive) au o specificare comportamentală, spre deosebire de cele compuse care au o specificare structurala.

Au fost propuse un numar mare de limbaje de tip ADL, printre cele mai cunoscute fiind C2 ([MORT96]), Darwin ([MDEK95], [MK96]), LILEANNA ([Tra93]), Rapide ([LV95]), UniCon ([SRG96]), Weaves([GR91]), Wright ([AG97]), Acme ([GMW00]).

Toate ADL enumerate mai sus modelează componente. Lucrarea [MT00] realizează o comparaţie a facilităţilor prezentate de aceste limbaje. Posibilitatea de a modela interfaţa componentelor este o capacitate fundamentală a tuturor ADL. Interfaţa este un set de *puncte de interacţiune* între componentă şi mediul înconjurător. Interfaţa specifică *serviciile* (mesaje, operaţii, variabile) furnizate de o componentă. Majoritatea ADL au de asemenea facilităţi pentru a specifica *cerinţele* unei componente, în sensul serviciilor cerute din partea altor componente. Interfaţa de-

BUPT

fineşte angajementele pe care o componentă şi le ia din punct de vedere computaţional şi constrângerile privind utilizarea sa. Punctele de interfaţă au diferite denumiri, de exemplu *port* în Wright şi Acme, *player* în UniCon. De obicei există puncte de interfaţă diferite pentru funcţionalitatea furnizată (*provided*) şi cea cerută (*requested*). Punctele de interfaţă pot fi tipizate (Darwin, UniCon) sau indiferente la tipuri (C2, Weaves).

Unele ADL suportă specificarea semanticii componentelor, în diferite grade. De exemplu, UniCon exprimă informaţiile semantice sub formă de liste de proprietăţi, Wright modelează dinamica comportării (utilizează CSP). Acme şi Weaves permit specificarea unor proprietăţi arbitrare. În ceea ce priveste punctele slabe ale ADL, numai puţine suportă specificarea unor proprietăţi non-funcţionale.

Limbajele ADL pot exprima constrângeri referitoare la utilizarea componentelor. Interfaţa specificată a unei componente cuprinde toate posibilităţile permise de interacţiune. Specificaţile semantice adaugă relaţii şi dependenţe între elementele interne ale componentei. Constrângeri pot fi specificate prin atribute în UniCon (de exemplu, la câte conexiuni poate fi conectat un port), invarianţi stilistici în C2 şi Wright (de exemplu, numărul şi tipurile porturilor pe care le poate avea o componentă).

Marea majoritate a ADL prezintă un model static al componentelor. În mod tipic, majoritatea ADL permit realizarea unor descrieri statice ale sistemului şi nu au facilităţi pentru a realiza modificări în arhitectură în timpul execuţiei. Limbajele şi uneltele care suportă dinamismul sunt încă în stare incipientă şi este vorba de descrierea unui dinamism "programat" al arhitecturii. Se permit modificări în timpul execuţiei, dar modificările sunt "codate" şi compilate în aplicaţie. In vederea realizării de arhitecturi dinamice, [Ore96], [OMT98] şi [Med96] consideră că mai sunt necesare în plus:

- Limbaje de descriere a modificărilor arhitecturale (Architectural Modification Language - AML), pentru descrierea operaţiilor de modificare ([Med96]).
- Limbaje de descriere a constrângerilor arhitecturale (*Architectural constraint languages* ACL), pentru descrierea condiţiilor în care pot fi efectuate modificările arhitecturale. Asemenea limbaje sunt utilizate pentru a exprima constrângeri asupra structurii unui sistem, utilizând specificaţii imperative [Bal94] şi declarative [MK96].

Oberleitner şi Gschwind propun limbajul ACL (Architectural Composition Language) (în [OG02], [OG03]), limbaj care combină aspecte de ADL şi de limbaj de compoziţie. Acest limbaj conţine construcţii pentru definirea de noi componente şi conectori, modelul rezultat fiind unul executabil.

BUPT

Se constată că pentru realizarea corectă a compoziţiilor de componente sunt necesare, pe de o parte, descrieri ale proprietăţilor interfeţelor componentelor individuale, iar pe de altă parte, descrierea constrângerilor arhitecturale, structurale, ale compoziţiei. Nici un limbaj de descriere cunoscut din literatură nu prezintă caracteristicile necesare pentru descrierea tuturor acestor aspecte într-un mod unitar, diverse limbaje prezintă facilităţi pentru aspecte singulare. În secţiunea 3.3 a acestei teze se defineşte limbajul CCDL, un limbaj de descriere propriu care combină facilităţi specifice limbajelor de descriere a interfeţelor cu facilităţile de limbaj de descriere arhitecturală.

2.2.4 Problema compoziţiei predictibile

Problema compoziţiei predictibile este un aspect important şi la ordinea zilei ([CSSW01], [CSSW02], [HMSW02]). Problema se referă la capacitata de a prezice proprietăţile unui ansamblu de componente pe baza proprietăţilor componentelor individuale.

Sistemul exemplu din figura 2.5 ([BBB+00]) conţine două componente, $C1$ şi $C2$. $C1$ are proprietăţile $p1$ şi $p2$, iar $C2$ are proprietatea $p3$. Trebuie demonstrat că sistemul compus prezintă proprietatea $p4$.

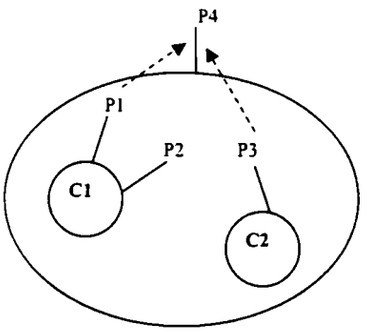


Figura 2.5: Problema compoziţiei predictibile

Proprietăţile unui sistem compus se datorează în general unei mulţimi de componente care interacţionează, şi nu unei componente singulare din cadrul sistemului compus. În consecinţă, proprietăţile părţilor trebuie combinate pentru a putea prezice proprietăţile întregului. In exemplul din figura 2.5, proprietăţile $p1$ şi $p3$ ale componentelor $C1$ şi $C2$ sunt legate cauzal de proprietatea $p4$ a ansamblului. Proprietatea $p2$ a lui $C1$ nu influenţează proprietatea $p4$ a ansamblului. Ca exemplu concret, dacă $p4$ reprezintă latenţa sistemului iar $p1$ şi $p3$ reprezintă anumiţi indicatori de performanţă ai componentelor $C1$ şi $C2$, se pot defini metode de calcul al lui $p4$ din $p1$ şi $p3$. Dacă proprietatea $p2$ se referă la calitatea documentaţiei componentei

BUPT

$C1$, aceasta nu poate servi în nici un fel la determinarea proprietăţii de latenţă a ansamblului.

În literatură se constată că rezultate practice în domeniul compoziţiei predictibile s-au obţinut în special pentru proprietăţi statice (latenţă [HMSW02], [LWNC02], consum de memorie [FEHC02], consistenţa versiunilor componentelor [LWNC02]), unde proprietatea ansamblului poate fi calculată din proprietăţile (de acelaşi tip) ale componentelor individuale constituente. Metodele de calcul sunt legate de domeniul de aplicaţie şi de caracteristicile impuse familiei din care fac parte sistemele compuse [LWNC02]. Metodele care abordează proprietăţi necantitative se concentrează pe câte o singură proprietate: deadlock [IT02a], fiabilitate (*reliability*) [SM02]. Modelul compoziţional dezvoltat în această teză permite compoziţia predictibilă a unor componente cu proprietăţi abstracte sau necantitative prin mecanismelle definirii proprietăţilor componentelor şi constrângerilor pentru componentele compozabile care vor fi descrise în capitolul 3.

2.3 Aspecte de automatizare a lucrului cu componente

Există o puternică tendinţa de a realiza automatizări în ingineria software ([ASE02], [ASE01], [ASE00]). Aspectele de automatizare a procesului de compoziţie se referă la verificarea automată a corectitudinii unei compozitii date sau la automatizarea în diverse grade a generării unei compoziţii de componente care să realizeze implementarea unui sistem care oferă un set de functionalităţi şi proprietăţi cerute.

2.3.1 Verificarea automată a corectitudinii unui sistem compus

Verificarea semantică

Incercări de a realiza verificări ale corectitudinii unui sistem compus pe baza verificării contractelor componentelor sunt menţionabile din lucrările lui Perry ([Per87], [Per89]) în contextul unui mediu pentru gestionarea evoluţiei sistemelor software.

După cum s-a prezentat în paragraful 2.2.2, Perry utilizează noţiunea de propagare a condiţiilor pentru a realiza o verificare semantică intuitivă. Batory [BG97] adaptează metoda introdusă de Perry pentru verificarea proprietăţilor statice ale unui sistem software. Cazurile considerate de Batory în [BG97] se referă la sisteme ierarhice, cu arhitecturi *GenVoca* (stratificate). Diferenţa faţă de lucrările lui Perry este

BUPT

scara componentelor: o componentă Inscape este o funcţie, pe când o componentă *GenVoca* este un subsistem.

În cazul arhitecturii *GenVoca*, componentele au precondiţii şi obligaţii care nu sunt neaparat satisfăcute local (de către componente adiacente). Precondiţiile şi obligaţiile unei componente k sunt satisfăcute "la distanţă", de către componente care sunt situate deasupra sau dedesubtul componentei k, imediat sau la distanţă. O particularitate accentuată de Batory e că proprietăţile exportate de către o componentă k spre straturile superioare nu sunt aceleaşi cu proprietăţile exportate către straturile inferioare. Din acest motiv, Batory face deosebirea între două tipuri de precondiţii şi postcondiţii. *Postcondiţii* sunt numite acele proprietăţi ale unei componente k, ce sunt expuse componentelor aflate sub k. *Precondiţiile* unei componente k definesc proprietăţi care trebuie să fie satisfăcute pentru ca acea componentă să funcţioneze corect, ele testeaza toate postcondiţiile componentelor aflate deasupra lui k. După cum este reprezentat în figura 2.6, o componentă k cere ca şi precondiţie faptul ca atributul A trebuie să aibă valoarea v. Pentru utilizarea corecta a lui k, trebuie să existe o componentă u, care este deasupra lui k, şi a cărei postcondiţie setează $A = v$. Se face precizarea că u nu este în mod necesar aflat imediat deasupra lui k ci în orice strat deasupra, chiar la distanţă. *Postrestricţii* sunt numite proprietăţi ale unei componente k, care sunt exportate către componente aflate deasupra lui k. *Prerestricţii* sunt precondiţii care testează în mod cumulativ postrestricţiile componentelor care se afla sub k. După cum este reprezentat în figură, pentru o componentă k ce are un singur parametru cu prerestricţia ca atributul A trebuie să aibă valoarea w, trebuie să existe o componentă d, care se află sub k, a cărei postrestricţie setează $A = w$.

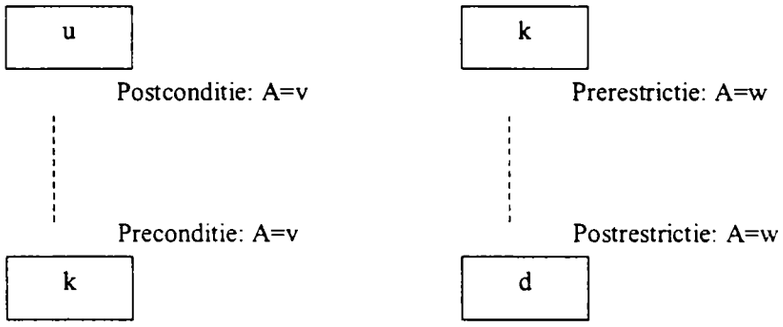


Figura 2.6: Precondiţii şi postcondiţii în arhitecturi stratificate

Fiind definite regulile de proiectare (precondiţii, postcondiţii, prerestricţii, postrestricţii) ale fiecărei componente, verificarea regulilor de proiectare implică o propagare *top-down* a postcondiţiilor şi testarea precondiţiilor, precum şi o propagare

BUPT

bottom-up a postrestricţiilor şi testarea prerestricţiilor. [BG95] prezintă algoritmi recursivi pentru propagarea *top-down* a postcondiţiilor şi testarea precondiţiilor, ca şi pentru propagarea *bottom-up* a postrestricţiilor şi testarea prerestricţiilor. De asemenea, în [BCRW00] se prezintă o variantă îmbunătăţită a acestui algoritm, nu doar pentru verificarea, dar şi pentru optimizarea unei compoziţii din perspectiva minimizării unei anumite funcţii de cost definite.

Verificarea integrării comportamentale a componentelor într-o compoziţie

Pentru a realiza descrierea comportamentală a componentelor, se utilizează în general metode bazate pe automate finite ([SR00], [Reu01]), sau pe *message sequence charts* (MSC) ([SVSJ03], [IT02b]).

Inverardi propune metode ([IWY97], [IT02b]) de a verifica integrarea comportamentală. Pentru fiecare componentă, sunt date atât o specificare a comportamentului propriu de interacţiune cât şi presupunerile (*assumptions*) pe care le face despre comportamentul de interacţiune al contextului în care se aşteaptă să opereze. În [IWY97] se propune un algoritm care, fiind date asemenea specificaţii pentru un set de componente, face teste de compatibilitate între presupunerile referitoare la context şi comportamentul de interacţiune propriu al componentei. Se formulează o metodă bazată pe modelul CHAM (Chemical Abstract Machine) ([Wer98]) ca şi formalism de specificare a arhitecturilor software. Conform metodei CHAM, o arhitectură este modelată ca şi o maşină abstractă dupa modelul reacţiilor chimice.

Pentru a realiza testul de compatibilitate între componente, [IWY97] propune ca formă de reprezentare a comportării curente (*actual behaviour, AC*) a unei componente şi a comportării presupuse (*assumed behaviour, AS*) a mediului să fie dată de grafuri derivate din modelul CHAM. Un graf AC modelează comportamentul în modul următor: nodurile reprezintă stări ale componentei şi sunt în consecinţă molecule. Nodul rădăcină e starea iniţială a componentei, iar fiecare arc reprezintă o posibilă tranziţie într-o nouă stare, utilizând o regulă de transformare din modelul CHAM al componentei. Grafurile AS sunt opusul celor AC. Pentru fiecare graf AC, există un graf AS corespunzător care modelează ce tip de comportare este cerută din partea contextului pentru a efectua operaţiile modelate de către graful AC. În general contextul poate fi furnizat de mai multe componente, grafurile AS se referă în general la comportarea a mai mult de o componentă. Pentru a face verificarea că o anumită configuraţie a unui sistem are ca rezultat un sistem corect, trebuie făcută o comparaţie între aşteptările asupra contextului extern făcute de către o componenta şi comportamentul actual al componentelor cu care aceasta interacţionează. Comparaţia se face pe baza unei relaţii de echivalenţă între grafurile AC şi AS.

BUPT

Metodele de verificare comportamentală testează în general compatibilitatea componentelor la nivelul sintactic şi de sincronizare, nivelul semantic (care descrie ce face componenta) nefiind acoperit. Ca rezultat, este posibil să se declare compatibile componente cu semantici foarte diferite, dar al căror comportament e descris de automate compatibile.

2.3.2 Generarea automată

Există diverse posibilităţi de a automatiza generarea unui sistem cu proprietăţi dorite: programarea generativă, compoziţia automată de componente, generarea automată a adaptărilor/conexiunilor unor componente. În cazul generării automate, pe langă problema stabilirii potrivirilor între componente (care este la fel ca şi la verificare) apar şi alte aspecte specifice, tratate în această secţiune.

Programarea generativă. Generatoare pentru familii de sisteme

Generatoarele de sisteme software automatizează dezvoltarea de software pentru familii de aplicaţii. Un generator transformă o specificare de nivel înalt a sistemului ţintă în cod sursă. Baza pentru aceasta o formează biblioteci de componente parametrizate şi care sunt compatibile pentru interconectare.

Un generator se adresează unui anumit domeniu de aplicaţie, componentele nu sunt *stand-alone*, ci sunt proiectate pentru a fi compatibile şi interoperabile. In plus, fiind vorba de generare de cod sursă, toate componentele sunt implementate în acelaşi limbaj.

Batory ([BO92], [BCRW00]) şi Czarnecki ([CE99b]) propun metode de programare generativă pentru arhitecturi stratificate (*GenVoca*). Acest tip de arhitectură poate descrie o mare varietate de sisteme, având totodată şi avantajul că este simplă, iar compunerea unui sistem de acest tip poate fi descrisă de reguli simple. Nu există încă descrise în literatura din domeniu abordări pentru compoziţii complet automatizate pentru alte tipuri de arhitecturi. Această teză propune o astfel de abordare pentru arhitecturi ierarhice multi-flux, o generalizare a arhitecturilor stratificate.

Domeniul programării generative abordează o problemă adiacentă celei avute în vedere în această teză (compoziţia software), în sensul că tratează procesul de producţie a unei familii de produse, dar pornind de la şi începând cu proiectarea componentelor special pentru a fi compuse într-un tip binecunoscut de aplicaţie. De asemenea, programarea generativă realizează generarea la nivelul codului sursă, metodele nefiind direct aplicabil pentru compunerea, din componente existente, a unor sisteme cu proprietăţi necunoscute în faza de analiză a problemei şi de realizare

BUPT

a generatorului. Este însă interesantă ideea privind sarcinile unui generator, prezentată in [CE99b]. Pentru a realiza generarea unui sistem finit alcătuit din componente pornind de la o specificaţie a sa, [CE99b] identifică paşii necesari, reprezentaţi în figura 2.7:

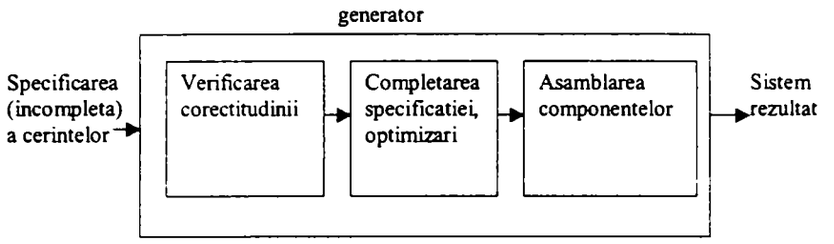


Figura 2.7: Sarcinile unui generator de compoziţii

Un generator trebuie să verifice dacă sistemul poate fi construit din cerinţele specificate (dacă nu sunt specificate combinaţii ilegale de proprietăţi dorite), completează şi optimizează specificaţia pe baza unor reguli implicite şi asamblează componentele.

Definirea rolului generatorului este foarte importantă şi general valabilă pentru orice modul care realizează compunerea automată a unui sistem pe baza unor cerinţe. Totuşi, în sisteme mai complexe, completarea specificaţiei până la nivelul determinării secvenţei de componente care vor fi utilizate la implementare poate fi în sine o operaţie foarte complexă, necesitând strategii de compoziţie bine definite.

În cadrul programării generative, unde un generator este *specific unui anumit domeniu de aplicaţie*, elementul care face legatura între spaţiul problemei şi cel al soluţiei este *configuration knowledge*.

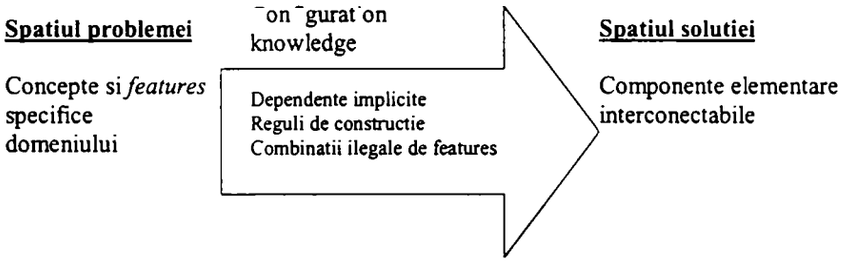


Figura 2.8: Locul informaţiilor de configurare utilizate în compoziţie

Configuration knowledge conţine informaţiile specifice domeniului, cum ar fi: combinaţiile nepermise de facilităţi (în caz că anumite combinaţii nu au sens), setările implicite (dacă anumite facilităţi nu sunt specificate, se vor considera valori implicite),

BUPT

dependenţe implicite (unele facilităţi neexprimate explicit vor fi deduse pe baza celor specificate), reguli de construcţie (reguli care specifică cum sunt traduse combinaţiile anumitor facilităţi cerute în anumite combinaţii de componente de implementare. Rezultă de aici că strategiile de generare bazate pe conceptul de *configuration knowledge* sunt fiecare specifice unui anumit domeniu de aplicaţie, unei anumite familii de produse.

Limbaje de descriere

În general comandarea oricărui produs nu implică descrierea părţilor din care este asamblat, ci se specifică anumite proprietăţi dorite ale produsului. În literatură ([CE99a], [CE99b]) se discută importanţa unor limbaje de specificare pe domenii de aplicaţii care să fie utilizate la specificarea cerinţelor pentru un sistem care trebuie generat automat.

Un aspect important al limbajelor de specificare pe domenii e nivelul de specificitate al limbajului. Acesta ar trebui să poată acoperi, în mod ideal, un spectru larg, de la nespecific şi până la specificarea în detaliu a componentelor. Cerinţele pentru sistemul dorit trebuie să poată fi specificate la niveluri diferite de detaliu. După caz, poate fi nevoie atât de specificarea sumară a cerinţelor, cât şi de posibilitatea ca utilizatorul să specifice în detaliu proprietăţile. Intr-o aplicaţie o anumită componentă va fi cerută specificând numai atâta informaţie câta este necesară. Sistemul nu trebuie să forţeze specificarea a prea multe detalii (ceea ce ar duce la o dependenţă sporită de implementarea bibliotecii de componente), dar, în caz de nevoie, trebuie să permită specificarea detaliilor. Suportarea unor niveluri diferite de specificitate se poate face prin implementarea unor setări implicite şi definirea unor reguli de comportare implicită.

Un alt posibil avantaj al utilizării unui limbaj de specificare a produsului finit în loc de o simplă enumerare a componentelor rezultă din faptul că în general, o anumită facilitate cerută produsului nu se va reflecta într-o unică componentă ce o realizează, ci mai degrabă într-o anumită combinaţie specifică de componente.

Compoziţia automată de componente

În domeniul compoziţiei automate de componente, cercetările existente propun în general construirea automată a unui sistem pornind de la o *descriere* a acestuia şi de la diverse forme adiţionale de *restricţii* sau *presupuneri iniţiale* referitoare la structura sistemului. În subcapitolul 2.4 se vor prezenta mai în detaliu diferite forme de a exprima restricţiile şi presupunerile iniţiale referitoare la structura unei compoziţii, în

BUPT

contextul managementului variabilităţii acesteia. Construcţia sistemului implică două acţiuni: determinarea selecţiei componentelor necesare şi determinarea conexiunilor între acestea. Intervenţia operatorului este de dorit să fie minimă şi să se exprime doar în termenii proprietăţilor dorite ale sistemului: operatorul nu trebuie să fie obligat să intervină direct cu decizii în procesul de compoziţie a sistemului în termeni de componente şi conexiuni.

Un exemplu semnificativ în acest sens al compoziţiei automate este mediul de dezvoltare bazată pe configurare *Aster* ([BISZ98], [IBS98], [IB96], [KI00]), utilizat în configurarea automată de aplicaţii distribuite şi middleware.

ASTER oferă uneltele necesare pentru selectarea şi integrarea de componente de middleware, pornind de la descrierea arhitecturală a aplicaţiei şi de la cerinţele non-funcţionale ale acesteia. Aster este compus din:

- Un limbaj declarativ pentru descrierea unei aplicaţii distribuite în termenii interconectării unor componente la nivelul interfeţelor lor, împreună cu cerinţele non-funcţionale ale aplicaţiei referitoare la nivelul de middleware

- Un set de unelte pentru generarea de cod pentru aplicaţie, pe baza selecţiei şi configurării automate a unui middleware adaptat conform cerinţelor aplicaţiei (după nivelul de securitate dorit în schimbul de date, gradul de disponibilitate dorit al modulelor, caracteristici ale canalelor de comunicaţie, etc).

Limbajul declarativ Aster este dezvoltat prin extinderea limbajului Interface Description Language al OMG [OMG95] cu construcţii care permit exprimarea proprietăţilor non-funcţionale cerute, respectiv furnizate de către module. Setul de proprietăţi recunoscute este expandabil, fiind conţinut într-o bază de date care poate fi actualizată.

Sistemele de middleware construite cu Aster constau dintr-un *base middleware* care furnizează mecanismele esenţiale de comunicare şi componente middleware utilizate la crearea sistemului final adaptat cerinţelor (*customized*).

Configurarea automată face selectarea, pe baza potrivirii specificaţiilor între cerinţele aplicaţiei şi ale componentelor middleware dintr-un repository, a unui *base middleware* şi a unor componente middleware adiţionale. După această fază de selectare are loc faza de generare propriu-zisă a codului aplicaţiei adaptate cerinţelor. Aceste etape sunt ilustrate în figura 2.9 (din [IBS98]).

Este specific pentru ASTER faptul că un sistem adaptat cerinţelor este construit pornind de la un *repository* de componente şi o magistrală sistem primitivă existentă (denumită *base middleware* sau *software bus*). Exprimarea cerinţelor aplicaţiei se

BUPT

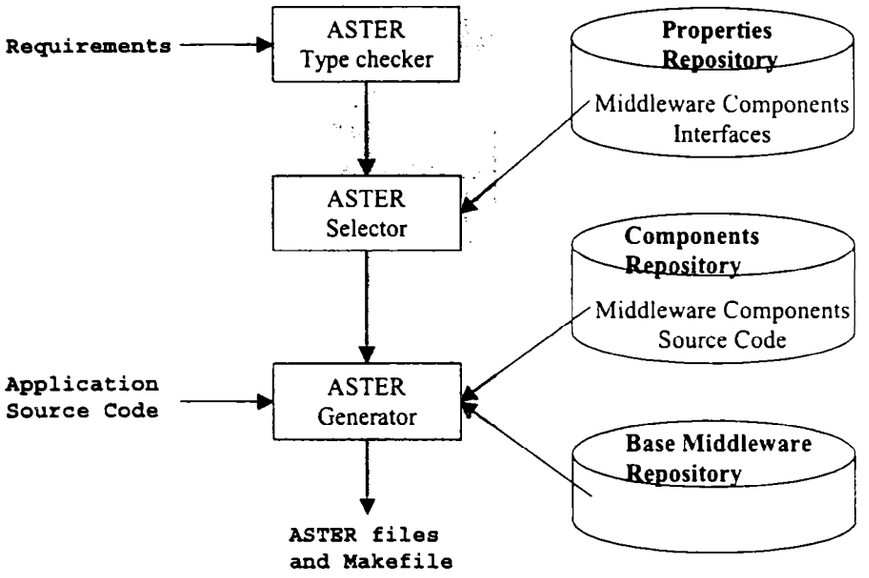


Figura 2.9: Mecanismul de generare automată în Aster

bazează pe existenţa unui arbore de proprietăţi ([IB96]), care reprezintă întregul set de proprietăţi recunoscute, organizate în categorii de proprietăţi ierarhice.

Adaptarea automată a componentelor

O situaţie care apare frecvent în practică este că funcţionalitatea furnizată de o componentă nu se potriveşte în totalitate cu cea necesară unei alte componente. In acest caz trebuie identificată partea care se potriveşte cerinţelor şi apoi trebuie realizate adaptări pentru a obţine o compatibilitate deplină. Deoarece adaptarea unei componente poate fi o muncă manuală foarte complexă, există cercetări în scopul automatizării acestei operaţii.

În [Sch01], [SR00], [RS02], [Reu01] se propune un model de componente adaptabile, care conţin suficiente informaţii în interfaţă pentru a permite adaptarea lor, prin extinderea sau prin reducerea funcţionalităţilor lor cu ajutorul unor adaptoare de interfaţă. Compatibilitatea componentelor este privită din punctul de vedere al unor contracte comportamentale exprimate cu ajutorul automatelor finite.

O altă metoda, propusă de Inverardi ([IS01], [IT02b]), abordează generarea corectă a *conectorilor* pentru a asigura comportarea corectă a unei compoziţii de componente. Se foloseşte o descriere comportamentală a componentelor prin LTS (*labeled transition systems*), reprezentând mesajele pe care componentele le pot trimite sau recepţiona pe canalele de comunicare (metoda derivată din MSC - *message sequence charts*).

BUPT

Metoda de generare porneşte de la un set dat de componente şi se generează posibilii conectori, analizând pentru fiecare proprietăţile sistemului. În lucrările citate se analizează doar proprietatea de *deadlock*.

2.3.3 Arhitecturi dinamice şi autoadaptive

Sistemele autoadaptive reacţionează la schimbări în mediu şi în disponibilitatea resurselor, reconfigurându-se ca să îşi continue funcţionarea cu bune performanţe. Astfel de sisteme îşi adaptează în mod dinamic comportamentul, adaptând părti din sistem, înlocuind sau adăugând anumite componente software acolo unde e nevoie. În sistemele convenţionale, un furnizor de servicii trebuie în general să oprească, să modifice şi să repornească un serviciu dacă doreşte să il actualizeze sau să il reconfigureze. În multe cazuri poate fi inacceptabil să se întrerupă un serviciu pe durata adaptării, în aceste cazuri fiind nevoie de implementarea lor ca servicii autoadaptive. Cercetarile din domeniul reconfigurării dinamice încearcă să gasească soluţii la această problemă.

Tipuri de dinamism

Un sistem software bazat pe componente îşi poate schimba comportamentul prin două mecanisme de schimbare a configuraţiei sale:

1. schimbarea comportamentului uneia sau mai multor componente individuale
2. schimbarea structurii compoziţiei.

Dacă se utilizează o reprezentare a sistemului de componente sub formă de graf orientat, unde nodurile sunt componentele iar arcele căile de transmisie de mesaje, ca în figura 2.10, se pot da următoarele reformulări ale acestor cazuri ([AW99]):

1. schimbarea comportamentului unei componente se realizează înlocuind un nod al grafului cu o altă variantă de implementare. În acest caz există un tip fixat al nodului, iar componentele care pot fi plasate în acel nod sunt implementări diferite ale aceluiaşi tip de componente.
2. schimbarea structurii compoziţiei ar putea fi realizată la rândul ei în două moduri: alterarea fluxului de control (schimbarea în mod dinamic a rutelor mesajelor ce trec prin aceleaşi componente) sau alterarea structurii grafului (prin adăugarea unor componente instanţiate în mod dinamic). Rezultatele prezentate în literatura de specialitate se referă în special la prima variantă, neexistând soluţii pentru alterarea în mod neanticipat a structurii grafului.

BUPT

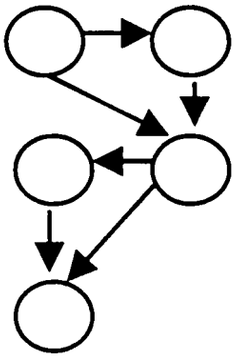


Figura 2.10: Exemplu de reprezentare a unui sistem de componente sub forma de graf orientat

O clasificare a modificărilor arhitecturale în funcţie de momentul de timp la care se operează modificarea este făcută de Oreizy ([Ore96]) în felul următor:

- *la momentul proiectării.* Modificările sunt făcute înainte ca modelul arhitectural şi codul sursă asociat sa fie compilate într-o aplicaţie.
- *La un moment înainte de execuţie.* Modificările sunt făcute după compilare dar înainte de execuţie. Asemenea modificări cer ca modelul arhitectural al aplicaţiei să fie inclus în aplicaţie şi să existe mecanisme pentru adăugarea sau înlăturarea de componente.
- *La momente ante-determinate, dar în timpul execuţiei.* Modificările se fac doar când anumite condiţii pre-specificate sunt îndeplinite, este o reconfigurare "programată".
- *In timpul execuţiei.* Modificările sunt făcute în timpul execuţiei, când nu s-au făcut cu anticipaţie nici un fel de presupuneri cu privire la starea aplicaţiei sau la topologia sa arhitecturală.

Sisteme auto-adaptive

O tendinţă actuală în sistemele software e că acestea trebuie să fie capabile să se autoconfigureze, adaptându-se la mediul în care se execută. Mediul de operare al sistemului poate fi reprezentat prin intrări provenind de la un operator, echipamente hardware externe sau senzori, evenimente programate.

Noţiunea de arhitectură auto-organizantă este înrudită cu reconfigurarea programată, scopul ambelor fiind de a minimiza pe cât posibil responsabilităţile externe de management. Sistemul ştie singur să execute o reconfigurare când are loc un anumit

BUPT

eveniment declanşator (defectarea unei componente, adăugarea de noi componente, etc), în sarcina etapei de proiectare ramânând doar specificarea constrângerilor de modificare. Gradul de autonomie poate varia, de la adaptarea complet automată până la includerea explicită a unor decizii ale utilizatorului în proces. Există sisteme *open-adaptive* sau *closed-adaptive*, conform taxonomiei din [OGT+99]. Un sistem este denumit *open-adaptive* dacă permite în timpul execuţiei să se introducă noi comportamente şi scenarii de adaptare. Un sistem este numit *closed-adaptive* dacă nu e capabil să suporte adăugarea din mers de noi comportamente.

Oreizy, Taylor, Medvidovic ş.a. propun în [OGT+99] o metodologie generală pentru realizarea de sisteme software autoadaptive. Ei consideră că abordarea potrivită pentru modificările pe scară largă este una bazată pe arhitectură: modificările sunt formulate şi evaluate pe un *model arhitectural* explicit, rezidând pe platforma implementării. Modificările în modelul arhitectural sunt reflectate în modificări în implementare şi trebuie asigurată în permanenţă consistenţa dintre model şi implementare. Serviciile de monitorizare şi evaluare observă aplicaţia şi mediul său de operare, dând *feedback* către procesul de management al adaptării. Includerea unui model arhitectural explicit în sistem este asemănatoare cu realizarea de sisteme reflective, metodă frecvent utilizată pentru sisteme dinamice şi adaptive ([HŞ97], [BCRP98], [BCCD00], [KC00], [TJJ00]) .

Modificarea simultană de componente, conectori şi tolpologie într-un mod fiabil necesită mecanisme şi un formalism arhitectural specifice. Sunt necesare facilităţi pentru coordonarea şi verificarea modificărilor, ca parte din infrastructura de adaptare. Pe măsură ce arhitectura se adapteaza şi evoluează, apare problema menţinerii unui model exact şi consistent al arhitecturii aplicaţiei şi al părţilor ei constituente. Pentru a putea aborda aceste probleme, este nevoie să se utilizeze, ca parte integrantă a aplicaţiei, un model arhitectural care descrie interconectările între componente şi conectori şi maparea lor pe elemente de implementare. Maparea aceasta permite ca modificările, făcute în termenii modelului arhitectural, să aibă ca efect modificări corespunzătoare în implementare.

Mediul de execuţie (*runtime system*) trebuie să asigure buna funcţionare a modificării sistemului în timpul execuţiei. Pentru aceasta, responsabilităţile sale trebuie să includă:

- Menţinerea modelului arhitectural al sistemului. Deoarece modelul arhitectural este necesar pentru a realiza modificările în timpul execuţiei, acest model trebuie inclus ca parte a sistemului şi întreţinut de către mediul de execuţie.

- Efectuarea modificărilor arhitecturale, utilizând facilităţile furnizate de mediu,

BUPT

ca încărcarea şi legarea dinamică a componentelor, mecanisme de comunicare interproces pentru comunicarea între componente etc.

În cazul reconfigurării dinamice a unui sistem prin înlocuirea unei componente pot să apară unele probleme majore de consistenţă. Prima problemă ar fi legată de garantarea faptului că nici o componentă înca în uz (care mai e utilizată de altă componentă) nu e scoasă din sistem. O a doua problemă este legată de faptul că unele componente conţin informaţii de stare, şi când o componentă este înlocuită, starea internă a celei vechi trebuie transmisă celei noi. Încă nu sunt descrise în literatură soluţii pentru transferul informaţiilor de stare în cazul înlocuirii componentelor.

2.4 Probleme de management al variabilităţii în procese de decizii automate

În cadrul proceselor care necesită decizii automate privitoare la construcţia sau configurarea/reconfigurarea unui sistem (cum sunt programarea generativă, compoziţia dinamică de componente – discutate în paragraful 2.3.2, arhitecturile dinamice şi auto-adaptive – discutate în 2.3.3) trebuie rezolvate unele probleme tehnice asemănatoare. Este vorba de respectarea dependenţelor între componente, precum şi a unor restricţii sau presupuneri iniţiale referitoare la structura sistemului şi a variabilităţii permise a caracteristicilor sistemului.

2.4.1 Identificarea şi reprezentarea dependenţelor

Oreizy, Taylor şi Rosenblum ([OGT+99], [OMT98]) au arătat necesitatea unei reprezentări explicite a arhitecturii sistemului, pentru a putea realiza reconfigurarea dinamică. În modelul lor componentele interacţioneaza prin conectori, care sunt utilizaţi atât pentru comunicare, cât şi pentru reprezentarea dependenţelor între grupuri de componente. Din cauză că aceşti conectori sunt utilizaţi şi pentru comunicare în grup, modelul nu face distincţia între situaţia când două componente legate printr-un conector depind una de alta sau doar comunică. Este necesar ca să existe posibilitatea de a realiza o analiză a dependenţelor ([SW01]). Kon ([KC00], [KC99], [KYH+01]) identifică o cerinţă esenţială în dezvoltarea de sisteme configurabile în mod automat bazate pe componente, şi anume gestionarea corectă şi completă a dependenţelor între componente. Este susţinută necesitatea unui model de reprezentare a dependenţelor între componente şi a unui mecanism de gestionare a acestor dependenţe. Sistemele trebuie să aibă reprezentări explicite ale dependenţelor

BUPT

între componente, ceea ce duce la posibilitatea de configurare dinamică şi automată a unui sistem construit din componente.

Kon face distincţie între două tipuri de dependenţe: cerinţe pentru încărcarea unei componente statice în sistem (*prerequisites*) şi dependenţe dinamice între componente active într-un sistem în execuţie. Cunoaşterea de către sistem a prerechizitelor pentru instalarea şi execuţia fiecărei componente duce la posibilitatea automatizării instalării şi configurării de noi componente. De asemenea, dacă un sistem cunoaşte dependenţele dinamice care există la un moment dat între componentele sale active, atunci el poate reacţiona mai bine la excepţii care pot afecta funcţionarea unor anumite componente şi poate suporta reconfigurarea dinamică prin înlocuirea anumitor componente.

În [KC00], [KC99] prerechizitele sunt exprimate ca dependenţe de componente persistente, hardware sau software, iar dependenţele dinamice se referă la dependenţe în timpul execuţiei, în care sunt implicate componente dinamice, posibil volatile. Prerechizitele conţin informaţii referitoare la natura şi capacităţile resurselor şi serviciile software care sunt necesare. Această ultimă categorie de prerechizite este echivalentă cu clauzele *requires* intâlnite la descrierile de componente. O idee importantă din [KC00] e că în implementarea prototipului pot fi suportate diferite tipuri de parsere de prerechizite, permiţând diferite limbaje de specificare şi chiar politici de rezolvare a cerinţelor.

Pentru gestionarea dependenţelor dinamice, se utilizează un configurator, care este responsabil cu expunerea deschisă a dependenţelor componentelor. Pentru fiecare componentă se păsteaza lista clienţilor săi (componentele care depind de ea) şi lista componentelor de care depinde ea, ca în figura 2.11.

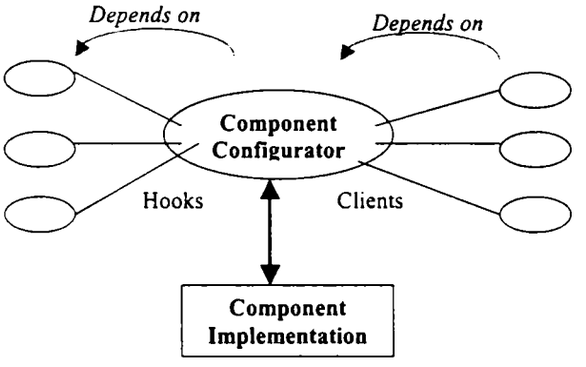


Figura 2.11: Expunerea dependenţelor între componente

Expunerea dependenţelor între componente este un factor favorizant pentru implementarea unui mecanism de reconfigurare automată. Un *framework* de configurarea

BUPT

automată este prezentat de Kon în [KYH+01]. Configurarea automată porneşte de la dependenţele statice existente (*prerequisites*). Se utilizează un format simplu de descriere a prerechizitelor, SPDF, care poate cuprinde cerinţele hardware şi software de instalare a unei componente. Scenariul de configurarea automată propus de Kon ([KYH+01]) este prezentat în figura 2.12.

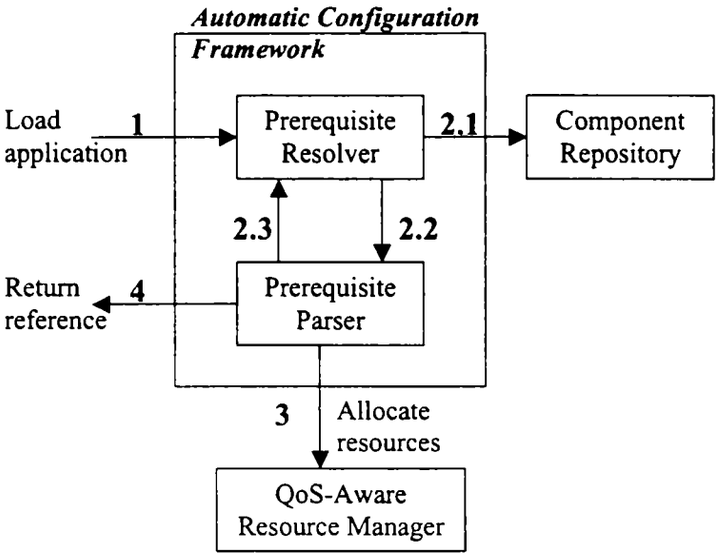


Figura 2.12: Framework de configurare automată

În primul pas, clientul trimite o cerere de a încărca o anumită aplicaţie, specificând numele componentei principale a respectivei aplicaţii. Această cerere este recepţionată de `PrerequisiteResolver`, care aduce codul componentei cerute împreună cu specificăţia prerechizitelor sale din repository.
Subsistemul `PrerequisiteResolver` apelează `PrerequisiteParser` pentru a procesa specificaţia prerechizitelor componentei curente. Pe măsură ce aceste prerechizite sunt procesate, Parserul apelează Resolverul pentru a încărca componentele cerute de prerechizite.

Scenariul de configurare este simplificat de faptul că se consideră dependenţe directe între componente, utilizarea unei componente se face încărcând toate componentele de care depinde aceasta, care rezultă imediat sub forma unui arbore de dependenţe. Nu se pune deloc problema arhitecturii sistemului.

Acelaşi mecanism de reificare a dependenţelor stă şi la baza unor soluţi de adaptare dinamică. Kon şi colectivul propun *dynamicTAO* ([KYH+01], [KC00]), un *Object Request Broker* (ORB) care suportă adaptarea dinamică prin modificarea din mers a strategiilor utilizate pentru diferite categorii de funcţii ale ORB cum sunt mul-

BUPT

tiplexarea cererilor, scheduling, managementul conexiunilor, concurenţa. Sistemul *dynamicTAO* este dezvoltat pe baza lui *TAO* ([SC99], [SLC99]) un ORB flexibil, extensibil şi configurabil. Strategiile utilizate de *TAO* pentru implementarea unor aspecte de concurenţă, scheduling etc sunt specificate cu ajutorul unui fişier de configurare citit la pornirea ORB. *TAO* a fost destinat pentru aplicaţii statice de tip *hard real time*, în care după configurarea iniţiala a ORB, strategiile ramân aceleaşi pe toată durata execuţiei, deci nu apare nevoia pentru reconfigurare din mers. Sistemul *DynamicTAO* e o extensie a TAO care permite reconfigurarea din mers a strategiilor utilizate. Arhitectura poate fi utilizată de asemenea şi pentru reconfigurarea dinamică a aplicaţiilor utilizator executate deasupra ORB. Se utilizează configuratori care realizează expunerea dependenţelor în maniera care a fost descrisă mai sus în figura 2.11.

În [SW98] e prezentat un model bazat pe fluxul de acţiuni (*workflow*) pentru aplicaţii distribuite. În acest model, un limbaj de scriptare [RSS98] e utilizat pentru a realiza descrieri de nivel înalt ale schemelor de workflow. Schemele reprezintă structura taskurilor într-o aplicaţie distribuită, în ceea ce priveşte compoziţia taskurilor şi dependenţele între taskuri. Interpretarea termenului de *dependenţă* în lucrările respective este direct legată de modelul de *workflow* utilizat, o dependenţă în acest caz fiind fie o dependenţă de notificare (un task îşi poate începe execuţia doar după ce o dependenţă a sa şi-a terminat execuţia) sau o dependenţă pe fluxul de date (un task are nevoie de anumite date de intrare înainte de a-şi începe execuţia). Modelul amestecă reprezentările fluxului de date, coordonarea, dependenţele între componente. Modelul exprimă dependenţe de workflow între componentele unei anumite aplicaţii, neabordând interacţiunile între componentele aplicaţiei şi serviciile sistemului.

2.4.2 Configurarea pornind de la puncte de variabilitate

Variabilitatea este calitatea unui sistem de a permite modificări sau adaptări la cerinţe (*customization*). Este necesar ca anumite tipuri de variabilitate să fie anticipate şi sistemele să fie construite astfel încât să le permită ([vGBS01]).

O categorie de soluţii pentru controlul variabilităţii permise în generarea automată porneşte de la tipurile de caracteristici (*features*) ale sistemului ţintă. Se face de dinainte o clasificare a categoriilor de proprietăţi, identificând posibilele valori alternative în cadrul acestora.

În [IB96], exprimarea cerinţelor unei aplicaţii e făcută în termenii unor proprietăţi solicitate. Acestea se bazează pe existenţa unui arbore de proprietăţi care specifică

BUPT

setul de proprietăţi recunoscute de sistem. Descendenţii unei proprietăţi din arbore constituie rafinări ale respectivei (clase de) proprietăţi. Frunzele acestui arbore reprezintă de fapt toate alternativele proprietăţilor.

Metode similare de analiză a claselor de proprietăţi sunt şi *diagramele de caracteristici (features-diagramms)* utilizate în *programarea generativă*. Batory ([BO92], [BCRW00], [BG97]) si Czarnecki ([CE99b], [CE99a]) propun metode de programare generativa pentru arhitecturi stratificate (GenVoca) şi care au fost discutata în paragraful 2.3.2.

Caracteristicile specifice unei anumite familii de produse împreună cu punctele lor de variaţie sunt descrise într-un *feature model*. Fiecare posibilă alternativă de proiectare e reprezentată de o caracteristică (*feature*). Caracteristicile sunt organizate în diagrame de caracteristici, care exprima tipul de variabilitate conţinut de spaţiul proiectului.

Un exemplu (din [CE99b]) este prezentat în figura 2.13, pentru modelul unei maşini. Orice maşina are caracteristicile obligatorii: şasiu, motor, sistem de transmisie. La acestea se poate adăuga, opţional, facilitatea de a avea remorcă. Există alternative, cum ar fi transmisie manuală sau automată (alternative exclusive) sau alternative simple (o maşina poate avea un motor electric, cu benzină sau ambele. Diagrama din figura 2.13 descrie 12 posibile variante de maşini (numărul combinaţiilor fiind dat de cele două tipuri de transmisie, trei tipuri de motoare şi opţionalitatea unei remorci). În plus, pe lânga constrângerile exprimate în diagrama de caracteristici, se mai pot enunţa şi altele suplimentare. De exemplu, s-ar putea impune constrângerea ca un motor electric sau hibrid necesită o transmisie automatică, ceea ce ar restrânge numarul combinaţiilor valide la opt.

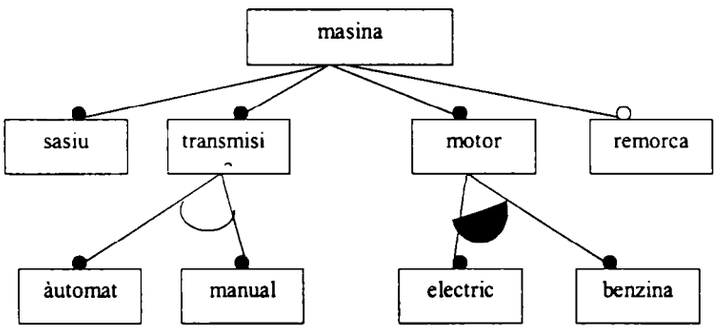


Figura 2.13: Exemplu de *features diagramm*

După identificarea modelului de caracteristici, următorul pas este stabilirea arhitecturii pentru linia de produse. Aceasta implică determinarea tipurilor de com-

BUPT

ponente necesare, a modului cum vor fi interconectate, a interfeţelor categoriilor de componente, etc. Czarnecki utilizează arhitecturi în straturi, tip GenVoca [BO92]. În [CE99b], Czarnecki propune o metodă simplă de determinare automată a arhitecturii GenVoca a unui sistem alcătuit din componente reutilizabile şi care să corespundă unui set de cerinţe.

În [dBvV02b], se propune o tehnică de generare (automată) a unui sistem pornind de la cerinţele funcţionale şi nefuncţionale ale acestuia. Un rol esenţial în cadrul strategiei de generare este deţinut de către un graf "caracterstici-soluţii" (*feature-solution graph*), care reprezintă cunoştiinţe din domeniul aplicaţiei în forma caracteristici dorite – soluţii care realizează aceste caracteristici (soluţii date sub forma unor tipare arhitecturale şi de proiectare). Acest graf defineşte fragmente de soluţii pentru rezolvarea fiecărei cerinţe.

2.4.3 Configurarea pe baza unui skeleton

O metodă de a controla variabilitatea structurii unui sistem, fie că este vorba de adaptare dinamică sau de generare automată, este impunerea unui *skeleton* generic. Arhitectura sistemului (*skeleton*-ul) este definită ca şi un şablon de compoziţie de tipuri de componente. Tipul unei componente include interfeţele care definesc dependenţele de context şi interfeţele pe care componenta le exportă (serviciile). În şablonul de compoziţie, locurile unde pot fi conectate instanţe specifice ale componentelor sunt rezervate prin menţionarea tipurilor de componente. Se face astfel o separare între structura arhitecturală a compoziţiei de componente şi instanţele componentelor care fac parte din implementare. Implementarea sistemului rezultă ca o specializare a unui şablon de componente prin selectarea câte unei instanţe corespunzătoare pentru fiecare tip de componentă. De cele mai multe ori, această metodă se bazează pe ipoteza că fiecare categorie de caracteristici (*feature*) a sistemului este realizată de un anumit tip de componentă, ceea ce permite reconfigurarea sistemului în funcţie de cerinţele aplicaţiei. Pot exista mai multe instanţe pentru fiecare tip de componentă, fiecare instanţă implementând diferite protocoale sau algoritmi cu diferite proprietăţi de calitate a serviciului.

Figura 2.14 prezintă schematizat aceasta situaţie. În exemplul din figură, arhitectura defineşte patru tipuri de componente şi relaţiile dintre ele, iar cerinţele actuale ale aplicaţiei vor fi utilizate pentru alegerea corespunzătoare a câte unei instanţe pentru fiecare tip de componentă. În figură s-a reprezentat situaţia în care se alege implementarea $I1B$ pentru tipul $CT1$, implementarea $I2A$ pentru tipul $CT2$, respectiv $I3C$ şi $I4B$ pentru $CT3$ şi $CT4$.

BUPT

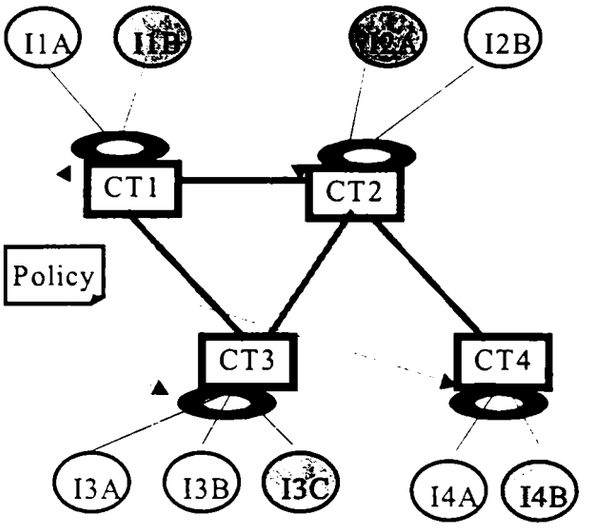


Figura 2.14: Exemplu de skeleton al unui sistem

În [TJJ00] se propune un model pentru reconfigurarea dinamica a *Object Request Brokers (ORB)*. Modelul acesta este bazat pe o arhitectură reflectivă, din componente, care permite integrarea flexibilă a capacităţii de a realiza diferite cerinţe nefuncţionale (security, real-time, tranzacţii, etc). ORB îşi adaptează implementarea în funcţie de preferinţele aplicaţiilor cu privire la diferite aspecte nonfuncţionale. (Cerinţele nonfuncţionale sunt cerinţe care nu sunt direct incluse în funcţionalitatea aplicaţiei - "ce face aplicaţia" - ci mai degrabă exprimă caracteristici suplimentare pe care ar trebui să le aibă aplicaţia). Procesul de reconfigurare dinamică este bazat pe găsirea concordanţelor dintre *policy* şi descriptorii de componente [JTMJ00]. Policy-urile si descriptorii de componente sunt definite într-un limbaj specific de exprimare a cerinţelor nonfuncţionale şi sunt interpretate în timpul execuţiei ORB, ducând la selectarea componentelor adecvate.

In [dBvV02b], [dBvV02a] se descrie o modalitate de compunere a unui sistem prin rafinări succesive. Se porneşte de la descrieri prin UCM (Use Case Maps) ale sistemului şi de la cerinţele funcţionale şi nefuncţionale ale acestuia. UCM conţin puncte de variabilitate denumite *stubs*, în care pot fi inserate alte componente *plug-in*. Fiecare *plug-in* la rândul său poate conţine *stubs*, în care se pot conecta diferite alte componente *plug-in*. Componentele pot fi inserate în sistem doar în punctele indicate de *stubs*. *Stubs*-urile şi *plug-in*-urile sunt caracterizate de tipuri, plasarea unui plug-in într-un stub se poate face doar dacă tipurile lor sunt compatibile. Există plug-in obligatorii, opţionale sau extensii.

Mecanisme asemănătoare de skeleton-uri pentru controlul structurii sistemului

BUPT

care se compune sunt întâlnite şi în [FP02], care utilizează componente configurabile cu puncte de interconectare şi în [PLV97], care propune un *Adaptive Configuration Pattern* care deconectează structura compoziţională a unui sistem de implementările modulelor sale.

În soluţiile de programare generativă a lui Batory ([BO92], [BCRW00], [BG97]) şi Czarnecki ([CE99b], [CE99a]), prezentate în paragraful 2.4.2, sistemele generate pornesc în mod implicit de la un skeleton liniar, subînţeles datorită arhitecturii GenVoca acceptate.

Mediul de configurare automată Aster (discutat în paragraful 2.3.2) porneşte construcţia sistemului şi de la o magistrală sistem primitivă existentă (*base middleware*), care are rolul de skeleton al structurii.

2.5 Concluzii. Obiectivele tezei

2.5.1 Concluzii

Noţiunea de componentă software este utilizată foarte frecvent şi în diferite contexte în literatura de specialitate. Ceea ce se prezintă sub denumirea de componente îmbracă forme, funcţionalităţi şi caracteristici diferite, de aici rezultând un mare număr de definiţii pentru ce este o componentă. În această lucrare noţiunea de componentă va fi cea definită de Szyperski (în [Szy99]) ca fiind o unitate de compoziţie independentă cu interfeţe specificate în mod contractual şi dependenţe explicite de context. De asemenea, în această lucrare se va utiliza în continuare noţiunea de componentă descrisă în termeni corespunzători *nivelului arhitecturii software*, componentele fiind totodată şi abstracţii arhitecturale.

Un aspect esenţial pentru a permite o utilizare corectă a componentelor în compoziţii, este o *specificaţie* bună a părţilor care se compun. Ori de câte ori componentele sunt asamblate, trebuie să existe un *contract* între acestea asupra termenilor colaborării. Un contract trebuie să specifice care sunt *serviciile furnizate* de către componentă şi care sunt *condiţiile cerute* pentru a putea funcţiona la parametrii contractaţi. Contractele trebuie să cuprindă mai multe aspecte. Este nevoie de contracte care să specifice nu doar tipurile componentelor implicate, ci şi contracte semantice şi de calitate a serviciilor oferite.

Modurile de specificare a componentelor pot utiliza metode din limbaje de descriere a interfeţelor, clauze *requires-provides*, limbaje de descriere arhitecturală, iar pentru descrierea comportamentală a componentelor, automate finite. O problemă care rezultă este creşterea complexităţii specificaţiilor. In plus, după cum arată Shaw

BUPT

în [Sha96], în mod intrinsec specificaţia unei componente arhitecturale este în cele mai multe cazuri *incompletă* (proiectantul componentei nu poate anticipa toate aspectele care i-ar putea interesa pe utilizatorii componentei) deci specificaţia trebuie să fie *extensibilă* (se pot descoperi noi tipuri de dependenţe în încercarea de a refolosi împreună componente dezvoltate independent).

O specificare adecvată este esenţială în realizarea unei compoziţii corecte. O bună specificare este şi una din premisele posibilităţii de a realiza *automatizări* ale unor procese de manipulare a componentelor - *verificări* ale corectitudinii unei compoziţii sau *compunerea automată*.

Verificarea corectitudinii compoziţiei poate fi făcută la diverse niveluri de complexitate şi prin diferite metode, de la o verificare de genul verificare de tipuri (*type-checking*) la verificarea semanticii compoziţiei şi la verificarea comportării compoziţiei. Fiecare metodă de verificare se bazează pe o metodă adecvată de specificare a componentelor.

Problema generării automate a unui sistem pornind de la descrierea proprietăţilor dorite ale sistemului este în general o problemă complexă. Generarea automată se poate referi atât la nivelul compunerii şi generării de cod sursă (programarea generativă), cât şi la compunerea şi integrarea de componente în mod dinamic. Deşi cele două situaţii sunt diferite, strategiile şi algoritmii de compoziţie se pot inspira reciproc. Deosebirile pornesc din scopul diferit al compunerii şi generării de cod sursă (programării generative) şi al compunerii de aplicaţii din componente: în general, în programarea generativă componentele au fost special proiectate pentru a fi utilizate într-o anumită "linie de produse", pe când compoziţia aplicaţiilor din componente trebuie să facă faţă situaţiilor care pot apare cu un set de componente "gata făcute" aflate la dispoziţie, dintre care trebuie găsite soluţiile.

Compoziţia unui sistem presupune selectarea componentelor adecvate si combinarea lor într-o structură adecvată pentru formarea sistemului dorit. Dată fiind complexitatea problemei, generatoarele cunoscute din literatură se limitează la a da soluţii pentru o structură impusă în care se determină doar componentele utilizate. Arhitecturile în straturi (*layered*, GenVoca) sunt foarte des utilizate ca şi structuri pentru compoziţia automată. De asemenea, se poate porni de la o structură de bază a sistemului, care apoi este completat cu componente pentru adaptarea sistemului la cerere (*customization*). Generarea automată a unui sistem nu se poate face pornind doar de la specificarea cerinţelor, întotdeauna trebuind să existe, sub diverse forme, restricţii, presupuneri iniţiale, constrângeri.

Pentru a realiza operaţii automate de verificare şi generare de compoziţii de componente, este importantă cunoaşterea şi gestionarea corectă şi completă a *dependen-*

BUPT

ţelor între componente. Această problemă apare şi în cazul arhitecturilor dinamice şi auto-adaptive. De asemenea, pentru a putea suporta configurarea dinamică, este necesară o reprezentare explicită a arhitecturii sistemului printr-un model arhitectural explicit rezidând pe platforma implementării.

Compunerea automată sau dinamică de componente poate fi folosită pentru realizarea sistemelor autoadaptive. Sistemele autoadaptive trebuie să fie capabile să reacţioneze la schimbări în mediu, la variaţii în disponibilitatea resurselor, adaptându-şi în mod dinamic comportamentul, înlocuind sau schimbând anumite componente din alcătuirea lor. Cercetările curente în domeniu au dus la realizări notabile în ceea ce priveşte metodele de realizare a *infrastructurilor* necesare suportării unui asemenea grad de dinamism. Mai puţin investigate au fost mecanismele prin care se poate automatiza luarea *deciziei* configurării. În majoritatea proiectelor în care configurarea unui sistem se face automat, pe baza unor policy-uri stabilite, configurarea are în general un caracter de variabilitate limitată, iar multe abordări descriu deciziile de configurare în termeni strâns legaţi de domeniul aplicaţiei.

2.5.2 Obiectivele tezei

Lucrarea de faţă propune un *model compoziţional pentru sisteme software cu arhitectură multi-flux.* Spre deosebire de alte abordări, care leagă foarte puternic un model compoziţional de un domeniu de aplicaţie, în această teză problema este abordată la *nivel arhitectural.* Problema compoziţiei este tratată pentru o întreagă clasă de sisteme, aparţinând unor domenii de aplicaţie oarecare, dar având acelaşi stil al arhitecturii lor software.

Un model compoziţional trebuie să cuprindă:

- o schemă şi un formalism de descriere a componentelor
- o strategie de compoziţie bazată pe proprietăţile cerute ale sistemului ţintă

Existenţa unui asemenea model este utilă în orice situaţie de dezvoltare bazată pe componente, dar este esenţială şi indispensabilă pentru realizarea de *unelte* care să sprijine activitatea de compunere a unui sistem, deoarece orice *automatizare* a unor operaţii de decizie necesită un *model sistematic.*

Se va ilustra utilizarea modelului propus în:

- generarea automată a unui sistem pornind de la proprietăţile dorite ale acestuia, în unelte implementând strategii de compoziţie automată incorporate în sisteme autoadaptive.

BUPT

- compunerea interactivă, în unelte de compoziţie vizuală cu verificări automate a corectitudinii

Principala provocare în realizarea unui asemenea model o constituie obiectivul de a permite compoziţii corecte neanticipate ca şi structură sau tipuri de componente. Pentru aceasta, trebuie stabilite metode adecvate de descriere a componentelor şi a constrângerilor pentru compoziţie, şi strategii de generare care să poată fi implementate prin program.

Obiectivele ştiinţifice ale acestei teze se pot rezuma astfel:

1. Realizarea la cerere de *configuraţii de compoziţie ne-anticipate* (ca şi structură a compoziţiei şi tipuri de componente participante). Acest lucru este util pentru realizarea unor sisteme open-adaptive (în timpul execuţiei să poată fi introduse noi comportamente şi scenarii de adaptare).

   - Problema cea mai dificilă de cercetare care se ridică aici este găsirea unui echilibru între necesitatea de a permite cât mai multe variaţii neanticipate ale structurii unui sistem şi necesitatea de a menţine identitatea şi corectitudinea sa. Un obiectiv important al acestei teze este îmbunătăţirea mecanismelor actuale de management al variabilităţii unui sistem (care au fost discutate în secţiunea 2.4).
   - Pentru creşterea flexibilităţii în ceea ce priveşte compoziţia, este de preferat ca dependenţele între componente să fie exprimate indirect, în termeni de servicii pe care anumite componente le cer să fie disponibile, evitând impunerea de dependenţe explicite ale unei componente de alte componente. Eliminarea dependenţelor explicite între componente este, după cum a rezultat şi din studiul literaturii, un pas esenţial înspre creşterea flexibilităţii. Abordarea propusă în această lucrare se bazează pe considerarea de *dependenţe anonime* între componente, exprimate în mod indirect prin intermediul unor proprieteti cerute/furnizate, care duc la dependenţe stabilite din mers.

2. Definirea unei scheme de descriere pentru componente care să fie adecvată manipulării cu ajutorul unor unelte de dezvoltare sau de către programe. Pentru a realiza acest lucru, componentele trebuie descrise ca triplete *descriere - reprezentare - implementare.*

   - Descrierea unei componente specifică interfaţa ei de interacţiune şi contractele pe care le impune şi pe care trebuie să le respecte aceasta. De-

BUPT

scrierea componentelor trebuie făcută în mod consistent cu o schemă acceptată de descriere. În literatură au fost prezentate diferite variante de descrieri de componente. Obiectivul acestei lucrări este de a defini o schemă de descriere a componentelor care *să furnizeze toate informaţiile necesare uneltelor automate* de lucru cu componentele - verificatoare şi generatoare.

- Pentru folosirea uneltelor automate, este necesară şi o *descriere a constrângerilor* impuse pentru compoziţia componentelor. Acest domeniu a fost mai puţin abordat în literatură. Pentru o utilizare simplă, se recomandă o schemă unitară pentru descrierea atât a componentelor cât şi a constrângerilor.
- Un alt obiectiv al acestei lucrări este promovarea unui *formalism redus al descrierilor*. Descrierea completă a tuturor aspectelor de interfaţă, funcţionale şi proprietăţilor non-funcţionale poate deveni o sarcină mult mai complexă decât implementarea componentei. Tendinţa de creştere a complexităţii specificaţiilor are ca şi consecinţă şi complicarea strategiilor de determinare a potrivirilor între două componente. Pe de altă parte, metodele complexe de descriere comportamentală a componentelor sunt insuficiente pentru a constitui o bază în unelte de generare automată: după cum s-a arătat este posibil să se declare compatibile componente cu semantici foarte diferite, dar a căror comportament e descris de automate compatibile. Descrierea doar comportamentală a componentelor nu poate fi utilizată pentru compoziţia automată. Ca şi puncte de plecare pentru metoda de descriere a componentelor propusă în această lucrare se utilizează clauzele requires-provides (discutate în paragraful 2.2.2), şi metoda propusă de Shaw pentru descrierea proprietăţilor componentelor arhitecturale (discutată în paragraful 2.2.1).
- În contextul creşterii dimensiunii şi complexităţii software-ului, modelul compoziţional propus trebuie să poată face faţă acestor probleme. Pentru a atinge obiectivul de *control al complexităţii*, modelul de componente definit este unul ierarhic, în care compoziţia are loc prin rafinări succesive.

3. Definirea şi implementarea unei *strategii de compoziţie* care să poată fi implementată prin program. Compoziţia automată poate servi în cazul *sistemelor software auto-configurabile*. La acestea, deciziile de compoziţie sunt luate de către o strategie implementată în sistem sub forma unei căutări automate. Din literatură se poate vedea că s-au desfăşurat cercetări în acest domeniu în special

BUPT

în ceea ce priveşte infrastructura necesară arhitecturilor şi compoziţiei dinamice, dar s-au investigat mai puţin aspectele decizionale ale problemei de compoziţie.

- Problema compoziţiei unui sistem din componente este următoarea: Fiind dată o mulţime de componente disponibile C şi o mulţime de proprietăţi P, se cere să se construiască din componentele disponible un sistem care să satisfacă proprietăţile din mulţimea P. Construcţia sistemului implică două acţiuni: determinarea selecţiei submulţimii componentelor necesare din C şi determinarea conexiunilor între acestea. Pentru a putea implementa în mod *automat* acest proces de decizie, trebuie formulate reguli precise care stau la baza lui.
- În cadrul sistemelor auto-configurante, intervenţia operatorului este de dorit să fie minimă şi să se exprime doar în termenii proprietăţilor dorite ale sistemului: operatorul nu trebuie să fie obligat să intervină direct cu decizii în procesul de compoziţie a sistemului în termeni de componente şi conexiuni.
- Pentru suportul compoziţiei automate este necesară o bună cunoaştere şi reprezentare a dependenţelor între componente, precum si o formă de exprimare a unor constrângeri structurale. Acestea trebuie să se facă în concordanţă cu primul obiectiv enumerat, al păstrării posibilităţilor de a realiza compoziţii neanticipate.

4. Oferirea unei soluţii cu caracter mai general decât al altor modele compoziţionale prezentate în literatură, marea majoritate a acestora fiind dedicate unui anumit domeniu de aplicaţie.

   - Această lucrare îşi propune să dezvolte un model compoziţional specific unui anumit *stil arhitectural*.
   - Modelul propus îşi realizează independenţa de domeniul de aplicaţie prin faptul că în descrierile de componente şi strategia de compoziţie lucrează cu proprietăţi abstracte (*semantic-unaware*).

Modelul compoziţional propus în această teză este necesar pentru asigurarea unui cadru sistematic de compoziţie a sistemelor din componente în arhitecturile multi-flux, pentru domenii de aplicaţie diferite.

BUPT

Capitolul 3

Schemă şi formalism de descriere a componentelor compozabile în arhitecturi multi-flux

3.1 Model de componente

3.1.1 Definirea conceptelor de bază ale modelului

În această lucrare se susţine ideea că un model compoziţional trebuie abordat la nivelul arhitectural. O metodă de compoziţie porneşte de la acceptarea unui anumit stil arhitectural respectat atât de către sistemul compus cât şi de către componente. Abordarea la acest nivel arhitectural permite o bună portabilitate a modelului compoziţional pentru diferite domenii de aplicaţie în care există sisteme ce sunt caracterizate de acelaşi stil arhitectural. Legarea abordării compoziţiei componentelor de nivelul arhitectural este o ideea susţinută în comunitatea ştiinţifică din domeniu [Ham02, IT02a, IT02b, Wil01, Wil03, IS01, KI00] deoarece permite stăpânirea complexităţii problemei şi elimină riscurile de nepotrivire arhitecturală (*architectural mismatch*) [GAO95] la compoziţie.

Conceptele şi terminologia utilizate sunt în concordanţă cu lucrări reprezentative din literatura de specialitate ([BBB+00], [Szy97]) şi care au fost prezentate mai pe larg în secţiunea 2.1. Acest paragraf scoate în evidenţa particularităţile şi restricţiile aplicate acestor concepte, aşa cum sunt ele utilizate în această lucrare.

Componentă

O *componentă* încapsulează implementarea unor anumite funcţionalităţi. Aceste funcţionalităţi sunt garantate numai în condiţiile respectării de către client a unui anumit *contract* specificat de componentă. Componenta este o entitate arhitecturală,

BUPT

care trebuie să respecte caracteristicile *modelului de componente.*

Model de componente

Modelul de componente utilizat în cadrul lucrării presupune că interacţiunile între componente au loc prin intermediul unor *porturi.* Fiecare componentă poate realiza una sau mai multe transformări funcţionale ale intrărilor sale, producând una sau mai multe ieşiri.

Contracte

La specificarea unei componente, se definesc care sunt serviciile furnizate şi funcţionalităţile implementate de componentă şi în ce condiţii pot fi acestea furnizate. Contractele sunt definite în termeni de *proprietăţi furnizate* (*provides*) şi *proprietăţi cerute* (*requires*) pentru a fi indeplinite ca precondiţii.

Proprietate a componentei

In modelul considerat în cadrul acestei teze, componentele sunt descrise prin intermediul proprietăţilor lor. Noţiunea de proprietate este una largă, în accepţiunea definiţiei din [HMSW02]: "ceva ce este cunoscut si constatabil despre o componentă". În modelul propus, o proprietate este exprimată printr-un nume simbolic. Proprietăţile sunt descrise cu ajutorul unor nume simbolice aparţinând unui vocabular convenţional şi extensibil.

Port

Porturile sunt "puncte de interacţiune logică între componentă şi mediu" ([AG97]). In modelul propus, orice componentă are cel puţin un port de intrare şi un port de ieşire, prin care are loc schimbul de date cu mediul. Intrările şi ieşirile sunt de tip flux de date. Din punct de vedere arhitectural, modelul presupune o compatibilitate de tip pentru interconectarea între orice port de intrare şi orice port de ieşire.

Flux

Un flux defineşte o traiectorie de date de la intrarea în sistem şi pană la ieşirea din sistem. Un flux are părţi care sunt interne unor componente precum şi părţi situate între componente. Un flux este o relaţie stabilită de traseele de date între perechi de porturi.

BUPT

Arhitectura multi-flux

În această lucrare se investighează sisteme aparţinând stilului arhitectural *multi-flux*, o variantă a arhitecturii clasice *pipes-and-filters*.

In stilul arhitectural pipes-and-filters ([GS94], [Gar01]) fiecare componentă are un set de intrări şi un set de ieşiri. Componenta citeşte fluxuri de date la intrări şi produce fluxuri de date la ieşiri. Aceasta se realizează prin aplicarea unei transformări specifice fluxurilor de intrare, de obicei printr-o prelucrare incrementală, astfel încat fluxul de ieşire începe să fie produs pe măsură ce se citeşte fluxul de intrare, de unde denumirea de "filtre" (*filters*) pentru componente şi de "conducte" (*pipes*) pentru conectorii care au rolul de a transmite ieşirea de la un filtru la intrarea următorului filtru.

O caracteristică importantă a acestui stil arhitectural este că filtrele trebuie să fie entităţi independente unele de altele, să nu işi partajeze starea cu alte filtre. De asemenea, modelul de comunicaţie este unul *anonim*, în sensul că un filtru nu cunoaşte identitatea filtrelor la care îşi are conectate intrarile şi ieşirile. La specificarea unui filtru se pot face precizări restrictive asupra datelor pe care poate să le primească la intrări şi a datelor pe care le furnizează la ieşirile sale, dar nu se pot identifica componentele conectate la intrările, respectiv ieşirile sale.

Sistemele pipes-and-filters permit proiectantului să conceapă comportamentul unui sistem ca sumă a comportamentelor tuturor filtrelor care intră în compoziţia sistemului. Un asemenea stil permite o bună reutilizare a componentelor, în sensul că oricare două filtre pot fi conectate, cu condiţia ca datele schimbate între ele să fie acceptate de ambele filtre. Asemenea sisteme sunt uşor de întreţinut şi dezvoltat, deoarece pot fi adaugate oricând filtre noi la sisteme existente sau se pot înlocui filtre vechi cu altele mai performante. Asemenea sisteme facilitează analiza unor proprietăţi precum capacitatea de trecere (*throughput*) şi analiza pentru determinarea situaţiilor de blocare (*deadlock*). Stilul pipes-and-filters permite execuţia concurentă, de multe ori fiecare filtru putând fi implementat ca un task separat şi executat în paralel cu alte filtre.

Cel mai cunoscut şi mai simplu exemplu de arhitectură pipes-and-filters sunt aplicaţiile scrise în scripturile shell Unix. Acesta oferă o notaţie pentru conectarea componentelor (care sunt procese Unix) şi un mecanism run-time pentru construcţia pipe-urilor. Arhitecturi pipes-and-filters apar în domenii ca prelucrarea semnalelor, compilatoare tradiţionale, sisteme distribuite. Printre domeniile de aplicaţii investigate în această lucrare sunt procesarea semnalelor şi transmiterea datelor în stiva de protocoale de reţea.

BUPT

Arhitectura sistemului *multi-flux* este definită de relaţiile de flux de date. Fluxurile de date prin sistem sunt fixe, iar componentele trebuie "potrivite" peste acestea. Pot exista mai multe variante de compoziţii de componente care să poată fi potrivite pe fluxurile de date fixe, sistemul rezultat alcătuind aceeaşi funcţionalitate. Pentru fiecare componentă, trebuie cunoscute fluxurile interne astfel încât componenta să poată fi integrată în sistem.

In figura 3.1 este prezentat un exemplu de sistem multi-flux. Acesta conţine patru fluxuri (de la $In1$ la $Out1$, de la $In1$ la $Out2$, de la $In2$ la $Out2$, de la $In2$ la $Out3$) şi pe acestea integrează trei componente $C1$, $C2$, $C3$.

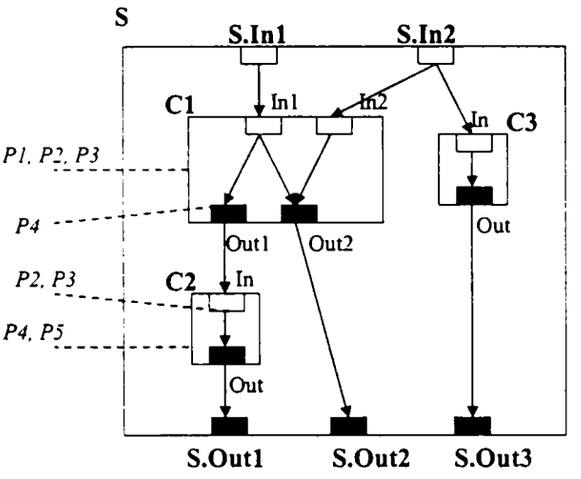


Figura 3.1: Exemplu arhitectura multi-flux

Se introduce cu această ocazie şi *notaţia grafică utilizată* în cadrul acestei lucrări. O componentă este reprezentată printr-un dreptunghi, având porturi reprezentate ca şi puncte de conexiune pe frontiera sa. Porturile de intrare sunt reprezentate sub formă de dreptunghiuri goale iar porturile de ieşire sunt reprezentate ca şi dreptunghiuri pline. Proprietătile furnizate de o componentă sunt asociate componentei prin notaţia cu linie punctată. Proprietăţile cerute sunt asociate porturilor printr-o notaţie cu linie punctată. În exemplul din figura 3.1 proprietăţile $P1$, $P2$, $P3$ sunt furnizate de componenta $C1$. Proprietatea $P4$ este cerută la portul $Out1$ al componentei $C1$.

3.1.2 Relaţii ierarhice între componente

Modelul propus promovează relaţii ierarhice între componente. Există componente *simple* (atomice) şi componente *compuse*. O componentă simplă este unitatea de bază de compoziţie, în timp ce componentele compuse apar ca şi un mecanism de

BUPT

grupare şi abstractizare.

O *componentă simplă* are un singur port de intrare şi un singur port de ieşire. Intre aceste porturi există implicit un flux de date direct. O *componentă compusă* e definită de o structură internă care e o compoziţie de componente. O componentă compusă poate avea N porturi de intrare şi M porturi de ieşire. Structura internă a unei componente compuse conţine un numar de fluxuri, care definesc relaţiile între perechi de porturi care îi aparţin. Stilul arhitectural de tip multi-flux e valabil şi în interiorul componentelor compuse. De fapt, orice sistem poate fi văzut ca şi o componentă compusă.

Componentele compuse sunt componente "normale" (*first-class*), în sensul ca au o identitate proprie, dată de faptul că au propriile proprietăţi, contracte proprii, şi fluxuri interne fixate. O componentă compusă, ca un întreg, este întotdeauna definită de un set de proprietăţi proprii. Relaţia dintre proprietăţile componentei compuse şi proprietăţile subcomponentelor este discutată în paragraful 3.2.

Structura internă a unei componente compuse, de cele mai multe ori, nu este fixă. Aceste componente sunt compozabile, în limitele unor constrângeri care vor fi precizate în paragraful 3.2.

3.1.3 Contracte exprimate cu ajutorul proprietăţilor

Contractele între componente sunt exprimate cu ajutorul proprietăţilor *furnizate* (*provides*), respectiv *cerute* ca precondiţii (*requires*) de către componente.

O proprietate, în modelul nostru, este caracterizată de un nume (identificator) simbolic, care face parte dintr-un vocabular bine cunoscut. Proprietăţile pot avea şi atribute valorice sau subproprietăţi care să le detalieze.

Proprietăţile furnizate sunt asociate componentei ca un întreg, descriind serviciile oferite de aceasta. In cazul componentelor compuse, se pot specifica şi proprietăţi furnizate de porturi, reflectând structura internă a componentei.

Proprietăţile cerute sunt asociate întotdeauna cu porturi.

Un contract pentru o componentă este respectat daca toate proprietăţile cerute de el işi gasesc *potriviri* între proprietăţile furnizate de componentele cu care aceasta este conectată.

O potrivire între o proprietate cerută şi o proprietate furnizată este constatată pe baza următoarelor criterii: în primul rând potrivirea numelui proprietaţii, apoi potrivirile eventualelor atribute. Un avantaj care decurge din acest mecanism de realizare a potrivirilor este că sistemul de compoziţie precum şi componentele nu trebuie să conştientizeze semantica proprietăţilor utilizate în descrieri. Prin aceasta,

BUPT

exprimarea regulilor de compoziţie este simplificată pentru că nu depinde de domeniul problemei.

Orice potrivire între cererea şi furnizarea unei proprietăţi trebuie considerată în cadrul *aceluiaşi flux*. Cerinţele asociate cu un port de intrare $Cx.In_y$ se adreseaza componentelor care au porturi de ieşire conectate la fluxul care intră în $Cx.In_y$. Analog, cerinţele asociate cu un port de ieşire $Cx.Out_y$ se adresează componentelor care au porturi de intrare conectate la fluxurile care ies din $Cx.Out_y$. În figura 3.2, componenta CX are următoarele cerinţe: cerinţele *pa* şi *pb*, asociate portului său de intrare In_y şi cerinţele *pd* şi *pe* asociate portului său de ieşire Out_y. Cerinţele *pa* şi *pb* sunt furnizate de către componente care se află pe fluxul care intră în portul In_y, iar *pd* şi *pe* sunt furnizate de componente care se află pe fluxul care iese din portul Out_y.

Dependenţele între componente, exprimate în acest fel, au o natura anonimă: niciodată o componentă nu este descrisă ca depinzând de o alta componentă, ci doar de un set de proprietăţi. In exemplul din figura 3.2, componenta CX depinde de proprietăţile *pa*, *pb*, *pd* şi *pe*. Componenta client nu face distincţia tipului sau identităţii componentelor care îi furnizează proprietăţile cerute de ea. Orice componentă care furnizează aceste proprietăţi poate fi utilizată în compoziţie pentru a asigura dependenţele de care are nevoie CX.

În mod implicit, se considera că o cerinţă pusă de un port se adresează unei componente *oarecare* de pe fluxul extern incident. În exemplul din figura 3.2, cerinţele puse de componenta CX nu au fost furnizate toate de către componentele imediat adiacente lui CX pe flux. Se pot specifica şi cerinţe aparţinând unei categorii mai restrictive, a cerinţelor imediate. O cerinţă imediată trebuie să fie realizată, să işi găsească potrivirea, prin componenta imediat învecinată.

În exemplul din figura 3.1, componenta $C1$ furnizează proprietăţile $P1$, $P2$, $P3$. Componenta $C2$ furnizează proprietăţile $P4$ şi $P5$. Componenta $C1$ cere la portul său de ieşire $Out1$ proprietatea $P4$; aceasta este furnizată de către componenta $C2$, conectată la fluxul care iese din $C1.Out1$. Portul $C2.In$ cere proprietăţile $P2$ şi $P3$, care sunt furnizate de către componenta $C1$.

Cerinţe negative. O cerinţă poate fi de forma *NOT proprietate*. Aceasta înseamnă că proprietatea specificată trebuie să lipsească de pe fluxul respectiv. Direcţia în care se adresează cerinţa este întotdeauna dinspre portul de unde este pusă cerinţa înspre exterior (în susul fluxului daca e pusă la un port de intrare sau în josul fluxului dacă e pusă la un port de ieşire).

Cerinţe pereche. Fie componentele $C1$ şi $C2$, unde $C1$ furnizează proprietatea $pr1$ şi $C2$ furnizează proprietatea $pr2$. Componentele au porturile lor $C1.P1$ şi $C2.P2$.

BUPT

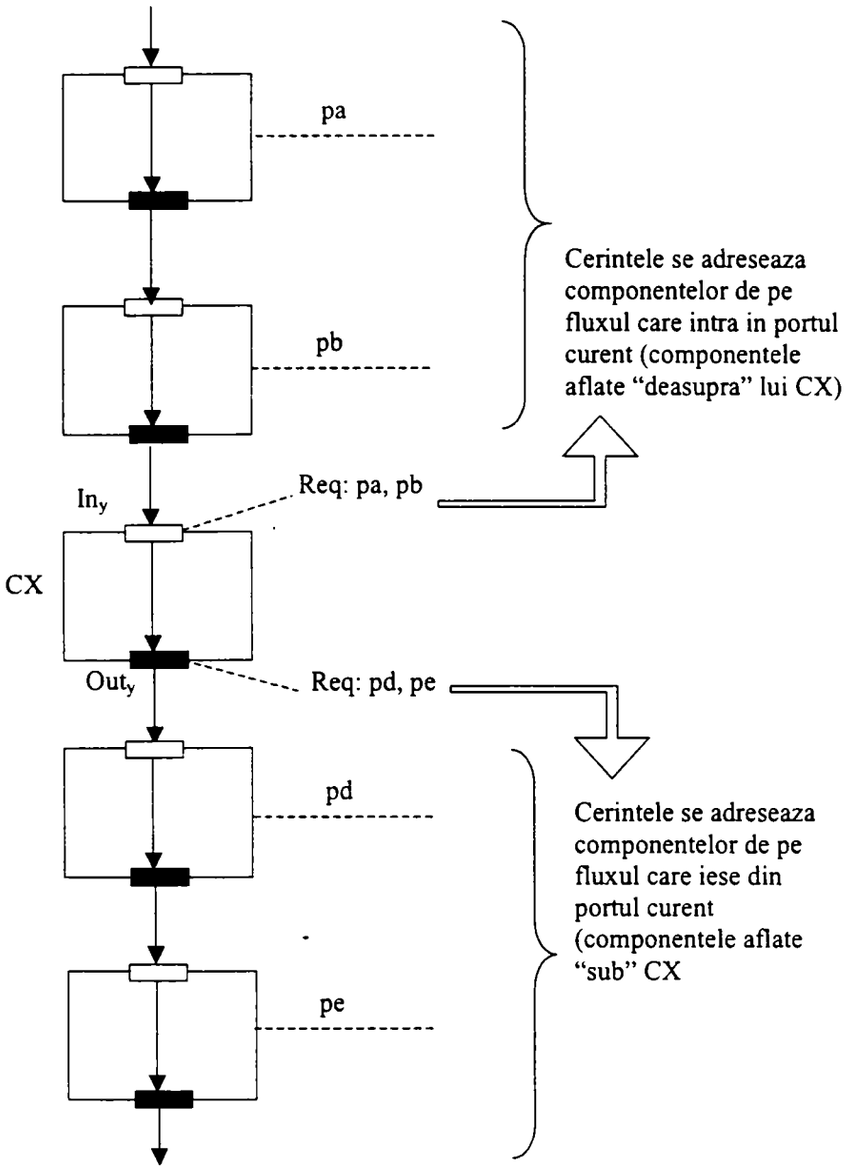


Figura 3.2: Cerinţe exprimate prin proprietăţi

La portul $C1.P1$ se impune cerinta $pr2$, *pereche cu* $pr1$, şi la portul $C2.P2$ se impune cerinţa $pr1$ pereche cu $pr2$. Se poate deci realiza o conexiune de la portul $P1$ al lui $C1$ spre portul $P2$ al lui $C2$. Se remarcă faptul că relaţia de pereche se stabileşte între proprietatea $pr1$ a lui $C1$ şi o proprietate $pr2$, nu între componente ($C1$ nu este pereche cu $C2$, ci cu "cineva" care furnizează proprietatea $pr2$). Particularitatea introdusă de declararea explicită a acestui tip de cerinţe ca şi pereche, constă în faptul că se introduc reguli privitoare la aranjarea acestora în compoziţii: dacă în general nu contează ordinea relativă în care sunt îndeplinite cerinţele acumulate pe un flux,

BUPT

în cazul particular al certinţelor pereche se impune restricţia ca perechile de cerinţe *sa nu se intersecteze.*

Se consideră sistemele din figura 3.3. In primul caz, componentele CA, CB, $CA*$ şi $CB*$ au urmatoarele contracte: CA furnizează proprietatea A şi are ca cerinţă la portul său de ieşire proprietatea $A*$. Componenta CB furnizează proprietatea B şi are ca cerinţă proprietatea $B*$. $CA*$ furnizează proprietatea $A*$ şi are ca cerinţă la portul său de intrare proprietatea A. Componenta $CB*$ furnizează proprietatea $B*$ şi are ca cerinţă proprietatea B. Chiar dacă componentele CA şi $CA*$, respectiv CB şi $CB*$ se solicită reciproc, cerinţele lor nu au fost declarate ca şi pereche. O compoziţie validă, care respectă toate contractele cerute, este cea din figură (primul exemplu).

Dacă se definesc componentele CAP, CBP, $CAP*$ şi $CBP*$ cu aceleaşi proprietăţi furnizate, dar cu cerinţele declarate explicit ca fiind de tip pereche, o compoziţie ca şi cea anterioară nu ar fi validă, pentru ca perechile de cerinţe se intersectează. Compoziţii valide, cu aceste cerinţe , ar putea fi ultimele 2 cazuri din figura 3.3, pentru ca perechile $A - A*$ şi $B - B*$ nu se intersectează.

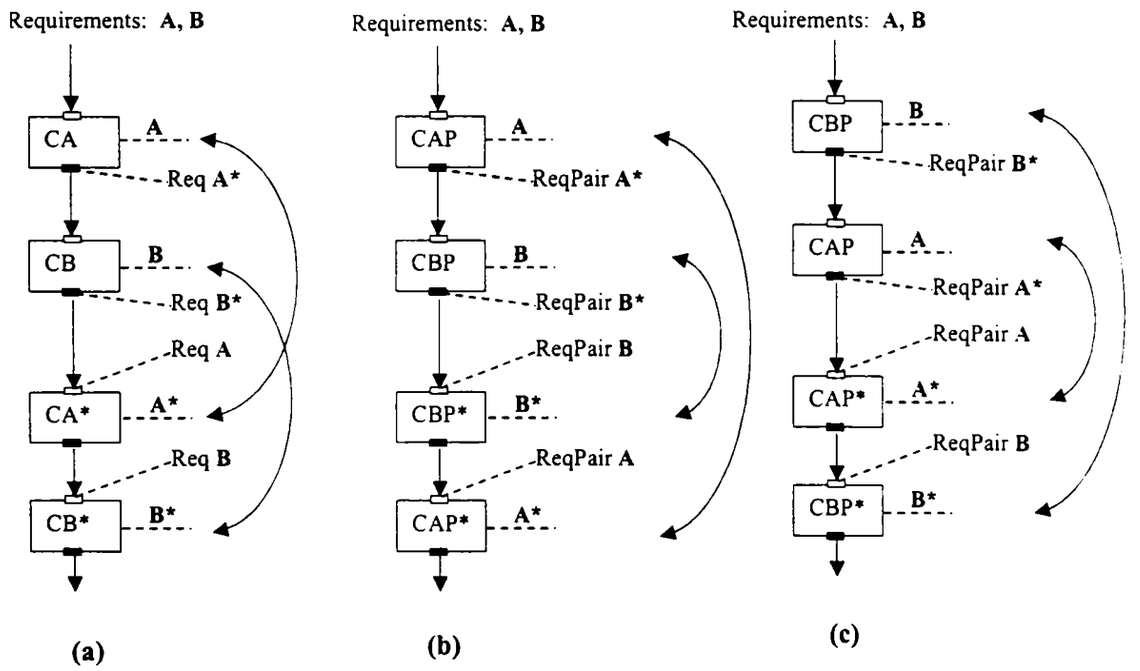


Figura 3.3: Cerinţe pereche

Se consideră următorul exemplu: fie o mulţime de componente care realizează funcţii de criptare, decriptare, compresie, decompresie. In specificarea acestor componente se vor utiliza cerinţele pereche. O succesiune reprezentând o compoziţie validă este una din variantele : *CRYPTO - COMPRESS- DECOMPRESS -*

BUPT

DECRYPTO sau *COMPRESS - CRYPTO - DECRYPTO - DECOMPRESS*. O combinaţie de tipul *CRYPTO - COMPRESS - DECRYPTO - DECOMPRESS*, care nu ar asigura integritatea transformărilor, este invalidă deoarece nu respectă restricţia impusă de cerinţele pereche.

Rafinarea prin subproprietăţi. După cum s-a prezentat anterior, proprietăţile sunt identificate prin nume. Uneori nu este suficient acest lucru, caz în care proprietăţile pot fi în continuare rafinate prin atribute sau subproprietăţi.

Atributele sunt un set de valori ataşate unei proprietăţi. De exemplu, considerăm componente care realizează fragmentarea – defragmentarea unui flux de date în pachete de o anumită dimensiune. O componentă `fragmentator` furnizează o proprietate `fragmentare`. Pentru a fi pe deplin specificată, această proprietate trebuie însoţită de un atribut `dimensiune-fragment`, care este o valoare numerică, reprezentând un parametru ce stabileşte dimensiunea pachetelor create de fragmentator.

Proprietăţile pot fi rafinate prin subproprietăţi. De exemplu, se poate defini o componentă *COMPRESS*, ce realizează compresia datelor sale de intrare. Compresia poate fi realizată prin diferiţi algoritmi, daţi ca şi subproprietăţi ale proprietăţii compresie, după cum se prezintă în figura 3.4.

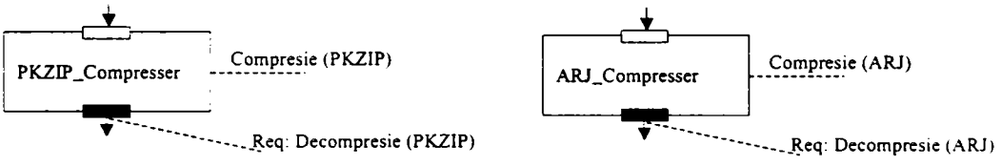


Figura 3.4: Subproprietăţi

Dacă apare o cerinţă nedetaliată referitoare la realizarea compresiei datelor, se poate utiliza orice componentă care furnizează proprietatea compresie. Va fi aleasă în mod arbitrar o rafinare a acesteia cu o anumită subproprietate. Dacă cerinţa este mai specifică (de exemplu, compresie cu PKZIP) se va alege o componentă care realizează exact aceasta proprietate.

Dacă o proprietate care are rafinări în formă de atribute sau subproprietăţi este definită într-o *pereche*, o potrivire înseamnă că şi atributele sau subproprietăţile lor trebuie să se potrivească. De exemplu, proprietatea compresie CU PKZIP constituie o potrivire doar cu proprietatea decompresie CU PKZIP, şi nu şi cu decompresie cu ARJ.

După cum se arată în exemplul din figura 3.5, în cazul proprietăţilor pereche nu mai este necesar să se specifice explicit ca şi cerinţă faptul că tipul compresiei

BUPT

şi decompresiei trebuie să fie acelaşi, pentru că acest lucru va fi verificat implicit, datorită declaraţiei proprietătilor ca şi pereche.

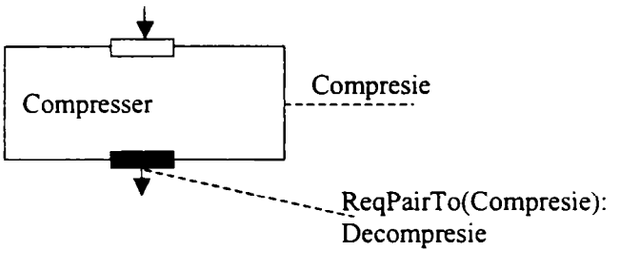


Figura 3.5: Proprietăţi pereche cu subproprietăţi

De multe ori, subproprietăţile rezultă în cazul componentelor compuse şi sunt o modalitate prin care "se scurg" informaţii despre proprietăţile subcomponentelor.

3.2 Componente compozabile

Relaţiile ierarhice între componente oferă o posibilitate de structurare şi permit un reglaj fin al compoziţiilor. O particularitate a modelului propus în aceasta lucrare este faptul că o componentă compusă nu are întotdeauna o structură internă fixă. O astfel de componentă se numeşte *componentă compozabilă*. În aceasta rezidă o parte importantă a capacităţii de configurare: configuraţia exactă a structurii interne a unei componente poate fi compusă în mod dinamic, ca rezultat al unor cerinţe externe, realizând un reglaj fin al proprietăţilor şi al funcţionalităţii componentei.

În paragraful anterior s-a menţionat exemplul unei componente *COMPRESSER* care realizează compresia datelor de pe fluxul său de intrare. Structura internă a acestei componente poate varia, în funcţie de metoda de compresie aleasă. Aceste variaţii nu se referă doar la utilizarea şi înlocuirea unei componente de tip `AlgoCompresie` cu diverse variante de implementare a unor algoritmi de compresie. Variaţia poate fi şi una structurală, implicând utilizarea unor tipuri diferite de componente. În exemplul dat în figura 3.6 se prezintă variante de realizare a componentei *COMPRESSER*, (a) printr-o metodă de compresie statică, implicând o analiză prealabilă a sursei, respectiv (b) printr-o metodă adaptivă, caz în care din structura componentei lipseşte partea de analiză a sursei.

Dupa cum s-a menţionat in paragraful 3.1.2, componentele compuse sunt entităţi independente, cu identitate proprie. Această identitate proprie trebuie păstrată aceeaşi, independent de modul în care se realizează compunerea structurii interne a componentei. In paragraful 3.1.2, au fost menţionate ca şi elemente definitorii ale

BUPT

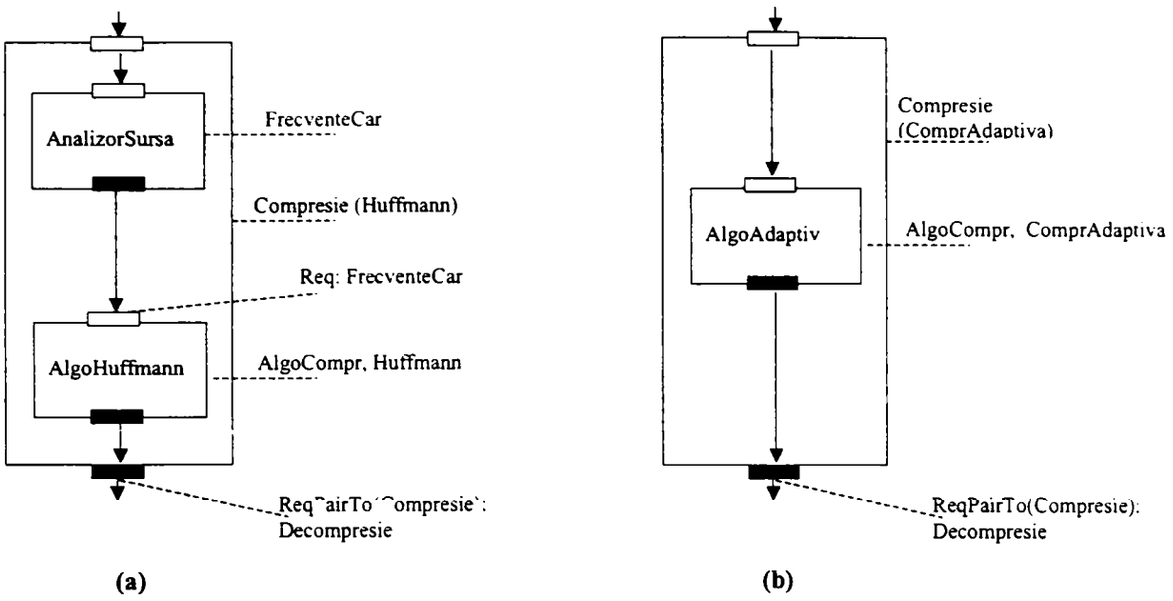


Figura 3.6: Exemplu de componentă compozabilă - COMPRESSER

identităţii unei componente compuse interfaţa cu porturile sale, proprietăţile furnizate şi cerute de componentă şi fluxurile interne. Aceste proprietăţi ale componentei compuse sunt de cele mai multe ori expresia, la un nivel de abstractizare mai înalt, a facilităţilor caştigate prin compoziţia subcomponentelor. Vocabularul folosit la descrierea proprietăţilor unei componente compuse este distinct faţă de vocabularul folosit pentru descrierea proprietăţilor subcomponentelor. Definirea acestor abstracţii de vocabular trebuie făcută de proiectantul componentei compuse. Proprietăţile subcomponentelor sunt întotdeauna cauza proprietăţilor componentei compuse, dar de multe ori proprietăţile componentei compuse nu pot fi exprimate sau deduse matematic sau cantitativ din proprietăţile subcomponentelor. Este vorba în special de proprietăţile funcţionale ale componentei compuse, care sunt exprimate în termenii unor abstracţii de nivel înalt, definite de proiectantul componentei. Multe proprietăţi sunt proprietăţi emergente compoziţiei, caracterizând comportarea globală a compozitiei şi rezultă din combinarea proprietăţilor mai multor subcomponente. De exemplu, considerăm o componentă *robot*. Aceasta este compusa din *braţ-stâng*, *braţ-drept* şi *unitate-comandă*. Faptul că asamblarea acestora rezultă într-un *robot* nu este ceva care se poate deduce sau calcula, este pur şi simplu o abstractizare de un nivel superior şi este un fapt definit de către proiectantul componentei compuse *robot*.

Pentru componenta *COMPRESSER* din exemplul dat în figura 3.6, aceasta poate avea diferite structuri interne, în funcţie de tipul de algoritm utilizat pentru

BUPT

compresie.

Pentru a asigura păstrarea identităţii componentei în cazul diverselor variante de compunere trebuie să existe anumite constrângeri structurale referitoare la structura internă a componentei compozabile. Aceste limitari ale variabilităţii sunt indispensabile pentru a putea garanta corectitudinea compoziţiei. Rezultă de aici existenţa a două cerinţe contradictorii: trebuie să existe un mecanism de exprimare a variabilităţii structurale care, pe de o parte, să permită cât mai multe variante neanticipate de configuraţii, iar pe de altă parte să exprime constrângeri suficiente pentru garantarea corectitudinii şi predictibilităţii compoziţiei. Găsirea unei soluţii optime din punctul de vedere al acestor două cerinţe contradictorii este o problemă de cercetare actuală. Paragrafele următoare prezintă o soluţie proprie la această problemă.

3.2.1 Necesitatea definirii unei noi metode de management a variabilităţii

După cum s-a prezentat în secţiunea 2.3.3, reconfigurarea unui sistem bazat pe componente se poate face ca structură sau ca şi componente. Soluţiile clasice pentru definirea variabilităţii unei configuraţii, descrise în literatură şi discutate în secţiunea 2.3.3, sunt utilizarea de *skeleton*-uri sau puncte de variabilitate.

Aceste tehnici limitează atât setul de proprietăţi care poate fi luat în considerare în procesul de configurare/compoziţie a unui sistem, cât şi tipurile de variaţii structurale posibile.

Utilizarea unui *skeleton* limitează posibilităţile de variaţii structurale, atât din punct de vedere al topologiei structurii cât şi din punct de vedere al tipurilor de componente care pot fi utilizate.

Metodele care se bazează pe diagrame de caracteristici (*features-diagramms*) implică faptul că proprietăţile şi clasificarea lor trebuie să fie cunoscute de dinainte. Metodele înglobează mecanisme care permit modificarea destul de simplă a setului de proprietăţi şi a categoriilor lor, dar această modificare trebuie făcută înainte de începerea procesului de configurare. O metodă bazată pe diagrame de caracteristici este adecvată pentru instanţierea membrilor unei familii de produse prin metoda programării generative, dar introduce limitări dacă s-ar prelua variantele bazate pe modele de proprietăţi pentru configurarea dinamica (la *run-time*) a unui sistem.

Utilizarea de skeleton-uri sau puncte de variabilitate nu reuşeşte să asigure un grad suficient de ridicat de variabilitate neanticipată, deoarece limitează următoarele aspecte importante:

- se limitează posibilităţile de variaţie a structurii, din punctul de vedere al topolo-

BUPT

giei şi a tipurilor de componente aşteptate.

- se limitează posibiliăţile de a descoperi şi utiliza în mod dinamic noi tipuri de componente
- se limitează setul de proprietăţi care pot fi luate în considerare în procesul de configurare.

Este necesar ca un mecanism de configurare dinamică să permită:

- modificarea într-un grad cât mai mare a structurii unei compoziţii, cu deschidere pentru structuri neanticipate. De exemplu, în figura 3.6 s-au prezentat două variante ale structurii unei componente compozabile *COMPRESSER*. Structura lor este diferită: dacă se utilizează un algoritm de compresie statică, COMPRESSER trebuie să conţină o componentă care face în prealabil analiza fluxului de intrare, apoi o componentă care implementează un algoritm static de compresie. Daca în schimb se utilizează o variantă de compresie dinamică, nu mai apare necesitatea utilizării unui analizor. Structura internă a componentei *COMPRESSER* nu pote fi descrisă printr-o constructie de tip *skeleton* cu locuri de inserţie a componentelor, de genul "loc pentru o componenta analizor" şi "loc pentru o componentă algoritm de compresie".
- ca tipurile de componente implicate să poată fi descoperite şi folosite în mod dinamic. Se include aici, pe de o parte, necesitatea ca să poată descoperi componente noi implementând tipuri cunoscute (în modelul propus, furnizând proprietăţi cunoscute), dar şi necesitatea de a putea descoperi şi utiliza noi tipuri de componente. Revenind la exemplul cu componenta compozabilă COMPRESSER, aceasta implică posibilitatea de a descoperi în mod dinamic noi componente care implementează şi alţi algoritmi de compresie. Mai mult, trebuie lăsată deschisă şi posibilitatea de a descoperi noi tipuri de componente – de exemplu, un nou tip de componentă care realizează un algoritm de compresie static şi care include şi funcţionalitatea de analiză a sursei.
- ca soluţia să fie independentă de domeniul aplicaţiei. Proprietăţile cerute sistemului nu trebuie să se încadreze neapărat într-o clasificare cunoscuta de dinainte.

Aceste criterii sunt îndeplinite de soluţia de control a variabilităţii propusă în această teză. În modelul dezvoltat în această teză, se consideră proprietăţile necantitative şi ne-calculabile. În acest caz, proprietăţile unei compoziţii, ale unui

BUPT

ansamblu de componente, sunt caracteristici abstracte ale acestuia, exprimate utilizând alt nivel de abstractizare în descriere. Constrângerile structurale propuse în această lucrare sunt un mijloc de a asigura o compoziţie corectă, predictibilă. Paragraful următor prezintă metoda propusş de control a variabilităţii unei compoziţii prin *constrângeri structurale.*

3.2.2 Definirea constrângerilor structurale ale unei componente compozabile

Pentru a permite mai multă flexibilitate decât în soluţiile clasice de tipul skeleton sau puncte de variaţie, în modelul dezvoltat în această teză se propune o soluţie bazată pe definirea de *constrângeri structurale* pentru o componentă compozabilă.

În principiu, rolul constrângerilor structurale pentru o componentă (un sistem) compozabilă este acela de a stabili elementele esenţiale ale structurii acesteia. Acestea nu sunt nici o descriere structurală completă şi nici un skeleton al structurii. Constrângerile structurale definesc un set de *cerinţe minime* pe care trebuie să le îndeplinească structura unei componente pentru a-i asigura acesteia o identitate precizată.

Constrângerile structurale ale componentelor compozabile cuprind: constrângeri structurale elementare, cerinţe structurale dependente de context şi dependenţe interflux.

Constrângerile structurale elementare descriu proprietăţile minimale care trebuie să fie prezente într-o structură de fluxuri, pentru ca proprietăţile furnizate ale componentei compuse să poată fi îndeplinite. Constrângerile structurale elementare:

- definesc care sunt fluxurile interne ale componentei
- precizează ce proprietăţi minimale trebuie să fie prezente pe fiecare flux
- pot stabili relaţii de ordine între proprietăţi.

Constrângerile structurale elementare trebuie specificate de cel care dezvoltă componenta compusă.

Figura 3.7 prezintă un exemplu de constrângeri structurale elementare pentru o componentă compozabilă C1. Aceste constrângeri arată că proprietatea FP1 trebuie să se gasească pe fluxul $In1 \rightarrow Out1$, proprietatea FP2 pe fluxul $In1 \rightarrow Out2$ şi proprietăţile FP3 şi FP4 pe fluxul $In2 \rightarrow Out2$, cu FP3 "deasupra" lui FP4.

Constrângerile structurale permit ca structura componentei să fie stabilită în mod dinamic, cu cât mai puţine limitări. O componentă compozabilă definită prin con-

BUPT

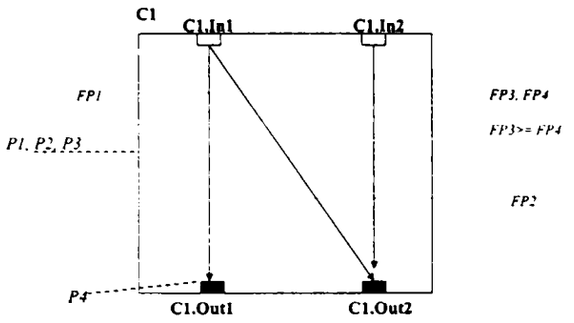


Figura 3.7: Exemplu - Constrângeri structurale pentru componenta compozabilă C1

strângerile ei structurale poate avea un număr nelimitat şi neanticipat de realizări concrete.

În figura 3.8 se prezintă două configuraţii posibile pentru structura internă a componentei $C1$, ambele configuraţii în concordanţă cu constrângerile structurale definite în figura 3.7.

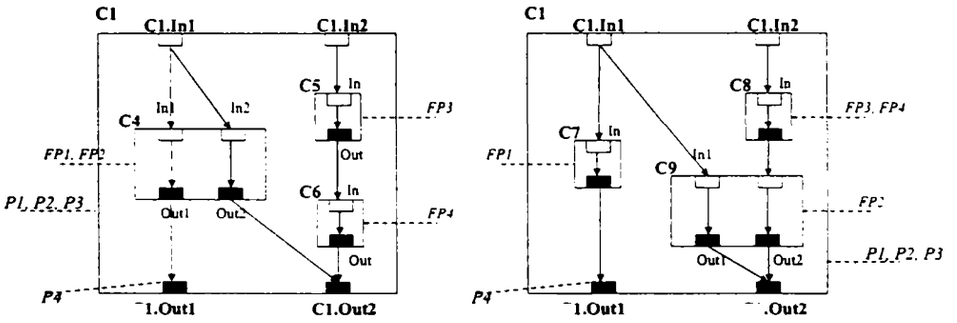


Figura 3.8: Exemplu - configuraţii interne pentru componenta compozabilă C1

Cerinţele structurale dependente de context exprimă cerinţele care se referă la alte componente folosite în cazul acesta ca şi subcomponente. Dacă o anumită componentă S poate fi utilizată ca şi subcomponentă în interiorul unei componente compuse C, şi dacă constrângerile structurale elementare ale lui C nu pot identifica singure rolul şi maniera corectă de integrare a lui S în interiorul lui C, atunci trebuie adaugate specificaţii suplimentare. Acestea se adaugă la constrângerile structurale ale lui C sub forma cerinţelor structurale dependente de context. Ele sunt adaugate de cel care dezvoltă subcomponenta S.

Cerinţele structurale dependente de context se exprimă în aceeaşi termeni ca şi constrângerile structurale elementare, adică în termeni de proprietăţi şi relaţii de ordine între acestea. Nu se pot defini fluxuri noi în cadrul acestei categorii. Există o deosebire importantă între interpretarea cerinţelor dependente de context şi

BUPT

a constrângerilor structurale: constrângerile structurale enumeră proprietăţi absolut indispensabile a fi îndeplinite pentru ca să existe componenta compusă. De exemplu, acestea precizează relaţii de felul "Pentru a exista, componenta X trebuie să conţină proprietatea p pe fluxul $f1$". Cerinţele dependente de context descriu situaţii de genul "dacă proprietatea q va fi utilizată in interiorul componentei X, q poate fi plasată numai pe fluxul $f1$ sub proprietatea p". Acest enunţ nu implică deloc obligaţia ca proprietatea q să fie prezentă în structura componentei X; se precizează doar care ar putea să fie locul lui q dacă ar apărea de undeva cererea ca proprietatea q să existe în interiorul lui X.

Dependenţele inter-flux specifică relaţii care pot să existe între fluxuri, de tip relaţii de continuitate logică.

Introducerea constrângerilor structurale este un element de bază în modelul compoziţional propus de aceasta lucrare. Acesta este un mecanism nou de a gestiona variabilitatea unui sistem, lăsând suficiente grade de libertate pentru a permite configurări ale căror detalii nu au fost planificate de dinainte. Configuraţia unei componente compozabile nu e limitată la alternative cunoscute în avans, nici nu se fixează numărul şi tipul componentelor care se pot utiliza.

Revenind la exemplul componentei compozabile $COMPRESSER$, discutat în paragrafele anterioare (figura 3.6), identitatea acesteia este stabilită de constrângerile structurale definite ca în figura 3.9.

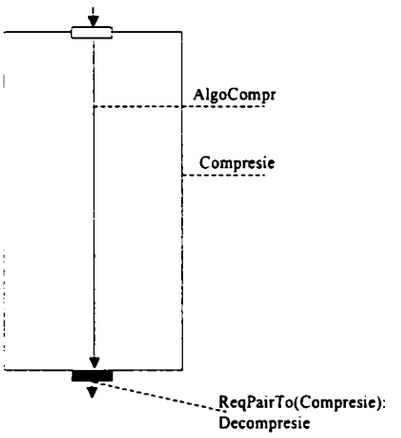


Figura 3.9: Exemplu - Constrângeri structurale elementare pentru componenta compozabilă COMPRESSER

Constrângerile structurale elementare descriu componenta compozabilă $COMPRESSER$ ca pe o entitate cu două porturi, un port de intrare şi unul de ieşire, între care există un singur flux, pe care trebuie să existe proprietatea *AlgoCompr*. Ambele

BUPT

variante de configuraţii structurale pentru componenta COMPRESSER, care au fost prezentate în figura 3.6, satisfac constrângerile structurale de bază. Proprietatea *AlgoCompr* este furnizată de către componenta $AlgoHuffman$ în cazul ilustrat în figura 3.6.a, respectiv de către componenta *AlgoAdaptiv* în cazul 3.6.b. Componenta $AnalizorSursa$, prezentă în cazul (a), este adăugată ca urmare a cerinţelor directe ale componentei $AlgoHuffman$ şi nu pentru că ar fi impusă de constrângerile structurale ale $COMPRESSER$.

Dacă se doreşte completarea funcţionalităţii lui $COMPRESSER$ cu facilitatea de a determina raportul de compresie, aceasta implică măsurarea dimensiunii înainte de comprimare şi compararea cu dimensiunea de dupa comprimare. Există componentele S (furnizează proprietatea $Size$) şi CS (furnizează proprietatea $CompareSize$) care realizează aceste lucruri. Componenta CS are ca şi cerinţa proprie să fie utilizată pe un flux după proprietatea $Size$). Aceste informaţii însă nu sunt suficiente pentru a putea utiliza componentele S şi CS în interiorul lui $COMPRESSER$ în scopul enunţat. Situaţia se rezolvă definind *cerinţe structurale dependente de context* în maniera indicată de figura 3.10.

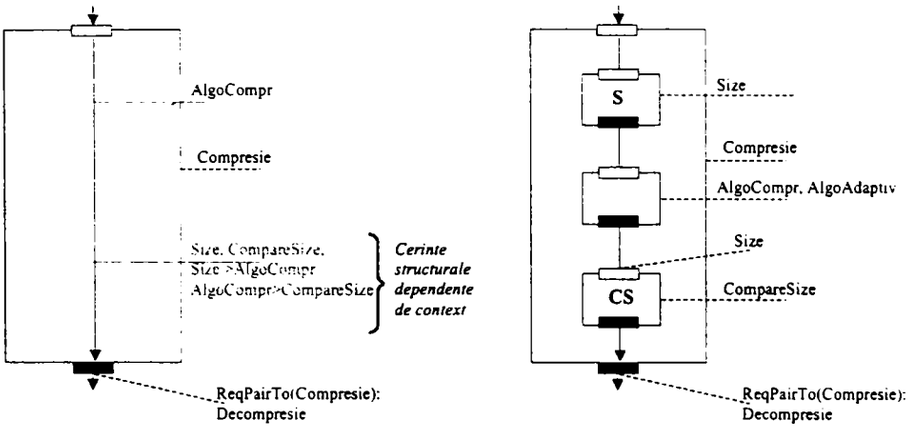


Figura 3.10: Exemplu - Cerinţe structurale dependente de context pentru componenta compozabilă COMPRESSER

Cerinţele structurale dependente de context precizează faptul că pe fluxul intern al componentei compozabile $COMPRESSER$ pot fi prezente proprietăţile $Size$ şi $CompareSize$, cu relaţiile de ordine $Size > AlgoCompr$ şi $AlgoCompr > CompareSize$ (proprietatea $Size$ deasupra proprietăţii $AlgoCompr$ şi $CompareSize$ sub $AlgoCompr$). Cerinţele structurale dependente de context se exprimă întermeni de proprietăţi pe fluxuri şi relaţi ide ordine între proprietăţi, la fel ca şi constrângerile structurale elementare. Spre deosebire de acestea, cerinţele dependente de context au un car-

BUPT

acter opţional, ele se activează doar dacă contextul extern impune prezenţa proprietăţilor specificate de ele. Numai dacă factori externi solicită prezenţa proprietăţii *CompareSize* aceasta va fi adăugată şi cerinţele dependente de context vor fi considerate.

3.3 Limbajul CCDL – Composable Components Description Language

3.3.1 Necesitatea definirii unui nou formalism de descriere

In cadrul cercetărilor efectuate pentru această teză s-a definit şi implementat un formalism pentru specificarea componentelor compozabile, în forma limbajului declarativ **CCDL (Composable Component Description Language)**. **CCDL** este un limbaj de descriere a contractelor componentelor precum şi a constrângerilor structurale ale acestora, în sensul în care acestea au fost definite în paragraful anterior.

Datorită faptului că definirea componentelor compozabile descrise prin constrângeri structurale este o contribuţie originală a acestei teze, **CCDL** este un limbaj de descriere cu caracteristici originale. La nivel conceptual, CCDL prezintă unele elemente care sunt comune şi familiei limbajelor de descriere a interfeţelor (*IDL - Interface Description Languages*) şi altele care sunt asemănatoare familiei limbajelor de descriere arhitecturală (*ADL - Architectural Description Languages*), dar nu poate fi încadrat total în nici una din aceste familii.

Necesitatea introducerii CCDL rezultă din incapacitatea limbajelor aparţinând familiilor limbajelor de descriere a interfeţelor respectiv a limbajelor de descriere arhitecturală de a exprima atât contractele componentelor cât şi constrângeri structurale pentru compoziţia lor.

În mod asemănător limbajelor din familia IDL, CCDL este utilizat pentru specificarea contractelor componentelor. Cele mai cunoscute limbaje din familia IDL nu furnizează suficientă informaţie de descriere a componentelor pentru a putea servi ca şi bază în compoziţia automată. În majoritatea *component framework*-urilor (cum sunt de exemplu COM şi CORBA cu IDL-urile lor) există specificări de tipuri pentru argumentele operaţiilor din interfaţa unei componente. După cum s-a discutat şi în paragraful 2.2.1 referitor la contracte între componente, specificarea tipurilor nu este suficientă, deoarece lipseşte posibilitatea de specificare a semanticii. Specificările formale ([ZW97]), pe de altă parte, sunt complexe şi lipsesc unelte care să uşureze aplicarea lor.

În ceea ce priveşte limbajele de descriere arhitecturală (ADL), acestea sunt uti-

BUPT

lizate la reprezentarea configuraţiilor arhitecturale. Medvidovici şi Taylor menţionează în [MT00] că o condiţie esentială ca un limbaj să facă parte din familia ADL este să poată fi utilizat la reprezentarea componentelor, conectorilor şi configuraţiilor arhitecturale. O configuraţie arhitecturală descrie o structură (topologie) de componente şi conectori. CCDL operează cu noţiunile de componente şi configuraţii arhitecturale, dar în cazul de faţă, rolul şi scopul limbajului de descriere CCDL este diferit de rolul şi scopul ADL. Structura unei componente compozabile descrisă în CCDL trebuie să *nu* fie reprezentată. Configuraţia internă a componentei compozabile trebuie să rămana deschisă. Limbajele din familia ADL nu pot fi utilizate pentru a reprezenta structura internă nedefinită, variabilă, a unei componente compozabile. În CCDL se specifică doar constrângerile structurale pentru configuraţie, aceasta rămânând deschisă pentru compoziţie. Constrângerile structurale descrise în CCDL nu sunt o descriere completă de configuraţie arhitecturală, ci exprimă doar linii directoare flexibile ce vor ajuta la stabilirea compoziţiilor.

O descriere ADL este utilzată de uneltele care realizează o analiză a proprietăţilor structurii arhitecturale. O descriere CCDL este utilizată de o unealtă care generează configuraţia arhitecturală compatibilă cu constrângerile structurale şi care raspunde unui set de cerinţe.

O descriere CCDL poate fi comparată cu o prescripţie arhitecturală. Un limbaj de descriere arhitecturală care poate face deosebire între o *prescripţie arhitecturală* (şablon pentru proiectare) şi o *descriere arhitecturală* (starea sistemului la execuţie) este xADL [DvdHT01]. Puterea prescripţiilor arhitecturale în xADL este redusă, gradul de variabilitate este limitat la a specifica dacă anumite componente sunt opţionale.

Limbajele de descriere a modificărilor arhitecturale (AML) şi limbajele de descriere a constrângerilor arhitecturale (ACL) a căror rol este descris în [OMT98] sunt mai apropiate de obiectivele limbajului CCDL, dar tratează problema arhitecturilor dinamice din punctul de vedere al exprimării operaţiilor de modificare, nu al exprimării de reguli pentru compoziţie.

Prin plaja de noţiuni acoperite de CCDL şi modul de realizare a acestora, o descriere CCDL este conceptual apropiată de ceea ce Shaw defineşte în [Sha96] că trebuie să fie o specificaţie arhitecturală de componentă. După cum s-a discutat şi în paragraful 2.2.1, specificaţiile componentelor arhitecturale sunt caracterizate de următoarele proprietăţi:

- specificaţia este în cele mai multe cazuri *incompletă*: de multe ori proiectantul componentei nu poate anticipa toate aspectele care i-ar putea interesa pe

BUPT

utilizatorii componentei.

- specificaţia trebuie să fie *extensibilă*, deoarece în timp se pot descoperi noi tipuri de dependenţe în încercarea de a refolosi împreună componente dezvoltate independent. O descriere CCDL este extensibilă, pentru că include constrângerile structurale dependente de context, parte opţională a descrierii unei componente compozabile şi care este mereu actualizată.

- specificaţia este *eterogena*, trebuie să descrie toate categoriile de proprietăţi. Modelul componentelor compozabile şi CCDL cuprind atât aspecte de "contract" funcţional şi nefuncţional, cât şi descrieri structurale.

3.3.2 Definiţia limbajului CCDL

Limbajul CCDL, adică formalismul propus în această teză pentru descrierea componentelor compozabile, este definit sub forma unei scheme XML.

Utilizarea facilităţilor XML

XML (EXtensible Markup Language) [W3C98] este o specificaţie dezvoltată de W3C ca metodă de marcare a documentelor conţinând informaţie structurată. Informaţia structurată este alcătuită din conţinut (cuvinte, imagini, etc.) şi indicaţii referitoare la rolul jucat de conţinut. Un limbaj de marcare oferă un mecanism pentru identificarea structurii unui document.

XML este extensibil pentru că nu e un format fix, cum este de exemplu HTML. XML este un metalimbaj - un limbaj pentru descrierea altor limbaje. Această caracteristică a XML permite să fie utilizat cu uşurinţă la crearea de limbaje proprii de descriere pentru orice fel de tipuri de documente. Utilizarea XML s-a extins de la scopul iniţial pentru care a fost creat (Web) şi poate fi utilizat pentru a stoca orice fel de informaţii structurate.

În XML, fragmente de text pot fi delimitate cu ajutorul unor etichete (*tag*) special formatate pentru a marca începutul şi sfârşitul anumitor regiuni de text. Sintaxa unui document XML poate fi definită cu ajutorul unui DTD (*Document Type Definition*) care specifică numele elementelor, atributelor şi entităţilor care pot fi folosite şi cum anume pot fi acestea folosite.

DTD a fost definit pentru utilizarea în documente text clasice, poate servi la descrierea doar a structurii unui document şi nu are un mecanism de definire a conţinutului elementelor în termeni de tipuri de date. Un DTD nu poate fi utilizat pentru a specifica domenii numerice sau pentru a defini limitări sau verificări

BUPT

asupra conţinutului text. Verificările ce se pot face pe baza DTD se referă doar la marcajul care încadrează (structurează) datele.

Rolul XML a evoluat în timp, de la o simplă metodă de marcare a documentelor text, la un mijloc de codificare a datelor şi management al meta-datelor. Deoarece DTD aveau o expresivitate limitată, W3C a propus înlocuirea acestora cu XML Schema [W3C01]. Recomandarea XML Schema furnizează un mijloc de a specifica şi conţinutul elementelor în termeni de tipuri de date, astfel încât proiectantul unui nou tip de document să poată da criterii pentru validarea conţinutului elementelor. XML Schema introduce în plus un sistem de tipuri pentru XML, cu tipuri simple şi compuse, ca şi posibilitatea de a defini tipuri prin moştenire. O Schema este la rândul ei scrisă ca un document XML, spre deosebire de DTD, care aveau un formalism propriu de declarare de sintaxă. Acesta este un avantaj în plus al XML Schema faţă de DTD, pentru că astfel se poate folosi acelaşi software atât pentru prelucrarea declaraţiei de sintaxă (Schema) cât şi pentru prelucrarea instanţelor XML (documentele).

Orice document XML trebuie să fie bine-format (*well-formed*) şi dacă există dată o descriere de sintaxă DTD sau Schema, poate fi verificat şi dacă este valid în raport cu aceasta. Un document e bine format dacă respectă sintaxa XML (în principiu, cele mai importante reguli de sintaxă impun ca orice element care are un tag de start trebuie să aibă şi un tag de sfarşit; elementele nu au voie să fie suprapuse (*overlapped*), ci doar încuibate (*nested*)). Un document e valid daca este bine format şi dacă este conform cu o definiţie de sintaxă dată în formă de DTD sau Schema.

Un DTD sau o XML Schema poate fi utilizat ca şi un meta-limbaj în care se pot defini limbaje de descriere. Există limbaje de descriere arhitecturală([DvdHT01], [DvdHT02]) dezvoltate recent, care sunt definite printr-un DTD sau XML Schema.

Decizia de a implementa CCDL ca şi o *schema XML* simplifică procesul de implementare, în primul rând datorită existenţei unei game largi de unelte, comerciale sau *freeware*, care asigură suportul operaţiilor de bază pentru crearea şi prelucrarea documentelor XML. O asemenea unealtă, folosită în realizarea CCDL, este XMLSpy [Alt02].

XMLSpy este un mediu integrat de dezvoltare pentru XML şi conţine un editor şi verificator pentru documente şi scheme XML. Documentele XML sunt texte ASCII şi pot fi scrise manual şi citite cu orice editor de texte, dar datorită volumului mare de informaţii de marcaj utilizarea lor directă este dificilă, greoaie şi supusă erorilor. De aceea se recomandă utilizarea facilităţilor de editare şi validare oferite de unelte precum XMLSpy. Mediul XMLSpy a fost utilizat la editarea şi verificarea schemei care defineşte limbajul CCDL precum şi la editarea descrierilor de componente CCDL conform acestei scheme.

BUPT

Limbajul de descriere CCDL, definit în această lucrare, necesită de asemenea implementarea unui *parser* adecvat. Acest *parser* trebuiesă fie capabil să interpreteze fişiere de descrieri de componente în formatul CCDL, generănd structurile de date corespunzătoare descrierilor de componente. Din nou, decizia de a alege pentru CCDL implementarea în forma unei scheme XML este susţinută de existenţa bibliotecilor de programare care oferă diverse variante de parsere şi verificatoare pentru XML.

Parserul pentru CCDL implementat în cadrul lucrării utilizează ca bază de plecare parserul XML Apache Xerces [Apa01]. Apache Xerces este o bibliotecă open-source pentru analiza lexicală şi utilizarea documentelor XML. Aceasta implementează un parser şi un validator pentru XML, ca şi interfeţele de programare standard SAX şi DOM. Utilizarea acestor biblioteci a uşurat munca de implementare a parserului pentru CCDL. Parserele XML se referă însă doar la partea de sintaxă, nu poate descrie semantica elementelor sau a relaţiilor dintre ele. Pentru a asigura verificarea semantică a descrierilor CCDL se utilizează metode tradiţionale specifice implementării unui limbaj. Detalii legate de implementarea acestuia se găsesc în paragraful 3.3.3.

Descrierea limbajului CCDL

Descrierea unei componente în CCDL cuprinde una sau două părţi principale: descrierea vederii externe a componentei şi, înmod opţional, descrierea vederii interne dacă este vorba de o componentă compusă. Vederea externă conţine informaţiile despre proprietăţile furnizate spre exterior şi porturile prin care interacţioneaza cu mediul. Vederea internă poate cuprinde fie constrângerile structurale, dacă este vorba de o componentă compozabilă, fie o descriere structurală completă, daca avem o componentă cu o arhitectură internă fixă.

Descrierea unei componente se face conform unei scheme XML definită ca în figura 3.11. Figura prezintă elementul rădacină al schemei.

La rădăcina schemei se află elementul `component`, care este compus din secvenţa de elemente `componentExternals` (element obligatoriu) şi `componentInternals` (element opţional). O componentă este identificată prin numele său, dat ca şi atribut al elementului `component`.

Vederea externă a descrierii unei componente

Elementul `componentExternals` este definit de secvenţa de elemente `provides` şi `port`. Exteriorul componentei este definit de proprietăţile furnizate, indicate sub eticheta `provides`, şi porturile componentei, fiecare port fiind descris sub o etichetă `port`.

BUPT

```
<xs:element name="component" type="componentType"/>

<xs:complexType name="componentType">
    <xs:sequence>
        <xs:element name="componentExternals"
                        type="componentExternalsType"/>
        <xs:element name="componentInternals"
                        type="componentInternalsType" minOccurs="0"/>
    </xs:sequence>
    <xs:attribute name="name" type="xs:string" use="required"/>
</xs:complexType>

<xs:complexType name="componentExternalsType">
    <xs:sequence>
        <xs:element name="provides" type="providesType"/>
        <xs:element name="port" type="portType" maxOccurs="unbounded"/>
    </xs:sequence>
</xs:complexType>

<xs:complexType name="componentInternalsType">
    <xs:sequence>
        <xs:element name="structuralConstraints"
                type="structuralConstraintsType" minOccurs="0"/>
        <xs:element name="structuralDescription"
                type="structuralDescriptionType" minOccurs="0"/>
    </xs:sequence>
</xs:complexType>
```

Figura 3.11: Radacina schemei de definiţie a unei componente

Proprietăţile cu care este descrisă o componentă sunt date de elemente de tip `property`, tip descris ca în figura 3.12.

O proprietate este identificată printr-un nume, specificat ca şi atribut al elementului. Proprietatea poate conţine în mod opţional elemente care să descriu parametrii săi, caracterizaţi de triplete de atribute (nume, tip, valoare). O proprietate poate fi rafinată prin subproprietăţi, încadrate într-un element `with`.

Pentru a simplifica notaţia şi a obţine o reprezentare mai compactă a elementelor schemei şi a relaţiilor dintre acestea, se va utiliza în continuare o notaţie grafică generată de XMLSpy pentru reprezentarea unei scheme. Elementul `componentExternals` este reprezentat grafic în această notaţie în figura 3.13.

BUPT

```
<xs:complexType name="propertyType">
        <xs:sequence>
            <xs:element name="parameter" minOccurs="0"
                                              maxOccurs="unbounded">
                <xs:complexType>
                    <xs:attribute name="name" type="xs:string"/>
                    <xs:attribute name="type" type="xs:string"/>
                    <xs:attribute name="value" type="xs:string"/>
                </xs:complexType>
            </xs:element>
            <xs:element name="with" minOccurs="0">
                <xs:complexType>
                    <xs:sequence>
                        <xs:element name="property" type="propertyType"
                                              maxOccurs="unbounded"/>
                    </xs:sequence>
                </xs:complexType>
            </xs:element>
        </xs:sequence>
        <xs:attribute name="name" type="xs:string" use="required"/>
    </xs:complexType>
```

Figura 3.12: Definirea tipului property

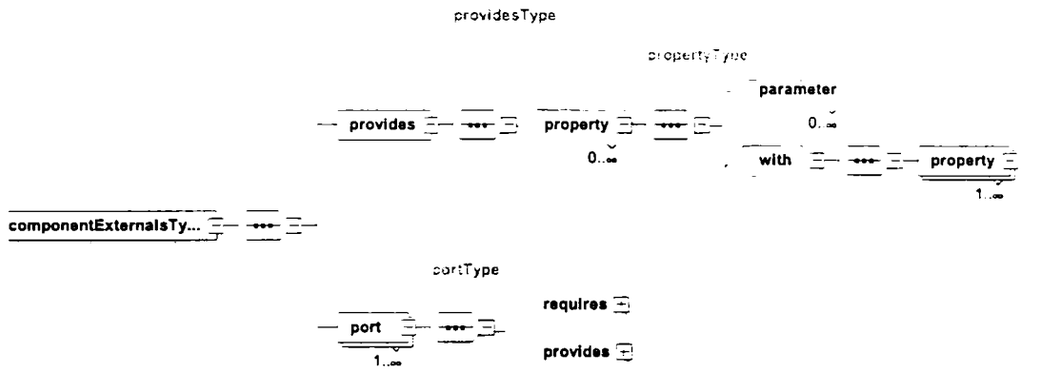


Figura 3.13: Vedere externă a unei componente

Vederea internă a descrierii unei componente compozabile

In figura 3.14 este reprezentat grafic elementul `componentInternals`.

Un element `componentInternals` poate avea ca şi subelemente `structuralConstraints` şi `structuralDescription`. Elementul `structuralConstraints` este obligatoriu, în timp ce `structuralDescription` este opţional.

BUPT

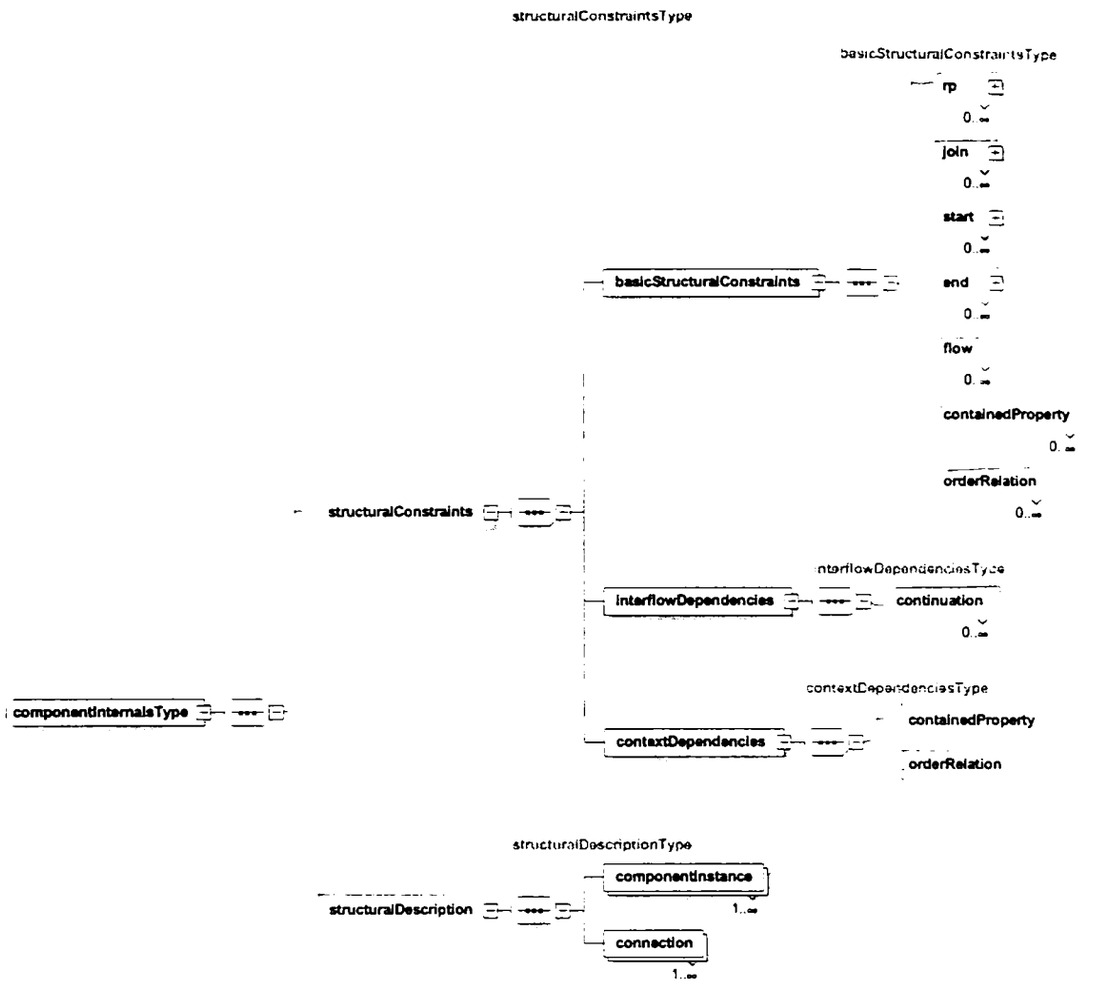


Figura 3.14: Vedere internă a unei componente

Elementul `structuralConstraints` este alcătuit dintr-o secvenţă de trei subelemente, `basicStructuralConstraints`, `interflowDependencies` şi `contextDependencies`. Constrângerile structurale de bază (basic structural constraints) definesc fluxurile interne ale componentei şi proprietăţile care trebuie să fie conţinute pe acestea.

Fluxurile sunt definite de elementele `flow`. Acestea sunt caracterizate prin trei atribute: numele unic prin care este identificat fluxul, şi două atribute (`from` şi `to`) care reprezintă numele nodurilor de conexiune care sunt extremităţile fluxului. Punctele de conexiune trebuie specificate explicit şi ele pot fi de tipurile: puncte de ramificare (*reflection point*), puncte de joncţiune (*join point*), puncte terminale (*end point*), puncte de start (*start point*).

Un punct de ramificare serveşte la transmiterea aceluiaşi flux pe două sau mai multe direcţii. Un punct de ramificare este descris prin elementul `rp`, se identifică

BUPT

printr-un nume unic în cadrul componentei şi are un nod de intrare şi un număr nelimitat de noduri de ieşire. Nodurile de intrare şi ieşire sunt descrise de subelementele `in` şi `out`, fiecare identificat în cadrul punctului de conexiune de atributul său `name`. Un punct de joncţiune serveşte la combinarea a două sau mai multe fluxuri într-unul singur. Este descris prin elementul `jp`, are un număr oarecare de noduri de intrare şi un nod de ieşire. Un punct terminal (descris printr-un element `end`) indică faptul că un flux de date este consumat de componenta căreia îi aparţine.

Pentru descrierea proprietăţilor care trebuie să fie prezente pe un anumit flux, se utilizează elemente `containedProperty`, care specifică proprietatea şi care au ca şi atribut numele fluxului de care aparţin. Elementele `orderRelation` definesc relaţii de ordine între proprietăţi.

Elementul `interflowDependencies` poate conţine elemente `continuation` care indică relaţii de continuare între perechi de fluxuri. Numele fluxurilor implicate în relaţie sunt identificate prin atributele `from` şi `to`.

Elementul `contextDependencies` descrie constrângeri structurale dependente de context, care se exprimă prin elemente `containedProperty` şi `orderRelation`, la fel ca şi în cazul constrângerilor structurale de bază.

Exemplu de componentă descrisă în CCDL

In figura 3.15 este dată ca exemplu descrierea CCDL a componentei $C1$ din figura 3.7.

Componenta este definită sub numele $C1$ (specificat ca şi atribut al elementului rădăcină). Descrierea componentei cuprinde vederea externă, încadrată între *tag*-urile `component_external`, şi vederea internă, încadrată între *tag*-urile `component_internal`.

Vederea externă cuprinde enumerarea proprietăţilor furnizate (între *tag*-urile `provides`) şi enumerarea porturilor de interacţiune (fiecare port fiind introdus de căte un *tag* `port`). Din exemplu se observă că $C1$ furnizează trei proprietăţi, $P1$, $P2$ şi $P3$, definite fără alte atribute. Aceasta înseamnă că atributele au valorile implicite care definesc proprietăţile ca propagabile şi fără subproprietăţi. Componenta are patru porturi, dintre care două porturi de intrare (numite $In1$ şi $In2$) şi două porturi de ieşire (numite $Out1$ şi $Out2$). La portul $Out1$ este asociată cerinţa $P4$.

Partea internă a componentei este descrisă prin constrângerile sale structurale. Se defineşte un punct de ramificare RP_{In1}, pentru a trata ramificarea intrării $In1$, şi un punct de joncţiune JP_{Out2} pentru a trata joncţiunea care apare la $Out2$. Componenta $C1$, după cum se observă din figura 3.7, are trei fluxuri interne: $In1 \rightarrow Out1$, $In1 \rightarrow Out2$ şi $In2 \rightarrow Out2$. In descrierea CCDL fluxurile se pot defini numai între puncte de conexiune, şi anume dintr-un punct de conexiune poate pleca sau intra un singur

BUPT

```
<?xml version="1.0" encoding="UTF-8"?>
<component xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:noNamespaceSchemaLocation="component.xsd"
           name="C1">
    <component_external>
        <provides>
            <property name="P1"/>
            <property name="P2"/>
            <property name="P3"/>
        </provides>
        <port name="In1" type="In"/>
        <port name="In2" type="In"/>
        <port name="Out1" type="Out">
            <requires>
                <required_property name="P4" assertion="yes" pair="false"/>
            </requires>
        </port>
        <port name="Out2" type="Out"/>
    </component_external>
    <component_internal>
        <basic_struct_constr>
            <rp name="RP_In1">
                <in name="RP.In1"/>
                <out name="RP.Out1"/>
                <out name="RP.Out2"/>
            </rp>
            <join name="JP_Out2">
                <in name="JP.In1"/>
                <in name="JP.In2"/>
                <out name="JP.Out"/>
            </join>
            <flow name="dummy1" from="In1" to="RP.In"/>
            <flow name="flow1" from="RP.Out1" to="Out1"/>
            <flow name="flow2" from="RP.Out2" to="JP.In1"/>
            <flow name="flow3" from="In2" to="JP.In2"/>
            <flow name="dummy2" from="JP.Out" to="Out2"/>
            <contained_property name="FP1" flowlocation="flow1"/>
            <contained_property name="FP2" flowlocation="flow2"/>
            <contained_property name="FP3" flowlocation="flow3"/>
            <contained_property name="FP4" flowlocation="flow3"/>
            <order below="FP4" above="FP3" flowlocation="flow3"/>
        </basic_struct_constr>
        <interflow_depend/>
        <context_depend/>
    </component_internal>
</component>
```

Figura 3.15: Exemplu de descriere CCDL

flux (de aceea se utilizează puncte de ramificaţie şi de joncţiune, pentru a multiplica punctele de conexiune acolo unde se întâlnesc mai multe fluxuri). In descrierea CCDL

BUPT

a componentei, în secvenţa de cod din figura 3.15, apar două fluxuri *dummy*, $In1 \rightarrow RP.In1$ (cu rolul de a lega intrarea $In1$ la punctul de ramificaţie) şi $JP.Out \rightarrow Out2$ (cu rolul de a lega punctul de joncţiune la ieşirea $Out2$. Celelalte fluxuri, $flow1$, $flow2$ şi $flow3$ corespund fluxurilor interne ale componentei $C1$. Constrângerile structurale elementare precizează că pe fluxul $flow1$ trebuie să fie prezentă proprietatea $FP1$, pe fluxul $flow2$ proprietatea $FP2$, iar pe fluxul $flow3$ proprietăţile $FP3$ şi $FP4$, cu relaţia de ordine dintre acestea $FP3$ deasupra lui $FP4$, în sensul fluxului care le conţine.

3.3.3 Implementarea parserului de CCDL

Rolul parserului de CCDL este de a prelucra fişiere cu descrieri de componente şi de a produce un obiect de tip `ComponentDescription`.

Descrierea unei componente cuprinde: proprietăţile furnizate de componentă, porturile de interacţiune cu exteriorul, fluxurile interne şi constrângerile structurale referitoare la structura sa internă, exprimate în termeni de fluxuri şi proprietăţi pe fluxuri.

In reprezentarea structurii fluxurilor interne unei componente mai intervine noţiunea de nod de conexiune internă. Un flux poate fi definit numai între două noduri de conexiune internă. Intr-un nod de conexiune internă poate să înceapă sau să se sfârşească un singur flux. Nodurile de conexiuni interne provin din porturile componentei şi din intrările şi ieşirile punctelor de ramificaţie şi punctelor de joncţiune.

Această modalitate de descriere a unei componente se regăseşte în clasa `ComponentDescription` şi relaţiile ei cu clasele reprezentate în figura 3.16.

O descriere de componentă conţine un număr de porturi (lista `ports` de elemente de tip `Port`), de fluxuri (lista `flows` de elemente de tip `Flow`)şi de noduri de conexiune (lista `connection_nodes` de elemente de tip `ConnectionNode`). Fluxurile pot fi definite numai specificându-le originea şi destinaţia sub forma de noduri de conexiune. O descriere de componentă mai conţine şi lista proprietăţilor furnizate (`provides`).

Clasa `Property` reprezintă noţunea de *proprietate* ca element central al descrierilor de componente. Orice proprietate este identificată printr-un nume, poate avea parametri valorici şi o listă de subproprietăţi. Lista de subproprietăţi este lista `refining_properties`. De asemenea, unei proprietăţi i se pot asocia relaţii de ordine (`below`, `above`) cu alte proprietăţi aflate pe acelaşi flux. Proprietăţile pot să apară la toate nivelurile, fiind asociate atât componentelor, cât şi porturilor şi fluxurilor. Un port (clasa `Port`) are o listă de proprietăţi cerute (`requires`) şi o listă de proprietăţi furnizate (`provides`). Un flux (clasa `Flow`) are o listă de proprietăţi (`contained_properties`) care trebuie să fie conţinute pe acel flux. Fluxu-

BUPT

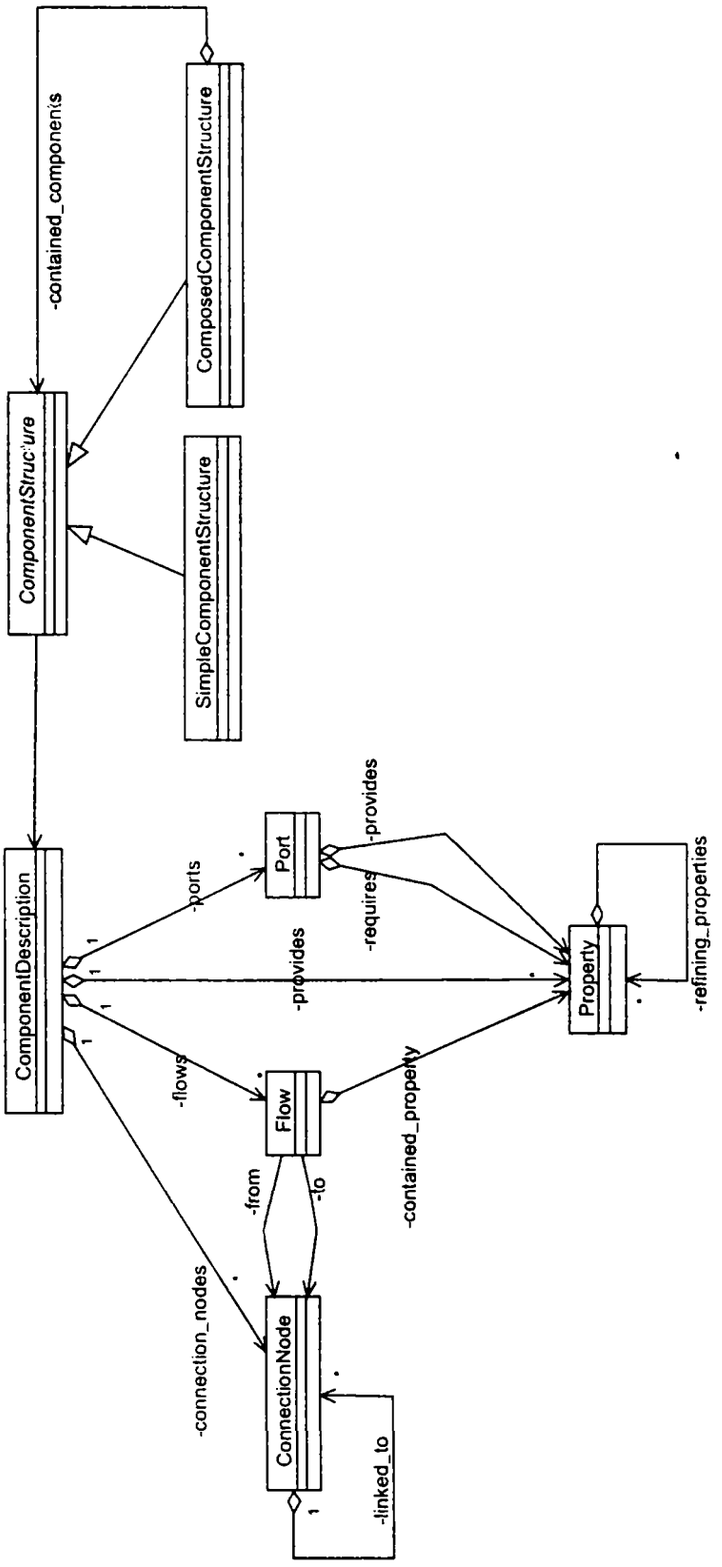


Figura 3.16: Diagrama de clase UML pentru descriptorii de componente

BUPT

rile sunt definite prin segmente orientate între două noduri de conexiune (`from`, `to` `ConnectionNode`).

Fiecărei descrieri de componentă i se poate asocia o descriere de structură internă, modelată prin clasa `ComponentStructure`. Aceasta poate fi fie o componentă simplă fie descrierea structurii prin lista de subcomponente conţinute (`contained_components`) în cazul componentelor compuse.

Descrierile de componente se găsesc în fişiere CCDL. Pentru ca aceste descrieri să poată fi utilizate de către unelte de manipulare automată a componentelor este nevoie de un parser care să fie capabil să le interpreteze şi să le valideze, producând pe baza lor obiecte de tipul `ComponentDescription` prezentat mai înainte.

Se defineşte interfaţa `ComponentDescriptionParser`, reprezentând un parser care analizează un fişier dat conţinând descrierea unei componente, îl validează şi construieşte un obiect de tip `ComponentDescription`:

```
public interface ComponentDescriptionParser {
  public ComponentDescription interpret (InputStream is)
          throws ComponentDescriptionException;
};
```

Pentru interpretarea descrierilor de componente compozabile, s-a definit o interfaţă `ComponentDescriptionParser`, a cărei metodă `interpret` are rolul de a interpreta un `InputStream` reprezentând o descriere a unei componente compozabile şi de a produce din aceasta un obiect de tip `ComponentDescription`. Rolul acestei interfeţe este de a decupla sintaxa limbajului de descriere de schema de descriere pentru componente compozabile. Se vor putea defini şi alte formalisme de descriere (diferite de CCDL) pentru schema de componente compozabile descrisă de modelul propus. Limbajul CCDL, fiind bazat pe XML, are avantajul că poate fi prototipizat rapid, dar în acelaşi timp are dezavantajul că XML nu este deloc o notaţie concisă. Pentru limbajul de descriere propus, CCDL, clasa `XMLComponentDescriptionParser` implementează interfaţa `ComponentDescriptionParser` iar intrarea descrierii este în formatul CCDL descris în paragraful precedent.

Diagrama de clase este prezentată în figura 3.17.

Implementarea parserului pentru CCDL `XMLComponentDescriptionParser` se bazează pe utilizarea interfeţei de programare de aplicaţii java pentru XML SAX [Sun].

Există două modele standard, SAX (Simple API for XML Parsing) şi DOM (Document Object Model) de prelucrare a datelor XML. Modelul SAX trimite datele

BUPT

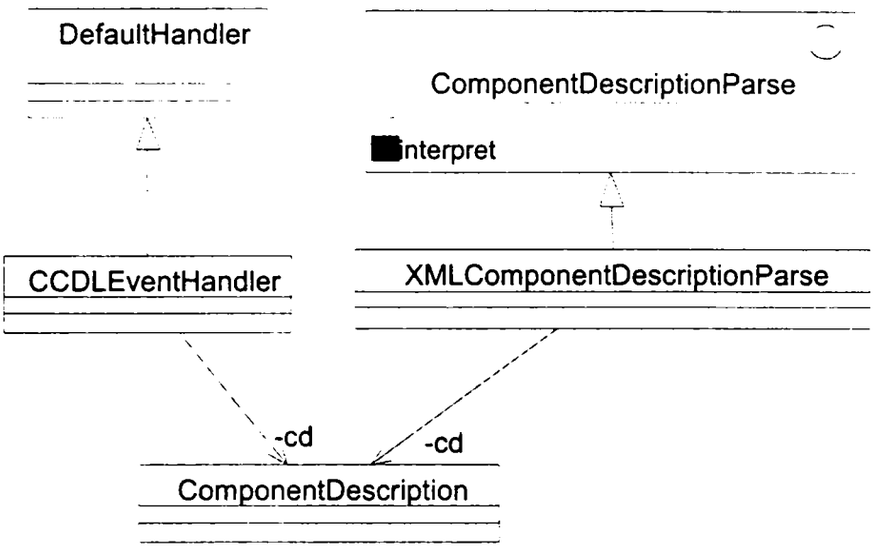


Figura 3.17: Diagrama de clase UML pentru parserul CCDL

citite către aplicaţie pe măsură ce le identifică, în continuare fiind rolul aplicaţiei să preia aceste date. Aceasta corespunde unui model de lucru bazat pe evenimente. Un parser de XML care implementează interfaţa SAX generează evenimente care corespund diferitelor elemente întâlnite în documentul scanat. Modelul DOM construieşte o reprezentare internă a documentului sub forma unui arbore. Modelul SAX are avantajul că este rapid şi eficient, iar necesarul de memorie este redus, pentru că nu construieşte o reprezentare internă a datelor XML, cum face DOM. Din aceste motive s-a ales modelul SAX pentru implementarea lui `XMLComponentDescriptionParser`.

Modelul SAX defineşte o interfaţă de programare pentru un parser de XML prin `javax.xml.parsers.SAXParser`. Principala metodă a lui `SAXParser` este:

```
parse(InputStream is, ContentHandler ch);
```

Aceasta declanşează operaţia de scanare a intrării `is`, conţinând un document XML, apelând pentru tratarea evenimentelor metodele definite în `ContentHandler`.

`ContentHandler` este o interfaţă de *callback* utilizată de parserele XML pentru a notifica aplicaţia despre evenimentele generate la întâlnirea elementelor din documentul XML. SAX furnizează o clasă `org.xml.sax.helpers.DefaultHandler` care constituie o implementare simplă a lui `ContentHandler`. Acest *DefaultHandler* conţine metode pentru tratarea evenimentelor generate de identificarea elementelor din structura unui document XML, cele mai importante metode care tratează evenimente sunt `startDocument`, `endDocument`, `startElement`, `endElement`.

BUPT

Ca pentru orice parser implementat pe baza interfeţei de programare SAX, şi pentru definirea parserului CCDL trebuie extinsă această clasă `DefaultHandler` pentru a realiza acţiuni specifice tipului de document care se defineşte. S-a definit clasa *CCDLEventHandler* derivată din *DefaultHandler* pentru interpretarea evenimentelor generate la scanarea unui document CCDL şi construirea şi iniţializarea, pe baza acestora, a membrilor obiectului *ComponentDescription*.

In modelul SAX, un anumit parser poate fi instanţiat prin intermediul unei `SAXParserFactory`. In implementarea utilizată, am ales implementarea `org.apache.xerces.parsers.SAXParser` ([Apa01]) pentru că realizează un parser ce permite validarea documentului bazată pe o Schema. Se setează ca schemă de referinţă pentru validarea intrării schema de definiţie a limbajului CCDL.

Implementarea operaţiei `interpret` din `XMLComponentDescriptionParser` realizează o transformare din XML în Java tipică modelului SAX:

```
public class XMLComponentDescriptionParser
                      implements ComponentDescriptionParser {

...

  public ComponentDescription interpret(InputStream is) throws
                                    ComponentDescriptionException
   {
       ...

       // Use an instance of our CCDL SAX event handler
       DefaultHandler handler = new CCDLEventHandler(comp_descr);

       System.setProperty("javax.xml.parsers.SAXParserFactory",
                      "org.apache.xerces.jaxp.SAXParserFactoryImpl");
       // Use the validating parser
       SAXParserFactory factory = SAXParserFactory.newInstance();

       factory.setValidating(true);
       factory.setNamespaceAware(true);

       ...
       // Parse the input
       SAXParser saxParser = factory.newSAXParser();

       ...

       saxParser.parse( is, handler);
```

BUPT

```
        ...

        return cd;

        }
...

}
```

Se crează un `SAXParser` având ca handler de tratare a evenimentelor un obiect de clasa `CCDLEventHandler` care a fost implementată să extindă în mod corespunzător `DefaultHandler` pentru interpretarea semantică a *tag*-urilor specifice CCDL.

3.4 Componente: descrieri – implementări

Utilizarea componentelor se bazează pe informaţiile conţinute în două baze de date (*repositories*), *component repository* şi *implementation repository*. *Component repository* conţine descrieri CCDL ale componentelor. *Implementation repository* conţine descrierea implementărilor componentelor.

Relaţiile între cele două *repositories* sunt ilustrate în figura 3.18.

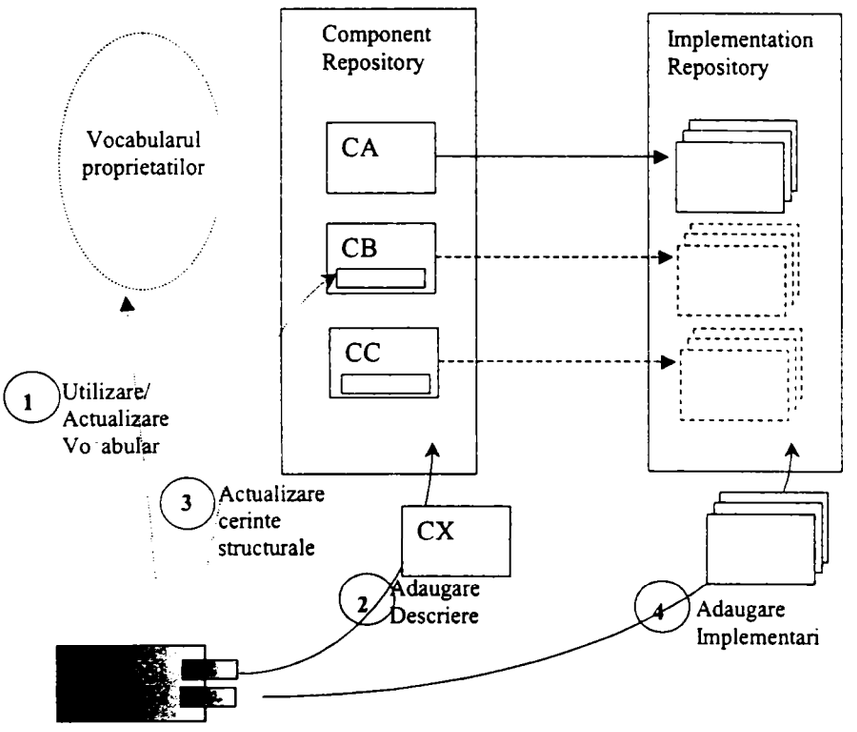


Figura 3.18: Component Repository – Implementation Repository

BUPT

Fiecare componentă cu o identitate proprie are o descriere unică de componentă, descriere ce se găseşte în *component repository* sub forma de descriere CCDL. Această descriere poate utiliza pentru proprietăţi termeni dintr-un vocabular consacrat.

Fiecărei descrieri de componentă din *component repository* îi poate corespunde una sau mai multe implementări cunoscute. Implementările cunoscute sunt înregistrate în *implementation repository.*

Sunt posibile următoarele modalităţi de descriere a unei implementări:

- specificarea unei clase ce furnizează implementarea
- o descriere de structură pentru o componentă compusă (componentele care o alcătuiesc şi configuraţia acestora)

O intrare în *implementation repository* mai conţine şi descrierea setului de proprietăţi specifice impelmentării. Este vorba de proprietăţi non-funcţionale care caracterizează varianta de implementare, cum sunt: viteza, memoria necesară, costul. Aceste proprietăţi nu sunt specifice tipului componentei, de aceea nu se specifică în descrierea din *component repository.* Este posibil ca pentru acelaşi tip de componentă (aceeaşi descriere din component repository) să existe variante diferite de implementare, fiecare prezentând caracteristici diferite ale implementării (variante de tip "varianta rapida" sau "varianta economica" a unui tip de componentă cu o funcţionalitate dată).

In cazul componentelor compozabile, nu fiecărei descrieri de componentă îi sunt asociate în avans şi înscrise în *implementation repository* toate implementările posibile. Componentele compozabile pot fi caracterizate de proprietăţi cu subproprietăţi variabile. O anumită implementare este stabilită de un set anume de subproprietăţi. Descrierea implementării unei asemenea componente va specifica seturile exacte de subproprietăţi la care corespunde implementarea. De exemplu, se consideră descrierea componentei compozabile $COMPRESSER$ (vezi exemplul din paragraful 3.2). Implementările cunoscute (sub formă de structură sau clasă) trebuie să specifice seturile de subproprietăţi pe care le implementează. De exemplu, se presupuncunoscute două implementări ale tipului $COMPRESSER$, o implementare care realizează subproprietatea $Huffmann$ şi alta care realizează subproprietatea $ComprAdaptiva$. Nu este însă obligatoriu să existe implementări cunoscute ale componentei compozabile.

Dacă nu există nici o implementare cunoscută pentru o componentă, sau dacă implementarea cunoscută nu este conformă cu toate cerinţele, implementarea componentei va fi compusă astfel încât să satisfacă aceste cerinţe şi în limitele constrângerilor sale structurale.

BUPT

3.4.1 Scenariul de utilizare a unei componente

Decizia de a utiliza o anumită componentă se ia pe baza descrierii componentei din *Component Repository*, iar apoi trebuie găsită în *Implementation Repository* o implementare pentru acea componentă. Alegerea unui anumit tip de componentă (selecţia din component repository) se face pe baza proprietăţilor funcţionale ale componentei. Cerinţele non-funcţionale specifice implementării (performanţă, cost) sunt tratate în momentul alegerii unei implementări pentru respectiva componentă. În *Implementation Repository* se pot găsi mai multe implementări care satisfac aceeaşi descriere din *Component Repository*, selectarea implementării adecvate dintre acestea se face pe baza cerinţelor non-funcţionale specifice implementării. Dacă în *Implementation Repository* nu se găseşte nici o implementare pentru componenta aleasă sau dacă implementările existente nu satisfac toate atributele cerinţelor funcţionale cerute, atunci componenta în cauză va fi compusă, dacă tipul ei permite acest lucru, adică este definită ca fiind compozabilă.

3.4.2 Scenariul pentru introducerea unei noi componente în repository

In sistem pot fi introduse oricând noi tipuri de componente, urmând un scenariu ca cel prezentat în figura 3.18. Această figură indică etapele necesare pentru ca o nouă componentă CX să fie introdusă în sistem.

In primul rând, componenta CX trebuie să aibă o descriere CCDL, realizată de proiectantul ei. Descrierea componentei utilizează termeni din vocabularul de proprietăţi. Acest vocabular se poate extinde cu introcucerea a noi termeni, dacă este nevoie. Extensia vocabularului trebuie să fie imediat publicată şi poate fi subiectul unui proces de certificare, pentru a se evita duplicarea de proprietăţi sub nume diferite.

Descrierea componentei se adauga la *Component Repository*. În cazul în care utilizarea lui CX în contextul structurii interne a altor componente compozabile (de exemplu CB) necesită unele condiţii speciale, nespecificate de descrierea lui CB, descrierea lui CB trebuie extinsă, adăugându-i la partea de constrângeri dependente de context regulile impuse de o posibilă utilizare a lui CX în structura ei internă.

Cade în sarcina proiectantului lui CX să furnizeze lista de actualizări a dependenţelor de context ale lui CB şi ale altor componente ca aceasta. Aceast lucru e în spiritul specificaţiilor extensibile definite în [Sha96]. Situaţia apare în cazul în care CX furnizează noi proprietăţi, necunoscute de către proiectantul lui CB, şi deci ne luate în calcul la descrierea constrângerilor structurale ale lui CB. Dacă pentru a-l putea folosi pe CX în structura internă a lui CB sunt necesare reguli suplimentare

BUPT

pentru a putea localiza corect pe CX pe fluxurile existente, în relaţii corecte cu celelalte proprietăţi existente acolo, atunci aceste noi reguli sunt introduse cu această ocazie.

Ultima etapă o constituie adăugarea unor implementări pentru componente a căror descriere este deja în *Component Repository*. Implementarea poate fi dată fie în formă de clasă, fie în forma unei descrieri structurale complete pentru o componentă compusă. Nu este obligatoriu ca toate componentele compozabile să aibe specificate implementări.

3.5 Concluzii

Obiectivul oricărui proces de compoziţie este construcţia unui sistem cu anumite proprietăţi impuse prin utilizarea de componente ale căror proprietăţi individuale sunt cunoscute. Pentru stabilirea unui cadru sistematic de compoziţie a componentelor sunt necesare *modele compoziţionale*.

In această lucrare se propune un model compoziţional pentru sisteme cu arhitectură multi-flux. Modelul compoziţional propus conţine o schemă şi un formalism de descriere a componentelor, precum şi o strategie de compoziţie bazată pe cerinţe.

In acest capitol s-a prezentat schema de descriere a componentelor şi sistemelor compozabile, precum şi formalismul dedescriere dezvoltat pe baza acesteia, limbajul CCDL.

O contribuţie importantă este introducerea noţiunii de componentă *compozabilă*. O astfel de componentă are o identitate proprie bine definită, dar structura sa internă nu este fixată, ci poate fi compusă în mod dinamic, în cadrul stabilit de un set de constrângeri structurale. Caracteristic acesteia este faptul că descrierea unei componente implică stabilirea interfeţei acesteia în termeni de *porturi*, a contractelor semantice în termenii unor *proprietăţi* cerute şi furnizate, şi a *constrângerilor structurale* pentru componentele compozabile.

O caracteristică şi un avantaj al schemei de descriere propuse este faptul că specificaţiile au o complexitate redusă. După cum indică şi alte articole din literatură, creşterea complexităţii specificaţiilor nu reuşeşte să facă o tratare exhaustivă a tuturor aspectelor compoziţiei ([MMM03], [Sha96]). În loc de a baza analiza pe modelarea comportamentului sistemelor, ceea ce ar necesita modelarea unui număr mare de stări de execuţie, se utilizează un model al proprietăţilor statice ale sistemului, care sunt în număr mai mic. De aceea s-a ales un model simplu bazat pe proprietăţi cunoscute ale componentelor, asemănător cu cel propus în [Sha96].

Un alt avantaj al schemei propuse este că mecanismul constrângerilor structurale,

BUPT

introdus în această teză, permite un grad ridicat de variabilitate *neanticipată*, spre deosebire de alte soluţii clasice prezentate în literatură pentru definirea variabilităţii unei configuraţii cum sunt sunt utilizarea de skeleton-uri şi puncte de variabilitate. Constrângerile structurale definesc un set de cerinţe minime pe care trebuie să le îndeplinească structura internă a unei componente pentru a-i asigura acesteia o identitate precizată. Constrângerile structurale *nu* descriu şi nu fixează configuraţia structurii interne. Constrângerile structurale fixează fluxurile interne ale unei componente şi stabilesc *proprietăţile* care trebuie să fie prezente pe fiecare din aceste fluxuri. Prin aceasta, identitatea şi numărul componentelor care ar putea face parte din configuraţia structurii interne a componentei compozabile nu este precizată şi deci nu este limitată.

Pentru descrierea componentelor compozabile s-a definit limbajul CCDL. După cum s-a precizat în secţiunea 3.3, acest limbaj prezintă unele caracteristici asemănătoare limbajelor din familia ADL (limbaje de descriere arhitecturală) şi limbajelor din familia IDL (limbaje de descriere a interfeţelor), dar fără a se încadra în totalitate în nici una din aceste categorii. Acest limbaj poate exprima atât contractele componentelor compozabile cât şi constrângerile structurale pentru compoziţia lor, lucruri care nu s-ar putea realiza în cadrul limbajelor existente din familiile limbajelor IDL şi ADL. Pentru a permite utilizarea descrierilor de componente de către unelte de prelucrare automată, s-a implementarea un parser pentru limbajul CCDL.

Pentru gestionarea componentelor sunt necesare sisteme de informaţii software care să asigure reprezentarea şi regăsirea componentelor. Descrierile de componente realizate în limbajul CCDL sunt grupate în *repositories* de unde pot fi utilizate de unelte de prelucrare automată cum sunt: unelte de compoziţie automată care implementează strategii de compoziţie bazate pe cerinţe şi unelte de verificare a corectitudinii semantice a unei compoziţii).

Capitolul următor prezintă strategia de compoziţie care defineşte algoritmul de stabilire a structurii unui sistem care trebuie să corespundă unui set de cerinţe.

BUPT

Capitolul 4

Strategie de compoziţie pentru arhitecturi multi-flux

In acest capitol se propune o strategie de compoziţie pentru compunerea automată a sistemelor cu arhitecturi multi-flux. Compunerea se face pornind de la setul de cerinţe impuse pentru sistemul care se compune, pe baza mecanismului propagării cerinţelor. Această strategie de compoziţie este o parte importantă a modelului compoziţional definit în teză.

Problema compoziţiei unui sistem din componente este următoarea: fiind dată o mulţime de componente disponibile C şi o mulţime de proprietăţi P, se cere să se construiască din componentele disponibile un sistem care să satisfacă proprietăţile din mulţimea P. Componentele disponibile precum şi cerinţele pentru sistemul compus sunt descrise în maniera prezentată în capitolul 3. Pe baza acestor informaţii, construcţia sistemului implică luarea de decizii privitoare la determinarea selecţiei submulţimii componentelor necesare din C şi la determinarea conexiunilor între acestea.

Strategia de compoziţie propusă în acest capitol reprezintă o soluţie de automatizare a procesului de decizie a compoziţiei. Compunerea automată este utilizată ca mijloc de realizare a sistemelor auto-configurante. Acest capitol se încheie cu definirea modelului Composer-Builder pentru realizarea sistemelor auto-configurante, în care Composer-ul implementează strategia de compoziţie propusă.

4.1 Domenii de aplicare pentru compoziţia automată

Compoziţia automată este utilizată în cazul sistemelor software auto-configurabile. La acestea, deciziile de compoziţie sunt luate în baza unei strategii implementate în

BUPT

sistem sub forma unei căutări automate. Intervenţia operatorului este de dorit să fie minimă şi să se exprime doar în termenii proprietăţilor dorite ale sistemului: operatorul nu trebuie să fie obligat să intervină direct cu decizii în procesul de compoziţie a sistemului în termeni de componente şi conexiuni.

Problema compoziţiei automate este, teoretic, aceeaşi şi aplicată în cadrul unor unelte de sinteză automată de software [IT02a]. Există realizări puţine şi limitate în acest sens, din lipsa unor modele de compoziţie suficient de complete care să permită înlocuirea totală cu succes a rolului proiectantului. La acest nivel, preocupările de automatizare abordează fie verificarea unui sistem, fie automatizarea generării unor aspecte punctuale (de exemplu, generarea automată a conectorilor [IT02a], generarea automată de adaptări pentru componente incompatibile [Sch01], [SR00], [JK03], aspecte discutate în secţiunea 2.3).

Sinteza automată devine importantă în cazul sistemelor auto-configurabile în timpul execuţiei. În asemenea situaţii accesul unui operator este de regulă limitat şi apare necesitatea unor sisteme de decizie automată.

Astfel de sisteme auto-configurabile sunt necesare în multe din sistemele computerizate actuale şi se pot da următoarele exemple de cazuri în care apar acestea:

- O aplicaţie de tip terminal generic permite unui utilizator să acceseze prin intermediul reţelei diferitele servicii puse la dispoziţie de un furnizor de servicii, din categoriile transmisii de voce şi multimedia, servicii de "directory" (white & yellow pages). Acest terminal generic trebuie configurat cu o anumită stivă de protocoale de comunicare în reţea, în funcţie de tipul serviciului solicitat. Selecţia protocoalelor în cele mai multe cazuri nu se poate face cu anticipaţie, pentru că depinde de aşteptările aplicaţiei client şi de serviciul accesat. Această alegere poate fi făcută de cele mai multe ori la momentul de start, la crearea stivei de protocoale. Există şi situaţii în care trebuie să fie posibil să se realizeze alegerea la momentul execuţiei *run-time*. De exemplu, un protocol de reducere a congestiilor la nivelul transport va fi selectat în mod dinamic doar când încărcarea reţelei depăşeşte un anumit prag. Un alt posibil caz este reorientrea automată la nivelul de acces fizic: dacă stabilirea unei sesiuni PPP prin modem de la un terminal mobil eşuează, se poate încerca automat o redirecţionare a sesiunii PPP peste GSM.

  Pentru a avea un acces universal uniform la serviciile furnizate de server, compoziţia şi construirea efectivă a stivei de protocoale corespunzătoare serviciului solicitat şi aşteptărilor clientului trebuie să fie transparentă pentru utilizator, să fie realizată în mod automat. Selecţia protocoalelor se face în mod dinamic şi

BUPT

automat pentru a răspunde cel mai bine cerinţelor serviciilor la care trebuie să se asigure accesul şi pentru ca utilizatorul să vadă întotdeauna acelaşi terminal generic.

- Un mediu inteligent de instrumentaţie virtuală folosit pentru construirea şi executarea de aplicaţii de măsură, monitorizare şi control.

  Un asemenea mediu trebuie să se poată auto-configura automat în mod corespunzător cu taskul curent de monitorizare, pornind de la o simplă enumerare a cerinţelor dorite şi cu cât mai puţine intervenţii din partea utilizatorului în operaţia de construcţie a circuitului de măsurare. De asemenea, în orice moment în decursul rulării aplicaţiei de monitorizare, mediul trebuie să fie capabil să opereze schimbări asupra circuitului de măsurare, ca reacţii la noi cerinţe introduse de condiţiile externe care s-au modificat între timp. De exemplu, în cazul detectării apariţiei unor perturbaţii pe semnalele de intrare, să introducă în mod automat filtre pe acele intrări.

Generalizând, un sistem auto-configurabil este un sistem care operează într-un mediu ale cărui cerinţe referitoare la proprietăţile sistemului sunt în permanentă schimbare şi care este capabil să se configureze în mod automat pentru a răspunde cel mai bine cerinţelor curente. Modul de evoluţie a mediului înconjurător nu poate fi întotdeauna prezis în momentul în care se face proiectarea sistemului. Cerinţele care apar referitoare la proprietăţile sistemului trebuie rezolvate în mod dinamic la pornirea sistemului sau chiar în timpul execuţiei acestuia.

În această lucrare se propune o metodă de realizare a sistemelor auto-configurante utilizând compoziţia automată a sistemelor, ghidată de cerinţele impuse. Principiile de realizare şi validare a unei compoziţii sunt aceleaşi ca şi în cazul când compoziţia componentelor este făcută de către un proiectant la crearea unui sistem. Decizia de compoziţie (care sunt componentele selectate şi ce relaţii/conexiuni se stabilesc între acestea) este acum implementată printr-un program. În consecinţă, cum în acest caz calculatorul nu are intuiţia unui proiectant, regulile de compoziţie trebuie să fie clar, riguros şi complet specificate. De asemenea, este nevoie de specificaţii clare ale componentelor, care să descrie toate informaţiile necesare luării deciziei de compoziţie. În lucrarea de faţă se utilizează pentru specificarea componentelor modelul CCDL dezvoltat şi prezentat anterior în capitolul 3. În baza acestui model, în acest capitol se defineşte o strategie de compoziţie automată.

Un obiectiv important al strategiei de compoziţie automate propuse este de a permite configuraţii neanticipate (posibilităţi de compoziţie care nu sunt prevăzute din start). Soluţiile nu trebuie limitate la utilizarea unui set de configuraţii cunoscute din

BUPT

start, deoarece există în continuu posibilitatea ca noi componente să fie descoperite, şi de asemenea ca cele existente sa fie utilizate în contexte diferite. Dificultatea constă în găsirea echilibrului între nevoia de a permite configurări neanticipate şi garantarea unei compoziţii corecte din punct de vedere al cerinţelor impuse sistemului.

Strategia de compoziţie propusă se adresează sistemelor de componente compozabile cu arhitectură multi-flux independent de domeniile de aplicaţii cărora le aparţin sistemele. In capitolul 5 vor fi prezentate două studii de caz, din două domenii de aplicaţii diferite, pe care este validat modelul compoziţional pentru arhitecturi multi-flux propus în această teză: utilizarea compunerii automate pentru realizarea de protocoale de reţea auto-configurante şi realizarea unui mediu inteligent de masurare si control bazat pe instrumente virtuale.

4.2 Mecanismul propagării cerinţelor

Unul din elementele esenţiale care stă la baza strategiei de compoziţie dezvoltate în lucrare este *mecanismul de propagare a cerinţelor*. Mecanismul presupune delegarea responsabilităţii de satisfacere a cerinţelor de la o componentă către altă componentă de-a lungul aceluiaşi flux, pană la întâlnirea unei componente care poate satisface cerinţele impuse. Pe baza acestui mecanism de propagare a cerinţelor se definesc algoritmi de generare a unei compoziţii de componente pornind de la cerinţele impuse (o strategie de compoziţie) şi de verificare a corectitudinii unei compoziţii date.

4.2.1 Definirea propagării cerinţelor. Cazul liniar

Se prezintă pentru început conceptul propagării cerinţelor în cazul simplificat al unui sistem cu un singur flux şi care conţine numai componente simple.

În acest caz, fiecare componentă are un port de intrare şi un port de ieşire. Cerinţele asociate portului de intrare se adresează componentelor care sunt pe flux înaintea ("deasupra") componentei curente. Denumim aceste cerinţe *cerinţe ascendente* (*upward requirements*). Cerinţele asociate portului de ieşire se adresează componentelor care urmează pe flux după ("sub") componenta curentă. Aceste cerinţe sunt numite *cerinţe descendente* (*downward requirements*). Figura 4.1 ilustrează aceste noţiuni.

După cum s-a prezentat în secţiunea 3.1.3, în general cerinţele se adresează unei componente oarecare aflată în direcţia bună a fluxului în raport cu componenta curentă şi nu sunt restrânse la componentele vecine (figura 3.2).

Fiind dată o componentă C, care la un moment oarecare în timpul procesului

BUPT

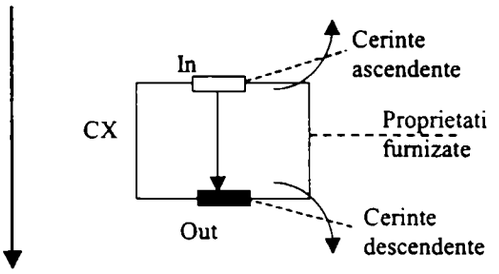


Figura 4.1: Cerinţe ascendente şi descendente

de compoziţie are cerinţele ascendente mulţimea de proprietăţi $CU = \{CU_i\}_{i=1...n}$ şi cerinţele descendente mulţimea de proprietăţi $CD = \{CD_i\}_{i=1...m}$, trebuie găsită mulţimea componentelor care situate deasupra lui C îndeplinesc toate cerinţele ei ascendente CU, precum şi mulţimea componentelor care situate sub C îndeplinesc toate cerinţele ei descendente CD.

Fie X o componentă care furnizează mulţimea de proprietăţi XP şi are cerinţele ascendente date de mulţimea de proprietăţi XU. Se determină mulţimea

$$XPCU = XP \bigcap CU$$

$XPCU$ este submulţimea cerinţelor ascendente ale lui C care sunt îndeplinite de X. Dacă mulţimea $XPCU$ este nevidă, componenta X poate fi conectată la portul de intrare a lui C. Mulţimea

$$XNPCU = CU - XPCU$$

reprezintă cerinţele ascendente ale lui C care nu au fost îndeplinite de X.

Toate proprietăţile din mulţimea $XNPCU$ vor fi adăugate la mulţimea cerinţelor ascendente ale lui X. Acest fenomen a fost denumit **propagarea ascendentă a cerinţelor**. In urma propagării, noua mulţime a cerinţelor ascendente ale lui X devine

$$XU' = XU \bigcup XNPCU$$

Mulţimea XU' este mulţimea de cerinţe cu care se continuă căutarea de componente pentru completarea părţii superioare a fluxului.

Fie Y o componentă care furnizează mulţimea de proprietăţi YP şi are cerinţele descendente date de mulţimea de proprietăţi YD. Rezultă mulţimea

$$YPCD = YP \bigcap CD$$

BUPT

submulţimea nevidă a cerinţelor descendente a lui C care sunt îndeplinite de Y. Componenta Y poate fi conectată la portul de ieşire a lui C. Mulţimea

$$YNPCD = CD - YPCD$$

reprezintă cerinţele descendente a lui C care nu au fost îndeplinite de Y.

Toate proprietăţile din mulţimea $YNPCD$ vor fi adăugate la mulţimea cerinţelor descendente ale lui Y. Acest fenomen a fost denumit **propagarea descendentă a cerinţelor**. In urma propagării, noua mulţime a cerinţelor descendente ale lui Y devine

$$YD' = YD \bigcup YNPCD$$

Mulţimea YD' va fi utilizată în căutarea restului de componente pentru completarea părţii inferioare a fluxului.

In figura 4.2 se prezintă un exemplu de propagare a cerinţelor.

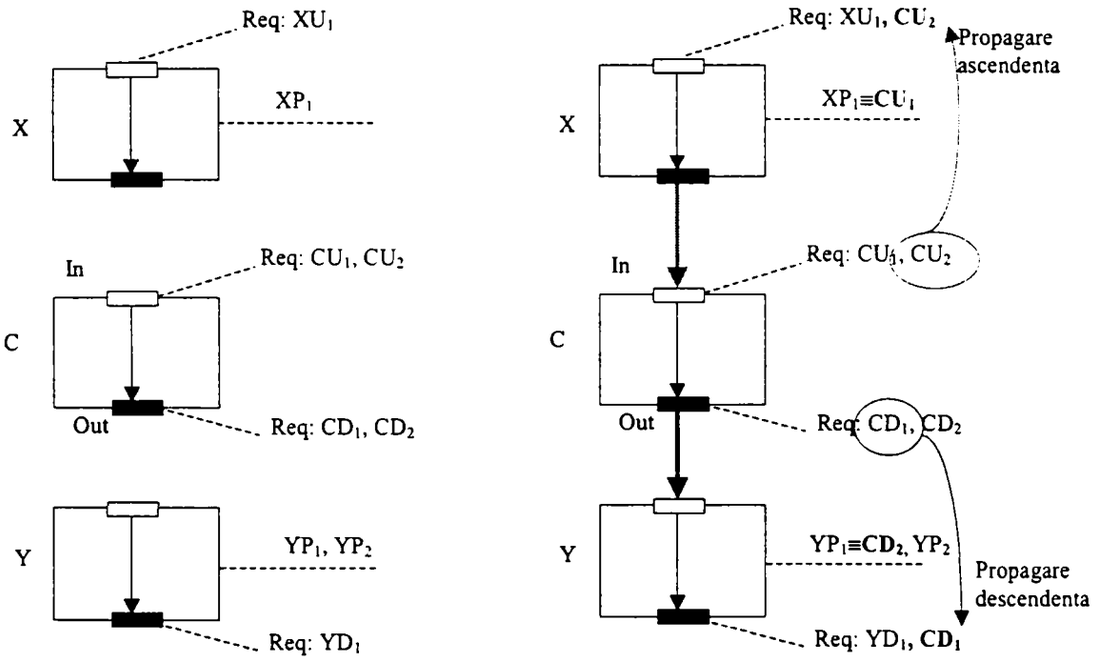


Figura 4.2: Propagarea cerinţelor - cazul liniar

Exemplul din figură consideră trei componente, C, X şi Y. Componenta C are cerinţele ascendente $CU = \{CU_1, CU_2\}$ şi cerinţele descendente $CD = \{CD_1, CD_2\}$. Componenta X furnizează $XP = \{XP_1\}$ şi are cerinţele ascendente proprii $XU = \{XU_1\}$. Componenta Y furnizează $YP = \{YP_1, YP_2\}$ şi are cerinţele descendente proprii $YD = \{YD_1\}$. Se ştie că proprietatea $XP_1 \equiv CU_1$ şi $YP_1 \equiv CD_2$, deci X satisface una din cerinţele ascendente a lui C şi Y satisface una din cerinţele

BUPT

descendente a lui C. Se realizează conexiunile următoare: portul $X.Out \rightarrow C.In$ şi $C.Out \rightarrow Y.In$.

Subulţimea cerinţelor ascendente ale lui C, rămase nesatisfăcute de X, este $XNPCU = \{CU_2\}$ şi aceste cerinţe vor fi propagate către $X.In$. Mulţimea cerinţelor descendente ale lui C rămase nesatisfăcute de Y este $YNPCD = \{CD_1\}$, ceea ce va fi propagat către $Y.Out$. După aceste propagări, cerinţele ascendente a lui X vor fi mulţimea $XU' = \{XU_1, CU_2\}$ iar cerinţele descendente a lui Y vor fi $YD' = \{YD_1, CD_1\}$.

4.2.2 Generalizarea mecanismului propagării cerinţelor pentru arhitecturile multiflux

Cazul propagării liniare a cerinţelor este rădăcini comune cu cu noţiunea de propagare prezentată de Perry în [Per87], [Per89] şi care a fost discutată în paragraful 2.2.2. O analogie a modelului de faţă cu modelul lui Perry – care se referă la operaţiile de pe fluxul de control al unui program (prezentat în secţiunea 2.2.2) – se poate face corelând precondiţiile din modelul Perry cu cerinţe ascendente, postcondiţiile cu proprietăţile furnizate şi obligaţiile cu cerinţe descendente.

Cazul propagării liniare a cerinţelor prezintă similarităţi şi cu mecanismul prezentat de Batory & Geraci în [BG97], [BG95], [BCRW00] şi discutat în paragraful 2.3.1 al lucrării de faţă. In modelul lui Batory & Geraci ([BG97], [BG95]) se consideră sisteme stratificate, în care proprietăţile furnizate de un strat pentru straturile de deasupra sa diferă de proprietăţile funizate straturilor de dedesubt şi din acest motiv Batory & Geraci fac distincţie între două tipuri de pre şi postcondiţii. Ei numesc postcondiţii proprietăţile furnizate componentelor care se află dedesubt şi numesc postrestricţii proprietătile furnizate componentelor care se află deasupra. Acestea îndeplinesc precondiţii care reprezintă cerinţele ascendente şi prerestricţii care reprezintă cerinţele descendente. În [BG97], [BG95] se propun algoritmi pentru *verificarea* corectitudinii unui sistem, bazaţi pe propagarea descendentă a postcondiţiilor şi propagarea ascendentă a postrestricţiilor (adică a proprietăţilor furnizate).

Modelul propus în această teză are un scop diferit şi mai dificil, şi anume nu doar verificarea unei compoziţii date ci facilitarea compunerii, *generarea* automată a atructurii unei compoziţii. Din acest motiv, în această teză elementele propagate nu sunt proprietăţile furnizate, ci *cerinţele*, acestea fiind elementele motoare în cautarea soluţiei de compoziţie. De asemenea, în modelul de componente propus în această teză, proprietăţile furnizate de o componentă sunt asociate componentei ca şi întreg (după cum a fost prezentat în secţiunea 3.1.3), deci sunt aceleaşi proprietăţi furnizate

BUPT

pentru orice punct de interacţiune.

O altă particularitate importantă a modelul de compoziţie propus în această teză şi care îl diferenţiază de cazurile menţionate este că o componentă nu face parte din compoziţii strict stratificate şi poate avea interacţiuni prin intermediul porturilor sale cu un număr oarecare de fluxuri de componente. Toate aceste fluxuri văd componenta ca furnizoare a aceluiaşi set de proprietăţi.

În acest paragraf se defineşte mecanismul propagării cerinţelor pentru cazul componentelor cu mai multe porturi de intrare şi ieşire, mecanism care este o generalizare a celui pentru cazul liniar prezentat în paragraful anterior.

Pentru prezentarea cazului general, se renunţă la termenii de cerinţe "ascendente" şi "descendente" deoarece nu au sens în acest context. Aceşti termeni au fost introduşi în paragraful anterior pentru simplificarea explicaţiei principiului şi corelarea cu metodele înrudite din literatură. In cazul general, nu se mai poate vorbi de componente aflate "deasupra" sau "dedesubtul" altor componente. Asemenea relaţii de ordine se mai stabilesc doar între porturi conectate pe acelaşi flux.

În cazul general, componentele au cerinţe asociate porturilor lor de intrare şi de ieşire. Cerinţele asociate porturilor de intrare se adresează în mod ascendent fluxului care intră în acel port, iar cerinţele asociate porturilor de ieşire se adresează în mod descendent fluxului care iese din acel port.

Propagarea cerinţelor se face de-a lungul fluxurilor interne care corespund porturilor care le solicită. Fluxurile interne ale componentelor compuse permit astfel determinarea porturilor de ieşire care sunt influenţate de fiecare port de intrare. Pentru a putea modela cu fineţe interacţiunile între componentele aflate pe un lanţ de conexiuni, e necesar să se cunoască pentru fiecare componentă relaţiile între porturile sale de intrare şi ieşire (*intracomponent pathways* - [SW01]). In modelul propus în această teză, aceste relaţii sunt date de fluxurile interne.

Fie o componentă C, care are NI_C porturi de intrare şi NO_C porturi de ieşire. La un port de ieşire $C.Out_o$, $o \in [1 \ldots NO_C]$ cerinţele asociate sunt mulţimea de proprietăţi CO_o.

Fie o componentă Y care are NI_Y porturi de intrare şi NO_Y porturi de ieşire şi furnizează mulţimea de proprietăţi YP. Fie mulţimea

$$YPCO_o = YP \bigcap CO_o$$

mulţimea cerinţelor portului $C.Out_o$ îndeplinite de Y. Se consideră că se conectează portul $C.Out_o$ la un port de intrare $Y.In_i, i \in [1 \ldots NI_Y]$ al lui Y. Fie mulţimea $YFI_i = \{Y.Out_o\}$ submulţimea acelor porturi de ieşire ale lui Y la care există fluxuri

BUPT

de la portul $Y.In_i$. Cerinţele rămase neîndeplinite,

$$YNPCO_o = CO_o - YPCO_o$$

se propagă la toate porturile din mulţimea YFI_i, acestea urmând ca după propagare să aibă cerinţele:

$$\forall Y.Out_o \in YFI_i : YO'_o = YO_o \bigcup YNPCO_o$$

In figura 4.3 se prezintă un exemplu de propagare a cerinţelor în cadrul unei componente generale.

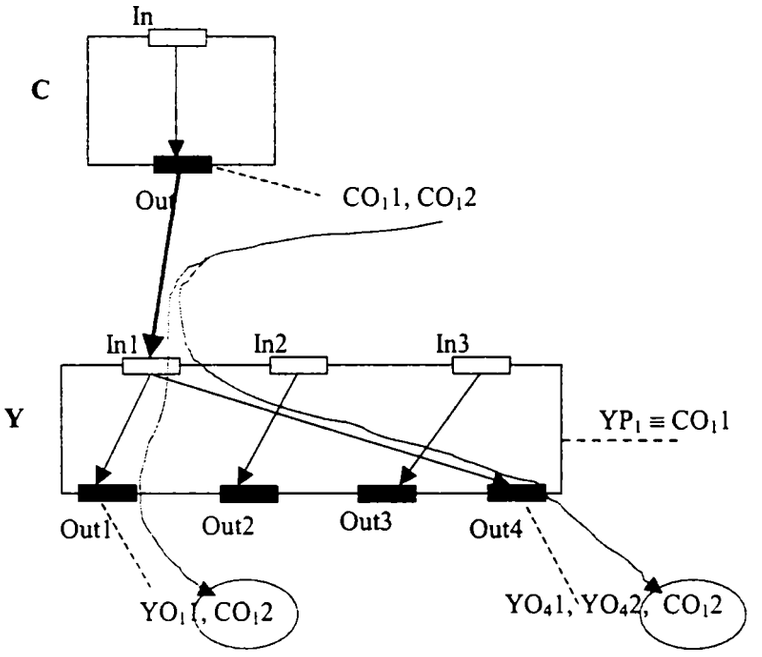


Figura 4.3: Propagarea cerinţelor - cazul general

În exemplu se consideră o componentă C care are un port de ieşire $C.Out_1$, la care există mulţimea cerinţelor $CO_1 = \{CO_1 1, CO_1 2\}$.

Componenta Y are $NI_Y = 3$ intrări şi $NO_Y = 4$ ieşiri. Mulţimea proprietăţilor furnizate de Y este $YP = \{YP_1\}$ şi proprietatea $YP_1 \equiv CO_1 1$. Fluxurile interne ale lui Y sunt următoarele: $In_1 \rightarrow Out_1$, $In_1 \rightarrow Out_4$ $In_2 \rightarrow Out_2$ şi $In_3 \rightarrow Out_3$. Dacă se conectează portul $C.Out_1$ la portul $Y.In_1$, rezultă că cerinţa $CO_1 2$ de la $C.Out_1$ nu a fost încă îndeplinită şi aceasta este propagată la porturile $Y.Ou1_1$ şi $Y.Out_4$, acele porturi de ieşire ale lui Y care sunt legate prin fluxuri interne de portul $Y.In_1$.

În mod analog se defineşte şi propagarea cerinţelor asociate porturilor de intrare.

BUPT

Definirea propagării proprietăţilor interne în arhitecturi multi flux

Un fenomen similar propagării cerinţelor se petrece în cazul componentelor compuse, unde proprietăţile furnizate sau solicitate ale subcomponentelor sunt propagate către porturile externe ale componentei compuse.

In figura 4.4 se prezintă un exemplu de propagare a proprietăţilor interne, în care compoziţia componentei K este realizată de componentele A, B, C, D şi E, dispuse pe cele trei fluxuri interne ale lui K.

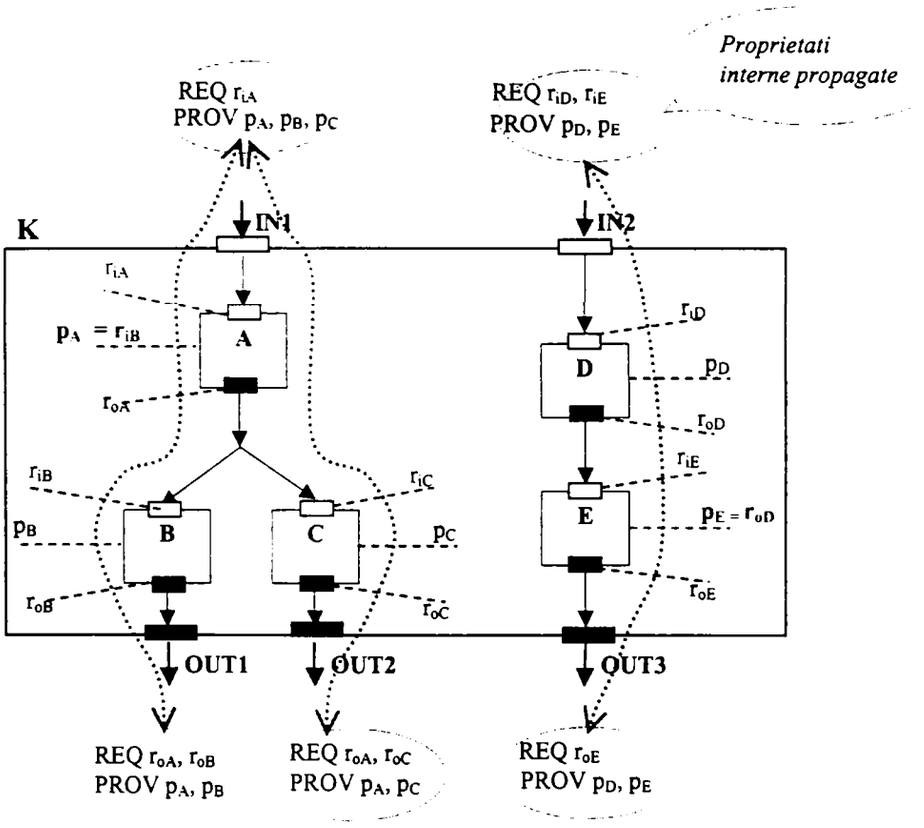


Figura 4.4: Propagarea proprietăţilor interne

Fiecare componentă furnizează şi solicită proprietăţi. Se notează cu ri_X o cerinţă la porturile de intrare a componentei X şi cu ro_X cerinţe exprimate la porturile de ieşire a componentei X. Notaţia p_X descrie o proprietate furnizată de componenta X. In exemplul din figură, s-a presupus că cerinţa ri_B este satisfăcută de proprietatea p_A şi cerinţa ro_D de către proprietatea p_E. In acest caz, proprietăţile(cerute sau furnizate) propagate către porturi sunt cele din figură.

Propagarea proprietăţilor interne furnizate are loc în felul următor. Către portul $In1$ se propagă proprietăţile furnizate de pe fluxurile $In1 \rightarrow Out1$ şi $In1 \rightarrow Out2$,

BUPT

mulţimea p_A, p_B, p_C. Către portul $Out1$ se propagă proprietăţile furnizate pe fluxul $In1 \rightarrow Out1$, adică mulţimea p_A, p_B iar către portul Out2 se propagă proprietăţile furnizate pe fluxul $In1 \rightarrow Out2$, p_A, p_C.

În ceea ce priveşte propagarea proprietăţilor interne cerute, acestea se pot propaga către porturile externe doar în sensul în care sunt adresate. De asemenea, nu sunt propagate acele proprietăţi interne cerute care sunt satisfăcute local in interiorul componentei. Spre portul $In1$ se pot propaga proprietăţi care sunt cerinţe ascendente pe fluxurile $In1 \rightarrow Out1$ şi $In1 \rightarrow Out2$; doar proprietăţile r_{iA} şi r_{iC} se propagă la $In1$, nu şi r_{iB}, deoarece $r_{iB} \equiv p_A$ (cerinţa ascendentă r_{iB} a lui B este satisfăcută de proprietatea p_A furnizată de A).

4.2.3 Verificarea unei compoziţii

Mecanismul propagării cerinţelor prezentat în paragrafele anterioare este utilizat în mod direct la verificarea corectitudinii semantice a unei compoziţii date. Verificarea unui sistem din componente ierarhic compozabile cu arhitectură multi-flux se face validând fiecare flux al său în parte. Potrivirile între proprietăţi furnizate şi proprietăţi cerute sunt stabilite conform regulilor modelului definit în capitolul 3.1.

Algoritmul de verificare rezultă imediat din aplicarea propagării cerinţelor pe toate fluxurile sistemului, fiind similar cu cel din [Per87, Per89] şi care a fost discutat în paragraful 2.2.2 şi cu cel din [BG97, BG95, BCRW00] discutat în paragraful 2.3.1. Un exemplu de utilizare a propagării cerinţelor pentru verificarea unei compoziţii va fi prezentat în paragraful 5.2.3.

O contribuţie importantă a acestei teze este utilizarea mecanismului de propagare a cerinţelor pentru generarea automată a structurii unui sistem ţintă pornind de la mulţimea proprietăţilor pe care trebuie să le aibă acesta. Acest aspect va fi prezentat în continuare în paragraful 4.3.

4.3 Generarea unei compoziţii

Scopul strategiei de compoziţie este de a se constitui într-un *algoritm de generare automată a structurii* unui sistem ţintă pornind de la cerinţele care i se impun. In acest subcapitol, prezentarea strategiei de compoziţie este făcută incremental, începând cu cazul particular al unei compoziţii liniare şi apoi continuând cu cazul general al compoziţiei în arhitectura multi-flux.

BUPT

4.3.1 Strategia de compoziţie – cazul liniar

În cazul liniar, problema compoziţiei este o problemă de determinare a unei secvenţe ordonate de componente simple pe un flux. Pentru stabilirea unei terminologii de prezentare convenţionale, fluxul se consideră "descendent" (orientat de "sus" în "jos"), componentele aşezându-se în straturi orizontale pe acest flux. Nivelul client se consideră primul (cel mai "înalt") strat, care exprimă proprietăţile dorite pentru sistemul de compus ca cerinţe descendente ale sale. Mulţimea R a acestor cerinţe îsi are originea atât în mulţimea CC a cerinţelor direct exprimate de client cât şi în cerinţele determinate de constrângerile structurale ale ţintei. În figura 4.5 se prezintă punctul de plecare pentru compoziţia în cazul liniar.

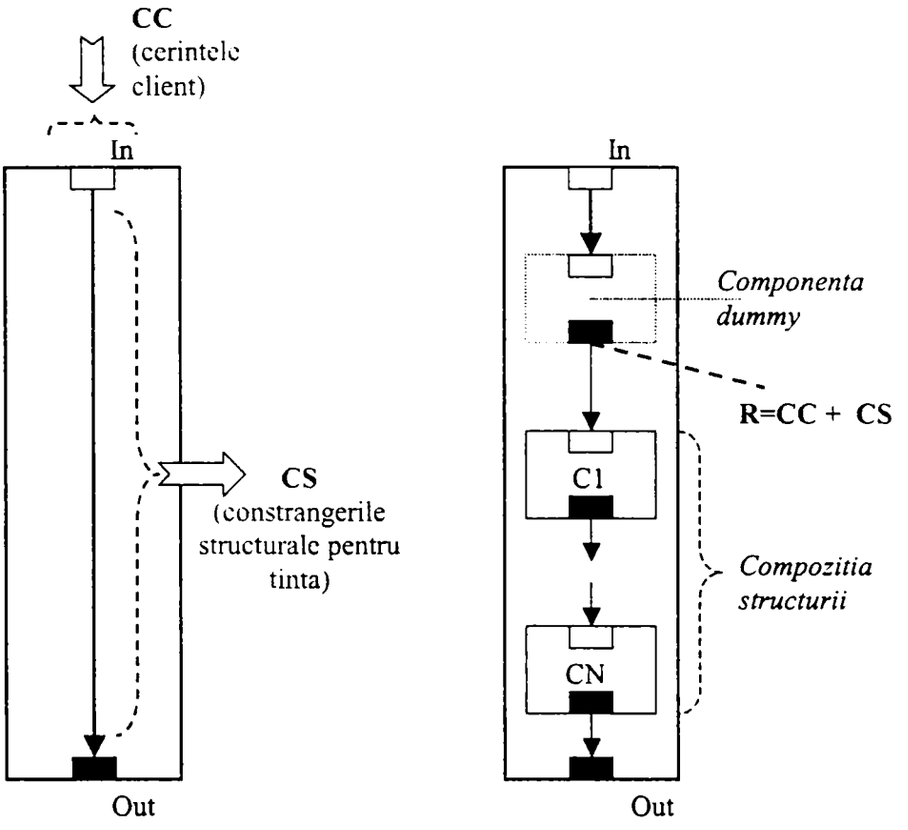


Figura 4.5: Strategia de compoziţie - cazul liniar

Ţinta (sistemul care trebuie compus) este o componentă compozabilă cu un singur flux, pe care sunt definite un set de constrângeri structurale CS. Se impun o mulţime de cerinţe client CC. Rezultă mulţimea $R = CC \bigcup CS$ cerinţele iniţiale. Acestea pot fi văzute ca cerinţe descendente ale unei componente de start *dummy*.

Dacă mulţimea cerinţelor este $R = \{R_1, R_2, \ldots, R_r\}$, o secvenţă ordonată de com-

BUPT

ponente $C_1, C_2, \ldots, C_N$ reprezintă o compoziţie bună dacă:

1. Toate cerinţele iniţiale $R_i \in R$ sunt îndeplinite de către componente din secvenţa $C_i, i = 1 \ldots N$.

2. Pentru fiecare componentă $C_j, j = 1 \ldots N$ lista cerinţelor sale descendente este îndeplinită de către o mulţime de componente $C_i, i \in [j+1 \ldots N]$ (componente C_i care urmează pe C_j în secvenţă)

3. Pentru fiecare componentă $C_j, j = 1 \ldots N$ lista cerinţelor sale ascendente este îndeplinită de către o mulţime de componente $C_i, i \in [1 \ldots j-1]$ (componente C_i care preced pe C_j în secvenţă)

4. Nu există nici o componentă $C_j, j \in [1 \ldots N]$ a cărei proprietăţi nu sunt cerute nici de setul de cerinţe iniţiale, nici de către cerinţele altor componente din secvenţă. (se adaugă doar componentele care sunt utile, care furnizează proprietăţi cerute, direct sau indirect).

Strategia de compoziţie se bazează pe căutarea de componente care să satisfacă cerinţele impuse, căutând componente care să furnizeze proprietăţile cerute. Direcţia de căutare este descendentă (*top-down*). Se crează o componentă de start C_0 (componentă fictivă, *dummy*), care are ca şi cerinţe descendente mulţimea cerinţelor iniţiale R. Mulţimea R cuprinde enumerări de proprietăţi şi relaţii de ordine între proprietăţi. Acestea pot proveni din cerinţele direct exprimate de client sau din cerinţele derivate din constrângerile structurale ale ţintei. Căutarea soluţiei este condusă în primul rând de cerinţele descendente, deoarece se consideră că nivelul i utilizează servicii oferite de nivelele $i+1$ (cere anumite proprietăţi de la nivelele de sub el) şi furnizează servicii nivelului $i-1$ (de deasupra lui).

Startul procesului de căutare este dat de găsirea unor componente care furnizează cel puţin o parte din cerinţele iniţiale. Dacă s-a găsit o asemenea componentă, în majoritatea cazurilor încă mai rămân cerinţe neîndeplinite. De asemenea, componenta care furnizează o parte din cerinţele iniţiale are şi propria mulţime de cerinţe. Din acestea şi prin propagarea cerinţelor iniţiale care au rămas neîndeplinite se formează noua mulţime de cerinţe curente.

Strategia de compoziţie exploatează faptul că dependenţele între componente formează o structură de graf. Acest graf are în noduri componentele iar arcele reprezintă dependenţe între ele. Exisă un arc orientat de la C_i la C_j dacă C_j furnizează cel puţin o proprietate cerută ca proprietate descendentă de C_i, direct sau ca şi cerere propagată. Acest graf are o structură foarte dinamică pentru că prin mecanismul de

BUPT

propagare pot să apară oricând noi cerinţe care generează noi dependenţe. Arcele apar şi dispar pe măsură ce cerinţele sunt propagate şi rezolvate.

In timpul procesului de compoziţie, dacă s-a adaugat secvenţei soluţie componenta din nodul N_1, problema este determinarea componentei care poate fi adăugată sub ea. Lista succesorilor potenţiali ai lui $N1$ sunt toate nodurile N care furnizează cel puţin o proprietate care este în lista curentă (proprie sau propagată) de cerinţe a lui $N1$.

Lista potenţialilor succesori ai lui N_1 se găseşte în graf în vecinătatea nodului. Această listă iniţială este redusă prin aplicarea unor criterii de excludere, cum sunt:

- redundanţa semantică (nu se adaugă componente care repetă proprietăţi deja existente pe flux)
- ordinea bună (dacă o proprietate p_1 este în lista de cerinţe, cu condiţia de ordonare $p_1 > p_2$, nu se permite încă adăugarea unei componente care furnizează p_2 înainte de adăugarea unei componente care furnizează $p1$; de asemenea, se ţine cont de restricţiile de ordonare introduse de cerinţe de tip *pereche*)
- cerinţele negative (nu se adaugă o componentă care furnizează proprietatea p_1 dacă există deja în soluţia parţială o componentă care are cerinţa negativă p_1.

Dacă componentele din lista potenţialilor succesori au şi cerinţe ascendente, se caută şi se inserează componente care le furnizează. Aceste componente, inserate în secvenţă ca urmare a cerinţelor ascendente, pot introduce noi cerinţe descendente, care actualizează lista cerinţelor propagate şi structura grafului dedependenţe între componente.

O soluţie este un drum de la nodul de start la un nod-frunză, în care nu au mai rămas cerinţe propagate şi când nu există nici un fel de conflicte între proprietăţile şi cerinţele componentelor existente în soluţie.

Algoritmul este descris în continuare utilizând o coadă în care sunt păstrate nodurile care au fost deja examinate dar care vor mai fi prelucrate şi ulterior. In această coadă, nodurile atinse sunt păstrate împreună cu calea originară în rădăcină pe care fiecare componentă a fost atinsă. Această cale defineşte de asemenea cerinţele propagate. Aceeaşi componentă poate avea prezenţe multiple în coadă, deoarece poate fi atinsă urmând drumuri diferite.

În principiu, algoritmul de determinare a compoziţiei pe un flux este schiţat în secvenţa următoare:

```
Algorithm FlowSearch ( StartNode )
```

BUPT

```
*   do specific actions for search initialization
*   initializes CurrentPathToStart <- void
*   initializes Queue with ( StartNode, CurrentPathToStart)

while (! ConditionStopSearching()) do
    *   dequeue ( CurrentNode, CurrentPathToStart) from Queue
    *   solve upward requirements
    if (CurrentNode is a leaf-node) then
        *   solve the strong upward requirements of CurrentNode
        *   a path to a leaf-node has been found -> do strategy
            specific actions
        *   determine list UnresolvedRequirements along the
            CurrentPathToStart
        if (list UnresolvedRequirements contains no properties) then
            *  CurrentPathToStart is a composition solution on flow
            *  verify upward requirements along CurrentPathToStart
            *  if a good solution has been found
              *  do strategy specific actions
    else
        *   computes LPA = ListOfAdjacencies (CurrentNode)
        for ( each node CurrentPANode in LPA) do
            if (CurrentPANode is not already in CurrentPathToStart) then
                *  CurrentPathToStart <-
                                CurrentPathToStart + {CurrentPANode}
                *  Propagate requirements
                                from CurrentNode to CurrentPANode
                    ( modify graph structure )
                *  enqueue (CurrentPANode, CurrentPathToStart)
End FlowSearch
```

Compoziţia începe de la nodul de start *StartNode*, care conţine cerinţele referitoare la ţinta compoziţiei, obţinute din cerinţele client şi din cerinţele deduse din constrângerile structurale pentru ţintă.

Se determină mulţimea de componente LPA care pot reprezenta noduri adiacente nodului curent (furnizează cel puţin o proprietate care este cerinţă descendentă a lui

BUPT

CurrentNode). Pentru fiecare din nodurile *LPA* care nu sunt redundante deja, se determină mulţimea cerinţelor din *CurrentNode* satisfăcute şi a celor rămase nesatisfăcute. Cerinţele nesatisfăcute sunt propagate către noul nod adăugat.

Condiţia de terminare a căutării poate fi stabilită în funcţie de politica aleasă. Căutarea se poate termina după găsirea primei soluţii, sau a unui numar limitat de soluţii, sau poate continua până la găsirea tuturor soluţiilor. De asemenea, de fiecare dată când se întâlneşte un nod terminal (dar nu neapărat o soluţie completă, care să satisfacă toate cerinţele, se pot întreprinde diferite acţiuni, în funcţie de politica curentă (de a lua în considerare soluţii parţiale sau nu).

4.3.2 Exemplu ilustrativ pentru algoritmul de compoziţie – cazul liniar

Se va ilustra principiul algoritmului pe un exemplu simplu. Se consideră compunerea unei stive de protocoale de reţea din componente de granularitate mare, reprezentând fiecare câte un protocol - un strat din stivă.

In acest caz simplificat, stiva de protocoale este o componentă compozabilă, care are un singur flux pe care trebuie să existe, la baza stivei, proprietatea *interf* (interfaţă de reţea).

Se presupune că o aplicaţie are nevoie de o stivă de protocoale care să asigure o legătură de comunicaţie la distanţă fiabilă. Utilizând vocabularul de proprietăţi, rezultă cerinţele client: *transp*, *non – local*, *rel*. La aceste cerinţe client se adaugă cerinţele impuse de constrângerile structurale ale componentei compozabile *Stiva* care cer ca la baza stivei să se găsească o interfaţă de reţea. Aceste cerinţe adaugă proprietatea *interf* şi relaţia de ordine $* > interf$.

Se consideră că în depozitul de componente sunt prezente următoarele componente: *UDP*, *TCP*, *IP*, *REL*, *ETH*, *ETHNI*. Fiecare este o componentă simplă având o intrare şi o ieşire şi furnizează proprietăţi conform funcţionalităţii lor cunoscute: *UDP* furnizează proprietatea *transp*, *TCP* furnizează proprietăţile *transp* şi *rel*, *IP* furnizează proprietatea *non – local*, *REL* este un protocol de fiabilitate care furnizează proprietatea *rel*, *ETHNI* este interfaţa de reţea de tip Ethernet, furnizează proprietatea *interf*, iar *ETH* asigură legătura de date (este un driver de reţea) Ethernet. Conţinutul depozitului de componente (inclusiv componenta compozabila Stiva) este reprezentat în figura 4.6.

În primul pas, ilustrat în figura 4.7 pornind de la cerinţele iniţiale, se găsesc cinci componente (*UDP*, *REL*, *IP*, *TCP* şi *EthNI*) care fiecare îndeplinesc un subset al acestor cerinţe. În figură sunt reprezentate relaţiile între componente pe parcursul

BUPT

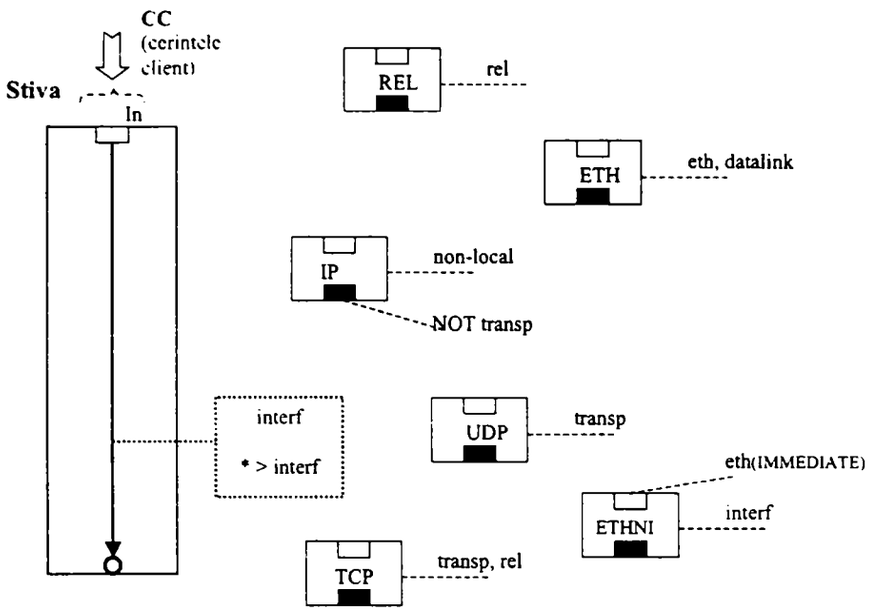


Figura 4.6: Exemplu - conţinutul *component repository*

algoritmului de căutare. Având în vedere că fiecare componentă are doar un port de intrare şi un port de ieşire şi că determinarea structurii compoziţiei înseamnă găsirea unei secvenţe ordonate de componente, se utilizează o notaţie grafică simplificată: componentele sunt reprezentate prin elipse; proprietăţile furnizate de o componentă apar lânga reprezentarea componentei, însoţite de eticheta `PROV` (*provides*); proprietăţile cerute la portul de ieşire (cerinţe descendente) apar cu eticheta `REQD` (*reqires down*) iar proprietăţile cerute la portul de intrare (cerinţe ascendente) apar cu eticheta `REQU` (*requires up*).

Deoarece constrângerile structurale impun relaţia $* > interf$, şi mai sunt cerinţe nesatisfăcute, $EthNI$ este exclusă din lista potenţialilor succesori. De asemenea, se exclude UDP deoarece furnizează proprietatea *transp* şi există constrângerea $rel >= transp$. In lista de potenţiali succesori rămân componentele REL, IP şi TCP, care nici una nu satisface în totalitate cerinţele. Componenta REL furnizează doar proprietatea *rel*, deci proprietăţile $non - local$ şi *transp* vor fi propagate către ea. Componenta TCP satisface proprietăţile *rel* şi *transp*, va fi propagată cerinţa $non - local$. Spre componenta IP se vor propaga cerinţele *transp* şi *rel*.

Al doilea pas al algoritmului presupune selectarea unei componente din cele selectate la primul pas şi încercarea de a îi rezolva setul de cerinţe. Dacă se încearcă o continuare a traseului început cu REL, aceasta are ca cerinţe propagate *transp* şi $non - local$, pentru care potenţiali furnizori sunt dintre componentele UDP, IP,

BUPT

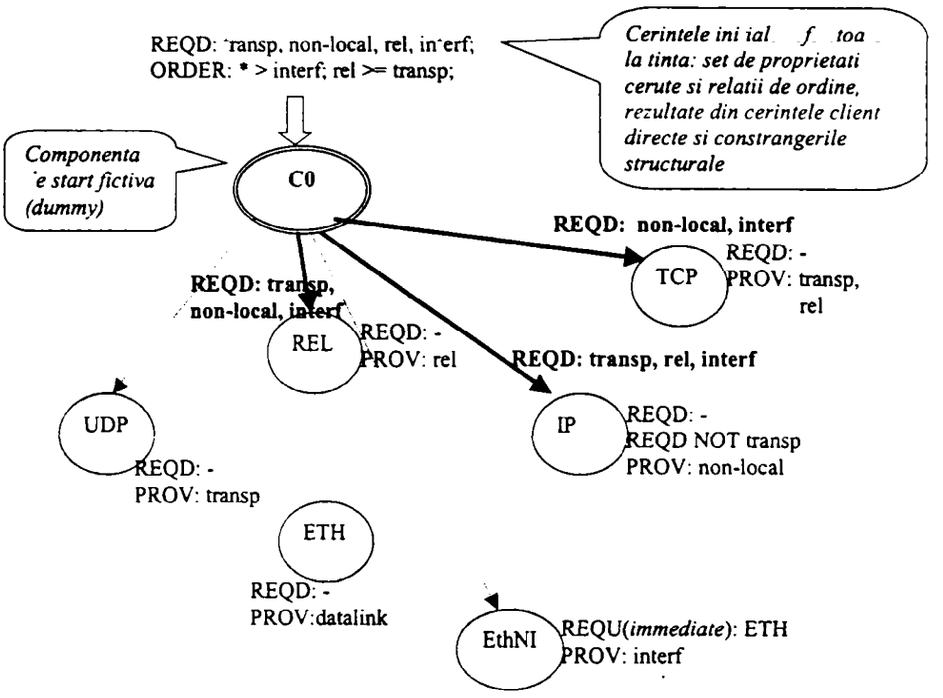


Figura 4.7: Compoziţia liniara a unei stive de protocoale. Pas 1.

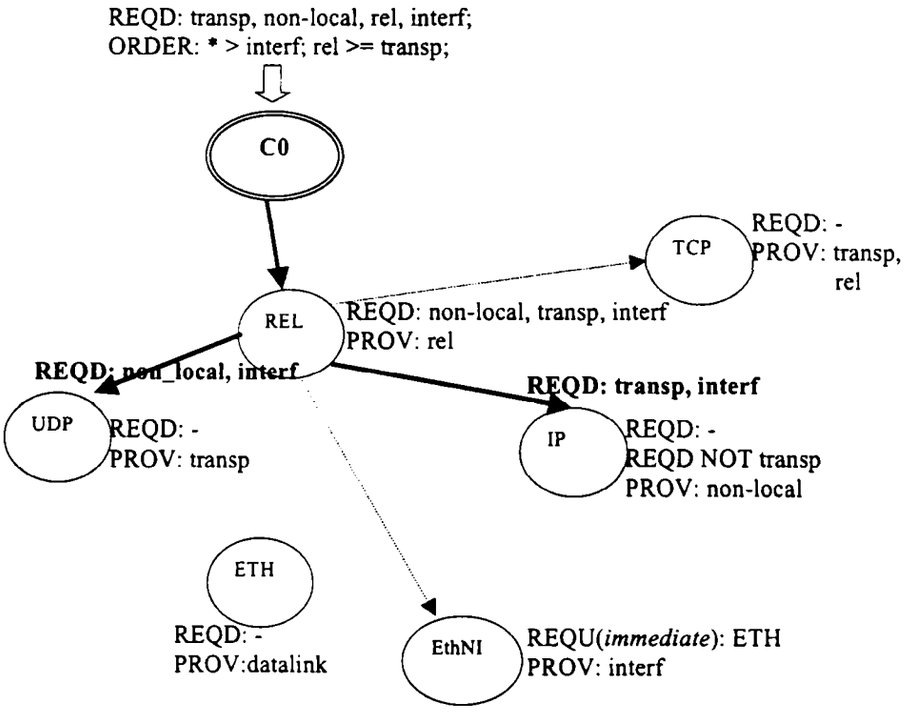


Figura 4.8: Compoziţia liniară a unei stive de protocoale. Pas 2.

BUPT

TCP, $EthNI$. Componenta $EthNI$ se elimină din nou datorită constrângerii care impune ordinea, iar TCP va fi eliminat din motive de redundanţă, pentru că repetă proprietatea *rel*. Pasul este ilustrat în figura 4.8.

Căutarea continuă până se găseşte secvenţa $REL-UDP-IP-ETH-EthNI$ ca o soluţie bună. De asemenea, se mai găseşte soluţia $TCP-IP-ETH-EthNI$. Ambele secvenţe reprezintă compoziţii valide pe flux, având îndeplinite toate proprietăţile cerute şi toate relaţiile de ordine respectate.

4.3.3 Strategia de compoziţie – cazul general

In cazul general, un sistem urmează modelul de componente compozabile în arhitecturi multi flux descris în capitolul 3. Următoarele aspecte diferenţiază cazul general de cazul liniar prezentat anterior:

- în sistem există mai multe fluxuri, pe fiecare flux are loc o compoziţie liniară;
- fiecare componentă poate fi compusă la rândul ei, interiorul ei fiind tot o structură de tip multi-flux.

Compoziţia sistemului rezultă prin rafinări sucesive, după cum se ilustrează în figura 4.9. Contribuţiile referitoare la stabilirea strategiei de compoziţie descrisă în acest capitol au fost publicate şi în lucrările [ŞVB02], [ŞJBV02], [ŞMBV03]. După ce un proces de căutare determină că se va utiliza o anumită componentă într-o compoziţie, poate începe o nouă problemă de compoziţie pentru a determina structura internă a componentei, dacă aceasta este compozabilă.

Intregul proces de decizie a compoziţiei este ghidat de cerinţe. Cerinţele pentru sistemul ţintă rezultă din cerinţele directe exprimate de client şi constrângerile structurale ale ţintei. Aceste cerinţe iniţiale sunt adresate pe fluxul principal al sistemului şi propagate, pe măsură ce se adaugă noi componente. În exemplul din figură, rezultă compoziţia structurii componentei C (ţinta iniţială) din componentele $C1$ şi $C2$.

Când o proprietate cerută r are subproprietăţi r_1, r_2, ..., r_n atunci componenta C_r care furnizează respectiva proprietate trebuie să fie adaptată astfel încât structura sa internă să corespundă cerinţelor impuse de subproprietăţi. Este posibil ca să existe o implementare cunoscută (în *Implementation Repository*, v. subcapitolul 3.4) a componentei C_r care satisface toate subproprietăţile cerute. Dacă o asemenea implementare nu este cunoscută, atunci componenta compozabilă C_r trebuie ajustată astfel încât să furnizeze toate subproprietăţile r_1, ..., r_n cerute. Componenta compozabilă C_r devine o nouă ţintă care trebuie compusă, pornind de la constrângerile sale structurale peste care se adaugă cerinţele directe client, r_1, ..., r_n.

BUPT

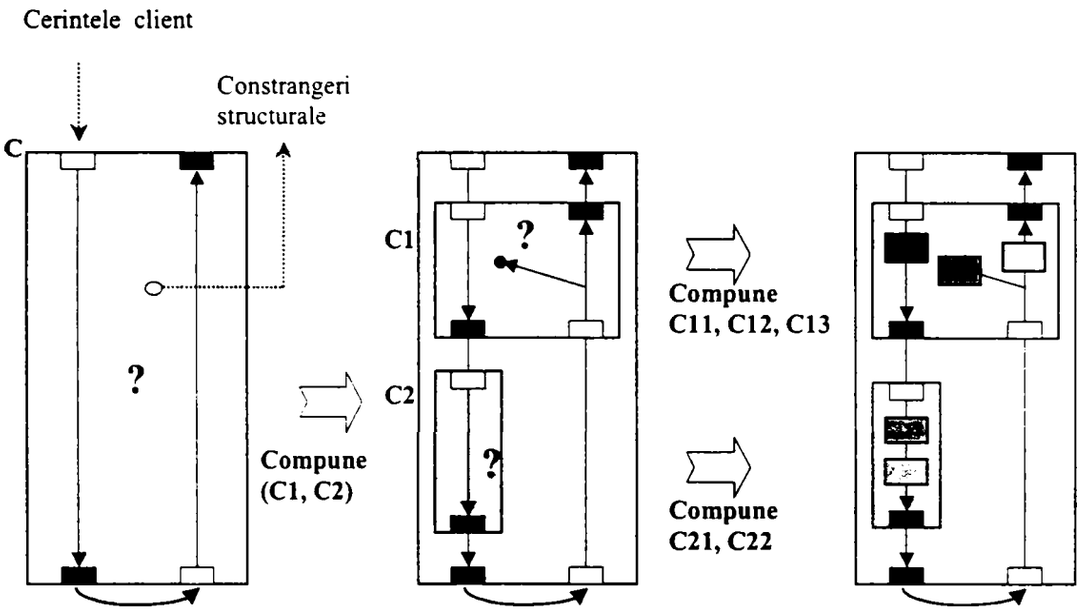


Figura 4.9: Compoziţia recursivă top-down

În exemplul dn figura 4.9, se presupune că cerinţele iniţiale adresate ţintei C conţin printre altele proprietatea $p1$ *WITH* ($p11$, $p12$) (proprietatea $p1$ cu subproprietăţile $p11$ şi $p12$) şi că componenta $C1$ a fost instanţiată în compunerea lui C deoarece furnizează proprietatea generică $p1$. Componenta $C1$ este o componentă compozabilă căreia nu îi sunt cunoscute implementări care să furnizeze $p11$ şi $p12$. Următorul pas este stabilirea ca ţintă a compoziţiei structura internă a lui $C1$. Aplicând cerinţele $p11$ şi $p12$ peste constrângerile structurale ale lui $C1$, rezultă structura lui $C1$ ca fiind alcătuită din componentele $C11$, $C12$, $C13$.

Descrierea de principiu a algoritmului de determinare a structurii interne a unei componente compozabile pornind de la un set de cerinţe client directe, este dată de următoarea secvenţă:

```
Algorithm ComposeComponent (aComposableComponentDescription,
                            theClientRequirements )

   do OnFlowComposition(theMasterFlow(aComposableComponentDescription),
                        theClientRequirements)

   foreach f = internal flow of aComposableComponentDescription do
      * determine fRemainingReq, the remaining external
        requirements for flow f
```

BUPT

```
        * do OnFlowComposition(f, fRemainingReq)

    foreach potentialComponentStructure do
        * find uncomplete subcomponents in potentialComponentStructure
        * foreach uncomplete subcomponent S
            * do ComposeComponent(S, SFinetuningRequirements)
        * revalidate potentialComponentStructure, elliminate if
          invalid

end ComposeComponent
```

Cerinţele client directe referitoare la proprietăţile sistemului ţintă al compoziţiei sunt adresate pentru început fluxului intern marcat ca şi flux principal şi este demarat un proces de compoziţie liniară pe acel flux. Un proces de compoziţie liniară pe un flux adaugă cerinţelor client directe şi setul de cerinţe introduse de constrângerile structurale care sunt legate de respectivul flux, apoi are loc un proces de căutare a componentelor înşirate pe acest flux.

Procesul de căutare pe un flux este similar cu algoritmul `FlowSearch` descris la pagina 98. Intervin modificări în operaţiile efectuate de `FlowSearch`, în principal la căutarea unei componente următoare şi determinarea potrivirii unei componente. La căutarea componentei următoare (determinarea listei potenţialilor succesori), căutara începe examinând componentele compuse existente deja în structura altor fluxuri interne ale ţintei şi care mai au porturi neconectate. Determinarea faptului că o anumită componentă se potriveşte cerinţelor curente existente pe flux trebuie avut în vedere faptul că acea componentă poate fi şi o componentă compusă C_x. Primul pas în determinarea potrivirii este dat de constatarea că proprietăţile furnizate de componenta C_x (furnizate ca proprietăţi globale) îndeplinesc un subset al cerinţelor curente pe flux. Următorii paşi în determinarea potrivirii constau în stabilirea perechii de porturi prin care componenta C_x va fi conectată la flux.

Procesul liniar de căutare este reluat pentru fiecare flux intern al ţintei. În stabilirea cerinţelor rămase şi în alegerea componentelor utilizate pentru fiecare flux se ţine cont de configuraţiile stabilite anterior pentru celelelte fluxuri.

Pot să rezulte un număr oarecare de potenţiale configuraţii pentru structura internă a ţintei. Pentru fiecare dintre aceste potenţiale configuraţii, se verifică dacă conţine subcomponente incomplete care mai trebuie ajustate conform unor subproprietăţi cerute. Pentru fiecare astfel de subcomponentă se declanşează un nou proces de compoziţie a structurii sale interne.

BUPT

4.4 Modelul Composer – Builder pentru sisteme autoconfigurante

In acest subcapitol se utilizează compoziţia automată a unui sistem din componente ca mijloc de realizare a sistemelor auto-configurante. Aceste sisteme se adaptează automat la condiţiile mediului în care se execută prin configurarea adecvată a structurii lor, configurarea care are loc într-un mod transparent pentru utilizator. In acest subcapitol se propune modelul Composer–Builder ca model general de realizare a sistemelor autoconfigurante care permite tratarea problemelor decizionale într-un mod independent de problemele tehnologice ale compoziţiei de componente Strategia de compoziţie definită în 4.3 este implementată în cadrul `Composer`-ului.

4.4.1 Definirea modelului Composer–Builder

Problema compoziţiei automate porneşte de la următoarele categorii de date aflate la dispoziţie:

- setul de cerinţe referitoare la proprietătile dorite ale sistemului ţintă
- un depozit de componente, conţinând descrierile componentelor cunoscute
- un depozit de implementări de componente

Pornind de la aceste date iniţiale, se pot lua decizii privitoare la:

- setul de componente necesare
- relaţiile de colaborare între componente

Toate componentele pot fi compozabile, deci determinarea structurii lor interne poate reprezenta o nouă problemă recursivă de compoziţie.

După ce s-a determinat setul de componente şi conexiunile între ele la toate nivelurile (s-a decis structura compoziţiei), se poate construi sistemul propriu-zis. Construcţia sistemului este posibilă cu ajutorul infrastructurii mediului de execuţie, care oferă suportul necesar pentru instanţieri şi legări dinamice de componente.

Compunerea automată a unui sistem implică deci rezolvarea a două aspecte diferite: aspectul decizional (determinarea structurii compoziţiei) şi aspectul tehnologic (asigurarea infrastructurii necesare). In această teză se propune separarea acestor aspecte, prin definirea modelului *Composer–Builder*. Se realizează astfel o împărţire a responsabilităţilor:

BUPT

- determinarea structurii (partea care se ocupă cu luarea deciziilor) este sarcina unui *Composer*,
- utilizarea infrastructurii pentru construcţia propriu-zisă a sistemului este sarcina unui *Builder*

Această relaţie *Composer-Builder* este prezentată în figura 4.10.

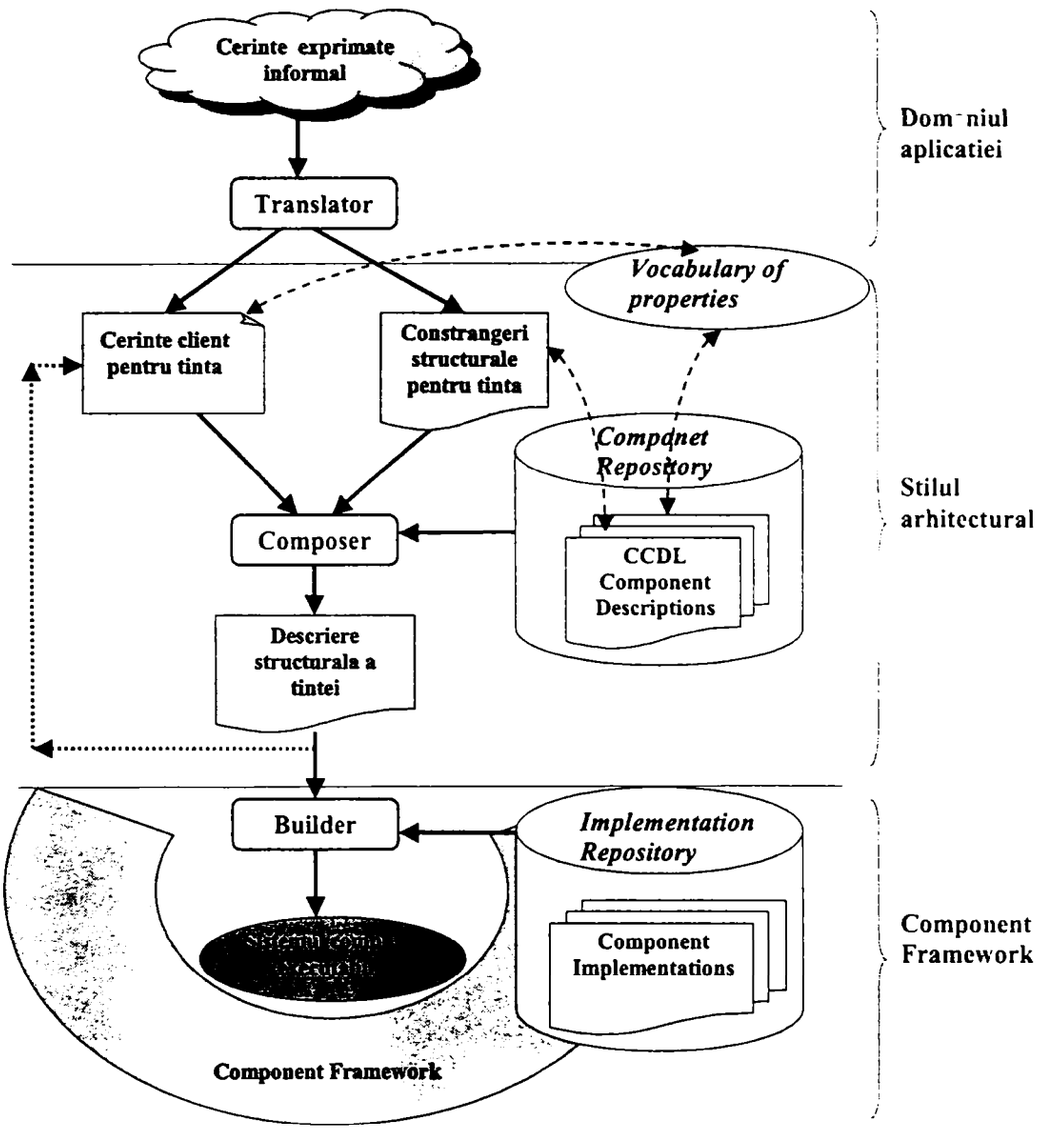


Figura 4.10: Composer-Builder

Strategia de compoziţie este implementată de subsistemul `Composer`. Acesta ia deciziile de compoziţie, operând asupra unui model arhitectural ([OGT+99]) al sistemului. Acest model arhitectural este o descriere structurală a sistemului compus.

BUPT

Composer-ul determină structura sistemului ţintă în concordanţă cu cerinţele impuse. **Composer**-ul este specific stilului arhitectural, deciziile de compoziţie fiind codificate după reguli care păstrează o totală independenţă de domeniul aplicaţiei. Strategia de compoziţie implementată va fi prezentată în detaliu în secţiunile următoare.

Un subsistem distinct, **Builder**, utilizează descrierea de structură produsă de **Composer** şi realizează construcţia propriu-zisă a sistemului. Ca implementare, **Builder**-ul depinde de tehnologia de componente utilizată. Din punct de vedere al **Composer**-ului, acesta impune pentru **Builder** doar cerinţa ca **Builder**-ul să poată instanţia în mod dinamic componente şi a să poată realizeza în mod dinamic conexiuni între porturi specificate ale acestora.

Composer-ul operează cu cerinţe exprimate ca expresii CCDL de proprietăţi. În cazul în care auto-configurarea sistemului are loc la momentul de start, conform cerinţelor unui utilizator, se poate adăuga în mod opţional un subsistem frontal de translatare a cerinţelor, dintr-o formă de specificare mai apropiată domeniului aplicaţiei, înspre forma de expresii CCDL.

Composer-ul are acces la un depozit de descrieri de componente (*Component Repository*), în care se găsesc descrieri CCDL ale tuturor componentelor, inclusiv ale celor compozabile. Ţinta compoziţiei este de asemenea o componentă compozabilă, şi pe lângă cerinţele client directe trebuie să fie cunoscute şi constrângerile structurale pentru aceasta.

Compoziţia sistemului ţintă va rezulta prin rafinări succesive *top-down*: componentele se selectează din depozit dacă rezolvă din cerinţele curente, apoi, dacă este necesar ca proprietăţile furnizate de o componentă să fie ajustate (*fine-tuning*), se poate începe un nou proces de compoziţie, având ca şi ţintă structura internă a respectivei componente compozabile.

Un avantaj al acestui model *Composer-Builder* este faptul că permite ca strategia de compoziţie implementată de **Composer** să fie definită în mod independent de domeniul aplicaţiei şi de aspectele tehnologice ale modelului de componente. Strategia de compoziţie este definită în termeni specifici arhitecturii sistemului. Strategia de compoziţie care a fost descrisă în subcapitolul 4.3 este valabilă pentru sisteme din diverse domenii de aplicaţie care au acelaşi stil arhitectural, stilul arhitectural multi-flux.

4.4.2 Implementarea Composer–Builder

S-a implementat un *Composer* pentru compunerea automată a sistemelor cu arhitecturi multi-flux din componente compozabile.

BUPT

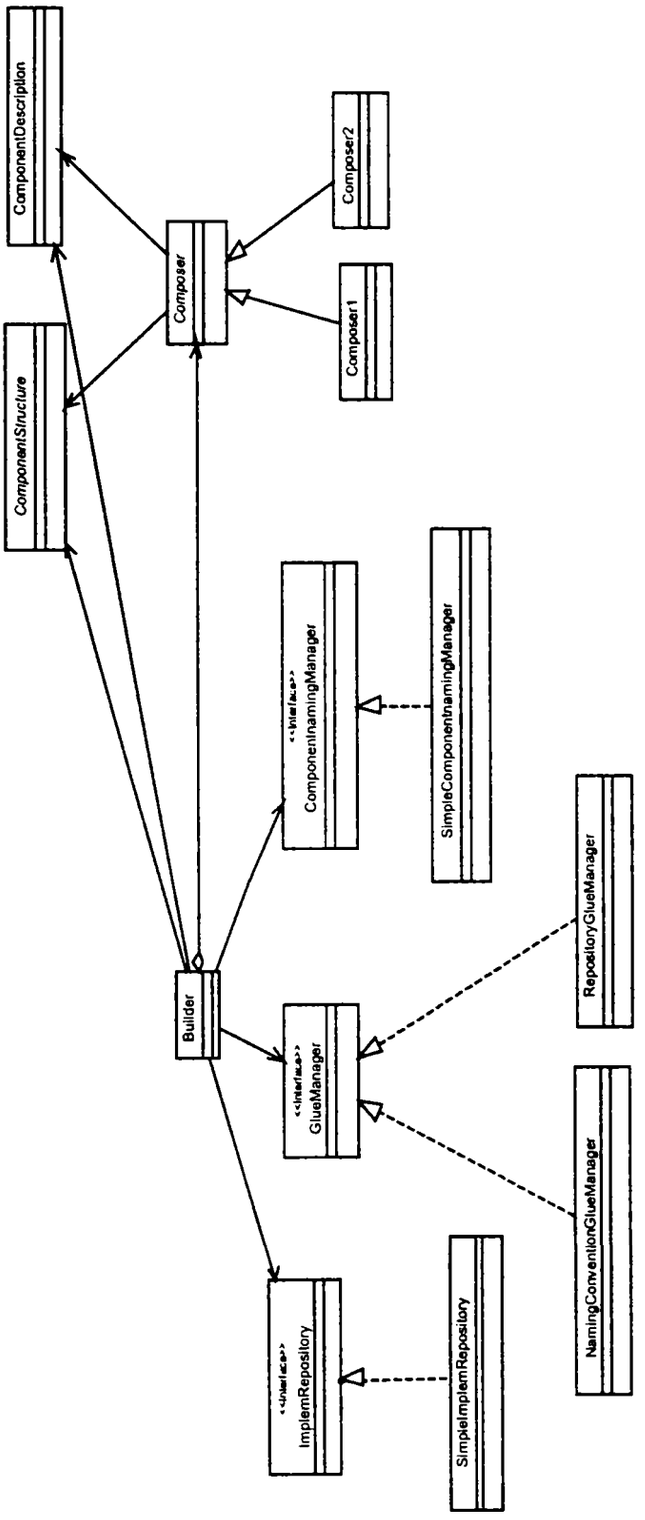


Figura 4.11: Diagrama de clase pentru Composer şi Builder

BUPT

Acest *Composer* este utilizat împreună cu un *Builder* care este capabil să instanţieze şi să interconecteze în mod dinamic componente.

Principalele clase care implementează sistemele menţionate sunt clasele `Composer` şi `Builder`. Relaţiile între `Composer`, `Builder` şi alte clase auxiliare sunt reprezentate în figura 4.11.

Strategia de compoziţie este implementată în `Composer`, prin metoda sa `compose` care realizează compoziţia structurii interne a unei componente compozabile a cărei descriere o primeşte ca şi parametru. Orice sistem care va fi compus trebuie descris în termenii de componente compozabile. Rezultatul operaţiei de compoziţie este o descriere de structură internă a componentei.

`Builder`-ul are rolul de a crea în mod dinamic o componentă compusă, conform descrierii structurii interne furnizate de `Composer`. Operaţiile principale pe care le execută `Builder`-ul, şi care trebuie să aibă suportul tehnic al tehnologiei de componente utilizate, sunt instanţierea unui tip de componentă şi realizarea de conexiuni între porturile a două instanţe. `Builder`-ul instanţiază componentele necesare cu ajutorul unei `ComponentFactory`, bazată pe `ImplemRepository`. Instanţele create trebuie să primească nume unice, lucru realizat cu ajutorul unui `ComponentNamingManager`. Conexiunile între componente sunt făcute cu ajutorul liantului oferit de un `GlueManager`.

4.5 Concluzii

Una dintre cele mai importante provocări în sistemele de calcul actuale este necesitatea de auto-configurare dinamică a acestora în scopul adaptării la ambianţa în care îşi desfăşoară execuţia. O posibilitate de realizare a auto-configurării este prin compoziţie automată de componente.

In acest capitol s-a prezentat o strategie de compunere automată a unui sistem din componente. Metoda este complementară schemei de definire a componentelor compozabile pentru arhitecturi multi-flux prezentată în capitolul 3.

Problema compunerii automate a unui sistem din componente este următoarea: Fiind dată o mulţime de componente disponibile C şi o mulţime de proprietăţi P, se cere să se construiască dintr-o mulţime de componente $C' \subset C$ un sistem S care să satisfacă proprietăţile din mulţimea P.

Strategia de compoziţie propusă în acest capitol se bazează pe mecanismul de propagare a cerinţelor şi reprezintă o contribuţie importantă a acestei teze.

Mecanismul propagării cerinţelor este definit ca o transpunere în domeniul componentelor şi o generalizare a principiului propagării cerinţelor definit de Perry pentru clauze requires-provides în [Per89]. Mecanismul propus al propagării cerinţelor este şi

BUPT

o generalizare a metodei lui Batory ([BG97]), prin adaptarea metodei pentru sisteme care nu sunt liniare. De asemenea, constituie o contribuţie importantă a acestei lucrărări faptul că mecanismul propagării cerinţelor este folosit aici ca motor de căutare a soluţiilor în *generarea* automată a unei compoziţii, spre deosebire de lucrările cunoscute din literatură unde variante ale acestui mecanism au fost folosite doar pentru *verificarea* unor compoziţii date.

Acelaşi principiu bazat pe mecanismul propagării cerinţelor poate fi utilizat şi pentru verificarea automată a corectitudinii unui sistem.

Un avantaj al strategiei de compoziţie propuse este că poate fi utilizată la generarea de sisteme din diferite domenii de aplicaţie, strategia fiind specifică unui anumit stil arhitectural, stilul multi-flux şi independentă de domeniul aplicaţiei. Un alt avantaj al metodei de compoziţie este că poate fi utilizată pentru generarea unor configuraţii neanticipate, atât în ceea ce priveşte tipurile componentelor implicate, în care noi tipuri de componente pot fi descoperite în orice moment, cât şi configuraţiile structurale noi care pot fi alcătuite, în cadrul permis de constrângerile structurale.

Metoda de compoziţie propusă poate fi folosită pentru realizarea de sisteme autoconfigurante. La acestea, deciziile de compoziţie sunt luate de către o strategie implementată în sistem sub forma unei căutări automate de către un `Composer`. Intervenţia operatorului este de dorit să fie minimă şi să se exprime doar în termenii proprietăţilor dorite ale sistemului: operatorul nu trebuie să fie obligat să intervină direct cu decizii în procesul de compoziţie a sistemului în termeni de componente şi conexiuni.

In acest capitol s-a definit şi modelul `Composer-Builder` pentru sisteme autoconfigurante (secţiunea 4.4). S-a realizat implementarea unui `Composer` bazat pe strategia de compoziţie definită.

BUPT

Capitolul 5

Validarea practica a modelului compozitional

Modelul compoziţional descris în capitolele 3 şi 4 şi reprezentând contribuţia teoretică a acestei teze a fost validat practic prin aplicaţiile care sunt descrise în acest capitol.

Modelul compoziţional propus suportă dezvoltarea de unelte pentru două procedee de compunere a sistemelor: compunerea automată pentru realizarea de sisteme auto-configurante (subcapitolul 5.1) şi compunerea interactivă, susţinută de unelte de compunere vizuală şi de verificare automată (subcapitolul 5.2).

Modelul este independent de domeniul aplicaţiilor, fiind specific sistemelor cu stil arhitectural multi-flux. Acest stil arhitectural caracterizează sisteme din domeniul stivelor de protocoale de reţea şi din domeniul instrumentaţiei virtuale pentru aplicaţii de măsură şi control.

Strategia de compoziţie propusă a fost validată prin implementarea `Composer`-ului şi integrarea lui într-un sistem tip *Composer-Builder*(v. paragraful 4.4) pentru realizarea de protocoale de reţea auto-configurante (subcapitolul 5.1).

In domeniul software-ului de măsurare şi control bazat pe instrumente virtuale, o strategie de verificare a compoziţiei bazată pe mecanismul de propagare a cerinţelor este utilizată la verificarea automată a circuitelor de procesare compuse în mod vizual din instrumente virtuale (subcapitolul 5.2).

5.1 Protocoale de reţea auto-configurante

5.1.1 Prezentarea domeniului aplicatiei

O situaţie în care apare necesitatea unor protocoale de reţea dinamic auto-configurante se regăseşte în aplicaţii de tip *terminal generic* [Pep02].

Acest concept devine de actualitate în contextul în care apar mereu noi tehnologii

BUPT

de acces la reţea şi se diversifică echipamentele care sunt părţi ale reţelei. In afară de clasicele calculatoare personale, apar noi tipuri de terminale utilizate pentru accesul la internet şi la servicii cu acces la distanţa (*remote access services*): WAP-GSM, web-phone, etc. Pe de altă parte, furnizorii din domeniul telecom işi extind ofertele de servicii. Pentru a putea utiliza noi servicii pe o gamă largă de terminale diferite este nevoie de un terminal generic. Acesta este o aplicaţie care se execută pe terminalul fizic şi care are rolul de a ascunde detaliile terminalului şi de a oferi o manieră de acces uniformă la serviciile la distanţă.

O aplicaţie de tip terminal generic permite unui utilizator să acceseze *în mod uniform* prin intermediul reţelei, independent de terminalul la care se află, diferitele servicii puse la dispoziţie de furnizorii de servicii la distanţă, ca de exemplu: transmisii de voce şi multimedia, servicii de "directory" (white & yellow pages), etc. Acest terminal generic trebuie configurat cu o anumită stivă de protocoale de comunicare în reţea, în funcţie de tipul serviciului solicitat şi de tipul terminalului fizic pe care se execută . Selecţia protocoalelor în cele mai multe cazuri nu se poate face cu anticipaţie, pentru că depinde de aşteptările aplicaţiei client şi de serviciul accesat. Nu sunt cunoscute cu anticipaţie proprietăţile necesare ale stivei de protocoale pe care o utilizează la un moment dat. Această alegere trebuie făcută la momentul de start al aplicaţiei când trebuie configurată stiva de protocoale. Pe de altă parte, acţiunea de configurare a stivei trebuie să se întâmple automat şi cât mai transparent pentru utilizator, care trebuie să fie scutit de sarcina configurării explicite a subsistemului de comunicaţie.

Infrastructura necesară pentru realizarea de protocoale de reţea dinamice a fost asigurată de un *component framework* pentru construcţia de protocoale de reţea, *DiPS* (*Distrinet Protocol Stacks framework*, [Mat99]) proiectat şi implementat de către colectivul DistriNet de la Katholieke Universiteit Leuven, Belgia. DiPS este un instrument util în cercetarea în domeniul stivelor de protocoale, permiţând prototipizarea rapidă a unor stive de protocoale adaptate unor cerinţe experimentale. DiPS este un cadru de componente pentru construcţia de protocoale de reţea. Abstracţiile de bază în DiPS sunt componentele şi conectorii. O componentă DiPS este elementul constructiv de bază al unei stive de protocoale. Operaţiile de manipulare de date şi operaţiile de control specifice prezente în stivele de protocoale (fragmentare, detecţia erorilor, ordonare, etc) sunt realizate de către componente diferite. O componentă simplă DiPS are o intrare prin care recepţionează pachete de date şi o ieşire prin care produce pachete de date. DiPS nu impune modelul clasic al stivei de protocoale organizată pe straturi (*layers*) dar oferă suport pentru acesta. DiPS furnizează suportul necesar pentru interconectarea straturilor şi componentelor, ca şi suportul pentru o

BUPT

compoziţie dinamică în timpul execuţiei, necesar în cazul protocoalelor dinamice.

Realizarea protocoalelor auto-configurante s-a facut peste infrastructura oferită de DiPS care include tehnologia necesară pentru protocoale dinamice. Partea de decizie automată necesară realizării de protocoale auto-configurante a fost implementată ca aplicaţie a strategiei de compoziţie prezentate în această teza. O stivă de protocoale auto-configurante este un sistem de tip `Composer-Builder`. *DiPS* furnizează facilităţile direct necesare pentru construirea sistemului `Builder`, şi anume facilităţile de a încărca şi lega în mod dinamic componentele din care asamblează protocoale. De asemenea, *DiPS* furnizează şi un punct de comunicare pentru aplicaţii (*socket generic*, [MMMV01]) care ascunde structura stivei de aplicaţie şi permite modificarea acesteia în mod transparent pentru utilizator.

In cadrul acestei teze s-a implementat `Composer`-ul care realizează strategia de compoziţie automată asigurând astfel partea de decizie necesară auto-configurării protocoalelor, peste infrastrctura furnizată de *DiPS*. Prin integrarea `Composer`-ului cu `Builder`-ul specific `DiPS` se realizează protocoale de reţea dinamic auto-configurante. Această soluţie a fost publicată în lucrările [ŞJBV02], [ŞVB02], [ŞMM01], [ŞM01], [ŞMBV03].

Alte cercetări în domeniul protocoalelor de reţea dinamice sunt prezente în literatură ([PLV97], [HF98], [HP90]). Faţă de acestea, soluţia de realizare a protocoalelor auto-configurante propusă pe baza strategiei de compoziţie automată descrisă în această teză se caracterizează prin flexibilitatea sporită şi posibilitatea de a face faţă unor situaţii de configurare conform cu cerinţe neanticipate.

5.1.2 Aplicarea strategiei de compoziţie pentru compunerea automată a unei stive de protocoale

Pentru exemplificarea metodei de compoziţie automată pe un caz real, se reia aici exemplul componentei compozabile *stiva de protocoale* care a fost prezentat simplificat în paragraful 4.3.2. Acum se consideră că granularitatea posibilă a componentelor este mult mai mică. Unităţile de compoziţie a stivei de protocoale pot fi componente oricât de mici din interiorul nivelurilor de protocoale.

O stivă de protocoale de reţea este văzută în cazul general ca şi o componentă compozabilă, având constrângerile structurale indicate în figura 5.1.

Dacă se consideră compunerea stivei de protocoale din componente de granularitate mică (se pot compune protocoalele aflate pe nivelurile stivei), la descrierea stivei trebuie luate în considerare două fluxuri, unul descendent şi unul ascendent, corespunzătoare căilor de transmisie respectiv recepţie de date din reţea. La bază stivei

BUPT

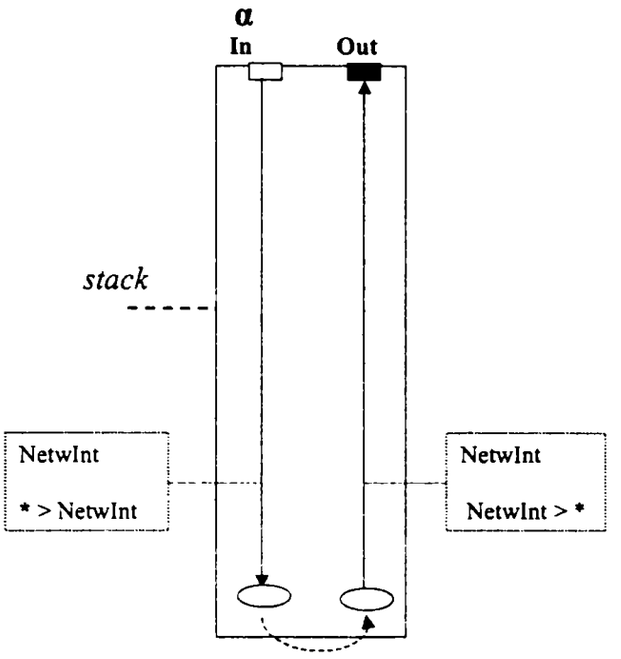


Figura 5.1: Constrângeri structurale pentru componenta compozabilă Stiva de Protocoale

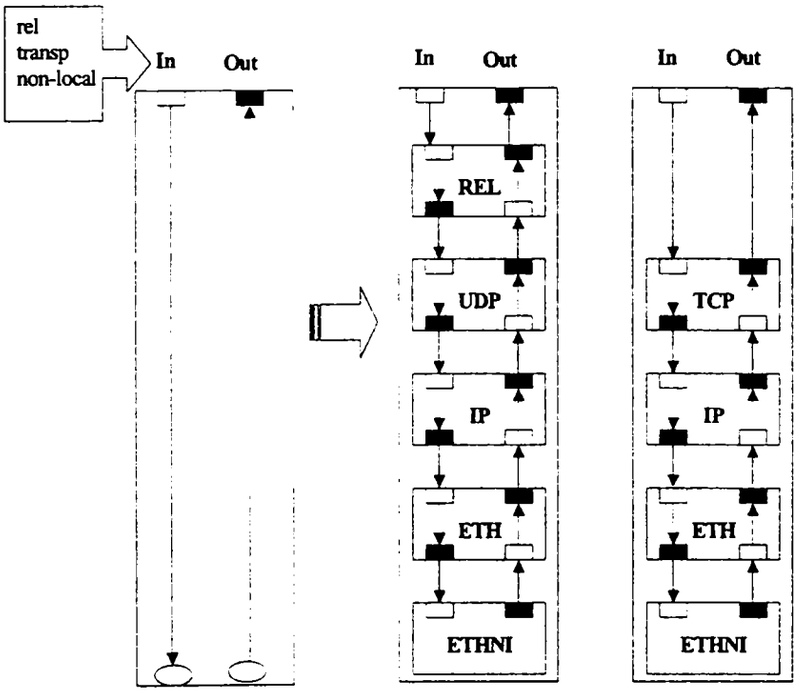


Figura 5.2: Configuraţii structurale pentru o Stiva de protocoale

BUPT

de protocoale trebuie să se afle o interfaţă de reţea, proprietatea *NetwInt*. Pe fluxul descendent, orice alta proprietate a stivei este deasupra proprietăţii *NetwInt*, iar pe fluxul ascendent *NetwInt* este deasupra (înaintea) oricărei alte proprietăţi.

Descrierea acestor constrângeri structurale în CCDL este prezentată în figura 5.3.

```
<?xml version="1.0" encoding="UTF-8"?> <!--CCDL for composable
Stack component--> <component
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="E:\comp.xsd" name="Stack">
    <componentExternals>
        <provides>
            <property name="stack"/>
        </provides>
        <port name="in" type="in" entrance="true"/>
        <port name="out" type="out"  entrance="true"/>
    </componentExternals>
    <componentInternals>
        <structuralConstraints>
            <basicStructuralConstraints>
                <start name="start_up">
                    <out name="out"/>
                </start>
                <end name="end_down">
                    <in name="in"/>
                </end>
                <flow name="downgoing" from="in" to="end_down.in"/>
                <flow name="upgoing" from="start_up.out" to="out"/>
                <containedProperty name="nw_interface"
                                        flowlocation="downgoing"/>
                <containedProperty name="nw_interface"
                                        flowlocation="upgoing"/>
                <orderRelation below="nw_interface" above="any"
                                        flowlocation="downgoing"/>
                <orderRelation below="any" above="nw_interface"
                                        flowlocation="upgoing"/>
            </basicStructuralConstraints>
            <interflowDependencies>
                <continuation from="downgoing" to="upgoing"/>
            </interflowDependencies>
            <contextDependencies/>
        </structuralConstraints>
    </componentInternals>
</component>
```

Figura 5.3: Descrierea CCDL a componentei Stiva de Protocoale

BUPT

Pentru configurarea stivei în concordanţa cu o mulţime de cerinţe client alcătuită din proprietăţile {`rel`, `transp`, `non-local`}, aceste cerinţe client sunt considerate împreună cu cerinţele introduse de constrângerile structurale ale componentei compozabile `Stiva`. Suma acestor cerinţe poate da ca şi rezultate configuraţiile din figura 5.2, obţinute prin propagarea cerinţelor într-un mecanism similar celui descris în exemplul din paragraful 4.3.1. Cazul nu se deosebeşte de cel prezentat la compoziţia liniară simplă.

O situaţie mai complexă este dată de un set de cerinţe client care conţine şi proprietăţi rafinate prin subproprietăţi, cum este:
REQUIRES `rel` WITH (`multimediarel`), `transp`, `non-local`.
Spre deosebire de situaţia anterioară, proprietatea `rel` trebuie să fie rafinată prin subproprietatea `multimediarel`. Aceste cerinţe înseamnă că este nevoie de o legătură de comunicaţie fiabilă, adecvată transmisiunilor multimedia. Pentru aceasta, în cadrul mecanismului de retransmitere specific asigurării fiabilităţii comunicaţiei trebuie adaptat astfel încât gestionarea pachetelor recepţionate şi pierdute se face printr-o strategie specială. Acest gen de adaptare nu ar fi posibilă dacă în compoziţie s-ar utiliza doar componente monolitice, de granularitatea unui nivel din stiva de protocoale. Cazul general permite o asemenea adaptare fină a proprietăţilor, prin intermediul componentelor compozabile. Soluţia iniţial posibilă $TCP - IP - ETH$ este respinsă, pentru că protocolul de fiabilitate, incorporat în componenta TCP, nu poate fi adaptat cerinţelor `multimediarel`. Cealaltă soluţie, $REL - UDP - TCP - ETH$ este procesată în continuare, pentru că componenta Protocol REL este o componentă compozabilă, şi se încearcă configurarea structurii acesteia astfel încât să corespundă cerinţei `multimedia`.

Constrângerile structurale pentru componenta compozabilă `ProtocolREL` sunt date în figura 5.4.

Structura componentei `ProtocolREL` este orientată pe două fluxuri principale, un flux descendent, corespunzător părţii de transmisie de date, şi un flux ascendent corespunzător părţi de recepţie de date. Există diverse strategii prin care se poate realiza asigurarea unei transmisii fiabile. In principiu, toate aceste strategii se bazează pe un mecanism care presupune că în partea de transmisie se aşteaptă un *feedback* de la partea de recepţie în legătură cu datele primite. Dacă acest *feedback* este necorespunzător, datele se consideră pierdute şi transmiţătorul trebuie să le retransmită. Există diverse politici de retransmisie, precum şi diverse strategii de feedback (*acknowledgements*). În cadrul constrângerilor structurale, se precizează doar faptul că o proprietate `Retransmission Strategy` trebuie să fie prezentă în partea superioară a fluxului descendent, iar pe fluxurile ascendente (partea de recepţie) trebuie să fie

BUPT

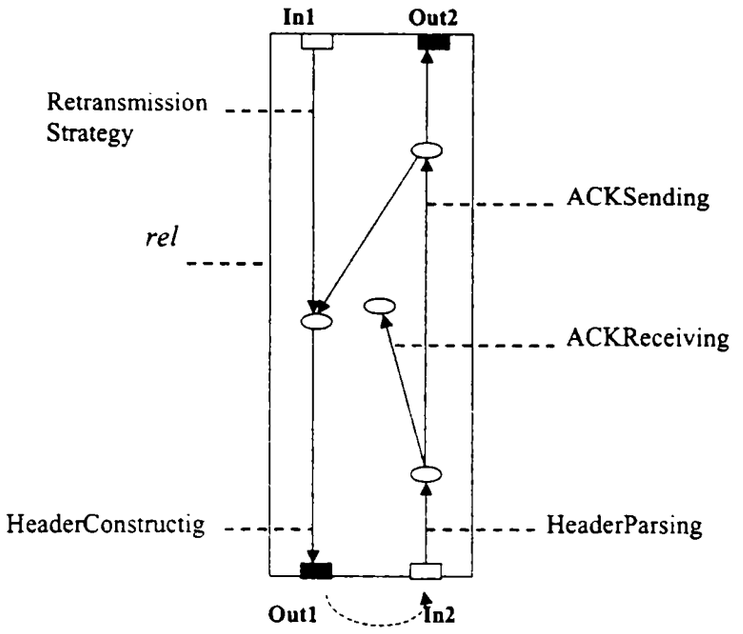


Figura 5.4: Constrângeri structurale pentru componenta compozabilă Protocol REL

prezente proprietăţile `ACKSending` şi `ACKReceiving`. Proprietatea `ACKSending` este furnizată de componente care implementează un mecanism de *acknowledgement* şi ţine evidenţa datelor recepţionate. Transmiterea *acknowledgement*-ului se face utilizând ieşirea de date din partea de transmisie. Proprietatea `ACKReceiving` este furnizată de componente care examinează datele care se recepţionează şi identifică datele propriu-zise de datele de semnalizare. Constrângerile structurale pentru componenta `ProtocolREL` mai specifică faptul că în partea inferioară a fluxului descendent este nevoie de o proprietate `HeaderConstruction`, iar pe fluxulascendent de `HeaderParsing`. Acestea sunt proprietăţile minime care trebuie să fie prezente în cadrul componentei `ProtocolREL`. Nu este însă impus tipul componentelor care pot furniza aceste proprietăţi, şi nici numărul componentelor care pot fi conectate pe fluxurile date: se pot adăuga la cerere componente care furnizează diferite alte facilităţi.

Pentru compunerea structurii componentei compozabile REL, cerinţa *multimediarel* este pusă pe fluxul principal, fluxul descendent cu originea în portul $In1$. Compunerea structuri acestei componente este ilustrată în figura 5.5.

Strategia de compoziţie decide într-un prim pas selectarea componentei *MultimediaRelStrategy* ca şi furnizor al unei strategii de retransmitere adecvate. La rândul său, componenta *MultimediaRelStrategy* are nevoie de suport pentru a putea ajusta în mod dinamic valoarea de *timeout* pentru retransmisii (are nevoie de

BUPT

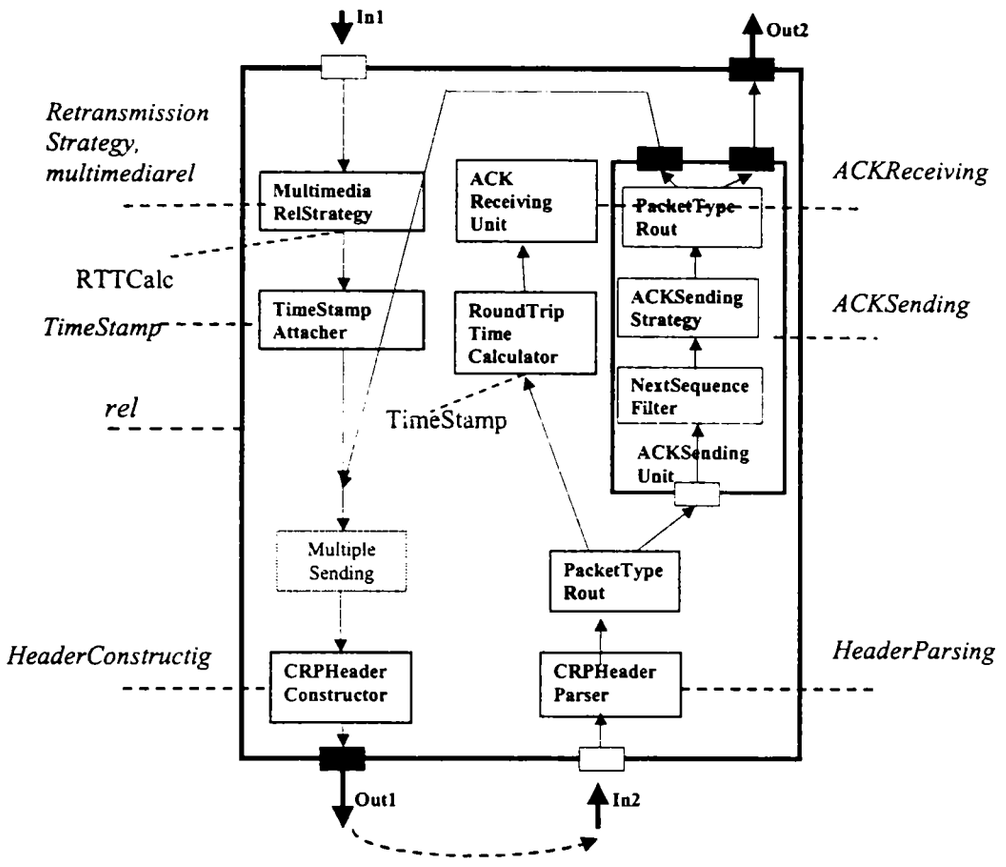


Figura 5.5: Compunerea structurii interne a componentei Protocol REL

calcularea timpului necesar pentru *roundtrip*). Aceasta duce la includerea componentei *RoundTripcalculator*, plasată pe fluxul ascendent, în concordanţă cu cerinţele proprii şi constrângerile structurale ale componentei *REL*. Componenta *RoundTripTimeCalculator* calculează timpul în care pachetele de date au parcurs traseul, pentru aceasta are nevoie să ştie care a fost momentul de timp la startul lor (exprimată prin cerinţa *TimeStamp*. Ca urmare, o componentă *TimeStampAttacher* este plasată pe fluxul descendent.

Conform constrângerilor structurale ale lui *REL*, trebuie asigurată o strategie de *Acknowledgement*. Deoarece nu s-au specificat preferinţe pentru acestea, se poate utiliza o strategie oarecare de *Acknowledgement*. In exemplul din figură se alege tipul *positive acknowledgements*. Componentele *AckReceivingUnit* şi *AckSendingUnit* realizează aceasta. Componenta *AckSendingUnit* este o componentă generică compozabilă la rândul său. Este necesară filtrare, şi se alege o componentă *NextSequenceFilter* pentru că aceasta are ca cerinţă ascendentă *multimediarel*.

BUPT

Pentru a ilustra modul în care această metodă de compoziţie tratează situaţiile care implică configuraţii neanticipate, se presupune că ulterior se dezvoltă o nouă componentă, *MultipleSending*. Pentru a îmbunătăţi performanţele protocolului de fiabilitate, se pot face din start transmisii multiple ale fiecărui pachet, pentru a reduce probabilitatea ca să fie necesară o retransmisie după trecerea intervalului de timeout. Cerinţele componentei *MultipleSending* impun ca intrarea ei să fie conectată la fluxul de la ieşirea unei componente care realizează o strategie de retransmisie. Dacă este cerută retransmitere multiplă, această cerinţa va fi îndeplinită conectând conectând o componentă de tip *MultipleSending* ca în figura 5.5.

5.2 Mediu integrat pentru instrumentaţie virtuală

Pentru dezvoltarea sistemelor prin compoziţie şi pentru întreţinerea lor, se pot defini *unelte* care asistă proiectantul într-un mod *interactiv* eventual având şi diverse grade de *automatizare*.

Acest capitol propune o unealta de compunere vizuală a sistemelor cu arhitectură multi-flux din componente compuse ierarhice, conforme modelului compoziţional care a fost definit în capitolul 3. Se prezintă o aplicaţie realizată utilizând această paradigmă, aplicaţia *SAAD-WIN*, un mediu integrat de măsurare bazat pe instrumente complexe. Acesta este o unealtă care facilitează descrierea vizuală şi desfăşurarea de diferite aplicaţii de măsură şi control.

Corectitudinea unei compoziţii construite vizual în mod interactiv este verificată aplicând regulile impuse de strategia de compoziţie definită în capitolul 4.

5.2.1 Prezentarea domeniului aplicaţiei

Un sistem de achiziţie şi procesare de date clasic trebuie să realizeze achiziţia, analiza şi prezentarea datelor. S-a impus treptat în ultima decadă ca un standard modelul *instrumentaţiei virtuale* pentru tratarea uniformă a acestor etape de procesări specifice acestor sisteme, dintre cele mai cunoscute pachete fiind LabView.

> Observaţia că programele pentru măsurări sunt ca şi instrumentele de măsură, dar cu deosebirea că utilizatorul interacţioneaza cu un terminal şi nu cu un panou frontal, a condus la conceptul de *instrument virtual*

declara Jeff Kodosky [KMR91], creatorul limbajului grafic de programare LabView.

Sistemele complexe alcătuite din instrumente virtuale sunt cel mai adesea descrise în mod vizual, modul lor de descriere este similar reprezentării prin scheme bloc

BUPT

corespunzând diferitelor nivele de abstractizare. Aceste scheme bloc determină modul de funcţionare şi interconectare al instrumentelor virtuale.

Concepte specifice domeniului aplicaţiei

Pentru aplicaţia descrisă în acest capitol s-a dezvoltat un model conceptual specific-domeniului şi s-a implementat **SAAD_WIN**, un mediu integrat de măsurare, bazat pe "instrumente". Contribuţiile în acest domeniu au fost publicate în [CMŞ95], [CŞM96], [GŞC+98], [CŞG97], [CJMŞ03].

Tratarea unitară a etapelor de achiziţie, procesare şi prezentare a informaţiilor este efectuată extinzând notiunea de "circuit de măsură" astfel încât acesta să conţină atât obiecte hardware, cât şi obiecte software. Informaţia parcurge un lanţ de transformări care sunt reprezentate schematic în figura 5.6.

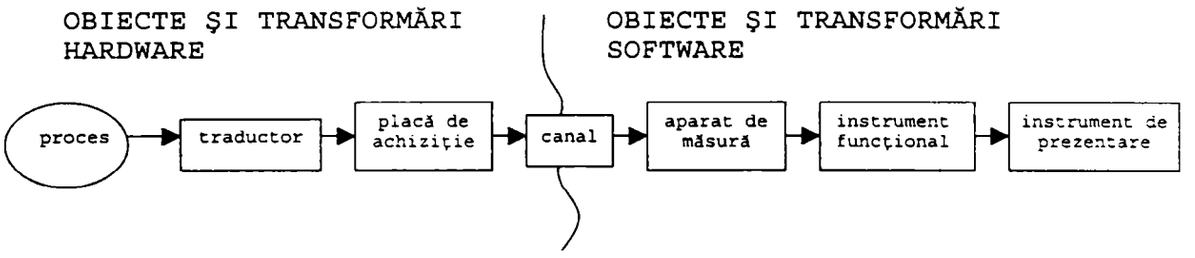


Figura 5.6: Lanţul de transformări în sistemul de achiziţie şi procesare

Fiecare obiect se referă la un lucru care poate fi identificat de utilizator fie ca un obiect concret din domeniul aplicaţiei, fie ca o abstracţie.

Sistemul software este construit pe baza următoarelor tipuri de componente fundamentale:

- canalele de intrare (CAN);
- aparatele de masura (AM);
- instrumentele funcţionale (IF);
- instrumentele de prezentare (IP).

Sistemul software trebuie să ofere utilizatorului imaginea următoarelor obiecte care există fizic: canalele de intrare şi aparatele fizice de măsură. La **canalele de intrare**, utilizatorul poate conecta **aparate fizice de măsură** corespunzătoare traductoarelor utilizate. Aceasta ar corespunde realizării unui montaj clasic. Un canal

BUPT

realizează transformarea numărului întreg corespunzător eşantionului achiziţionat în valoarea reală a tensiunii de la intrarea plăcii. Funcţia realizată de canal depinde de parametrii convertorului analog-numeric de pe placa de achiziţie. Aparatul de măsură calculează valoarea mărimii de intrare care este măsurată prin intermediul unui traductor. Funcţia aparatului de măsură, care este inversa funcţiei traductorului, este simplă dacă traductorul folosit este unul liniar, dar poate fi foarte complexă în caz contrar. La fiecare canal se poate conecta un aparat de măsura corespunzător traductorului utilizat. De fapt, aparatele de măsură sunt singurele care pot fi conectate la un canal din conexiunile de intrare. Ele încapsulează proceduri de extragere a datelor din bufferul conexiunii de intrare, realizeazând scalarea valorilor în conformitate cu parametrii traductorului asociat canalului şi asocierea valorii reconstituite a momentului de timp la care a fost măsurată fiecare valoare a mărimii de intrare. Mai multe canale de intrare constituie un set de conexiuni de intrare. Toate canalele apartinând aceluiaşi set de conexiuni de intrare vor fi citite în acelaşi ritm.

În unele situaţii este utilă şi simularea achiziţiei, adică generarea artificială a unor mărimi primare. Pentru aceasta, ansamblul canal de intrare - aparat fizic de măsură poate fi înlocuit cu un generator configurabil.

În plus faţă de aceasta, se pot defini şi implementa software diferite **instrumente matematice**, care încapsulează anumiţi algoritmi numerici de prelucrare a datelor ce sosesc pe "conexiunile lor de intrare", şi care furnizează rezultatele prelucrării. Aceste instrumente matematice se vor conecta la aparatele fizice de măsură sau la alte instrumente matematice, integrându-se în montajul experimental pentru a descrie complet fluxul de date de la achiziţia mărimilor primare pâna la obţinerea rezultatelor finale, a mărimilor derivate care interesează.

Un instrument matematic se caracterizează prin faptul că are n intrari şi o ieşire şi încapsulează un anumit algoritm numeric de prelucrare a datelor ce sosesc pe cele n conexiuni de intrare, furnizând datele prelucrării. S-au implementat instrumente matematice concrete, care pot realiza următoarele familii de operaţii: funcţii matematice, operaţii algebrice, metode de prelucrare a funcţiilor periodice, transformate Fourier şi altele.

Aparatele fizice de măsură, generatoarele şi instrumentele matematice au în comun faptul că furnizează valori numerice ale unor mărimi fizice măsurate direct sau derivate şi pot fi denumite generic **instrumente funcţionale**.

Prezentarea rezultatelor se poate face sub diferite forme. Stabilirea mărimilor care se reprezintă şi descrierea modului în care se reprezintă se poate face introducând conceptul de **instrument de prezentare**. Instrumentele de prezentare de diferite tipuri specializate pot fi conectate în cadrul circuitului de procesare la ieşirile instrumentelor

BUPT

funcţionale.

Instrumente din categoriile enumerate anterior pot fi instanţiate şi utilizate într-un **montaj** care descrie **circuitul de procesare**. Circuitul de procesare poate fi văzut ca un graf orientat aciclic, având în noduri instrumentele iar arcele fiind reprezentate de fluxurile de date existente între acestea. Pentru a obţine rezultatele finale ale procesărilor este necesară parcurgerea grafului într-o ordine care să reprezinte o sortare topologică ([Cre92]). În timpul parcurgerii în ordine topologică, asupra nodurilor se aplică o funcţie de prelucrare care este comanda de operare a instrumentului conţinut în nod. Un exemplu de circuit - graf orientat aciclic este reprezentat în figura 5.7.

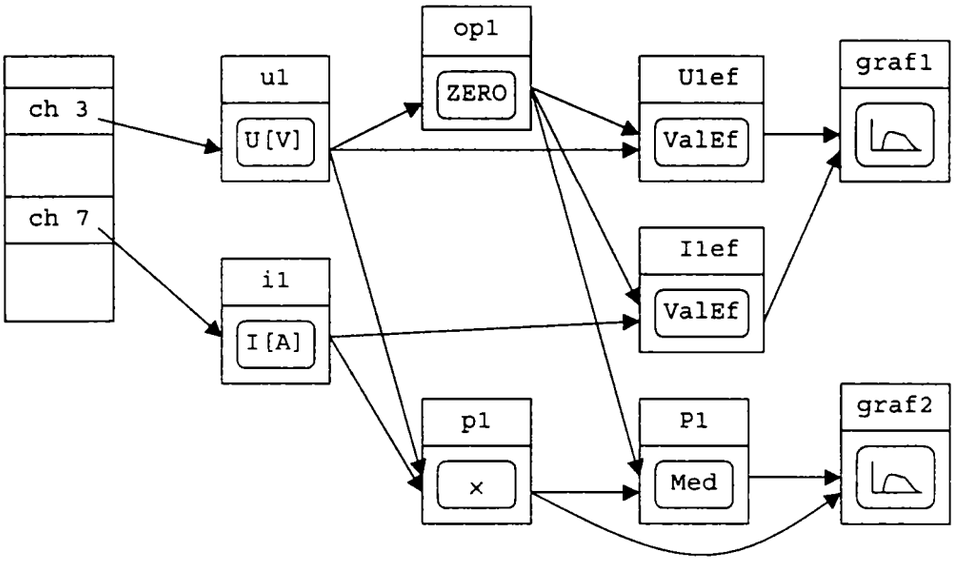


Figura 5.7: Circuit de procesare - graf orientat aciclic

În exemplul de circuit din figură, achiziţia se face pe două canale ($ch3$ şi $ch7$). La aceste două canale de intrare sunt conectate instrumentele de măsură $u1$ şi $i1$, care sunt de tip voltmetru (notat $U(V)$) şi ampermetru (notat $I(A)$) şi care redau mărimile de intrare sub formă tensiunii respectiv curentului măsurat. Instrumentul $op1$ este de tip detector de treceri prin zero (de tip $ZERO$) şi este folosit la determinarea perioadelor semnalului măsurat de $u1$, perioadă care este aceeaşi şi la semnalul $i1$. Se foloseşte un instrument multiplicator (de tip x) pentru a determina $p1$, valoarea momentană a puterii. Pentru calculul valorilor efective ale curentului $i1$ şi tensiunii $u1$ se folosesc două instrumente de tip $ValEf$, care realizează calculul pe perioadele detectate de $op1$.

O categorie mai specială o constituie **instrumentele complexe**. Acestea sunt instrumente a căror funcţionare nu este descrisă de un algoritm specific, ca şi în cazul celorlalte tipuri, ci de o schemă internă, compusă din alte instrumente. Instrumentele

BUPT

complexe pot fi atât predefinite, cât şi construite de utilizator. Un instrument complex este deci un instrument compus din alte instrumente, atomice sau alte instrumente complexe, care sunt tratate în continuare ca o singură nouă entitate. Rolul unui instrument complex este de a ajuta la structurarea circuitului de procesare, la simplificarea construirii circuitelor complexe. Instrumentele complexe sunt implementate de regulă prin componente compuse.

Instrumentele complexe pot fi tratate, din punct de vedere al includerii în circuitul de măsura, prin aceleaşi operaţii ca şi celelalte instrumente. Se defineşte conceptul general de "**instrument**" ca un obiect software, care, văzut din exterior este un *black-box* cu N intrări şi M ieşiri, pe care este capabil să le calculeze. Un instrument poate fi inserat într-un circuit de măsurare şi procesare, prin conectarea intrărilor sale la ieşirile altor instrumente. Circuitul de măsurare şi procesare defineşte complet fluxul de date, de la achiziţie şi până la prezentarea rezultatelor.

Conceptele prezentate în acest paragraf sunt specifice domeniului instrumentaţiei virtuale. Implementarea lor se face prin componente compatibile cu modelul compoziţional al arhitecturilor multi-flux cu componente ierarhice. Majoritatea aparatelor de măsură şi instrumentelor matematice elementare sunt implementate ca şi componente simple, iar instrumentele complexe sunt implementate de componente compuse.

Aplicaţii ale sistemului SAAD-WIN la testarea maşinilor electrice

Sistemul `SAAD_WIN` a fost utilizat în numeroase proiecte de testare a maşinilor electrice, rezultatele obţinute fiind publicate în [GBC+98], [BCG+94], [BCG+95]. Sistemul a fost utilizat printre altele pentru studiul regimului tranzitoriu al maşinilor electrice şi pentru studiul regimului electric deformat.

Regimurile tranzitorii care pot fi înregistrate cu acest program trebuie să fie de forma: iniţial (în regimul anterior considerat stabil), mărimea de intrare variază între anumite limite bine cunoscute. În momentul instaurării regimului tranzitoriu valorile mărimilor de intrare depăşesc aceste limite, fapt care declansează procesul de înregistrare care ne interesează. Sistemul poate fi configurat pentru utilizarea cu succes în aplicarea acestor regimuri în special la maşinile sincrone de putere mare (hidro şi turbogeneratoare). În cazul acestor maşini, din regimurile tranzitorii statice se obţine în scurt timp şi cu precizie ridicată un mare număr de parametri electromagnetici ([BCG+94]). În majoritatea regimurilor tranzitorii statice apare o atenuare a unui curent de la o valoare iniţială (curent continuu) la o valoare finală (de regulă zero). Este fundamentat teoretic şi verificat practic faptul că în cazul maşinilor sincrone, curba de atenuare a acestui curent este descrisă de expresia [BCG+94]:

BUPT

$$i(t) = I_1 \cdot e^{-\frac{t}{T_1}} + I_2 \cdot e^{-\frac{t}{T_2}} + \ldots + I_n \cdot e^{-\frac{t}{T_n}}$$

în care constantele I_k şi T_k, $k = 1 \ldots n$ sunt necunoscute. Determinarea acestora este importantă pentru aflarea unui mare număr de parametri ai maşinii sincrone. Algoritmul de separare a componentelor are la bază faptul că pentru aceste circuite constantele de timp diferă foarte mult:

$T_1 >> T_2 >> \ldots >> T_k$

Algoritmul de separare a acestor constante poate fi realizat sub forma unui circuit de procesare ca cel prezentat în figura 5.8.

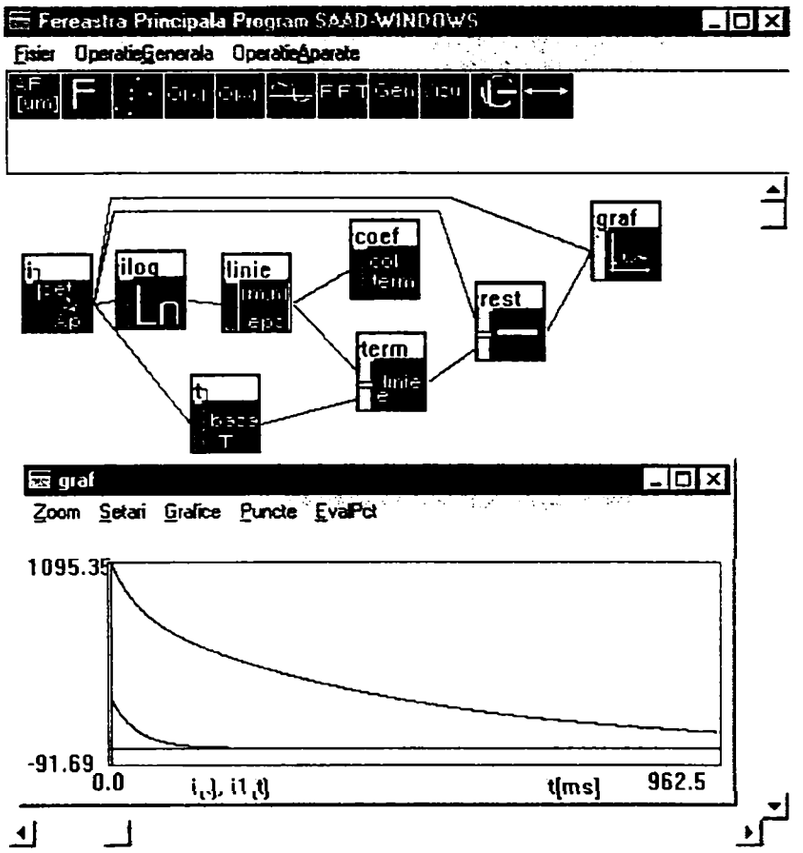


Figura 5.8: Circuit pentru separarea componentelor curentului

5.2.2 Compunerea vizuală

Pentru dezvoltarea şi întreţinerea interactivă a sistemelor din componente se utilizează frecvent sisteme de compoziţie vizuală ([dM95], [MCC03], [VSWJ03], [SVSJ03]). Aceată tehnologie derivă din limbajele vizuale şi programarea vizuală [Bur99], [Cit96].

BUPT

In acest subcapitol se prezintă compunerea vizuală a sistemelor din componente în arhitecturi multi-flux.

Metoda compoziţiei vizuale

De Mey defineşte în [dM95] compoziţia vizuală ca fiind:

> construcţia interactivă a unor aplicaţii funcţionale prin manipularea şi interconectarea directă a unor componente software prezentate în mod vizual.

Conexiunile între componente sunt realizate în acord cu regulile de compatibilitate stabilite de către un model compoziţional. Mediul vizual poate realiza operaţii de verificare a corectitudinii compoziţiei la diverse grade de complexitate [VSWJ03], [SVSJ03].

În procesul de dezvoltare de software orientat pe componente accentul se mută de pe partea tehnică de programare spre partea de aplicaţie. In acest context, realizatorul sistemului integrat e cel mai adesea mai apropiat de domeniul aplicaţiei decât de tehnicile de programare, ceea ce face ca utilizarea unor unelte de compoziţie vizuală să constituie o facilitate importantă pentru simplificarea procesului de dezvoltare.

Metodele de compoziţie vizuală presupun că o componentă are un *comportament* (*behaviour*) şi una sau mai multe *prezentări*.

Compunerea vizuală a sistemelor presupune realizarea de instanţe ale componentelor şi realizarea conexiunilor între componente prin manipularea exclusivă a reprezentărilor grafice ale componentelor. Prezentarea trebuie să fie sugestivă şi să conţină toate informaţiile necesare utilizatorului, referitoare la tipul componentei şi al interfeţei sale.

În general o unealtă de compoziţie vizuală este dedicată construcţiei interactive de aplicaţii dintr-un anumit domeniu. Uneltele de compozitie vizuală existente sunt în general dedicate câte unui domeniu specific (interfeţe grafice, aplicaţii pe flux de date etc) şi nu pot fi adaptate unui domeniu arbitrar. O generalizare a compunerii interactive trebuie să poată parametrizea unealta de compoziţie cu diferite *component framework*-uri. DeMey [dM95] indică posibilitatea realizării unei compoziţii vizuale generale prin separarea uneltei de *component framework* şi de regulile de compozitie. Ramân însă destule dificultăţi tehnice, şi datorită faptului că sistemele complexe sunt greu de vizualizat şi necesită tehnici flexibile de reprezentare.

BUPT

Compunerea vizuală a sistemelor multi-flux din componente ierarhice

Rezultatele prezentate în aceast subcapitol se referă la compoziţia vizuală a sistemelor cu arhitectură multi-flux din componente ierarhice. Modelul de componente cu care se lucrează este cel definit în capitolul 3.

Pentru compunerea vizuală, se extinde noţiunea de componentă - prezentată în capitolul 3 ca fiind o entitate dual ă {*descriere*, *implementare*} - cu aspectul de *prezentare*. O componentă se va considera caracterizată prin următoarea tripletă: {descriere, implementare, prezentare} (Figura 5.9).

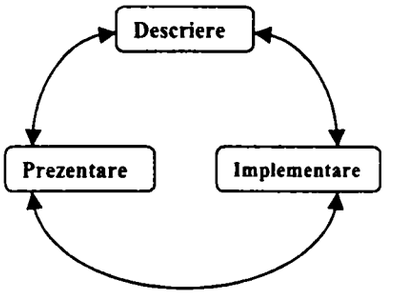


Figura 5.9: Componenta=descriere+implementare+prezentare

Prezentările vizuale ale componentelor sunt manipulate grafic in timpul compunerii vizuale a unui sistem. Realizarea de instanţe ale componentelor şi realizarea de legături între porturile acestora se fac exclusiv prin operaţii asupra prezentărilor componentelor. Legăturile se "desenează" printr-o operaţie cu semnificaţia *conectează portul de intrare pi al componentei X la portul de ieşire po al componentei Y*. Compoziţia sistemelor şi componentelor este realizată prin "desenarea" unor grafuri în care componentele sunt nodurile iar arcele sunt legăturile între componente.

In cazul compunerii vizuale (sau a unei metode de compunere interactivă în general), *descrierile* componentelor sunt utilizate la validarea unei compoziţii. Validarea compoziţiei implică verificarea faptului că toate conexiunile între componente sunt realizate în acord cu regulile de compatibilitate stabilite de către modelul compoziţional şi că toate contractele specificate în descrierile tuturor componentelor implicate sunt satisfăcute.

Modelul compoziţional cu care se lucrează permite conectarea oricărui port de intrare la orice port de ieşire. Nu sunt permise conexiuni între două porturi de intrare sau două porturi de ieşire, precum şi conexiuni care determină apariţia unor cicluri tranzitive în graful de componente. Validarea unei compoziţii pe baza contractelor specificate în descrierile componentelor se face conform unei strategii bazate pe mecanismul propagării cerinţelor, care va fi prezentată în capitolul 4.

BUPT

Se face distincţie între tipul unei componente (determinat de descrierea componentei) şi o instanţa a acelui tip de componentă. Utilizatorul uneltei de compoziţie vizuală are acces la un catalog al tuturor tipurilor de componente cunoscute la un moment dat, catalog actualizat pe măsură ce se crează noi tipuri de componente. Catalogul este accesibil utilizatorului uneltei de compoziţie vizuală în forma prezentărilor componentelor. Pentru fiecare tip de componentă pot fi realizate un număr oarecare de instanţe ale sale, utilizate în compoziţia sistemului. Modelul nu permite definirea recursiva a unui tip de componentă (definirea unui tip de componentă compusă având înstructura sa o instanţa a aceluiaşi tip).

Crearea unui nou tip de componentă poate fi făcută *prin programare* sau *prin compoziţie*. Modelul de componente presupune crearea şi utilizarea de componente compuse ierarhice. Componentele simple (atomice) sunt create numai prin programare, în timp ce componentele compuse pot fi create prin programare sau prin compoziţie vizuală.

Când un nou tip de componentă este creat prin programare trebuie specificate comportarea şi prezentarea acesteia. Comportarea se defineşte implementând o funcţie specifică de transformare a intrărilor în vederea producerii ieşirilor componentei. Prezentarea este definită creînd un simbol grafic (*icon*) de descriere a componentei. Acest *icon* trebuie să facă vizibile porturile componentei şi să reprezinte informaţiile esenţiale din descrierea acesteia. Prin intermediul acestui *icon* se manipuleaza componenta în unealta de compunere vizuală, care are acces şi la codul de implementare al componentei.

Când un nou tip de componentă compusă este creat prin compoziţie vizuală, se construieşte structura internă a acesteia. Structura unei componente compuse este o compoziţie în care sunt instanţiate componente de tipuri existente (atomice şi compuse). Comportarea componentei compuse rezultă din comportarea componentelor individuale care o alcătuiesc. Pentru o componentă compusă în mod vizual trebuie definită o interfaţă de compoziţie (specificând ce porturi ale setului de componente interne devin porturi ale componentei compuse) şi o prezentare vizuală a noii entităţi.

Figura 5.10 ilustrează ideea de construcţie vizuală a unei componente compuse şi de utilizare a acesteia ca şi o entitate nouă într-o compoziţie.

Figura 5.10.a prezintă un exemplu de sistem multi-flux care conţine şi componente compuse. Structura sistemului este compusă din nouă componente, $C1$–$C9$. In figură sunt reprezentate legăturile fizice existente între componente, care se realizează conectând intrările componentelor la porturile de ieşire de la care preiau date. Reprezentarea conexiunilor se face în acest fel (în sens opus fluxurilor de date) pentru că relaţia intrări-ieşiri este de N:1 (o ieşire poate fi conectată la N intrări, o intrare

BUPT

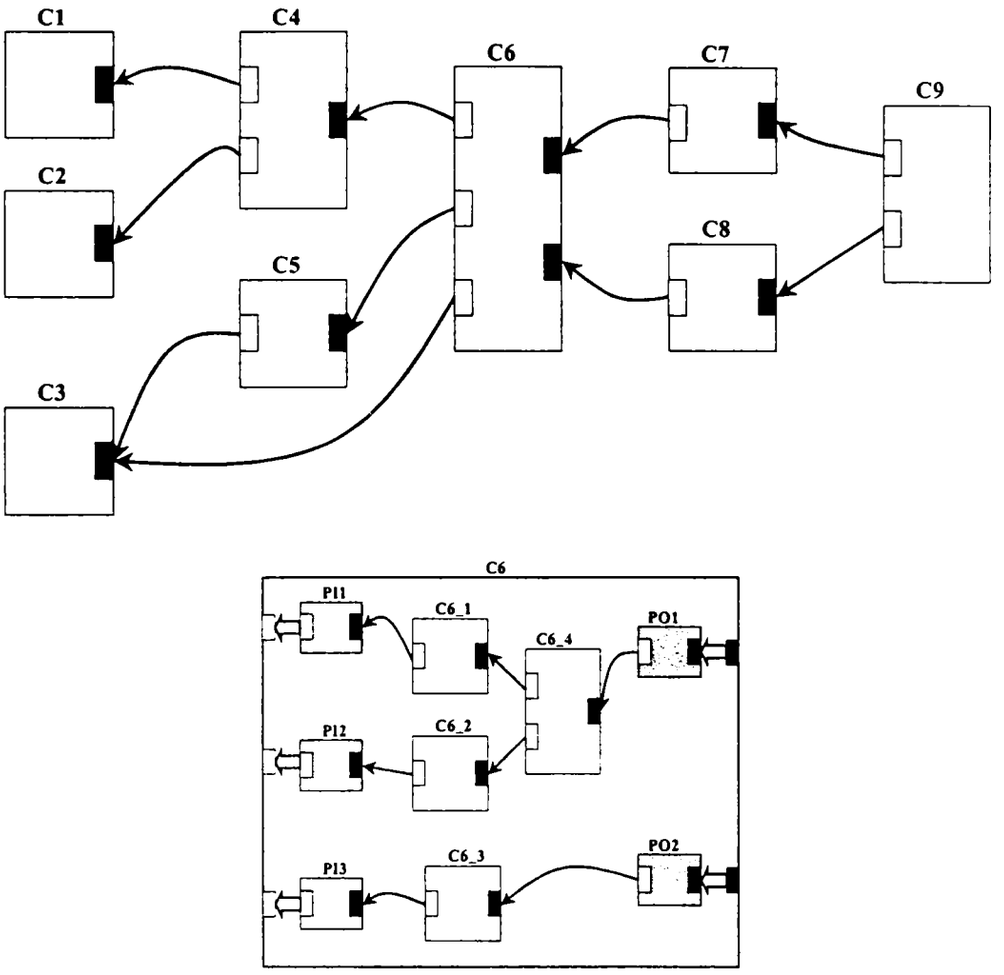


Figura 5.10: Exemplu de compunere vizuala. a. Sistem multi-flux continând şi componente compuse. b. Structura internă a unei componente compuse

poate fi conectată la o singură ieşire). Legăturile fizice exprimă relaţii de dependenţă, fiind în sens invers fluxurilor de date.

Structura internă a unei componente compuse este construită vizual în acelaşi mod ca şi sistemul, însă intervine în plus necesitatea de a specifica care sunt conexiunile de intrare şi ieşire pentru componenta compusă. Pentru aceasta, se folosesc componentele tip `PinIn` (pin de intrare) şi `PinOut` (pinde ieşire). Aceste componente se constituie în porturi de intrare–ieşire pentru componenta compusă şi au următoarele două sarcini: specificarea intrărilor şi ieşirilor care sunt externe componentei compuse şi asocierea fiecăreia dintre acestea cu un port al "black-box"-ului reprezentând componenta compusă. O componentă `PinIn` sau `PinOut` este caracterizată de un parametru având semnificaţia de "numarul portului", are o intrare şi o ieşire şi nu

BUPT

realizează o prelucrare, având doar rol de element de legatură.

In figura 5.10 se prezintă construcţia structurii interne a componentei compuse $C6$. Are trei porturi de intrare, corespunzătoare pinilor $PI1$, $PI2$, $PI3$ şi două porturi de ieşire, corespunzătoare pinilor $PO1$, $PO2$. Structura internă este determinată de compoziţia a patru componente, $C6_1$, $C6_2$, $C6_3$, $C6_4$. Legarea porturilor acestor componente interne la pinii de intrare şi ieşire a componentei compuse stabileşte interfaţa de compoziţie a acesteia.

Modul în care implementarea realizează apoi efectiv legăturile unei componente compuse cu restul schemei este ilustrat în figura 5.11.

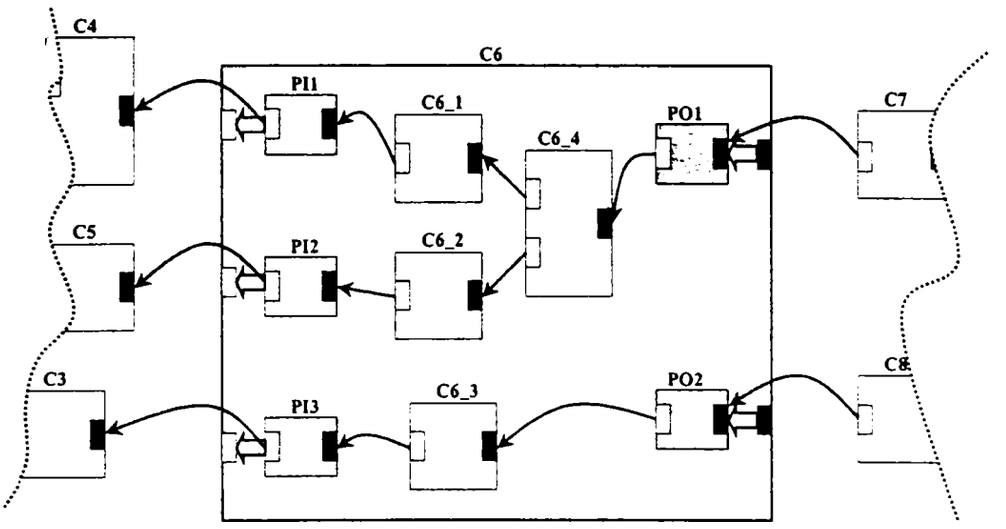


Figura 5.11: Exemplificarea modului de realizare a conexiunilor unei componente compuse cu restul sistemului

În această figură este redat un fragment din sistemul prezentat anterior, fragment care prezintă desfăşurarea legăturilor cu componenta compusă $C6$. La crearea prin compoziţie vizuală a lui $C6$ s-au trasat conexiunile subcomponentelor sale interne spre porturile externe. Cand componenta $C6$ e utilizată într-o compoziţie, se vor genera automat legăturile sale cu exteriorul, utilizându-se porturile sale externe ca în figura 5.11. În figură, săgeţile reprezintă legăturile fizice existente între componente: fiecare port de intrare are o legătură către portul de ieşire de la care preia date.

Compunerea vizuală a circuitelor de procesare

Compoziţia vizuală a circuitelor de procesare sau a unor noi tipuri de instrumente compuse în cadrul mediului SAAD-WIN are loc după procedeul prezentat în paragraful anterior referitor la compunerea vizuală a sistemelor multi-flux. Există o listă

BUPT

de prototipuri ale instrumentelor existente, o bibliotecă a tipurilor de componente existente, atât atomice cât şi construite prin compoziţie. Utilizatorul poate consulta această listă de prototipuri, din care selectează instrumentele pe care doreşte să le instanţieze.

Interfaţa cu utilizatorul a întregului sistem o constituie un editor de aplicaţii care transmite nucleului de achiziţie şi prelucrare comenzile şi parametrii pentru definirea circuitului de măsurare şi a experimentului. Acesta a fost dezvoltat ca un editor grafic, pentru descrierea în mod grafic a circuitului. Această descriere este asemănatoare cu desenarea sub formă de diagramă-bloc a unui montaj experimental.

Fereastra principală a aplicaţiei conţine doua zone: Fereastra de prototipuri şi fereastra pentru construcţia circuitului. Fereastra de prototipuri prezintă o listă a tuturor tipurilor de elemente de circuit care pot fi folosite. Fereastra pentru construcţia circuitului conţine o listă de referinţe la reprezentările elementelor de circuit. Pentru definirea interiorului unui instrument compus se deschide o fereastră de editare de acelaşi tip.

În figura 5.12 este reprezentat un ecran al programului `SAAD_WIN`, în care se disting elementele amintite anterior, şi anume: fereastra aplicaţiei; fereastra de prototipuri; fereastra pentru construcţia circuitului; reprezentările instrumentelor, canalelor şi legăturilor dintre acestea.

Mai există, dar nu sunt surprinse în figura 5.12, şi următoarele tipuri de ferestre: fereastra pentru definirea experimentului, care este legată de fereastra aplicaţiei, dar nu este aratata decât la cerere şi ferestre pentru definirea instrumentelor compuse.

Descrierea montajului implică operaţii de instanţiere a instrumentelor, de setare şi modificare a parametrilor acestora, precum şi precizarea conexiunilor dintre acestea. Modul de lucru caracteristic este următorul: utilizatorul poate consulta o colecţie în care se găsesc prototipurile tuturor instrumentelor cunoscute la un moment dat. Utilizatorul poate folosi şi instanţia în aplicaţia sa oricâte tipuri de instrumente şi oricâte instrumente de acelaşi tip ("stocul de instrumente" de orice tip este inepuizabil). Singura resursă limitată este numărul canalelor de intrare, al căror număr este impus de placa de achiziţie. Fiecărui instrument îi corespunde o reprezentare grafică sub forma de *black-box* cu un număr de intrări şi/sau ieşiri, în funcţie de tipul instrumentului. Pentru descrierea funcţiei realizate de instrument, aceste *black-box*-uri sunt etichetează cu simboluri (*icon*-uri) sugestive. Pentru descrierea instrumentelor compuse, se deschide o nouă fereastră de editare pentru schema internă a instrumentului. Interfaţa grafică cu utilizatorul furnizează uneltele de editare, modificare, configurare şi utilizare a montajului (circuitului de procesare).

BUPT

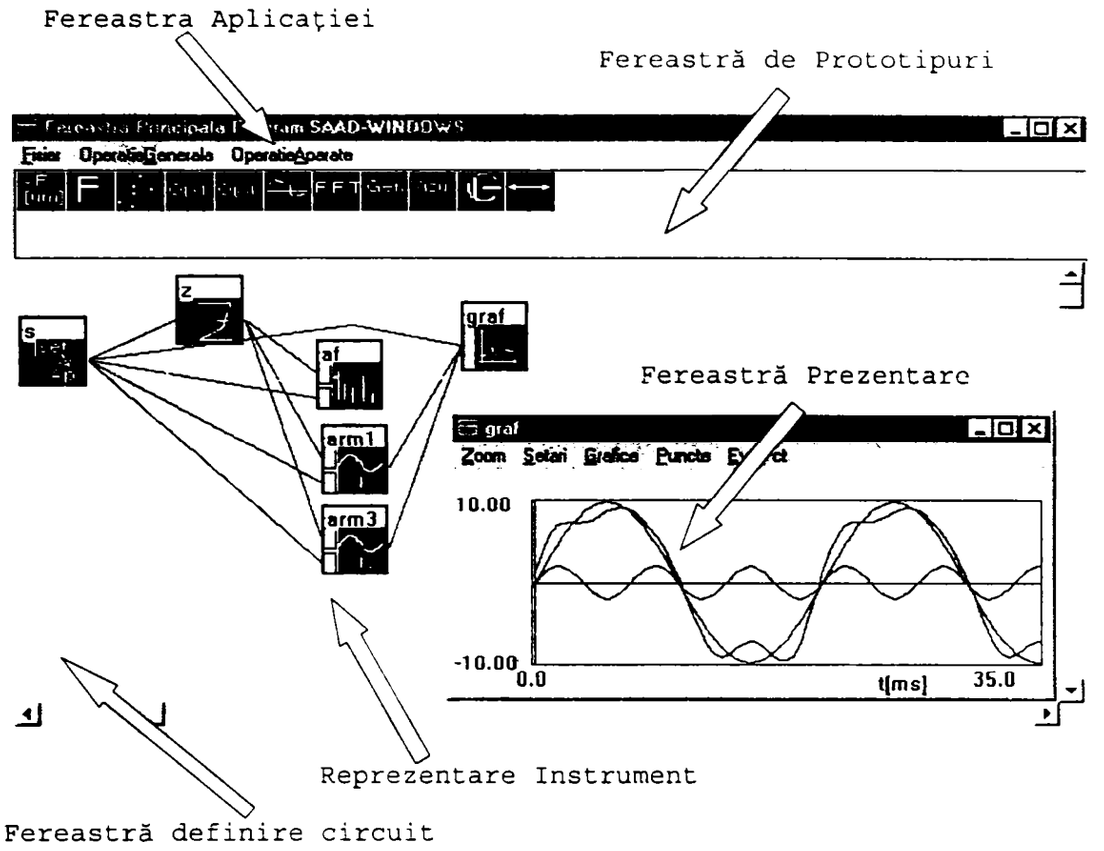


Figura 5.12: Compoziţia vizuală în mediul integrat de măsurare

5.2.3 Automatizări în construcţia unui circuit de procesare din instrumente virtuale

Cu ajutorul mediului SAAD_WIN bazat pe instrumente virtuale descrierea unei aplicaţii de măsură sau control se face compunând un circuit de procesare din instrumente ierarhic compozabile. În timpul operaţiei de "desenare" a schemei reprezentând montajul se fac automat şi unele operaţii de verificare şi validare a acţiunilor utilizatorului. Aceste operaţii de verificare se încadrează în două mari categorii:

- Verificarea respectării arhitecturi multi-flux
- Verificarea corectitudinii semantice a circuitului compus.

Verificarea respectării arhitecturii multi-flux

In această categorie se încadrează operaţii precum:

BUPT

- verifică apariţia unor situaţii de conexiuni nepermise între anumite tipuri de instrumente;

- verifică apariţia unor cicluri în montaj. Aceasta este o operaţie foarte importantă, având în vedere faptul că montajul defineşte fluxul de date prin sistem. Existenţa unui ciclu în schema care defineşte montajul este echivalentă cu existenţa unui ciclu infinit în program. Aceasta rezultă din modul de interpretare a montajului, al cărui rol este de a impune ordinea de operare a instrumentelor.

Menţinerea consistenţei montajului mai implică şi următoarele operaţii pentru asigurarea certitudinii că starea internă a instrumentelor individuale este cea corectă:

- la stergerea unei instanţe care este un nod în graful-montaj, se asigură faptul că toate instrumentele aflate în noduri spre care există un drum în graful orientat vor fi aduse în starea interna "date nedisponibile";

- acelaşi lucru este valabil şi în cazul în care starea internă a unui nod devine "date nedisponibile" din diverse motive (schimbarea parametrilor, comanda explicită a utilizatorului şi altele).

Verificarea corectitudinii semantice a unui circuit compus

Se face şi o verificare automată a corectitudinii semantice a circuitului de procesare, pornind de la descrierile $CCDL$ ale fiecărei componente implicate prin aplicarea modelului compoziţional propus în această teză ([ŞBVC02]).

Verificarea automată a corectitudinii semantice a unui circuit de procesare care a fost construit interactiv se face aplicând mecanismul propagării cerinţelor, aşa cum s-a descris în paragraful 4.2.3. Se prezintă în continuare un scenariu–exemplu pentru ilustrarea modului în care se face această verificare.

În figura 5.12 a fost descris un exemplu de circuit pentru studiul regimului deformat. Pentru analiza mărimilor periodice nesinusoidale se utilizează descompunerea semnalelor periodice nesinusoidale în armonici (semnale periodice sinusoidale). În `SAAD_WIN` sunt definite, printre altele, instrumente care realizează următoarele operaţii: calculul spectrului unui semnal periodic (amplitudinile tuturor armonicilor sunt reprezentate sub forma de histograme); extragerea semnalului corespunzător unei anumite armonici, etc. Pentru extragerea armonicilor care compun un semnal periodic nesinusoidal se foloseşte un montaj ca cel din figura 5.12. Utilizând acest montaj, se pot vizualiza atât amplitudinile armonicilor care compun semnalul periodic nesinusoidal

BUPT

sub formă de histograme, precum şi armonicile 1 si 3 în formă grafică. Instrumentele **arm1** şi **arm3** extrag armonicile 1 şi 3 din semnalul original. Deoarece se doreşte reprezentarea acestor armonici şi a semnalului original, s-a utilizat instrumentul `graf`, care reprezintă toate aceste semnale. Pentru vizualizarea amplitudinilor armonicilor care compun semnalul original se va seta ca parametru al instrumentului denumit `af` un întreg care reprezintă numărul de armonici care se doreşte să fie vizualizate.

Circuitul de procesare din figură 5.12 este reluat aici, din punct de vedere al componentelor conţinute, în figura 5.13.

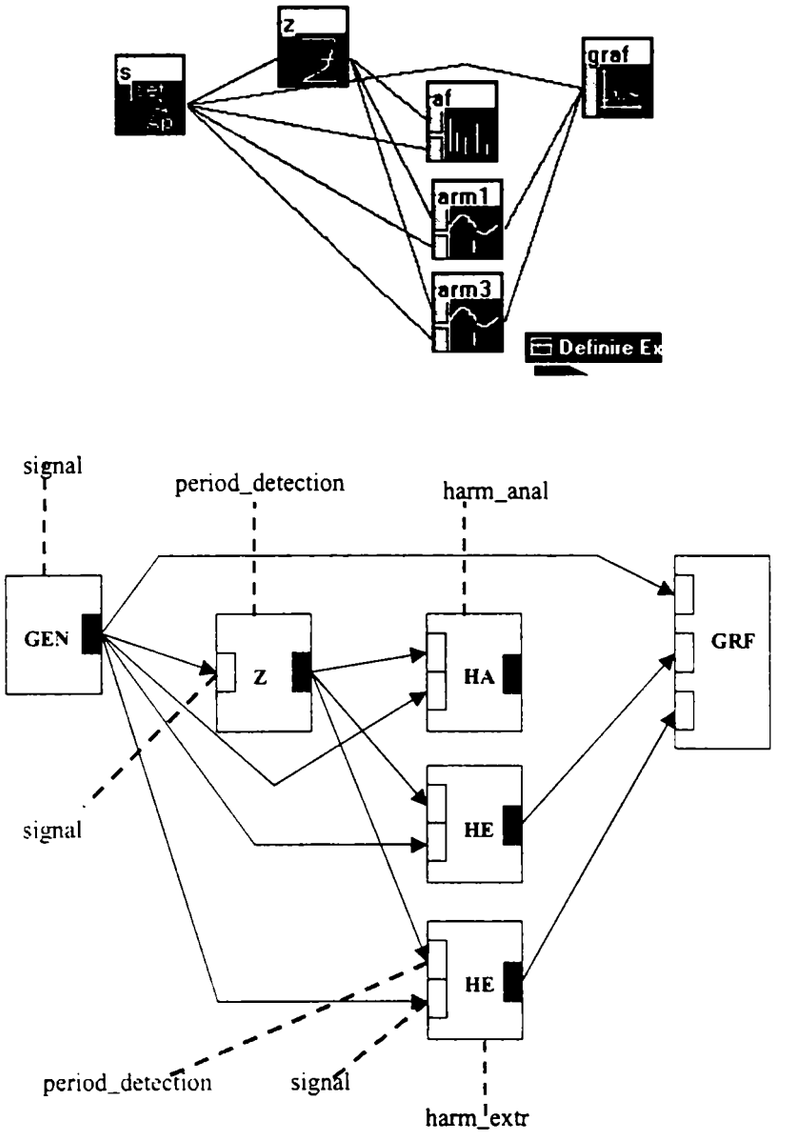


Figura 5.13: Componentele din care este compus circuitul de procesare

Acest circuit efectuează analiza Fourier a unui semnal şi realizează separarea

BUPT

primei şi a celei de-a treia armonici. Circuitul din figură conţine şase instanţe de componente: un generator de semnal (`GEN`), un analizor de spectru Fourier (`HarmonicsAnalyzer HA`), două instanţe ale tipului decomponentă extractor de armonici (`HarmonicsExtracter HE`), o componentă de tip `ZeroDetector` şi o componentă de reprezentări grafice. Pentru decongestionarea figurii, în figură nu sunt reprezentate grafic toate proprietăţile cerute şi furnizate de către toate componentele. Componenta `GEN` este caracterizată de proprietatea furnizată `signal`. Componenta `HarmonicsAnalyzer HA` este caracterizată de proprietatea furnizată `harm_anal`, iar componentele `HarmonicsExtracter HE` sunt caracterizate fiecare de proprietatea furnizată `harm_extract`. Toate aceste componente (`HA`, `HE`) au câte două intrări cu cerinţe similare: au nevoie la intrări de semnalul analizat şi de date despre perioada semnalului analizat, exprimate prin proprietăţile cerute la porturile de intrare (`signal`, `period_detection`). Perioada semnalului este furnizată de componenta detector de treceri prin zero (care funcţionează pe principiul că se presupune că semnalul îşi schimbă polaritatea de un anumit număr de ori intr-o perioadă; această componentă contorizează trecerile prin zero.) Cum însă nu din orice măsurătoare se poate extrage direct informaţia referitoare la perioadă direct din semnalul analizat, uneori se folosesc pentru aceasta alte semnale cu aceeaşi perioadă cu rol de pilot. Din aceste considerente există un tip separat de componentă, tipul detector de perioade `ZeroDetector`, care furnizează proprietatea `period_detection`, această funcţie neputând fi incorporată în analizorul de armonici.

Fiecare componentă din cele cunoscute are asociată o descriere CCDL care îi specifică interfaţa de conectare şi contractele pe care trebuie să le respecte, în termeni de proprietăţi furnizate–proprietăţi cerute. Este dată aici ca exemplu descrierea componentei extractor de armonici în figura 5.14.

După cum se observă din descrierea CCDL a componentei *HarmonicsExtracter*, aceasta are două porturi de intrare (portul $In2$ pentru semnalul de analizat şi portul $In1$ pentru informaţiile referitoare la perioadă) şi un port de ieşire, *Out*. Cerinţele (*requirements*) asociate porturilor de intrare indică faptul că la portul de intrare $In1$ trebuie să fie conectată o componentă care furnizează proprietatea $period-detection$, iar la portul de intrare $In2$ o componentă care furnizează un semnal, proprietatea *signal*.

In circuitul din figură, se utilizează o componentă de tip *ZeroDetector*. O altă posibilitate este ca un alt semnal, cu rolul de pilot al semnalului analizat, să furnizeze informaţiile de detectare a perioadei dacă aceste informaţii nu pot fi extrase direct din semnalul de analizat.

Chiar dacă *HarmonicsExtracter* nu este o componentă compozabilă, ea este o

BUPT

```
<component name="HarmonicsExtracter">
    <component_external>
        <provides>
            <property name="harm_extr">
                <parameter name="n" type="integer"/>
            </property>
        </provides>
        <port name="In1" type="In">
            <requires>
                <required_property
                name="period_detection" assertion="yes"/>
            </requires>
        </port>
        <port name="In2" type="In">
            <requires>
                <required_property
                name="signal" assertion="yes"/>
            </requires>
        </port>
        <port name="Output" type="Out">
        </port>
    </component_external>

    <component_internal>
        <basic_struct_constr>
            <flow name="extractor" from="In2" to="Out"/>
        </basic_struct_constr>
    </component_internal>
</component>
```

Figura 5.14: Descrierea CCDL a componentei HarmonicsExtracter

componentă compusă, având mai multe intrări, şi este necesar să se descrie constrângerilesale structurale pentru a-i specifica relaţiile interne de dependenţă între porturi. S-a definit un singur flux intern, fluxul $In2 \rightarrow Out$. Nu există un flux intern $In1 \rightarrow Out$ pentru că portul $In1$ are rol de intrare de control, necreând o relaţie între porturile $In1$ şi Out.

Pentru verificarea compoziţiei, se parcurg toate fluxurile existente în circuit (care sunt drumuri în graful orientat care reprezintă circuitul), verificând că pe fiecare flux toate proprietăţile cerute sunt satisfăcute. In circuitul din figura 5.13 se identifică şapte fluxuri. Acestea sunt:

BUPT

$GEN.Out \rightarrow GRF.In1$
$GEN.Out \rightarrow Z.In \rightarrow Z.Out \rightarrow HA.In1$
$GEN.Out \rightarrow HA.In2$
$GEN.Out \rightarrow Z.In \rightarrow Z.Out \rightarrow HE(1).In1$
$GEN.Out \rightarrow HE(1).In2 \rightarrow GRF.In2$
$GEN.Out \rightarrow Z.In \rightarrow Z.Out \rightarrow HE(2).In1$
$GEN.Out \rightarrow HE(2).In2 \rightarrow GRF.In3$

Pe fiecare dintre aceste fluxuri se verifică faptul că toate cerinţele de pe un flux sunt satisfăcute de proprietăţi furnizate pe respectivul flux şi toate relaţiile de ordine sunt satisfăcute . De exemplu, pentru ultimele două fluxuri din lista enumerată, situaţia proprietăţilor cerute şi furnizate se prezintă astfel:

PROV *signal* $\rightarrow$ REQ-UP *signal*, PROV *period – detection* $\rightarrow$ REQ-UP *period – detection*

respectiv

PROV *signal* $\rightarrow$ REQ-UP *signal* $\rightarrow$ PROV *harm – extr*

De exemplu, pe primul flux din cele două analizate, cerinţele existente sunt *signal*. asociat portului de intrare a componentei Z $Z.In$ (deci este o cerinţa ascendentă pe flux) şi *period – detection*, asociată portului $In1$ al componentei $HE(2)$. Ambele cerinţe sunt îndeplinite, componenta GEN furnizează proprietatea *signal* iar componenta Z furnizează proprietatea *period – detection*.

Strategia de compoziţie descrisă în acest capitol se poate aplica şi pentru generarea circuitului de procesare. În dezvoltări ulterioare ale mediului integrat de măsurare şi control bazat pe instrumente virtuale, se poate implementa generarea automată a circuitului de procesare pornind de la enunţarea cerinţelor care descriu scopul principal al experimentului. Această facilitate de compoziţie automată transformă `SAAD_WIN` într-un mediu de măsurare inteligent. În dezvoltări ulterioare ale `SAAD_WIN`, strategia de compoziţie poate fi utilizată şi pentru adaptarea dinamică a circuitului de procesare. De exemplu, această posibilitate de adaptare dinamică ar putea fi utilă în cazul în care semnalul de intrare este afectat la un moment dat de perturbaţii. Componenta detectoare de zerouri are nevoie de un semnal cât mai neted, deci circuitul de procesare va fi ajustat astfel încât sa se includă un filtru la intrarea detectorului de zerouri.

5.3 Concluzii

Pentru dezvoltarea şi întreţinerea sistemelor prin compoziţie de componente, se pot defini *unelte* care asistă compunerea într-un mod *interactiv* sau unelte care au şi

BUPT

diverse grade de *automatizare* a compoziţiei. Asemenea unelte pot fi dezvoltate numai pe baza unui model compoziţional precis definit.

In acest capitol s-a ilustrat validarea modelului compoziţional propus în această teză pe cazuri de generare automată, respectiv de verificare a unei compoziţii. De asemenea, cazurile sunt din două domenii de aplicaţii diferite, modelul compoziţional propus fiind specific stilului arhitectural multi-flux, independent de domeniile de aplicaţii.

Primul domeniu de aplicaţie prezentat (în subcapitolul 5.1) este cel al protocoalelor de reţea auto-configurante. Auto-configurarea este realizată prin ajustarea, prin compoziţie automată de componente, a structurii, astfel încât să răspundă cel mai bine cerinţelor curente ale mediului. Pentru atingerea acestui obiectiv s-a integrat `Composer`-ul dezvoltat în această teză (descris capitolul 4) cu un cadru de componente pentru realizarea de protocoale de reţea, obţinând în acest fel configurarea automată a acestora.

Un alt domeniu în care se poate utiliza modelul compoziţional propus în această teză este domeniul instrumentaţiei virtuale. Pentru dezvoltarea şi întreţinerea interactivă a sistemelor din componente se utilizează frecvent sisteme de compoziţie vizuală. Compoziţia vizuală este construcţia interactivă a unor aplicaţii funcţionale prin manipularea şi interconectarea directă a unor componente software prezentate în mod vizual. In subcapitolul 5.2 este prezentat sistemul SAAD-WIN, un mediu integrat de măsurare bazat pe instrumente realizat (proiectat şi implementat) aplicând principiile compoziţiei vizuale a componentelor ierarhic compozabile. Conexiunile între componente sunt realizate în acord cu regulile de compatibilitate stabilite de către un model compoziţional. Mediul de compoziţie vizuală poate realiza operaţii de verificare a corectitudinii compoziţiei la diverse grade de complexitate. Verificarea semantică a unei compoziţii se face pe baza mecanismului de propagare a cerinţelor introdus în capitolul 4.

BUPT

Capitolul 6

Concluzii. Principalele contribuţii ale tezei

Concluzia principală a acestei teze o constituie faptul că pentru stabilirea unui cadru sistematic de compoziţie a componentelor sunt necesare *modele compoziţionale*. Obiectivul oricărui proces de compoziţie este construcţia unui sistem cu anumite proprietăţi impuse prin utilizarea de componente a căror proprietăţi individuale sunt cunoscute.

În această teză se propune un model compoziţional independent de domeniul de aplicaţie, specific pentru compunerea sistemelor cu un anumit stil arhitectural: sisteme din componente compozabile în arhitectură multi-flux. Modelul compoziţional propus conţine o schemă şi un formalism de descriere a componentelor în forma imbajului CCDL, precum şi o strategie de compoziţie bazată pe cerinţe.

Prin intermediul componentelor ierarhice compozabile, modelul propus poate stăpâni complexitatea sistemelor, realizând compoziţii adaptate unor cerinţe nuanţate (*fine-tuned*).

Dezideratul realizării compoziţiei automate pornind doar de la cerinţele exprimate de client este un lucru foarte greu de atins fără introducerea unor specificaţii sau constrângeri suplimentare. In modelul propus în această lucrare, acest lucru a fost realizat prin precizarea constrângerilor structurale care se impun, detaliind structura entităţilor compozabile la nivel de fluxuri şi noduri de conexiune interne. Introducerea constrângerilor structurale este un element de bază în modelul compoziţional propus de aceasta lucrare. Acesta este un mecanism nou de a gestiona variabilitatea unui sistem, lăsând suficiente grade de libertate pentru a permite configurări ale căror detalii nu au fost planificate de dinainte. Configuraţia unei componente compozabile (a unui sistem compozabil) nu e limitată la alternative cunoscute în avans şi nici nu se fixează numărul şi tipul componentelor care se pot utiliza. Rezultă de aci un avantaj al modelului propus, faptul că este deschis compoziţiilor neanticipate.

BUPT

Pentru gestionarea componentelor, sunt necesare sisteme de informaţii software care să asigure reprezentarea şi regăsirea componentelor. Pentru dezvoltarea şi întreţinerea sistemelor, se pot defini unelte care asistă realizarea compoziţiei într-un mod *interactiv*, cum sunt sistemele de compoziţie vizuală, sau unelte care au diverse grade de *automatizare* mergând până la generarea automată. Modelul compoziţional propus în această teză a fost utilizat în ambele situaţii.

Modelul compoziţional propus a fost definit în urma experienţei dobândite în dezvoltarea unor aplicaţii de compoziţie din două domenii de aplicaţii diferite: protocoale de reţea şi instrumentaţie virtuală.

Principalele contribuţii ale tezei

Contribuţia majoră a acestei teze este definirea teoretică şi validarea practică a unui model compoziţional original pentru sisteme cu arhitecturi multi-flux bazate pe componente compozabile. Introducerea conceptului de componentă compozabilă este un element esenţial care stă la baza acestui model.

Contribuţiile acestei lucrări se înscriu în următoarele trei categorii:

1. Definirea schemei şi formalismului de descriere cu componente compozabile
2. Definirea strategiei de compoziţie bazate pe cerinţe
3. Validarea modelului compoziţional

Contribuţiile aduse prin definirea schemei şi formalismului de descriere cu componente compozabile (capitolul 3) pot fi rezumate astfel:

- Introducerea noţiunii de componente *compozabile* ierarhice. O componentă compozabilă (secţiunea 3.2) are o identitate proprie bine definită de porturile sale de interacţiune cu mediul, serviciile pe care le furnizează şi cele de care depinde. Structura internă a unei asemenea componente nu este fixată, ci poate fi compusă în mod dinamic, în cadrul stabilit de un set de constrângeri structurale. Descrierea unei componente compozabile înseamă mai mult decât descrierea interfeţei sale:

  Componentă compozabilă = Interfaţă + Constrângeri structurale.

  Modelul de componente compozabile a fost publicat în articolele: [ŞVB03], [ŞJBV02]. Lucrarea [ŞVB03] a fost citată într-o lucrare publicată de Ivers şi Wallnau ([IW03]) şi contribuţiile ei analizate de Matejka într-un seminar ştiinţific ţinut la Charles University, Praga, Cehia ([Mat03]).

BUPT

- Introducerea mecanismului de *constrângeri structurale* (paragraful 3.2.2). Constrângerile structurale pentru o componentă compozabilă exprimă linii directoare flexibile ce ajută la stabilirea compoziţiei structurii interne a componentei. În principiu, rolul constrângerilor structurale pentru o componentă (un sistem) compozabilă este de a stabili elementele esenţiale ale structurii acesteia. Constrângerile structurale definesc un set de cerinţe minime pe care trebuie să le îndeplinească structura internă a unei componente pentru a-i asigura acesteia o identitate precizată. Constrângerile structurale *nu* descriu şi nu fixează configuraţia structurii interne. Mecanismul constrângerilor structurale reprezintă o metodă nouă de management al variabilităţii. Această metodă realizează un echilibru optim între necesitatea de a permite variaţii neanticipate ale structurii unei componente sau ale unui sistem şi necesitatea de a menţine identitatea şi corectitudinea sa.

- Propunerea unei noi scheme de descriere a componentelor şi sistemelor compozabile. Caracteristic acesteia este faptul că descrierea unei componente implică stabilirea interfeţei acesteia în termeni de *porturi*, a contractelor semantice în termenii unor *proprietăţi* cerute şi furnizate, şi a *constrângerilor structurale* pentru componentele compozabile.

  Principalele avantaje ale acesteia sunt:

  - Se bazează pe specificaţii cu o complexitate redusă.
  - Realizează descrierea componentelor compozabile, permiţând un grad ridicat de variabilitate *neanticipată*, spre deosebire de alte soluţii clasice prezentate în literatură pentru definirea variabilităţii unei configuraţii cum sunt sunt utilizarea de skeleton-uri şi puncte de variabilitate.

- Definirea *limbajului CCDL* de descriere a componentelor compozabile. După cum a fost prezentat în secţiunea 3.3, acest limbaj prezintă unele caracteristici asemănătoare limbajelor din familia ADL (limbaje de descriere arhitecturală) şi limbajelor din familia IDL (limbaje de descriere a interfeţelor), dar fără a se încadra în nici una din aceste categorii. Acest limbaj poate exprima atât contractele componentelor compozabile cât şi constrângerile structurale pentru compoziţia lor, lucruri care nu s-ar putea realiza în cadrul limbajelor existente din familiile limbajelor IDL şi ADL. Descrierile de componente realizate în limbajul CCDL pot fi utilizate de unelte de compoziţie automată care implementează strategii de compoziţie bazate pe cerinţe. De asemenea, aceste

BUPT

descrieri pot fi utilizate şi de unelte care realizează verificarea corectitudinii semantice a unei compoziţii. Conceptele caracteristce limbajului CCDL au fost publicate în [ŞVB03].

- Implementarea unui parser pentru limbajul CCDL şi a unui *repository* de descrieri de componente (secţiunile 3.3.3, 3.4).

Contribuţiile principale referitoare la definirea strategiei de compoziţie pentru arhitecturi multi-flux din componente compozabile (capitolul 4) sunt:

- Definirea mecanismului *propagării cerinţelor în arhitecturi multi-flux cu componente compozabile* (secţiunea 4.2). Mecanismul propagării cerinţelor este definit ca o transpunere în domeniul componentelor şi o generalizare a principiului propagării cerinţelor definit de Perry pentru clauze requires-provides în [Per89]. Mecanismul propus este de asemenea o generalizare a metodei lui Batory ([BG97]), prin adaptarea metodei propagării pentru sisteme care nu mai sunt strict liniare.

- Definirea unei *strategii de compoziţie* bazată pe mecanismul propagării cerinţelor (secţiunea 4.3). O contribuţie importantă a acestei lucrărări o constituie faptul că mecanismul propagării cerinţelor este folosit aici şi ca motor de căutare a soluţiilor în *generarea* automată a unei compoziţii, spre deosebire de lucrările cunoscute din literatură unde variante ale acestui mecanism au fost folosite doar pentru *verificarea* unor compoziţii date.

  Avantajele strategiei de compoziţie propuse în această lucrare sunt:

  - permite sinteza automată a sistemelor pe baza unui repository *deschis* de componente, cu un set *expandabil* de proprietăţi
  - metoda este independentă de domeniul aplicaţiei, fiind aplicabilă sistemelor caracterizate de stilul arhitectural multi-flux

  Strategia de compoziţie a fost publicată în lucrările: [ŞMBV03], [ŞJBV02], [ŞVB02].

- Definirea modelului `Composer-Builder` pentru sisteme autoconfigurante (secţiunea 4.4) obţinute prin compoziţie automată de componente. Implementarea unui `Composer` bazat pe strategia de compoziţie definită în secţiunea 4.3 a acestei teze.

Contribuţiile aduse pentru validarea modelului (capitolul 5):

BUPT

- Realizarea unei aplicaţii de protocoale de reţea autoconfigurante prin integrarea **Composer**-ului într-un *framework* dat care asigură infrastructura pentru construcţia de protocoale dinamice de reţea. Acest lucru s-a făcut ca parte a unui proiect de cercetare amplu referitor la definirea unui *terminal generic* [Pep02]. Descrierea aplicării strategiei de compoziţie automată în acest domeniu al protocoalelor de reţea autoconfigurante a fost făcută şi în lucrările [ŞMM01], [ŞM01].

- Realizarea (proiectarea şi implementarea) unui mediu integrat de măsurare bazat pe instrumente virtuale **SAAD_WIN** (secţiunea 5.2), aplicând principiile compoziţiei vizuale a componentelor ierarhic compozabile. Rezultatele îndelungatei experienţe în acest domeniu al instrumentaţiei virtuale au fost publicate în lucrările: [CJMŞ03], [GŞC+98], [CŞG97], [CŞM96], [CMŞ95]. Programele implementate au fost utilizate în diverse aplicaţii de testare a maşinilor electrice, permiţând determinarea mai eficientă a parametrilor acestora [GBC+98], [BCG+95], [BCG+94].

- Aplicarea strategiei de compoziţie bazate pe mecanismul de propagare a cerinţelor la verificarea automată a circuitelor de procesare din instrumente virtuale din **SAAD_WIN**. Aceast aspect a fost descris în lucrarea [ŞBVC02].

Direcţii viitoare de cercetare

Cercetările conţinute în această teză pot fi continuate pe următoarele direcţii:

- Validarea modelului compoziţional propus pe sisteme din alte domenii de aplicaţii.

- Perfecţionarea strategiei de compoziţie bazate pe mecanismul propagării cerinţelor; investigarea utilizării unor tehnici de căutare euristică.

- Dezvoltarea modelului compoziţional pentru a permite reconfigurarea dinamica (prin înlocuire de componente în timpul execuţiei) a unui sistem. Pentru a putea realiza acest lucru, trebuie găsite metode de specificare universală a informaţiilor de stare caracteristice unui tip de componentă precum şi a transferului stării între componente de tipuri diferite.

- Studiul extinderilor necesare modelului pentru a trata şi alte stiluri arhitecturale.

BUPT

Bibliografie

[AG97] Robert Allen and David Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, 1997.

[Alt02] Altova GmbH. XML Spy. http://www.xmlspy.com, 2002.

[Apa01] Apache Group. Xerces Java Parser. http://www.apache.org, 2001.

[ASE00] *The 15th IEEE International Conference on Automated Software Engineering (ASE 2000)*, Grenoble, France, 11–15 September 2000. IEEE Computer Society.

[ASE01] *The 16th IEEE International Conference on Automated Software Engineering (ASE 2001)*, Coronado Island, San Diego, CA, USA, 26–29 November 2001. IEEE Computer Society.

[ASE02] *The 17th IEEE International Conference on Automated Software Engineering (ASE 2002)*, Edinburgh, Scotland, UK, 23–27 September 2002. IEEE Computer Society.

[AW99] Noriki Amano and Takuo Watanabe. An approach for constructing dynamically adaptable component-based software systems using LEAD++. In *OOPSLA'99 Workshop on Reflection and Software Engineering*, 1999.

[Bal94] R. Balzer. Enforcing architectural constraints. In *Second International Software Architecture Workshop ISAW-2*, San Francisco, October 1994.

[BBB+00] Felix Bachman, Len Bass, Charles Buhman, Santiago Comella-Dorda, Fred Long, John Robert, Robert Seacord, and Kurt Wallnau. Technical concepts of component-based software engineering, CMU/SEI-2000-TR-008. Technical report, Carnegie Mellon Software Engineering Institute, May 2000.

[BCCD00] Gordon Blair, Geoff Coulson, Fabio Costa, and Hector Duran. On the design of reflective middleware platforms. In *IFIP/ACM Middleware'2000 Workshop on Reflective Middleware*, Palisades, NY, April 2000.

BUPT

[BCG+94] M. Biriescu, V. Creţu, V. Groza, I. Marzoca, M. Moţ, and **Ioana Şora**. Transient regime recording and processing using a data acquisition and processing system. In *Proceedings 8.Kongressmesse fur industrielle Messtechnik, MessComp94*, pages 466–452, Wiesbaden, Germania, 1994.

[BCG+95] M. Biriescu, V. Creţu, V. Groza, I. Marzoca, M. Moţ, and **Ioana Şora**. Determination of torque characteristic of induction machine using a data acquisition and processing system. In *Proceedings 9.Kongressmesse fur industrielle Messtechnik, MessComp94*, pages 359–364, Wiesbaden, Germania, 1995.

[BCRP98] Gordon S. Blair, Geoff Coulson, Phillippe Robin, and Michael Papathomas. An architecture for next generation middleware. In J. Seitz N. Davies, K. Raymond, editor, *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98)*, Lake District, UK, 1998. Springer-Verlag.

[BCRW00] Don Batory, Gang Chen, Eric Robertson, and Tao Wang. Design wizards and visual programming environments for GenVoca generators. *IEEE Transactions on Software Engineering*, 26(5), May 2000.

[BG95] Don Batory and Bart Geraci. Validating component compositions in software system generators, UT/CS TR-95-03. Technical report, University of Texas at Austin, 1995.

[BG97] Don Batory and Bart Geraci. Composition validation and subjectivity in GenVoca generators. *IEEE Transactions on Software Engineering*, 23(2), February 1997.

[BISZ98] Christophe Bidan, Valerie Issarny, Titos Saridakis, and Apostolos Zarras. A dynamic reconfiguration service for CORBA. In *Proceedings of the 4th International Conference on Configurable Distributed Systems*, Annapolis, Maryland, USA, May 1998.

[BO92] Don Batory and Sean O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Transactions on Software Engineering and Methodology*, October 1992.

[Bur99] Margaret Burnett. Visual programming. In John Webster, editor, *Encyclopedia of Electrical and Electronics Engineering*. John Wiley & Sons, 1999.

[BW98] Alan W. Brown and Kurt C. Wallnau. The current state of CBSE. *IEEE Software*, pages 37–46, September-October 1998.

[CE99a] K. Czarnecki and U. Eisenecker. Synthesizing objects. In *Proceedings ECOOP'99*, pages 18–42, Lisbon, Portugal, June 1999. Lecture Notes in Computer Science 1628, Springer.

BUPT

[CE99b] Krzysztof Czarnecki and Ulrich Eisenecker. Components and generative programming. In *Proceedings ESEC/FSE'99*, pages 2–19. Lecture Notes in Computer Science 1687, Springer, 1999.

[Cit96] Wayne Citrin. Strategic directions in visual languages research. *ACM Computing Surveys*, 28(4es), December 1996.

[CJMŞ03] Vladimir Creţu, Traian Jurca, Mihai Micea, and **Ioana Şora**. Instrumentation and measurement in Romania. *IEEE Instrumentation and Measurement Magazine*, 6(3):41–48, September 2003.

[CŞG97] Vladimir Creţu, **Ioana Şora**, and Voicu Groza. Designing an integrated mesurement environment. In *IEEE Workshop on Emergent Technologies & Virtual Systems for Instrumentation and Measurement (ET&VSIM/97)*, pages 105–111, Niagara Falls, Ontario, Canada, May 1997.

[CŞM96] Vladimir Creţu, **Ioana Şora**, and Ioan Marzoca. Concepts and principles used in the software development of an integrated measurement environment. In *The 2nd International Conference on Technical Informatic Timisoara, CONTI'96*, pages 193–200, Timisoara, Romania, November 1996.

[CMŞ95] Vladimir Creţu, Ioan Marzoca, and **Ioana Şora**. Object oriented measurement environment for quantities with slow or medium variation speed. *Buletinul Stiintific si Tehnic al Universitatii Tehnice din Timisoara, Seria Automatica si Calculatoare*, 40(54):121–128, 1995.

[Cre92] Vladmir Cretu. *Structuri de date si tehnici de programare avansate.* Universitatea Tehnica Timisoara, 1992.

[CL02] Ivica Crnkovic and Magnus Larsson. *Building reliable Component-Based Software Systems.* Artech House, July 2002.

[Crn01] Ivica Crnkovic. Component-based software engineering - new challenges in software development. In *Software Focus.* John Wiley & Sons, December 2001.

[CSSW01] Ivica Crnkovic, Heinz Schmidt, Judith Stafford, and Kurt Wallnau, editors. *Proceedings of the 4th ICSE Workshop on Component-Based Software Engineering dedicated to Component Certification and System Prediction*, Toronto, Canada, 14–15 May 2001.

[CSSW02] Ivica Crnkovic, Heinz Schmidt, Judith Stafford, and Kurt Wallnau, editors. *Proceedings of the 5th ICSE Workshop on Component-Based Software Engineering dedicated to Benchmarks for Predictable Assembly*, Orlando, Florida, USA, 19–20 May 2002.

BUPT

[CSSW03] Ivica Crnkovic, Heinz Schmidt, Judith Stafford, and Kurt Wallnau, editors. *Proceedings of the 6th ICSE Workshop on Component-Based Software Engineering*, 2003.

[dBvV02a] Hans de Bruin and Hans van Vliet. The future of component-based development is generation, not retrieval. In *Workshop on Component Based Software Engineering at ECBS*, Lund, Sweden, April 2002.

[dBvV02b] Hans de Bruin and Hans van Vliet. Top-down composition of software architectures. In *Proceedings 9th Annual IEEE International Conference on the Engineering of Computer-Based Systems (ECBS)*, pages 147–156, Lund, Sweden, April 2002.

[dM95] Vicky de Mey. Visual composition of software applications. In O. Nierstrasz and D. Tsichritzis, editors, *Object-Oriented Software Composition*, pages 3–28. Prentice Hall, 1995.

[Dou00] Bruce Powel Douglass. Components, states and interfaces, oh my ! *Software Development*, 8(4), April 2000.

[DvdHT01] Eric M. Dashofy, Andre van der Hoek, and Richard N. Taylor. A highly-extensible, XML-based architecture description language. In *Proceedings of the Working IEEE/IFIP Conference on Software Architectures (WICSA 2001)*, Amsterdam, Netherlands, 2001.

[DvdHT02] Eric M. Dashofy, Andre van der Hoek, and Richard N. Taylor. An infrastructure for the rapid development of XML-based architecture description languages. In *Proceedings of the 24th International Conference on Software Engineering (ICSE2002)*, Orlando, Florida, USA, 2002.

[FEHC02] A. V. Fioukov, E. M. Eskenazi, D.K. Hammer, and M.R.V. Chaudron. Evaluation of static properties for component-based architectures. In *Proceedings 28th EUROMICRO conference on Component-based Software Engineering*, Dortmund, Germany, September 4th 6th 2002.

[FoC97] Proceedings of the ESEC/FSE workshop on Foundations of Component Based Systems (FoCBS), September 1997.

[FP02] Alexander Fried and Herbert Prahofer. Describing and reusing software design assets for system family engineering. In *Proceedings of Workshopf on Model-based Software Reuse at ECOOP 2002*, Malaga, Spain, June 2002.

[Fra98] Xavier Franch. Systematic formulation of non-functional characteristics of software. In *Proceedings 3rd IEEE International Conference on Requirements Engineering (ICRE)*, pages 174–181, Colorado Springs (Colorado, USA), April 1998.

BUPT

[GAO95] David Garlan, Robert Allen, and John Ockerbloom. Architectural mismatch, or, why it's hard to build systems out of existing parts. In *Proceedings of the 17th International Conference on Software Engineering*, pages 179–185, Seattle, Washington, April 1995.

[Gar01] David Garlan. Software architecture. In J. Marciniak, editor, *Wiley Encyclopedia of Software Engineering*. John Wiley & Sons, 2001.

[GBC+98] Voicu Groza, Marius Biriescu, Vladimir Creţu, **Ioana Şora**, and Martian Mot. Testing of electrical machines in periodical and quasi-periodical conditions, using a data aquisition and processing system. In *Proceedings of the 15th IEEE Instrumentation and Measurement Technology Conference*, St. Paul Minnesota, USA, May 1998.

[GŞC+98] Voicu Groza, **Ioana Şora**, Vladimir Creţu, Emil Petriu, and Dan Ionescu. A software architecture for an integrated measurement environment. In *Proc. ETIMVIS'98, 1998 IEEE International Workshop on Emerging Environment Technologies, Intelligent Measurements and Virtual Systems for Instrumentation and Measurement*, pages 166–172, St.Paul, Minnesota,USA, 1998.

[GMW00] David Garlan, Robert Monroe, and David Wile. Acme: Architectural description of component-based systems. In Gary Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, pages 47–68. Cambridge University Press, 2000.

[GR91] M. Gorlick and R. Razouk. Using Weaves for software construction and analysis. In *Proceedings 13th International Conference Software Enginering ICSE*, pages 23–34, May 1991.

[GS94] David Garlan and Mary Shaw. An introduction to software architecture. Technical Report CMU-CS-94-166, Carnegie Mellon University, 1994.

[Ham02] Dieter K. Hammer. Component-based architecting for component-based systems. In Mehmet Askit, editor, *Software Architectures and Component Technology*. Kluwer, 2002.

[HŞ97] Carmen Holotescu and **Ioana Şora**. Aspects of using reflection in distributed real-time systems. *Buletinul Stiintific al Universitatii Poliehnica din Timisoara, Seria Automatica si Calculatoare*, Tom 42(56):203–214, 1997.

[HF98] Richard Hayton and Matthew Faupel. Flexinet: Automating application deployment and evolution. In *Workshop on Compositional Software Architectures*, Monterey, California, USA, January 6-8 1998.

BUPT

[HMSW02] Scott A. Hissam, Gabriel A. Moreno, Judith A. Stafford, and Kurt C. Wallnau. Packaging predictable assembly. In *IFIP/ACM Working Conference on Component Deployment (CD2002)*, Berlin, Germany, June 20-21 2002.

[Hoa69] C.A.R. Hoare. An axiomatic approach to computer programming. *CACM*, 12(10):576–583, October 1969.

[HP90] Norman Hutchinson and Larry Peterson. The x-kernel: An architecture for implementing network protocols. *IEEE Computer*, pages 23–33, May 1990.

[IB96] Valrie Issarny and Christophe Bidan. Aster: A framework for sound customization of distributed runtime systems. In *Proceedings of the 16th International Conference on Distributed Computing Systems*, pages 586–593, Hong-Kong, May 1996.

[IBS98] Valerie Issarny, Christophe Bidan, and Titos Saridakis. Achieving middleware customization in a configuration-based development environment: Experience with the ASTER prototype. In *Proceedings of the International Conference on Configurable Distributed Systems*, pages 207–214, 1998.

[IS01] Paola Inverardi and S Scriboni. Connectors synthesis for deadlock-free component based architectures. In *Proceedings of the 16th ASE*, Coronado Island, California, USA, November 2001.

[IT02a] Paola Inverardi and Massimo Tivoli. Correct and automatic assembly of COTS components: an architectural approach. In *Proceedings of the 5th ICSE Workshop on Component-Based Software Engineering*, Orlando, Florida, USA, May 19-20 2002.

[IT02b] Paola Inverardi and Massimo Tivoli. The role of architecture in component assembly. In *Proceedings Seventh International Workshop on Component-Oriented Programmin (WCOP) at ECOOP*, Malaga, Spain, June 2002.

[IW95] Paola Inverardi and Alexander L. Wolf. Formal specification and analysis of software architectures using the chemical abstract machine model. *IEEE Transactions on Software Engineering*, 21(4), 1995.

[IW03] James Ivers and Kurt Wallnau. Preserving real concurrency. In *Proceedings of the 2003 ECOOP Workshop on Correctness of Model-Based Software Composition (CMC)*, pages 15–23, Darmstadt, Germany, July 21-25 2003.

BUPT

[IWY97] P. Inverardi, A. Wolf, and D. Yankelevich. Checking assumptions in components dynamics at the architectural level. In D. Garlan and D. LeMetayer, editors, *Proc. 2nd Int. Conf. on Coordination Models and Languages*, volume 1282, pages 46–63, Berlin, Germany, 1997. Springer-Verlag, Berlin.

[JK03] Jens Jahnke and Luay Kawasme. Generation of asynchronous component adapters. In *Proceedings of Workshop on Software Composition, affiliated with ETAPS 2003*, volume 82 of *Electronic Notes in Theorethical Computer Science*, Warsaw, Poland, 2003. Elsevier.

[JSS03] *Journal of Systems and Software, Special Issue on Component-Based Software Engineering*, 65(3):169–238, March 2003.

[JTMJ00] B.N. Joergensen, Eddy Truyen, Frank Matthijs, and Wouter Joosen. Customization of object request brokers by application specific policies. In *Proceedings of the IFIP International Conference on Distributed Platform and Open Distributed Processing (Middleware200)*, 2000.

[KC99] Fabio Kon and Roy Campbell. Supporting automatic configuration of component-based distributed systems. In *5th USENIX Conference on Object-Oriented Technologies and Systems (COOTS'99)*, San Diego, California, USA, May 3–7 1999.

[KC00] Fabio Kon and Roy H. Campbell. Dependence management in component-based distributed systems. *IEEE Concurrency*, 8(1):26–36, January-March 2000.

[KI00] Christos Kloukinas and Valerie Issarny. Automating the composition of middleware configurations. In *Automated Software Engineering*, pages 241–244, 2000.

[KMR91] J. Kodosky, L. MacCreisken, and G. Rymar. Visual programming using structured dataflow. In *IEEE Workshop on Visual Languages (VL'91)*, pages 34–39, Kobe, Japan, 1991.

[KYH+01] Fabio Kon, Tomonori Yamane, Christopher Hess, Roy Campbell, and Dennis Mickunas. Dynamic resource management and automatic configuration of distributed component systems. In *6th USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*, January 2001.

[LCK99] L.Bass, P. Clemens, and R. Kazman. *Software architecture in practice.* Addison Wesley, 1999.

[LLH02] Frank Luders, Kung-Kiu Lau, and Shui-Ming Ho. On the specification of components. In Ivica Crnkovic and Magnus Larsson, editors, *Building reliable Component-Based Software Systems.* Artech House, July 2002.

BUPT

[LV95] D.C. Luckham and J. Vera. An event-based architecture definition language. *IEEE Transactions on Software Engineering*, Vol.21(No.9):717–734, September 1995.

[LWNC02] Magnus Larsson, Andreas Wall, Christer Norstrom, and Ivica Crnkovic. Using prediction-enabled technologies for embedded product line architectures. In *Proceedings of the 5th ICSE Workshop on Component-Based Software Engineering*, Orlando, Florida, USA, May 19-20 2002.

[Mat99] Frank Matthijs. *Component Framework Technology for Protocol Stacks.* PhD thesis, Katholieke Universiteit Leuven, Belgium, December 1999.

[Mat03] Petr Matejka. A Description Language for Composable Components, Seminar at Distributed Systems Research Group, Charles University, Prague, Czech Republic. http://nenya.ms.mff.cuni.cz/teaching/meetings-2002-2003.phtml, May 2003.

[MCC03] James Mitchell, Brett Cowan, and Stephane Collart. Interactive visual components for server-side web application development. In Theo D'Hondt, editor, *Technology of Object-Oriented Languages, Systems and Architectures*, pages 185–197. Kluwer Academic Publishers, 2003.

[McI69] M. D. McIlroy. Mass produced software components. In P.Naur and NATO Science Committee B. Randell, editors, *Software Engineering*, pages 138–150. January 1969.

[MDEK95] Jeff Magee, Naranker Dulay, Susan Eisenbach, and Jeff Kramer. Specifying distributed software architectures. In *Proceedings of 5th European Software Engineering Conference (ESEC '95)*, pages 137–153, Sitges, Spain, September 1995. Springer LNCS989.

[Med96] Nenad Medvidovic. ADLs and dynamic architecture changes. In *Joint Proceedings of the SIGSOFT'96 Workshops*, pages 24–27. ACM Press, 1996.

[Mey88] Bertrand Meyer. *Object Oriented Software Construction.* Prentice Hall, Englewood Cliffs, 1988.

[Mey99] Bertrand Meyer. The significance of components. *Software Development*, 7(11), November 1999.

[Mey00a] Bertrand Meyer. Contracts for components. *Software Development*, 8(7), July 2000.

[Mey00b] Bertrand Meyer. What to compose. *Software Development*, 8(3), March 2000.

BUPT

[MK96] Jeff Magee and Jeff Kramer. Dynamic structure in software architectures. In *Proceedings ACM SIGSOFT '96: Fourth Symposium on the Foundations of Software Engineering*, pages 3–14, October 1996.

[MMM03] James Montgomery, Rune Meling, and Daniela Mehandijska. Semi-formal, not semi-realistic: A new approach to describing software components. In Theo D'Hondt, editor, *Technology of Object-Oriented Languages, Systems and Architectures*, pages 197–208. Kluwer Academic Publishers, 2003.

[MMMV01] Sam Michiels, Tom Mahieu, Frank Matthijs, and Pierre Verbaeten. Dynamic protocol stack composition: Protocol independent addressing. In *Proceedings of the 4th ECOOP Workshop on Object-Orientation and Operating Systems*, June 2001.

[MORT96] Nenad Medvidovic, P. Oreizy, J.E. Robbins, and R.N. Taylor. Using object-oriented typing to support architectural design in the C2 style. In *Proceedings ACM SIGSOFT 96: Fourth Symposium Foundations of Sopftware Engineering*, pages 24–32, 1996.

[MT00] N. Medvidovic and R. Taylor. A classification and composition framework for software architecture description languages. *IEEE Transactions on Software Engineering*, Vol.26(No.1):70–93, January 2000.

[ND95] Oscar Nierstrasz and Laurent Dami. Component-oriented software technology. In O. Nierstrasz and D. Tsichritzis, editors, *Object-Oriented Software Composition*, pages 3–28. Prentice Hall, 1995.

[Nie95] Oscar Nierstrasz. Research topics in software composition. In A. Napoli, editor, *Proceedings of Languages et Modeles a Objet*, pages 193–204, Nancy, 1995.

[OG02] Johann Oberleitner and Thomas Gschwind. Requirements for an architectural composition language. In *Second International Workshop on Composition Languages at ECOOP 2002*, Malaga , Spain, June 2002.

[OG03] Johann Oberleitner and Thomas Gschwindt. Transforming application compositions with XSLTs. In *Proceedings of Workshop on Software Composition, affiliated with ETAPS 2003*, volume 82 of *Electronic Notes in Theorethical Computer Science*, Warsaw, Poland, 2003. Elsevier.

[OGT+99] Peyman Oreizy, Michael M. Gorlick, Richard N. Taylor, Dennis Heimbigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S. Rosenblum, and Alexander L. Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems*, 14(3):54–62, May/June 1999.

BUPT

[OMG95] OMG. The common object request broker: Architecture and specification - revision 2.0. Technical report, OMG, 1995.

[OMT98] Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor. Architecture-based runtime software evolution. In *Proceedings of the International Conference on Software Engineering 1998 (ICSE'98)*, pages 177–186, Kyoto, Japan, April 19-25 1998.

[Ore96] Peyman Oreizy. Issues in the runtime modification of software architectures. Technical Report Technical Report UCI-ICS-TR-96-35, Department of Information and Computer Science, University of California, Irvine, 1996.

[Pah03] Claus Pahl. An ontology for software component matching. In Mauro Pezze, editor, *Fundamental Approaches to Software Engineering, 6th International Conference, Proceedings*, number 2621 in LNCS, pages 6–22. Springer Verlag, 2003.

[Pep02] Platform for enhanced provisioning of terminal-independent applications (PEPiTA). Research project of Katholieke Universiteit Leuven, Belgium with Alcatel Bell and The Flemish Institute for the advancement of scientific-technological research (IWT)ITEA #99007 (IWT), http://pepita.objectweb.org/index.html, 2001–2002.

[Per87] Dewayne E. Perry. Software interconnection models. In *Proceedings of the 9th International Conference of Software Engineering*, pages 61–69, Monterey CA, USA, May 1987.

[Per89] Dewayne E. Perry. The logic of propagation in the Inscape environment. In *Proceedings of SIGSOFT '89: Testing, Analysis and Verification Symposium*, Key West FL, USA, December 1989.

[PLV97] Edward J. Posnak, Greg Lavender, and Harrick M. Vin. An adaptive framework for developing multimedia software components. *Communications of the ACM*, 40(10):43–47, October 1997.

[PW92] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architectures. 17(4):40–52, 1992.

[Reu01] Ralf Reussner. Enhanced component interfaces to support dynamic adaptation and extension. In *Proceedings of the 34th Hawaii International Conference ob System Science*, 2001.

[RS02] Ralf H. Reussner and Heinz W. Schmidt. Using Parameterised Contracts to Predict Properties of Component Based Software Architectures. In Ivica Crnkovic, Stig Larsson, and Judith Stafford, editors, *Workshop On Component-Based Software Engineering (in association*

BUPT

with 9th IEEE Conference and Workshops on Engineering of Computer-Based Systems), Lund, Sweden, apr 2002.

[RSS98] F. Ranno, S.K. Shrivastava, and S.M.Wheater. A language support for specifying the composition of reliable distributed applications. In *Proceedings of the 18th International Conference on Distributed Computing Systems (ICDCS'98)*, Amsterdam, The Netherlands, May 1998.

[SB02] Judith Stafford and Jan Bosch. Architecting component-based systems. In Ivica Crnkovic and Magnus Larsson, editors, *Building reliable Component-Based Software Systems*. Artech House, July 2002.

[SC003] *Proceedings of Workshop on Software Composition, affiliated with ETAPS 2003*, volume 82 of *Electronic Notes in Theorethical Computer Science*, Warsaw, Poland, April 2003. Elsevier.

[SC97] Mary Shaw and Paul Clements. A field guide to boxology: Preliminary classification of architectural styles for software systems. In *Proceedings of COMPSAC '97*, pages 6–13, 1997.

[SC99] Douglas Schmidt and Chris Cleeland. Applying patterns to develop extensible ORB middleware. *IEEE Communications Magazine*, 1999.

[Sch01] Heinz Schmidt. Trusted components: Towards automated assembly with predictable properties. In *CBSE4 Proceedings*, Toronto, Canada, May 2001.

[Sha96] Mary Shaw. Truth vs knowledge: The difference between what a component does and what we know it does. In *Proceedings of the 8th International Workshop on Software Specification and Design*, pages 181–185, 1996.

[Sha01] Mary Shaw. The coming-of-age of software architecture research. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 656–664, Toronto, Canada, 2001. IEEE Computer Society.

[SLC99] Douglas Schmidt, David Levine, and Chris Cleeland. Architectures and patterns for high-performance real-time ORB endsystems. In Martin Zelkovitz, editor, *Advances in Computers*. Academic Press, 1999.

[SM02] Judith Stafford and John McGregor. Issues in predicting the reliability of composed components. In *Proceedings of the 5th ICSE Workshop on Component-Based Software Engineering*, Orlando, Florida, USA, May 19-20 2002.

[SN99] Jean-Guy Schneider and Oscar Nierstrasz. Components, scripts and glue. In Leonor Baroca, Jon Hall, and Patrick Hall, editors, *Software Architectures Advances and Applications*, pages 13–25. Springer, 1999.

BUPT

[SPR01] Heinz W. Schmidt, Iman Poernomo, and Ralf H. Reussner. Trust-by-contract: Modelling, analysing and predicting behaviour in software architectures. *Journal for Integrated Process and Development Science*, pages 25–51, September 2001.

[SR00] Heinz Schmidt and Ralf Reussner. Automatic component adaptation by concurrent state machine retrofitting. Technical Report 2000/81, School of Computer Science and Software Engineering, Monash University, Melbourne, Australia, 2000.

[SRG96] Mary Shaw, R.DeLine, and G.Zelesnik. Abstractions and implementations for archirtectural connections. In *Proceedings of the 3rd International Conference on Configurable Distributed Systems (CDS'96)*, Annapolis, Maryland, USA, May 1996.

[SVSJ03] Pieter Schollaert, Wim Vanderperren, Davy Suvee, and Viviane Jonckers. Online reconfiguration of component-based applications in PacoSuite. In *Proceedings of Workshop on Software Composition, affiliated with ETAPS 2003*, volume 82 of *Electronic Notes in Theorethical Computer Science*, Warsaw, Poland, 2003. Elsevier.

[SW98] S Shrivastava and S Wheater. Architectural support for dynamic reconfiguration of large scale distributed application. In *The 4th International Conference on Configurable Distributed Systems (ICCDS'98)*, Annapolis, Maryland, USA, May 4-6 1998.

[SW01] J. A. Stafford and A. L. Wolf. Architecture-level dependence analysis for software systems. *International Journal of Software Engineering and Knowledge Engineering*, 11(4):431–452, August 2001.

[Sun] Sun Microsystems. (Java API for XML processing, in The Java web services tutorial.

[Szy97] Clemens Szypersky. *Component Software: Beyond Object Oriented Programming*. Addison-Wesley, 1997.

[Szy99] Clemens Szypersky. Components and objects together. *Software Development*, 7(5), May 1999.

[Szy00] Clemens Szypersky. Point, counterpoint. *Software Development*, 8(2), February 2000.

[ŞM01] **Ioana Şora** and Frank Matthijs. Automatic composition of software systems from components with anonymous dependencies. Technical Report CW314, Katholieke Universiteit Leuven, Belgium, May 2001.

[ŞMM01] **Ioana Şora**, Sam Michiels, and Frank Matthijs. Policies for dynamic stack composition. Technical Report CW313, Katholieke Universiteit Leuven, Belgium, February 2001.

BUPT

[ŞBVC02] **Ioana Şora**, Yolande Berbers, Pierre Verbaeten, and Vladimir Cretu. Requirements driven, architecture specific compositional model. *Periodica Politechnica, Transaction on Automatic Control And Computer Science*, Tom 47(61)(2):29–34, 2002.

[ŞJBV02] **Ioana Şora**, Nico Janssens, Yolande Berbers, and Pierre Verbaeten. A component composition model to support unanticipated customization of systems. In *Proceedings - Workshop on Unanticipated Software Evolution (USE) at ECOOP 2002*, Malaga , Spain, June 2002.

[ŞMBV03] **Ioana Şora**, Frank Matthijs, Yolande Berbers, and Pierre Verbaeten. Automatic composition of systems from components with anonymous dependencies. In Theo D'Hondt, editor, *Technology of Object-Oriented Languages, Systems and Architectures*, pages 154–169. Kluwer Academic Publishers, 2003.

[ŞVB02] **Ioana Şora**, Pierre Verbaeten, and Yolande Berbers. Using component composition for self-customizable systems. In I. Crnkovic, J. Stafford, and S. Larsson, editors, *Proceedings - Workshop On Component-Based Software Engineering at IEEE-ECBS 2002*, pages 23–26, Lund, Sweden, 2002.

[ŞVB03] **Ioana Şora**, Pierre Verbaeten, and Yolande Berbers. A description language for composable components. In Mauro Pezze, editor, *Fundamental Approaches to Software Engineering, 6th International Conference, Proceedings*, number 2621 in Lecture Notes in Computer Science, pages 22–36. Springer Verlag, 2003.

[TJJ00] E. Truyen, B.N. Joergensen, and W. Joosen. Customization of object request brokers through dynamic reconfiguration. In *Proceedings TOOLS Europe 2000*, Mont St. Michel, France, June 2000.

[Tra93] W. Tracz. Parametrized programming in LILEANNA. In *Proceedings of ACM Symposium on Applied Computing SAC'93*, 1993.

[vGBS01] J. van Gurp, J. Bosch, and M. Svahnberg. On the notion of variability in software product lines. In *Proceedings of WICSA 2001*, August 2001.

[VSWJ03] Wim Vanderperren, Davy Suvee, Bart Wydaeghe, and Viviane Jonckers. PacoSuite and JAsCo: A visual component composition environment with advanced aspect separation features. In Mauro Pezze, editor, *Fundamental Approaches to Software Engineering, 6th International Conference, Proceedings*, number 2621 in LNCS, pages 166–169. Springer Verlag, 2003.

[W3C98] World Wide Web Consortium W3C. eXtensible Markup Language (XML) 1.0. http://ww.w3.org/TR/1998/REC-xml-19980210, 1998.

BUPT

[W3C01] World Wide Web Consortium W3C. Xml schema. http://ww.w3.org/XML/Schema, 2001.

[Wer98] Michel Wermelinger. Towards a chemical model for software architecture reconfiguration. In *Proc. of the 4th Intl. Conf. on Configurable Distributed Systems*. IEEE Computer Society Press, 1998.

[Wil01] D. S. Wile. Ensuring general-purpose and domain-specific properties using architectural styles. In *CBSE4 Proceedings*, Toronto, Canada, May 2001.

[Wil03] David Wile. Revealing component properties through architectural styles. *Journal of Systems and Software, Special Issue on Component-Based Software Engineering*, 65(3):209–214, March 2003.

[ZW97] Amy Moormann Zaremski and Jeannette M. Wing. Specification matching of software components. *ACM Transactions on Software Engineering and Methodology*, 6(4):333–369, 1997.

BUPT

Lista lucrărilor

Ioana Şora

Lucrări ştiinţifice publicate

1. **Ioana Şora**, Pierre Verbaeten, Yolande Berbers, "A Description Language For Composable Components", in **Fundamental Approaches to Software Engineering** 2003, pp. 22-37, Springer Verlag, seria *Lecture Notes in Computer Science (LNCS) Numarul 2621*, ISBN 3-540-00899-3, ISSN 0302-9743
2. Vladimir Creţu, Traian Jurca, Mihai Micea, **Ioana Şora**, "Instrumentation and Measurement in Romania", **IEEE Instrumentation and Measurement Magazine**, Vol. 6, No. 3, September 2003, pp. 42-48.
3. **Ioana Şora**, Frank Matthijs, Yolande Berbers, Pierre Verbaeten, "Automatic Composition of Systems from Components with Anonymous Dependencies Specified by Semantic-Unaware Properties", in **Technology of Object Oriented Languages, Systems and Architectures**, pp. 154-170, *The Kluwer International Series in Engineering and Computer Science*, Kluwer Academic Publishers, 2003, ISBN 1-4020-7428-X .
4. **Ioana Şora,** Y.Berbers, P.Verbaeten, V.Creţu, "Requirements Driven, Architecture Specific Compositional Model", **Periodica Politechnica, Transaction on Automatic Control And Computer Science**, Tom 47(61), No.2, 2002, ISSN 1224-600X, Timisoara, pp.(29-34).
5. **Ioana Şora**, Nico Janssens, Pierre Verbaeten, Yolande Berbers, „A Component Composition Model to Support Unanticipated Customization of Systems", Proceedings of **Workshop on Unanticipated Software Evolution (USE 2002) at the European Conference on Object Oriented Programming (ECOOP) Malaga, Spain**, June, 2002.
6. **Ioana Şora**, Pierre Verbaeten, Yolande Berbers, „Using Component Composition for Self-customizable Systems", Proceedings of **Workshop on Component Based Software Engineering, at the 9th IEEE Conference and Workshops on Engineering of Computer Based Systems (ECBS)**, pp. 23-26, **Lundt, Sweden**, 8-11 April 2002
7. **Ioana Şora**, Frank Matthijs, A*utomatic composition of software systems from components with anonymous dependencies,* Technical Report CW314, May 2001, **Katholieke Universiteit Leuven, Belgium**
8. **Ioana Şora**, Sam Michiels, Frank Matthijs, *Policies for dynamic stack composition,* Technical Report CW313, February 2001, **Katholieke Universiteit Leuven, Belgium.**
9. **Ioana Şora**, Vladimir Creţu, An Analysis of Performances of Object Request Brokers, **Buletinul Stiintific al Universităţii "Politehnica" din Timişoara**, Seria Automatică şi Calculatoare,1999
10. **Ioana Şora**, V.Creţu, "Object Request Brokers and Their Perspectives in Real-Time Systems", 5th International Conference on Engineering of Modern Electric Systems, EMES'99, Mai 27-29, Oradea
11. V.Groza, **Ioana Şora,** V.Cretu E.Petriu, D. Ionescu, "A Software Architecture for an Integrated Measurement Environment", The 15th **IEEE Instrumentation and**

BUPT

Measurement Technology Conference, St. Paul Minnesota, USA, May 18-21 1998

12. V.Groza, M.Biriescu, V.Cretu, **Ioana Şora**, M. Mot, Testing of Electrical Machines in Periodical and Quasi-Periodical Conditions, Using a Data Aquisition and Processing System, The 15th **IEEE Instrumentation and Measurement Technology Conference, St. Paul Minnesota, USA**, May 18-21 1998
13. V.Creţu, **Ioana Şora,** V. Groza, "Designing an Integrated Mesurement Environment", **IEEE Workshop on Emergent Technologies & Virtual Systems for Instrumentation and Measurement (ET&VS-IM/97), Niagara Falls, Ontario, Canada**, May 15-16 1997, Proceedings, pp. 105-111
14. **Ioana Şora,** Carmen Holotescu, "Formal Verification of Real-Time Systems using Temporal Model Checking", The 2nd International Conference on Microelectronics and Computer Science, **ICMCS-97, Chişinău**, Oct 30-31, 1997, in Proceedings, Volume 1, pp 199-202
15. Carmen Holotescu, **Ioana Şora,** Issues in Building Hard Real-Time Systems, The 2nd International Conference on Microelectronics and Computer Science, **ICMCS-97, Chişinău**, Oct 30-31, 1997, in Proceedings, Volume 1, pp. 180-185
16. V.Creţu, **Ioana Şora,** Carmen Holotescu, "Trends in formal verification of real-time systems: temporal model checking", **Buletinul Ştiinţific al Universităţii "Politehnica" din Timişoara**, Seria Automatică şi Calculatoare, Tom 42(56), 1997, pag.193-202
17. Carmen Holotescu, **Ioana Şora,** Aspects of using reflection in Distributed real-time systems, **Buletinul Ştiinţific al Universităţii Politehnica din Timişoara**, Seria Automatică şi Calculatoare, Tom 42(56), 1997, pag. 203-214
18. V.Creţu, **Ioana Şora,** I.Mârzoca, Concepts and Principles Used in the Software Development of an Integrated Measurement Environment, **The 2nd International Conference on Technical Informatic Timişoara, CONTI'96**, Timisoara, Nov.1996, Proceedings, Computer Science and Engineering, Vol.1, p.193-200
19. G. Liuba, M.Biriescu, V.Groz, V.Creţu , I.Marzoca, M.Moţ, **Ioana Şora,** Asupra încercării maşinilor electrice cu un sistem de achiziţie şi prelucrare a datelor, Volumul **Analele Universităţii "Eftimie Murgu" Reşiţa,**Fasc.III, Oct. 1996, pag.5-12
20. Ioan Şora, Ioana Şora, "Problems regarding the electromagnetic compatibility in the presence of the non-sinusoidal electroenergetical regime", ICEH'96, 28-29 Noiembre 1996, Universitatea "Lucian Blaga" Sibiu, pag. 285- 289.
21. Vladimir Creţu, Ioan Mârzoca, **Ioana Şora, "**Object Oriented Measurement Environment for Quantities with Slow or Medium Variation Speed", **Buletinul Ştiinţific şi Tehnic al Universităţii Tehnice din Timişoara**, Tom 40(54), Seria Automatică şi Calculatoare, 1995, p.121-128
22. M.Biriescu, V.Groza, V.Creţu, I.Marzoca, M.Moţ, **Ioana Şora, "**Asupra încercării maşinilor asincrone cu un sistem de achiziţie şi prelucrare a datelor", **Buletinul Ştiinţific şi Tehnic al Universităţii Tehnice din Timişoara**, Tom 40(54), Seria Electrotehnică,Electronică şi Telecomunicaţii, 1995, p.67-74
23. M.Biriescu, V.Creţu, V.Groza I.Marzoca, M.Moţ, **Ioana Şora ,** Determination of Torque Characteristic of Induction Machine Using a Data Acquisition and Processing System, **9.Kongressmesse fur industrielle Messtechnik, MessComp95, Wiesbaden, Germania**, Sept.1995 Tagungsband (Proceedings) p.359-364

BUPT

24. M.Biriescu, V.Crețu, V.Groza , I.Marzoca, M.Moț, **Ioana Șora,** Transient Regime Recording and Processing Using a Data Acquisition and Processing System, **8.Kongressmesse fur industrielle Messtechnik, MessComp94, Wiesbaden, Germania**, Sept. **1994** Tagungsband (Proceedings) p.466-452

Manuale:

1. Carmen Holotescu si **Ioana Șora**, *Structuri de date si analiza algoritmilor*, Indrumator de laborator, Universitatea Politehnica Timişoara, 1998.

Contracte:

Membru în echipele de cercetare ale următoarelor granturi / proiecte:

1. Proiect de cercetare al Katholieke Universiteit Leuven, Belgia cu Alcatel Bell & The Flemish Institute for the advancement of scientific-technological research (IWT). *Service Centric Acces Networks (SCAN)*, proiect IWT SCAN #010319, 2001 –2002. Membru in echipa de cercetare Katholieke Universiteit Leuven, Belgia.
2. Proiect de cercetare al Katholieke Universiteit Leuven, Belgia cu Alcatel Bell & The Flemish Institute for the advancement of scientific-technological research (IWT). *Platform for Enhanced Provisioning of Terminal-independent Applications (PEPiTA)*, ITEA #99007 (IWT), 2000 – 2001. http://pepita.objectweb.org/index.html. Membru in echipa de cercetare Katholieke Universiteit Leuven, Belgia.
3. Grant MCT 666/ 1996-98. Tema A1: Sistem de achiziție şi prelucrare de date pentru determinarea parametrilor electromagnetici
4. Grant 36/ 1998. Cod CNCSU 261: Tema 3: Produs program pentru înregistrarea şi prelucrarea informațiilor obținute din regimurile tranzitorii ale maşinilor electrice. Faza: Realizarea practica a placii de achizitii de date cu performante imbunatatite. Testarea placii.
5. Grant 7004/1997. Cod CNCSU 798. Tema 798: Produs program pentru înregistrarea şi prelucrarea informațiilor obținute din regimurile tranzitorii ale maşinilor electrice. Faza: Proiectarea placii de achizitii de date cu performante imbunatatite
6. Grant 5004/1996. Cod CNCSU 344. Tema 344: Produs program pentru înregistrarea şi prelucrarea informațiilor obținute din regimurile tranzitorii ale maşinilor electrice. Faza 1: Implementare produs program. Faza 2: Testare produs program
7. Grant 4004/ 1995 CNCSU. Tema 2B: Produs program pentru înregistrarea şi prelucrarea informațiilor obținute din regimurile tranzitorii ale maşinilor electrice. Faza 1: Specificații de definiție Faza 2: Descriere produs program
8. Contract MCT 487/B1995. Soluții noi în concepția, testarea şi utilizarea maşinilor şi echipamentelor electrice industriale. Tema 3: Sistem de achiziție şi prelucrare de date pentru determinarea parametrilor electromagnetici. Faza 3.2: Programe de achiziție şi prelucrare a datelor pentru studiul experimental al regimurilor tranzitorii la maşina asincronă

BUPT