LATVIJAS UNIVERSITĀTE

Guntis Arnicāns

Informācijas apstrādes rīku izveide neviendabīgai un dinamiskai videi

Promocijas darbs

datorzinātņu doktora (Dr.sc.comp.) zinātniskā grāda iegūšanai

Nozare: datorzinātnes

Apakšnozare: datoru un sistēmu programmatūra

Zinātniskais vadītājs:

Jānis Bičevskis

prof., Dr. sc. comp.

Rīga - 2004

Anotācija

Mūsdienu informatīvajām sistēmām ir jābūt funkcionāli daudzveidīgām, nepārtraukti strādājošām strauji mainīgā un neviendabīgā vidē, spējīgām dinamiski pielāgoties gan lietotāja, gan citu informatīvo sistēmu, gan vides un tehnoloģiju prasībām. Šī mērķa sasniegšanai kā viena no iespējām tiek piedāvāta intensīva specializētu rīku izmantošana. Darbā kā pamatpiemērs tiek apskatīta Komunikāciju servera, kas nodrošina integrētas un konsolidētas informācijas iegūšanu no daudziem datu avotiem neviendabīgā un sadalītā vidē, izveide.

Kvalitatīvu rīku izveide ir tieši vai netieši saistīta ar domēnspecifisku valodu pielietošanu. Domēnspecifiskas valodas var kalpot gan kā rīka funkcionalitātes specifikācija, gan kā ieejas datu vai rezultāta pieraksta valoda, gan arī kā rīka vadības komandu pieraksta valoda. Ir apskatītas šo valodu izmantošanas un implementēšanas iespējas, īpaši uzsverot valodas semantikas definēšanas un tās izpildāmā koda iegūšanas noteicošo lomu.

Demonstrējot pamatidejas lietojumprogrammu ģenerācijas rīka izveidei, ir apskatīta datubāzes pārlūka izveidošanas tehnika. Piedāvātais pārlūks vai datu pārvaldības sistēma var tikt uzģenerēta automātiski no fiziskā datu modeļa, kas ir pierakstīts ar ER diagrammas palīdzību. ER diagramma satur gan tradicionālos vienkāršākos elementus, gan dažus papildus atribūtus, kas būtiski atvieglo lietojuma ģenerēšanu un uzlabo gala sistēmas lietojamību. Visi ER diagrammas elementi tiek pārveidoti par noteiktu lietojuma komponenti, ir definēta tieša attiecība starp diagrammas elementiem un lietotāja saskarni. Lietojuma kvalitāte ir atkarīga no autora izstrādāto saskarnes ekrānu tipu izmantošanas. Darbā ir doti piemēri svarīgākajiem ekrānu veidošanas principiem.

Lielu sistēmu integrācijai, lai daudzus datu avotus varētu aplūkot un apstrādāt kā vienotu veselu informatīvu sistēmu, tiek piedāvāts izmantot rīku grupu, kas nodrošina datu avotu aprakstīšanu, to pārvaldību, lietotāju pieprasījumu izpildi un rezultāta attēlošanu lietotājam. Visa informācija par datu avotu loģisko struktūru un datu ieguves iespējām no tiem tiek glabāta speciālā repozitorijā. Lietotāja saskarnes ģenerācijai tiek piedāvāta īpaši veidota domēnspecifiska valoda.

Viena no būtiskākajām darbā apskatītajām jomām ir semantikas definēšana daudzvalodu interpretatoram (MLI – *Multi-Language Interpreter*), kas nodrošina programmas izpildi, dinamiski saņemot izmantotās valodas sintaksi un vēlamo semantiku. Semantika tiek definēta kā kompozīcija no daudziem semantiskajiem aspektiem, ņemot vērā valodas pragmatiku. Semantisko aspektu apraksti, tos apvienojot, tiek transformēti uz semantiskām funkcijām, lai izpildi varētu veikt analoģiski *Visitor pattern* principam, apstaigājot iekšējo programmas attēlojumu un izsaucot semantiskās funkcijas.

Semantikas pierakstā tiek izmantotas abstraktas komponentes jeb abstraktie datu tipi, kas meta līmenī tiek saistīti ar konektoriem. Šo komponenšu un konektoru implementēšana var būt ļoti dažāda atkarībā no uzstādītajām prasībām. MLI saņem sintakses un semantikas aprakstus jau kā nokompilētus un izpildāmus objektus, kurus var dinamiski piesaistīt un mainīt MLI darbības laikā. Darbā doti piemēri gan imperatīvas programmēšanas valodas tradicionālās semantikas definēšanai, gan arī specifiskas semantikas definēšanai. Autors piedāvā rīku veidošanā izmantot MLI darbības principus.

Summary

Modern information systems must be varied in terms of their functionality, they must operate without interruption in a rapidly changing and non-uniform environment, and they must be able to adapt in a dynamic way to the demands of users, other information systems, the surrounding environment and the technologies that are used. One way of reaching this goal is the intensive use of specialised tools. In this paper, a communications server is used as a basic example of this process. The server ensures that integrated and consolidated information can be obtained from many different data sources in a non-uniform and distributed environment.

The design of high-quality tools is directly or indirectly associated to the use of domain-specific languages. Domain-specific languages can serve as a specification of a tool's functionality, as the language for entry data or recordings of results, or as the language in which the tool's management commands are recorded. The author has looked at ways in which these languages can be utilised and implemented. He has particularly emphasised the need to define the semantics of the language, as well as the key role which the semantics play in terms of obtaining the relevant code.

In demonstrating the way in which the basic idea can be used in designing a tool, the author has looked at techniques that are used in establishing a database survey tool. The survey tool or data management system can automatically be generated from a physical data model that is recorded with the aid of an ER diagram. An ER diagram contains traditional simple elements, as well as a few additional attributes which make it much easier to generate the application and to improve the usability of the end system. All elements in the ER diagram are transformed into specific application components. Direct relations between the diagram's elements and the user interface are defined. The quality of the application depends on the use of various kinds of interface screens that have been designed by the authors. The paper provides examples of the most important principles in designing the screens.

When it comes to the integration of major systems so that many different data sources can be surveyed and processed as a unified and whole information system, the idea is that a group of tools can be used which ensure the description of data sources, the management of those sources, the implementation of user requests and the depiction of results for the user. All information about the logical structure of the data sources an about opportunities for extracting data from the data sources is stored in a special facility. A special domain-specific language is used to generate the user interface.

One of the most important areas of activity to be discussed in the paper is the definition of semantics for a Multi-Language Interpreter (MLI) so as to ensure the carrying out of the programme by dynamically receiving the utilised language's syntax and desired semantics. Semantics are defined as a composition of many semantic aspects, taking into account the pragmatics of the language. Descriptions of semantic aspects are merged and transformed into semantic functions, so that the performance of the tool can be analogous to the "visitor pattern" principle - surveying the internal representation of the programme and calling up the semantic functions.

In the record of the semantics, abstract components or abstract data types are used - at the meta level these are linked by connectors. The implementation of these components and connectors can be highly varied, depending on the requirements that are put into place. The MLI receives descriptions of syntax and semantics as compiled and executable objects - ones which can be involved and changed dynamically during the period in which the MLI is operating. The paper provides examples of ways in which the traditional semantics of an imperative programming language can be defined, as well as of ways in which specific semantics can be defined. The author proposes that the operating principles of the MLI be put to use in designing tools.

Аннотация

Современные информационные системы должны быть функционально разнообразными, обеспечивать непрерывную работу в быстро меняющейся и неоднородной среде, быть способными динамически изменяться в соответствии с требованиями пользователей, других информационных систем, среды и технологий. Как одна из возможностей для достижения данной цели предлагается интенсивное использование специализированных программных средств (СПС). Основным примером, рассмотренным в работе, является построение Сервера Коммуникаций, который бы обеспечивал получение интегрированной и консолидированной информации из различных источников данных в распределенной и неоднородной среде.

Разработка качественного продукта в данной области прямо или косвенно связана с использованием доменспецифических языков. Доменспецифические языки могут служить функциональной спецификацией СПС, средством записи входных данных или результата или средством записи управляющих команд. Рассмотрены возможности использования и реализации данных языков, особенно выделяя ведущую роль определения семантики языка и получения его выполнимого кода.

Для демонстрации основных идей разработки СПС генерации приложений, рассмотрена техника создания программы просмотра базы данных. Предложенная программа просмотра или система управления данными может быть автоматически сгенерирована из физической модели данных, представленной в виде ER диаграммы. ER диаграмма содержит как традиционные простейшие элементы, так и некоторые дополнительные атрибуты, которые значительно облегчают генерацию приложения и повышают удобство использования конечной системы. Все элементы ER диаграммы преобразуются в определенные компоненты приложения, определена прямая зависимость между элементами диаграммы и пользовательским интерфейсом. Качество приложения зависит от использования типов экранов интерфейса, разработанных автором. В работе даны примеры наиболее важных принципов разработки экранов.

Для интеграции больших систем, чтобы иметь возможность рассматривать и обрабатывать несколько источников данных как единую информационную систему, предлагается использовать группу СПС, которые обеспечивают описание источников данных, управление ими, выполнение запросов пользователей и отображение результатов. Вся информация о логической структуре источников данных и возможностях получения из них данных сохраняется в специальном репозитории. Для генерации пользовательского интерфейса предлагается специально разработанный доменспецифический язык.

Одной из существенных областей, рассмотренных в работе, является определение семантики для многоязыкового интерпретатора (MLI – *Multi-Language Interpreter*), который обеспечивает выполнение программы, динамически получая синтаксис и желаемую семантику используемого языка. Семантика определяется как композиция многих семантических аспектов, принимая во внимание прагматику языка. Описания семантических аспектов, объединяя их, трансформируются в семантические функции, для того чтобы выполнение можно было бы производить аналогично принципу *Visitor pattern*, обходя внутреннее отображение программы и вызывая семантические функции.

В записи семантики используются абстрактные компоненты или абстрактные типы данных, которые на метауровне связаны конекторами. Реализация данных компонент и конекторов может быть очень различной в зависимости от предъявленных требований. MLI получает описания синтаксиса и семантики уже как откомпилированные и исполняемые объекты, которые можно динамически подключать и изменять во время работы MLI. В работе даны примеры как определения семантики традиционного императивного языка программирования, так и определения специфической семантики. Автор предлагает использовать принципы работы MLI в создании СПС.

Pateicības

Es vēlos izteikt vislielāko pateicību visiem, kas mani ir atbalstījuši daudzu gadu garumā, lai taptu šis darbs. Tieša vai netieša palīdzība ir nākusi no daudzām personām, taču dažas man gribās atzīmēt īpaši.

Pirmām kārtām gribu pateikties savai ģimenei un vecākiem par pacietību daudzu gadu garumā, sapratni un atbalstu. Paldies sievai Vinetai un bērniem Amandai Agnesei, Lienei Beātei un Aijai Egijai. Viņas visvairāk izjuta šī darba un ar to saistīto publikāciju tapšanu, jo samazinājās viņām veltītai laiks. Vinetu vēl jāatzīmē, kā daudzu darbā aprakstīto ideju līdzautoru, kā arī visa darba un publikāciju galveno neformālo recenzentu.

Ne mazāk nozīmīgs atbalsts un palīdzība ir saņemta no mana darba vadītāja Jāņa Bičevska. Paldies par nepārtraukto un ilgstošo izglītošanu gan zinātnes, gan profesionālā darba, gan arī cilvēcisko attiecību jomā (mūsu sadarbība ir bijusi vismaz 13 gadus). Viņš ir bijis klāt pie daudzu ideju šūpuļa un būtiskākais – ir bijis galvenais dzinējspēks, lai šīs idejas materializētos publikāciju un reālu projektu veidā.

Īpaši jāatzīmē arī ciešā sadarbība ar Ģirtu Karnīti, kurš ir daudzu publikāciju līdzautors. Pateicoties viņam daudzas idejas tika pārbaudītas praksē, izveidojot Komunikāciju serveri, kas nodrošināja informācijas ieguvi no vairākiem Latvijas reģistriem.

Pateicību gribu izteikt visiem saviem skolotājiem un tagad arī kolēģiem, kas mani ir izglītojuši un stimulējuši darboties zinātnē. Jānis Bārzdiņš, Rūsiņš Mārtiņš Freivalds, Māris Treimanis, Audris Kalniņš, Juris Borzovs un Juris Strods ar savu personīgo piemēru un padomiem ir atbalstījuši manu darbību. Atsevišķi vēl jāmin, ka abi Juri bija ļoti aktīvi šī darba pabeigšanas atbalstītāji. Jurim Strodam jāpateicas arī par daudzu ideju vai to prezentēšanas sākotnējo izvērtēšanu, kas palīdzēja labot pieļautās kļūdas un neprecizitātes.

Lai šis darbs taptu, bija nepieciešama arī dažāda cita palīdzība un atbalsts. Paldies Uldim Straujumam, Ārijai Sproģei, Rudītei Ekmanei, Dzidrai Reidzānei, Leo Seļāvo, Baibai Apinei, Kārlim Streipam, Maksimam Kravcevam un vēl daudziem citiem.

Gribu atzīmēt arī Ralfu Lammelu, kuram es, organizējot apmešanos Rīgā, atbalstīju viņa doktora disertācijas uzrakstīšanu. Daudzas diskusijas par mums kopīgajām tēmām (programmēšanas valodu semantikas) palīdzēja koriģēt manu zinātnisko darbību. Arī turpmāk Ralfs ar savu darbību un publikācijām (pašlaik viņš ir kļuvis par vienu no vadošajiem zinātniekiem valodu specifikāciju jomā) bija man paraugs un apliecinājums, ka daudzi darbā apskatītie virzieni ir aktuāli un nepieciešami.

Šo darbu es gribu veltīt Marijas Arnicānes gaišajai piemiņai, kas bija mana pirmā skolotāja un audzinātāja. Lielā mērā, pateicoties viņai, es esmu tas, kas es esmu. Kopā ar maniem vecākiem viņas izveidotie manas personības pamati un tieksme pēc zināšanām noteikti ir pamatā visam manam tālākajam dzīves ceļam un vienam tā starpfinišam – šim darbam.

Saturs

Zīmējumu rādītājs

Tabulu rādītājs

Ievads

Mūsdienu pasaule mainās un attīstās ļoti strauji. Pēdējos simts gados īpaši būtiskas izmaiņas ir notikušas informācijas apstrādes jomā. Informācijas vērtība nemitīgi aug un tuvākajā nākotnē tā būs viena no vērtīgākajām lietām. Informācijas apstrādē būtiskāka loma ir datoru un atbilstošas programmatūras izmantošanai.

Strauji mainīgās un arvien pieaugošās cilvēku prasības liek programmatūras izstrādātājiem radīt jaunas informācijas apstrādes sistēmas, kas spēj, nepārtraukti strādājot, pielāgoties apkārtējai mainīgajai videi. Dabīgi, ka, uzticot datoriem arvien vairāk pārvaldīt dažādas cilvēkam kritiskās jomas, tiek prasīta programmatūras augsta lietojamības pakāpe, kvalitāte un uzticamība.

Kā veikt topošās sistēmas projektēšanu un ātru izveidi? Kā pamainīt sistēmas uzvedību pēc tās ekspluatācijas uzsākšanas, to neapstādinot? Kā nodrošināt izmaiņu korektumu? Kā nodrošināties pret katastrofālām sekām, ja kādā etapā būs kļūda? Šādus jautājumus var uzdot desmitiem, un nepārtraukti uz tiem tiek meklētas atbildes.

Viens no iespējamajiem veidiem, kas mūs atbalstītu informatīvo sistēmu izveidē atbilstoši mūsu vēlmēm un prasībām, ir veidot papildus specializētus rīkus. Darba motivācija ir atrast tehnoloģiju rīku radīšanai, kas atvieglotu dinamisku sistēmu izveidi un uzturēšanu heterogēnā un telpā sadalītā vidē. Piedevām tiek meklēti līdzekļi, kas ir saprotami ierindas programmatūras izstrādātājiem, citiem IT speciālistiem vai pat sistēmu lietotājiem, nevis tikai atsevišķiem attiecīgās jomas ekspertiem.

Darbā būtisks akcents likts uz domēnspecifisko valodu izmantošanu, kas ir būtisks faktors labu rīku izveidei. Domēnspecifiskās valodas pēc mūsu domām ir atslēga dinamiskai sistēmu izveidei un integrācijai, jo var noteikt gan rīka uzvedību, gan tā saskarni ar citām sistēmām vai lietotāju.

Pēdējos gados strauji pieaug jaunu ar IT jomu saistīto valodu izveide (piemēram, programmēšanas valodas vai datu aprakstīšanas valodas) un attiecīgi ar to implementāciju un lietošanu saistītās problēmas. Jaunās valodas pārsvarā ir klasificējamas kā domēnspecifiskās valodas. Turklāt tām ir nepieciešams ne tikai kompilators vai interpretators, bet arī virkne valodu atbalstošo rīku, jo programmatūras kvalitātes problēmas mūsdienās ir īpaši aktuālas un tās grūti risināt bez specializētiem un automatizētiem palīglīdzekļiem.

Datoru pielietojumi kļūst arvien dinamiskāki, sadalīti laikā un telpā, darbības vide ir heterogēna, jānodrošina paralēla procesu izpilde, jāorganizē sadarbība starp eksistējošām komponentēm vai sistēmām, jāpielāgojas mainīgajiem apstākļiem, nepārtraucot darbu, utt. Arvien vairāk tiek izmantoti interpretatori vai arī rīki koda ģenerēšanai un kompilēšanai izpildes laikā. Diemžēl formālie valodas semantikas apraksta līdzekļi nespēj pilnībā atbalstīt mūsdienu praktisko problēmu risināšanu un sāk zaudēt savas pozīcijas.

Domēnspecifisko valodu implementēšanai mēs izvēlamies interpretatoru izveidi, balstot to uz oriģinālu semantikas pierakstu, kas citur šādā formā līdz šim nav sastapts. Tiek domāts par semantikas pieraksta sarežģītības samazināšanu, tuvinot to lietotāja operacionālajai domāšanai un dodot viņam lielāku rīcības brīvību, salīdzinot ar klasiskajām formālajām semantikām. Pamatproblēma, lietojot programmēšanas valodu formālās specifikācijas, ir pārāk liela sarežģītība, neskaidra to pārvaldība,

nespēja izteikt visas praktiskās vajadzības un beigās tik un tā paliek problēma – kas pierādīs šīs grūti aptveramās specifikācijas korektumu?

Darbā ir aplūkotas arī sistēmu integrācijas problēmas. Pirmām kārtām ir apskatīti izveides principi un tehnika universālam daudzu datubāzu pārlūkam. Pārlūks operē ar informāciju datu avotus aprakstošajā repozitorijā un ar datiem, kas tiek saņemti no eksistējošām datubāzēm. Katrs elements, tāds kā entītija, lauks, relācija, tiek attēlots kā HTML lapas sastāvdaļa ar vēlamo struktūru un izklājumu. Ir izveidoti daudzi paraugi informācijas attēlošanai, kas ļauj dinamiski nomainīt vēlamo prezentāciju. Kā nopietnākais praktiskais pielietojums ir minama Komunikāciju servera izveidošana, kura viens no galvenajiem uzdevumiem ir integrēt kopā daudzas informatīvās sistēmas un piedāvāt no tiem konsolidēto informāciju lietotājam saprotamā formā.

Dati no reālajām datubāzēm pārlūkam tiek piegādāti ar speciālu funkciju palīdzību, kas aprakstītas repozitorijā. Tas ļauj ātri aprakstīt jaunus datu avotus, izveidot datu ieguves funkcijas, veikt vēlākas modifikācijas un pārvaldīt informācijas attēlošanu vienkāršā un vienotā veidā. Pārlūka arhitektūra ir pietiekoši elastīga, lai varētu integrēt visdažādāko tipu datubāzes un izmantot visdažādākos saziņas līdzekļus starp sistēmām. Ar repozitorijā aprakstītajām saitēm ir iespējams loģiski sasaistīt informāciju no dažādiem datu avotiem.

Mūsdienās daudzi pētnieki informatīvo sistēmu integrēšanai piedāvā izmantot globālus darbu vadības procesus. Parasti aprakstītajam procesam ir tikai viena fiksēta semantika, t.i. darbs, kas tam jānodrošina. Mēs piedāvājam veidot un izmantot vairākas semantikas, piemēram, primārā darba plūsmas semantiku, statiskās informācijas vākšanas semantiku, semantiku kļūdu meklēšanai un procesu simulācijai, dažādas semantikas kvalitātes uzlabošanai. Semantikas var kombinēt savā starpā un izpildīt vienlaicīgi.

Darbs sastāv no piecām nodaļām un pēc būtības tas ir autora ar doto problemātiku saistīto publikāciju vai publicēšanai sagatavoto rakstu un rezultātu apkopojums. Darbā apskatīti vairāki nepieciešamo rīku iegūšanas etapi un aspekti, bet pārsvarā koncentrēšanās ir uz lietām, kurās autors mēģinājis atrast jaunus risinājumus problēmām.

Pirmajā nodaļā tiek nosaukti pamatjēdzieni, kas ir nepieciešami problemātikas aprakstīšanai, kā arī aplūkota darba problemātika. Šajā nodaļā paskaidrots, ko mēs saprotam ar jēdzienu *rīks* un kā to varētu veidot. Tā kā rīka implementēšanā mēs balstāmies uz domēnspecifiskajām valodām, tad ir dots ieskats to veidošanā un implementācijā. Ar doto nodaļu ir saistīti sekojoši autora darbi [**Error! Reference source not found.**, Arn03b].

Otrajā nodaļā aplūkojam dažas iespējas, kā darbā aprakstītie risinājumi varētu tikt izmantoti praksē. Galvenais akcents ir likts uz Komunikāciju servera, kas apstrādā informāciju sadalītā un mainīgā vidē, apskatīšanu. Īsumā ir sniegts ieskats arī tālākajos darbības virzienos sistēmu integrācijā – darba plūsmas pārvaldībā ne tikai vienas organizācijas ietvaros, bet arī starp daudzām organizācijām. Ir dota arī pamatinformācija par datu avotiem, to aprakstīšanu un informācijas ieguvi no tiem. Ar doto nodaļu ir saistīti sekojoši autora darbi [ABK99, ABK00a, ABK00b, AK00a, AK00b, ABKK01, ABKK02, AK02a, AK02b].

Trešajā nodaļā detalizētāk ir apskatīta informācijas ieguve un attēlošana lietotājam. Īpašs akcents ir likts uz to, lai izveidotu vienkāršu mehānismu, kas

apmierinātu gan dinamiskuma, gan programmatūras korektuma, gan arī lietotāja saskarnes prasības. Ideja balstās uz informācijas prezentēšanu dažāda tipa ekrānos, izmantojot noteiktus ekrāna objektus ar precīzi definētu atbilstību datu avotu aprakstošajam modelim. Ar doto nodaļu ir saistīti sekojoši autora darbi [Arn98, AK00a, AK00b].

Ceturtajā nodaļā ir aplūkota informācijas apstrādes lietojumu izveide. Faktiski tiek piedāvāti varianti nepieciešamās programmatūras ģenerēšanai. Autors piedāvā risinājumus gan statiskas, gan dinamiskas programmatūras ģenerēšanai. Statiska programmatūra minimāli maina savas īpašības, t.i. tā tiek kompilēta, savukārt dinamiska programmatūra maina savas īpašības atkarībā no ārējiem apstākļiem, t.i. tā tiek interpretēta. Abos gadījumos ģenerēšana balstās uz datu avotu modeļiem, kas glabājas īpašā repozitorijā. Dinamiskas programmatūras ģenerēšanā lielu lomu spēlē domēnspecifisku valodu izmantošana. Šajā nodaļā ir dots viens domēnspecifiskas valodas piemērs, kas var kalpot par rīka specificēšanas līdzekli. Ar doto nodaļu ir saistīti sekojoši autora darbi [Arn98, AK00a, AK00b, ABK00b].

Piektajā nodaļā ir apskatīta rīka darbības semantikas pierakstīšanas veids un tās iespējamā implementācija. Šajā nodaļā tiek definēts daudzvalodu interpretators (MLI), kas nodrošina programmas izpildi, saņemot arī atbilstošās programmēšanas valodas sintaksi un nepieciešamo semantiku. Daudzvalodu interpretators var kalpot par pamatu lielākajai daļai mums nepieciešamo rīku izveidei. Galvenā uzmanība ir veltīta mūsu daudzvalodu interpretatora semantikas (uzvedības) uzdošanai, kā arī šīs semantikas pārvēršanai par izpildāmu kodu. Semantika tiek definēta kā kompozīcija no daudziem semantiskajiem aspektiem, ņemot vērā valodas pragmatiku. Semantisko aspektu apraksti, tos apvienojot, tiek transformēti par semantiskām funkcijām, lai izpildi varētu veikt, traversējot iekšējo reprezentāciju un izsaucot semantiskās funkcijas, analoģiski *Visitor pattern* principam. Semantikas pierakstā tiek izmantotas abstraktas komponentes, kas tiek saistītas meta līmenī ar konektoriem. Savukārt šo komponenšu un konektoru implementācija var būt ļoti dažāda. Darbā doti piemēri semantikas definēšanai gan parastai, gan specifiskai izpildei. Ar doto nodaļu ir saistīti sekojoši autora darbi [AAB96a, AAB96b, **Error! Reference source not found.**].

1 Pamatjēdzieni un problemātika

Eksistē daudzi un dažādi veidi, kā mēs varam būvēt informatīvās sistēmas. Mēs piedāvājam skatīties uz informatīvajām sistēmām kā uz rīku komplektu, kas ir integrēts vienā izpildes vidē kopīgai sadarbībai. Bez tam tiek izmantoti arī citi rīki, lai projektētu, implementētu vai testētu iepriekš minēto rīku komplektu un to darbības vidi.

Rīku izmantošana palielina izstrādes ērtumu, produktivitāti un kvalitāti, nodrošinot mērķa rīku izveidi, to savstarpēju integrēšanu un atbilstību programmatūras izstrādes metodoloģijai un prasībām. Izmantoto rīku kopa veido specifisku programmatūras vidi.

Dotajā nodaļā tiek dots ieskats zināmos un autora jaunievestos jēdzienos, kas nepieciešami tālākā darba izpratnei. Ir aplūkots rīka jēdziens; klasiskie rīku integrēšanas principi; rīku izveides, pārvaldības un kontroles principi; iespējamā rīka arhitektūra; kā arī apskatīta domēnspecifisko valodu, kas varētu būt rīku veidošanas pamatā, veidošana, realizēšana un izmantošana.

1.1 Dati, informācija un neviendabīga vide

Dati ir faktu vai ideju reprezentācija formalizētā veidā ar spēju būt komunicētiem vai manipulētiem ar kādu procesu.

Informācija ir jēga, ko cilvēki piešķir datiem automatizētajā datu apstrādē, izmantojot zināmu paražu līdzekļus, lai to prezentētu.

Tikai cilvēki var apstrādāt informāciju. Savukārt mašīnas var apstrādāt tikai datus.

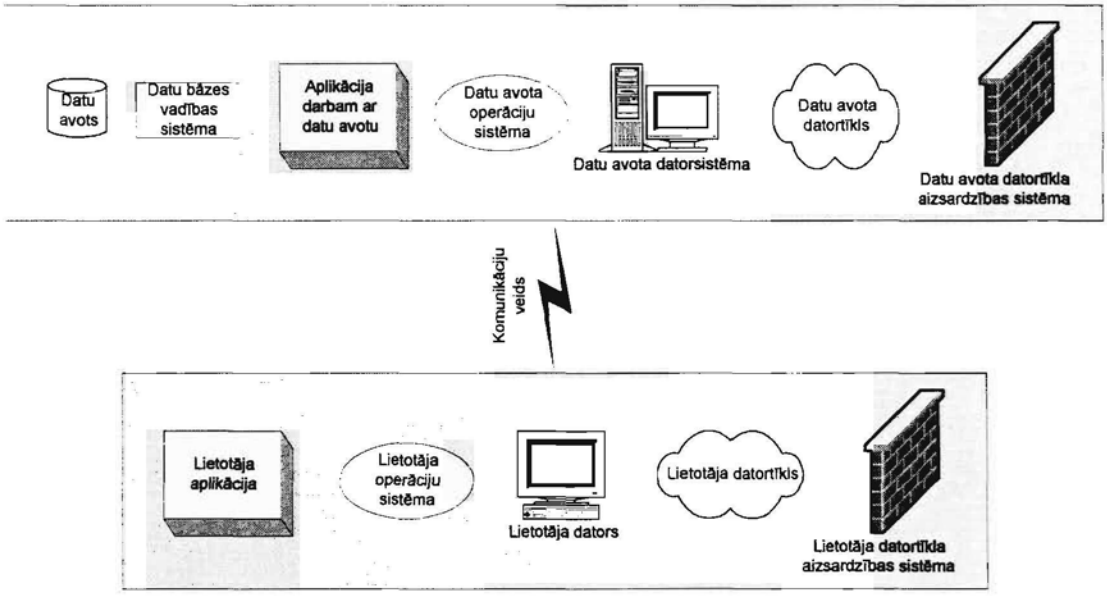
Runājot par datiem un informāciju, uzreiz jāmin arī jēdziens *dezinformācija*. Tā rodas tad, kad dati (fakti, idejas) ir sagrozīti un/vai tiem tiek piešķirta nepareiza jēga. Tātad datu kvalitāte nosaka arī informācijas kvalitāti. Tāpat informācijas kvalitāti nosaka arī datu apstrādes sistēmas un tajās ietverto algoritmu un prezentēšanas kvalitāte.

Dotajā darbā, risinot informācijas apstrādes problemātiku, daudz jādomā arī par datu kodēšanu un to glabāšanas formātiem. Dati tiek glabāti lielās loģiskās krātuvēs – datu avotos. Avotiem tiek būvēti dažādi modeļi, kas atšķiras pēc abstrakcijas līmeņa (konceptuālais, loģiskais, fiziskais), vai attēlojuma formas (ER, Objektu modelis, u.c.))

Kādu modeli labāk saprot lietotājs? Mums gribas apgalvot - tādu, kas ir tuvāk reālās dzīves atainojumam. Līdz ar to mēs meklējam iespējas darboties ar informatīvās sistēmas lietotājam saprotamākiem jēdzieniem.

Ir ļoti grūti novilkt līniju apkārt informatīvai sistēmai un teikt, ka jūs to kontrolējat un pārvaldāt. Dati glabājas ļoti heterogēnā un telpā sadalītā vidē, bieži vien autonomās informatīvās sistēmās. Datu vai informācijas apstrādei tas rada papildus grūtības. Dati tiek pārvietoti telpā. Vide var būt ļoti neviendabīga un katrā vietā var notikt nevēlama datu modifikācija vai pat zaudēšana.

Zīm. 1 attēlota samērā vienkārša situācija – datu avots un servisi ir novietoti vienā organizācijā (vai organizācijas struktūrvienībā), bet lietotājs atrodas citā organizācijā (vai tās pašas organizācijas citā struktūrvienībā).



Zīm. 1 Piemērs neviendabīgās vides elementiem

Process, kura rezultātā dati tiek transformēti par informāciju un tā savukārt attēlota informācijas patērētājam, ir garš un grūts, jo jāiet cauri daudzām lietojumprogrammām, operētājsistēmām, aizsardzības sistēmām, datu pārraides protokoliem, utt. Ja sistēmas veidotājs pielaidīs kļūdu kādā no sistēmas daļām, tad patērētājs var saņemt dezinformāciju vēlamās informācijas vietā, vai pat nesaņemt neko.

Problēma kļūst daudz nopietnāka un grūtāk risināma, ja informācija tiek radīta, izmantojot daudzus datu avotus un informācijas piegādes servisiem nepieciešams strādāt nepārtrauktā režīmā. Tas nozīmē, ka pielāgošanās datu avotiem un servisu izmaiņas jāveic dinamiski, neapstādinot sistēmu.

Šīs visas problēmas uzsver nepieciešamību, kāpēc nepieciešami rīki, kas strādā kā starpnieki starp datubāzēm, serveriem un gala lietojumprogrammām. Mums ir nepieciešams uzturēt aprakstus datu struktūrām, to saturam, datu īpašībām, pieejamajiem servisiem (tie ir meta-dati par datu avotiem un servisiem). Tas palielina nepieciešamību dinamiskai manipulācijai gan ar datiem, gan meta-datiem. Bez tam mums ir jāsatraucas par sistēmas kvalitāti, ko varētu nodrošināt ar testēšanas, simulācijas un uzraudzības rīkiem.

1.2 Informācijas apstrādes rīka un vides jēdzieni

Rīku kolekciju veidošana ir aktuāla jau daudzus gadus [How82]. Šīs jomas nozīmīgums aug dienu no dienas arī pašlaik [HOT00]. Apskatīsim īsi, kādi ir svarīgākie jēdzieni un principi, lai organizētu rīkus vienotā darboties spējīgā vidē.

Laikam ritot, dažādu rīku skaits pakāpeniski aug. To pielietojamības sfēras ir dažādas, sākot no tradicionālajiem rīkiem (teksta redaktori, kompilatori, interpretatori,

atkļūdotāji) un beidzot ar specifiskiem atbalsta rīkiem prasību vākšanai, projektēšanai, lietotāja saskarnes ģenerācijai, informācijas pieprasījumu veidošanai, ziņojumu definēšanai, sistēmu arhitektūras veidošanai, testēšanai, programmatūras versiju kontrolēšanai, konfigurācijas pārvaldībai, datubāzu administrēšanai, reversai inženierijai, programmas vizualizēšanai, metriku vākšanai u.c. [HOT00].

Jebkura sistēma, kas asistē programmētājam kaut kādā programmēšanas aspektā var tikt uzskatīta par *programmēšanas rīku*. Līdzīgi, sistēma, kas asistē kaut kādā programmatūras izstrādes fāzē, var tikt uzskatīta par *programmatūras rīku*. *Programmēšanas vide* ir programmēšanas rīku kopums, kas ir izveidots, lai atvieglotu programmēšanu un celtu programmētāja produktivitāti. Savukārt *programmatūras vide* ir programmēšanas vides paplašinājums ar programmatūras rīkiem un atbalsta visu programmatūras izstrādes procesu.

Iepriekš minētās definīcijas ir attiecināmas uz jebkuru programmatūru. Apskatīsim šaurāku programmatūras virzienu – programmatūru, kas domāta informācijas apstrādei neviendabīgā un dinamiskā vidē. Atbilstoši var ieviest jēdzienus *informācijas apstrādes rīks* un *informatīvo sistēmu programmatūras vide*.

Informācijas apstrādes rīks ir jebkura sistēma, kas nodrošina kādu no uzdevumiem informācijas apstrādē. ***Informācijas apstrādes rīku vide*** (informatīvā sistēma) ir informācijas apstrādes rīku kopums. ***Informatīvo sistēmu programmatūras rīks*** ir sistēma, kas asistē informatīvās sistēmas izveides procesā. Savukārt ***informatīvo sistēmu programmatūras vide*** ir informācijas apstrādes rīku vides paplašinājums ar informatīvo sistēmu programmatūras rīkiem.

Turpmāk tekstā jēdziena *informācijas apstrādes rīks* vietā izmantosim vienkārši vārdu *rīks* un jēdziena *informatīvo sistēmu programmatūras vide* vietā – *vide*.

Rīki var tikt klasificēti vai grupēti pēc informācijas apstrādes fāzēm vai problēmām, ko tie risina. Vide var tikt raksturota ar rīku tipiem, ko tā satur, un informācijas apstrādes problemātiku, ko var atrisināt ar vides palīdzību. Protams, ka vides var būt ļoti dažādas arī pēc rīku integrācijas principa un rīku savstarpējās saistības pakāpes. Rīki, kas tiek iekļauti vidē, var nebūt tieši saistīti ar informācijas apstrādi, bet tie var nodrošināt vides lietošanas ērtumu un produktivitāti, t.i. tie ir papildrīki.

1.2.1 Informācijas apstrāde rīku klasifikācija

Dabiskākais rīku klasifikācijas princips ir tos grupēt pēc funkcionalitātes un problēmām vai uzdevumiem, ko tie risina. Tāpat tos varētu grupēt arī pēc izveides principa, integrēšanās principa vai izpildes uzvedības.

Rīku klasifikācija pēc funkcionalitātes (dots tikai piemērs iespējamajai klasifikācijai):

1. Datu ieguves rīki no fiziskiem datu avotiem
   - 1.1. Relāciju datubāzes
     - 1.1.1. Konkrētas datubāzes rīks
     - 1.1.2. Universāls rīks daudzām datubāzēm
   - 1.2. Objektorientētās datubāzes
   - 1.3. Strukturēti faili
2. Komunikāciju rīki darbam ar datu avotiem
   - 2.1. Specifiska klienta programmatūra darbam ar datu avotu
   - 2.2. Pārlūkprogramma
   - 2.3. Elektroniskais pasts

3. Datu avotu aprakstīšanas rīki
   3.1. Datu avotu repozitorijs un tā pārvaldnieks
   3.2. Datu avotu definētājs, grafiskais redaktors
   3.3. Datu avotu servisu aprakstīšanas rīks
   3.4. Vārdnīcas un to pārvaldības rīki
4. Pieprasījumu apstrādes rīki
   4.1. Pieprasījumu definēšanas rīki
   4.2. Pieprasījumu izpildes rīki
5. Lietotāja interfeisa ģenerācijas rīki
   5.1. Statiska lietojumprogramma
   5.2. Dinamiska lietojumprogramma
6. Lietotāju pārvaldības rīki
   6.1. Lietotāju reģistrācijas un vispārējas pārvaldības rīki
   6.2. Lietotāja tiesību pārvaldības rīki
   6.3. Lietotāju apkalpošanas uzskaites un pārvaldības rīki (izmaksas, laiks, statistika)
   6.4. Lietotāju konfigurācijas rīki
7. Sistēmas uzraudzības rīki
   7.1. Reģistrācijas žurnālu pārvaldības rīki
   7.2. Statistikas uzskaites rīki
   7.3. Drošības kontroles rīki
8. Sistēmas kvalitātes rīki
   8.1. Sistēmas testēšanas rīki
   8.2. Dokumentēšanas rīki
9. Lietotāja pagaidu apstrādes pieprasīto vai piederošo datu pārvaldības rīki

Rīku klasifikācija pēc uzvedības

Pēc rīka uzvedības jeb darbības principa varētu izdalīt divas pamatklases:

- **Statisks rīks**. Rīks, kurš veic konkrētu fiksētu funkcionalitāti, kas laika gaitā būtiski nemainās un kuru var ietekmēt ar nelielām iepriekš fiksētām konfigurēšanas iespējām. Ieejas parametri ir samērā stingri definētā formātā. Tehniskā izpildījumā tas parasti ir kompilēts statisks kods.
- **Dinamisks rīks**. Dinamisks rīks savu funkcionalitāti jeb izpildes semantiku var diezgan krasi variēt. Ir lielas konfigurēšanas iespējas. Ieejas parametri laika gaitā var būtiski mainīties. Tehniskā izpildījumā tas varētu būt interpretators, kas realizē savu darbību interpretēdams iedotās komandas.

1.2.2 Rīku integrācija

Lai iegūtu visu informatīvo sistēmu, rīki (gan sistēmā iekļautie, gan izstrādi atbalstošie) tiek apvienoti vienotā vidē, t.i. rīki tiek integrēti kopā. Katras vides izveidotāji integrāciju veido pēc saviem ieskatiem, pārsvarā vadoties pēc praktiskiem apsvērumiem. Apkopojot tipiskākās pieejas, var runāt par līdz šim brīdim populārākajiem jeb klasiskajiem principiem [Rei96].

Klasiskās integrācijas iespējas

Eksistē trīs galvenās pieejas, kā daudzus rīkus integrēt kopā, lai iegūtu vienotu un pietiekami elastīgu vidi:

- **Datu integrācija**. Tiek veidots kopīgs datu repozitorijs, ar kuru strādā lielākā daļa rīku.

- **Vienots lietotāja skats**. Lietotājs redz un lieto visu sistēmu kopumā, nevis kopu ar atsevišķiem rīkiem, tiek operēts ar datu objektiem un tiem konkrētajā brīdī pieļaujamajām operācijām (rīkiem).
- **Kontroles integrācija**. Rīki sadarbojas ar ziņojumu palīdzību, var tikt nosūtīti gan dati, gan komanda, kas jādara ar datiem. Izmanto gan tiešo rīku sadarbību, gan sadarbību caur centrālo ziņojumu serveri.

Modernām vidēm jāizmanto visu augstāk minēto integrāciju iespējas, atrodot labākās integrēšanas attiecības konkrētās vides komponentēs.

1.2.3 Rīku izstrādes, vadības un kontroles principi

Veidojot dinamisku informācijas apstrādes vidi, ir jāņem vērā, ka apkārtējā vide pastāvīgi mainās. Arī prasības pašai videi pastāvīgi mainīsies. Piedevām mūsdienās servisiem ir jāstrādā nepārtraukti, t.i. visām izmaiņām vajadzētu notikt, neapstādinot esošos servisus. Rīkiem vajadzētu būt dinamiskiem, t.i. ērti konfigurējamiem, ar iespēju mainīt to semantiku, izmantot dažādus ieejas un izejas formātus, utt.

Lai vienkāršāk, ērtāk, kvalitatīvāk un drošāk varētu pārvaldīt rīkus un tos savstarpēji integrēt kopīgā vidē, ieteicams izmantot vienotus rīku izstrādes/izvēles, vadības un kontroles principus. Šī varētu būt cita integrācijas dimensija (*saistības dimensija*), salīdzinot ar klasiskajiem integrācijas principiem.

Būtiskākie aspekti, kas vieno rīkus *saistības dimensijā* varētu būt:

- **Vienota rīka arhitektūra**. Ir vienoti principi, kā rīks tiek izveidots, kas ļauj vienotā stilā radīt daudzus rīkus ar kopīgām komponentēm. Rezultātā ir ātrāka un kvalitatīvāka rīku izstrāde un turpmākā uzturēšana (modificēšana).
- **Kopīga programmatūra**. Ja rīkiem ir vienota ideoloģija un arhitektūra, tad to implementācija var saturēt kopīgus moduļus, apakšsistēmas, bibliotēkas un citu programmatūru.
- **Vienots rīku konfigurēšanas mehānisms**. Tā kā rīki lielākoties ir dinamiski, tad tiem ir nepieciešama elastīga un ātra konfigurēšana (rīka darbības semantikas uzdošana) atbilstoši nepieciešamajai apstrādei. Vēlams ir vienots princips, jo tas atvieglo rīku izmantošanu, integrēšanu un tālāku uzturēšanu.
- **Vienots rīku vadības mehānisms**. Rīkam var būt dažādi interfeisi, kā to varētu vadīt, t.i. dot komandas. Vēlami ir vienoti vadības standarti.
- **Vienots rīku kontroles mehānisms**. Lietojot dinamiskus rīkus, kuriem jāstrādā nepārtrauktā režīmā, ļoti liela nozīme ir to korektības kontrolei, t.i. kvalitātes pārbaudei (īpaši testēšanai). Kļūdaina rīka darbība var radīt neparedzamas sekas, jo var tikt sabojāta visa informatīvā sistēma (darbības apstāšanās, daļēja funkcionēšana vai nepareiza rezultāta izdošana).
- **Vienots testēšanas princips un izmantojamie līdzekļi**. Sistēmu kvalitāte ir kritisks faktors jebkurai sistēmai. Mums ir nepieciešams rīkus testēt pirms to ieviešanas ekspluatācijā. Vienoti principi un testēšanas rīki var būtiski samazināt resursu (laiks, nauda, cilvēki) patēriņu. Pat vēl vairāk, mums nepieciešams turpināt testēšanu reālās sistēmas darbības laikā un pārbaudīt

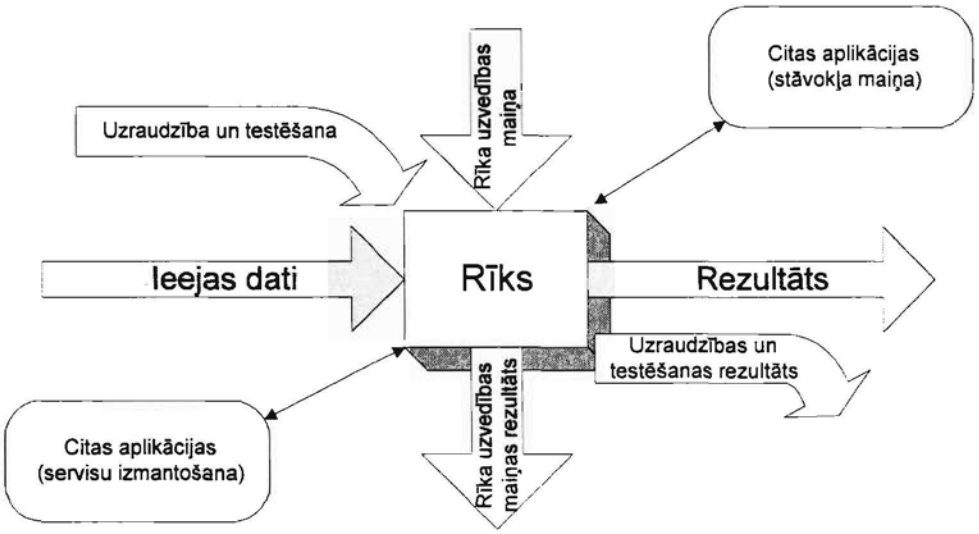
katru operāciju, ja mēs lietojam dinamisku koda ģenerāciju un tūlītēju šī koda kompilāciju vai interpretēšanu.

1.2.4 Rīka vispārinātā arhitektūra

Ir ļoti dažādi viedokļi par to, kāda varētu būt rīka konceptuālā arhitektūra. Pirms mēs piedāvājam savu skatu uz rīku, definēsim pamatjautājumus, kas mums ir svarīgi, lai izprastu rīka būtību:

- Kāds ir rīka pamatuzdevums? Kāpēc tas ir vajadzīgs?
- Kā nepieciešamo rīku var uzbūvēt? Kādas ir iespējamās tehnoloģijas, datu struktūras un algoritmi? Vai mēs varam adaptēt eksistējošus rīkus vai moduļus?
- Kas ir rīka ieejā? Vai rīks saņem visus ieejas datus pirms darba sākuma, vai arī ieejas dati tiek saņemti pa porcijām?
- Kas ir rīka izejā? Vai rīks izdod visus rezultātus pēc visu skaitļojumu pabeigšanas, vai arī rezultāts tiek dots pa porcijām?
- Vai rīkam ir stāvokļi vai arī tas ir bezstāvokļu? Vai starp secīgām komandām rīks atceras vēsturi par iepriekšējiem skaitļojumiem?
- Kā rīku var vadīt un kontrolēt? Kāda ir vēlamā rīka saskarne?
- Vai nepieciešama dinamiska rīka uzvedības maiņa bez visas izpildes vides apturēšanas vai pat rīka uzvedības maiņa tā darbības laikā? Kas ir dinamiski jāmaina, piemēram, ieejas datu struktūru, rīka semantiku, konfigurāciju?
- Vai ir nepieciešama rīka darbības kontrole pirms, pēc vai pat darba laikā?
- Vai rīks sadarbojas ar citām lietojumprogrammām, neņemot vērā pamata ieejas datus vai rezultātu? Vai informācija tiek tikai saņemta vai arī sūtīta? Vai sadarbība ar citām lietojumprogrammām / rīkiem ir sinhrona vai asinhrona?
- Vai rīks spēj mainīt citu lietojumprogrammu / rīku stāvokli vai konfigurāciju? Kādā laikā to var izdarīt - pirms cita rīka darbības sākšanas vai tā darbības laikā?

Balstoties uz šiem jautājumiem, mēs piedāvājam sekojošu vispārināta rīka arhitektūru, atstājot šajā līmenī tikai būtiskākās lietas (Zīm. 2).



Zīm. 2 Vispārināta rīka arhitektūra

1.3 Domēnspecifisku valodu loma rīku izveidē

Veidojot dinamiskus rīkus, kuriem ir jābūt arī ar augstu kvalitāti un drošību visās nozīmēs, liela nozīme ir rīka uzvedības formalizēšanai. Viens no populārākajiem veidiem ir aprakstīt rīka uzvedību ar speciālas valodas palīdzību. Šai valodai savukārt tiek definēta semantika, kas faktiski arī nosaka rīka funkcionalitāti. Līdz ar to par atslēgas jautājumu kļūst problēma, kā vieglāk pierakstīt šīs specifiskās valodas semantiku un kā ar mazākām pūlēm varētu iegūt dažādas rīka uzvedības, balstoties tikai uz šīs valodas sintaksi.

1.3.1 Programmēšanas valodu attīstības dinamika

Dažādu programmēšanas valodu skaits pakāpeniski arvien pieaug. *Jean E. Sammet* 1993.gadā fiksēja stāvokli programmēšanas valodu lietošanas jomā iepriekšējos 15 gados [Sam96]. Saskaņā ar viņa pētījumiem 1978. gadā ASV tika izmantotas aptuveni 170 valodas, no kurām kādas 80 bija vispārējas lietošanas valodas, bet pārējās 90 tika lietotas specializētās lietojumprogrammās. 1993.gadā jau bija aptuveni 1000 valodas, kurām bija nopietnas implementācijas un kuras tika lietotas. Vēl kādas 500 valodas bija projektēšanas vai izstrādes stadijā. Un, protams, ka bija milzīgs daudzums aprakstītu, bet reāli nerealizētu valodu. Aptuvenie novērtējumi rādīja, ka kādas 300 valodas tika izmantotas reālā praksē.

Pāris gadu vēlāk savus pētījumus publicēja *Kinnersley* [Kin95]. Viņa valodu sarakstā jau figurēja vairāk kā 2000 vienības. No tām 360 tika attiecinātas uz specifisko lietojumprogrammu domēnu. Vēl vairāk bija vispārēja pielietojama valodu dialekti ar būtiskiem uzlabojumiem specifiskām vajadzībām, tātad faktiski arī ir attiecināmas uz domēnspecifisko valodu saimi.

Patreizējā brīdī ir grūti novērtēt reālo valodu skaitu, bet tas noteikti mērāms vairākos tūkstošos. Šādu secinājumu var izdarīt sastopot literatūrā jauno valodu aprakstus arvien biežāk (un lielākā daļa no tām ir attiecināma uz domēnspecifiskām valodām). Arī vispārējā IT straujā ieiešana visās sfērās ir sekmējusi jaunu domēnspecifisku valodu tapšanu.

Valodas eksistence visbiežāk nav iedomājama bez reālas implementācijas, t.i. bez kompilatora vai interpretatora. Izņēmumi parasti ir specifiskās meta-valodas, piemēram, BNF. Kā nodrošināt tik daudzas valodas ar kompilatoru vai interpretatoru? Kā nodrošināt vēl papildu servisu šo valodu izmantošanā? Vai daudzas labas idejas nav mirušas tikai tāpēc vien, ka tehnisku iemeslu pēc valodu nav izdevies iedzīvināt reālajā praksē?

Dotajā darbā apskatīsim problēmas, kas rodas domēnspecifisko valodu implementācijā, un piedāvāsim iespējamos variantus to risināšanā. Apskatītos risinājumus varētu attiecināt uz vidējas sarežģītības valodām, jo ļoti plaša pielietojuma rūpnieciskām valodām parasti tiek meklēti specifiski risinājumi, lai panāktu augstu produktivitāti. Savukārt triviālām valodām implementācija jau ir pietiekoši apskatīta specializētajā literatūrā.

1.3.2 Kāpēc rodas jaunas valodas?

Jaunu valodu veidošanas iemesli ir vairāki, un svarīgākos no tiem ir atzīmējis daudzu nozīmīgu un populāru valodu autors un izstrādātājs *Frederick Brooks* [Bro96]. Viens no iemesliem ir eksistējošo valodu uzlabošana un pārstrāde, lai izlabotu kļūdas un palielinātu valodas produktivitāti – gan programmu rakstīšanā, gan arī to izpildē. Sevišķa loma ir produktivitātes uzlabošanā, jo domēnspecifiskās valodas, piemēram, SQL, Excel izklājlapa, LISP vai kāda objektorientēta valoda, kardināli paaugstina produktivitāti no daudziem aspektiem, veicot atbilstošo specifisko darbu [DK98, Sal98].

Nākošais svarīgais aspekts ir nepieciešamība palielināt programmatūras uzticamību. Jo, piemēram, strādā princips – "ja jūs to nevarat pateikt vispār, tad jūs to nevarat pateikt kļūdaini". Līdz ar to nopietni tiek strādāts pie valodas sintakses un pieļaujamajiem izteiksmes līdzekļiem, lai īsi un skaidri definētu, kas datoram jādara, bet būtu arī aizsardzība no daudzām iespējamajām kļūdām. Bez tam augsta un abstrakta līmeņa valodas nodrošina lielu operāciju korektu veikšanu (piemēram, SQL pieprasījumi datubāzēm vai valodas sintaktiskā koka iegūšana ar LEX/YACC palīdzību).

Cits svarīgs iemesls ir jaunu ideju realizācija. Piemēram, jēdziens *saistīšanās laiks* (*binding time*). Asamblera līmenī par tādām lietām nerunā, bet augsta līmeņa valodām var būt mainīgo piesaiste kompilēšanas laikā vai arī tā tiek atlikta līdz pēdējam brīdim izpildes laikā, nodrošinot ļoti lielu dinamiskumu. Cits piemērs ir jaunu algoritmu vai tehnoloģiju parādīšanās, kuras ir jānodrošina ar valodām, piemēram, paralēlā skaitļošana, dalītā skaitļošana vai kvantu skaitļošana.

Lielam daudzumam valodu vienīgais mērķis ir bijis un būs publicēšanās iespēja. Valodas tiek ieviestas ar mērķi, lai būtu iemesls veidot publikācijas. Nereti šāda darbība ir traucējoša, kas maldina potenciālos izmantotājus.

Daļa no valodām ir tapusi izklaidējoties. Eksistē cilvēki, kuriem ir hobijs veidot valodas, pētīt tās, bet ar to bieži vien arī viss beidzas.

Bez tam ir vēl liela valodu grupa, kurām it kā nav tiešais praktiskais pielietojums uz datora, bet tās ir izveidotas izglītošanās pēc. Parasti šādas valodas top apmācības mērķiem, lai atvieglotu kādu specifisku zināšanu apguvi, vai arī studentiem ir dots uzdevums izveidot valodu ar konkrētu specializāciju.

Pēdējos gados parādās nepieciešamība izveidot līdzekļus, kas ļauj ar datoru "sarunāties" un dot komandas ne tikai programmētājiem, bet arī citiem speciālistiem,

piemēram, gan HTML, gan valodas *virs tās* lapu izveidei (dizaineriem), specifisku ekonomisko vai statistisko aprēķinu valodas (finansistiem), XML – zināšanu un informācijas pierakstīšanai.

Reizēm domēnspecifiskas valodas netiek formāli fiksētas uz papīra, bet faktiski ir izstrādāti rīki, kas balstās uz kāda izstrādātāja galvā izveidotu valodas modeli. Protams, ka valoda var būt nepilnīga, nekonsekventa, vietām pat pretrunīga. Iespējams, ka rīks neatbilst tai pilnībā, taču tas ir izmantojams praksē konkrētu uzdevumu veikšanai.

Daļa no valodām bieži vien kalpo tikai kā specifikācijas valoda, t.i. ar tās palīdzību tiek pateikts, kas jāizstrādā, bet implementācija tiek realizēta atbilstoši apstākļiem.

1.3.3 Domēnspecifiskas valodas jēdziens

Kā nošķirt domēnspecifiskas valodas no citām valodām vai datora "vadīšanas līdzekļiem"? Vispirms jānosauc svarīgākās uz domēnu orientētās risinājumu grupas:

- Funkciju vai metožu bibliotēkas;
- Objektorientēts karkass (sistēma) vai komponentes karkass;
- Domēnspecifiskas valodas.

Domēnspecifiska valoda ir programmēšanas valoda vai izpildāma specifikāciju valoda, kas, izmantojot atbilstošus apzīmējumus un abstrakciju, piedāvā izteiksmīgu spēku (jaudu) konkrēta apgabala problēmu risināšanā, fokusējoties un parasti pat ierobežojoties tikai uz šo problēmu apgabalu [DKV00].

Literatūrā var atrast arī citas definīcijas, taču šī šķita viena no skaidrākajām. Turpmāk tekstā domēnspecifiskas valodas apzīmēšanai izmantosim saīsinājumu DSL. Salīdzinot ar vispārēja pielietojuma valodām DSL piemīt gan priekšrocības, gan trūkumi.

Domēnspecifisko valodu priekšrocības

- DSL atļauj risinājumu izteikt ar problēmas apgabala jēdzieniem un abstrakcijas līmeni. Līdz ar to problēmas apgabala speciālisti, kas var nebūt arī datorspeciālisti, var saprast, pārbaudīt, modificēt un pat izstrādāt programmas šajā valodā;
- DSL programmas ir kodolīgas, īsas, pašdokumentējošas plašā apjomā un var tikt izmantotas ļoti dažādiem mērķiem;
- DSL uzlabo produktivitāti, uzticamību, uzturamību un portabilitāti;
- DSL sevī ietver apgabala zināšanas un tādējādi nodrošina to konservāciju un šo zināšanu lietošanu;
- DSL nodrošina validāciju un optimizāciju apgabala līmenī;
- DSL nodrošina labāku sistēmas testējamību.

Domēnspecifisko valodu trūkumi

- DSL projektēšanai, implementēšanai un uzturēšanai ir nepieciešamas papildus izmaksas;
- DSL lietotājiem ir nepieciešama apmācība;
- DSL ir ierobežota pielietojamība;
- Ir grūtības nospraust precīzas DSL pielietošanas apgabala robežas;
- Ir grūtības, balansējot starp DSL un vispārējas nozīmes programmēšanas valodas konstrukcijām;
- Eksistē potenciālais efektivitātes zaudējums, ja salīdzina ar tieši rakstītu programmatūru.

DSL analīze, problēmas un liels daudzums anotētas bibliogrāfijas ir atrodams [DKV00].

1.3.4 DSL projektēšana un implementācija

Lai nonāktu līdz kvalitatīvai DSL, jāveic rūpīga tās plānošana, projektēšana un pēc tam implementēšana. DSL attīstība ir līdzīga citu programmēšanas valodu attīstībai.

DSL attīstības metodoloģija

Domēnspecifisku valodu attīstība parasti satur sekojošus etapus [DKV00]:

- **Analīzes posms**
  1. Identificēt problēmas apgabalu;
  2. Savākt visas atbilstošās zināšanas par izvēlēto problēmas apgabalu;
  3. Apkopot savāktās zināšanas parocīgos semantiskos jēdzienos, apzīmējumos un operācijās;
  4. Uzprojektēt domēnspecifisko valodu, kas precīzi apraksta lietojumprogrammas problēmu apgabalā.
- **Implementācijas posms**
  5. Izveidot bibliotēku, kas implementē semantiskos jēdzienus;
  6. Uzprojektēt kompilatoru (interpretatoru), kas domēnspecifiskās valodas programmas translē uz (izpilda ar) secīgiem izstrādātās bibliotēkas izsaukumiem;
- **Lietošanas posms**
  7. Uzrakstīt programmas domēnspecifiskajā valodā visām nepieciešamajām lietojumprogrammām un vajadzības gadījumā nokompilēt tās.

DSL implementācija

Vispirms ir būtiski, vai ir pilnīgi jauna valoda vai ir papildināta kāda esoša ar jaunām konstrukcijām. Esošai valodai bieži vien var izmantot jau esošos kompilēšanas vai interpretēšanas rīkus.

Jaunai valodai pamatjautājums ir, ko veidot - kompilatoru vai interpretatoru? Tiesa mūsdienās arvien biežāk tiek veidoti interpretatori, kas lielā mērā jau pārveido kodu tuvu mašīnvalodai. Arī kompilatori tiek veidoti tādi, kas kompilē programmas kodu izpildes laikā, tiklīdz atbilstošais kods jāizpilda (*just-in-time compilers*). Rezultātā grūti nospraust klasisko robežu starp interpretatoru un kompilatoru. Ņemot vērā datoru jaudas, var atļauties lietot interpretatorus, iegūstot dinamiskuma priekšrocības [RLV+96]. Šī darba autors vairāk ir tieši interpretatoru piekritējs DSL implementācijai, jo interpretators ir daudz elastīgāks par kompilatoru un bieži vien arī vienkāršāks realizācijā [Kam90].

Citi populāri DSL implementēšanas varianti ir:

- Iebūvētas valodas vai domēnspecifiskas bibliotēkas (piemēram, izmantojot lietotāja definētas funkcijas jau kādā esošā valodā)
- Preprocēšana vai makro procesēšana (piemēram, veidojam mums nepieciešamās DSL konstrukcijas, kas faktiski ir tikai īsāks pieraksts veseliem programmas fragmentiem)
- Paplašināts kompilators vai interpretators (piemēram, Tcl)

Turpmāk darbā tiks apskatītas vairākas lietas, kas ir nepieciešamas rīku izveidei pēc vienotiem principiem, īpašu akcentu liekot uz DSL izmantošanu un DSL implementēšanu.

2 Informācijas sistēmu integrācija un pārvaldība

Dotajā nodaļā aplūkosim konkrētu rīku, kas domāts informācijas apstrādei neviendabīgā, sadalītā un mainīgā vidē. Autors ir piedalījies komunikāciju servera jēdziena un tā implementēšanai nepieciešamās teorētiskās bāzes izveidošanā. Komunikāciju serveri var uzskatīt par augsta līmeņa rīku, kuru pašu veido vai var veidot daudzi specializētāki rīki.

2.1 Komunikāciju serveris

Ar komunikāciju serveri (turpmāk tekstā - KS) tiek saprasta programmatūra un datortehnika, kas plašam lietotāju lokam nodrošina iespēju saņemt informāciju no vairākiem avotiem (valsts reģistriem, datubāzēm, informatīvām sistēmām), izmantojot vienu kontaktpunktu. KS veic lietotāju identifikāciju, datu izmantošanas autorizāciju, pieprasījumu, kas prasa vēršanos pie vairākiem informācijas avotiem, izpildi, kā arī šo pieprasījumu izpildes izmaksu novērtējumu finansu norēķiniem. KS lietotājiem nodrošina, pirmkārt, iespēju iepazīties ar to, kur un kāda informācija glabājas un, otrkārt, pieprasīt un saņemt informāciju no dažādie reģistriem, neiedziļinoties tās glabāšanas tehniskajos jautājumos [ABK99].

Nepieciešamība pēc KS izveides ir izvirzījusies nacionālās programmas "Informātika" [MoT98a, MoT98b], kā arī divu lielu projektu: Baltijas valstu valdību datu pārraides tīkla (turpmāk tekstā - Tīkls) [R98a, R98b] un Integrētās valsts nozīmes informācijas sistēmas (turpmāk tekstā - Megasistēma) [R98c] projektu izstrādes laikā.

Tīkla izveides mērķis ir kardināli uzlabot telekomunikāciju un datu apmaiņas iespējas starp Baltijas valstu administratīvajām institūcijām. Piemēram, vienas valsts dienestiem, lai iegūtu informāciju par citā valstī reģistrētiem objektiem (uzņēmumiem, personām, transporta līdzekļiem). Ir lietderīgi, neiedziļinoties citas valsts datubāzu struktūrās, iegūt nepieciešamās ziņas no viena informācijas avota. KS pielietošana, kā to ir pierādījusi Integrētās valsts nozīmes informācijas sistēmas projekta izstrāde, ir aktuāla arī vienas valsts ietvaros kā universāls līdzeklis informācijas apmaiņai starp dažādiem valsts nozīmes reģistriem vai informatīvām sistēmām.

Projektā Megasistēma kā pirmie vienotā tīklā informācijas apmaiņai tika fiksēti sekojoši LR reģistri: Uzņēmumu reģistrs, Transporta līdzekļu reģistrs, Meklējamo transporta līdzekļu reģistrs, Iedzīvotāju reģistrs, Meklējamo personu reģistrs, Pazaudēto personas dokumentu reģistrs, Izglītības dokumentu bāze, Vīzu datubāze, Valsts statistikas informatīvā sistēma, Konsulārās informācijas sistēma, Veselības aizsardzības informācijas sistēma, Informācijas sistēma par narkotikām. Sadarbībai ar ārvalstīm pirmām kārtām tika izvēlēti: Eiropas Biznesa Reģistrs (EBR), Eiropas Transporta līdzekļu reģistrs (EuCaris) un Interpol datubāzes.

Iespēju robežās piekļuvei pie reģistriem, datubāzēm vai informatīvajām sistēmām jāizveido vienots kontaktpunkts, jo lietotājs nav informēts par dažādajiem informācijas avotiem, informācijas glabāšanas tehniskajām detaļām, saistību starp dažādiem informācijas avotiem, iespējām dabūt integrētu informāciju no vairākiem datu avotiem. Liela uzmanība jāpievērš lietotāja interfeisam, jo pieprasījumus sistēmai veiks ierindas datorsistēmu lietotāji ar dažādu sagatavotību IT jomā, kā arī lietotāju autorizācijai un lietotāja tiesību pārvaldībai, atbilstoši likumdošanas prasībām.

Sākotnēji tika izdalītas 5 galvenās Komunikāciju servera funkcijas:

1. lietotāja identifikācija;
2. informācijas izmantošanas pilnvarošana un sankcionētā piekļuve;
3. lietotāju tiesību vadība;
4. pieprasījumu, kas prasa vēršanos pie vairākiem informācijas avotiem, izpilde;
5. pieprasījumu izmaksu novērtējums finansu norēķiniem.

Komunikāciju serverī būtiska ir lietotāja tiesību pārvaldība. Piemēram, skatīšanās režīmā lietotāju tiesības var sadalīt vairākās daļās:

1. tiesības iegūt informāciju par to, kas glabājas konkrētā reģistrā (šī ir publiski pieejama informācija);
2. tiesības iegūt informāciju no viena reģistra vienas tabulas informāciju par vienu ierakstu pēc tā unikālā identifikatora;
3. tiesības iegūt ierakstu kopu (sarakstu) no viena reģistra vienas tabulas, kas atlasīti pēc noteikta kritērija;
4. tiesības iegūt ierakstu kopu no viena reģistra vairākām tabulām, kas atlasīti pēc noteikta kritērija;
5. tiesības iegūt in formāciju vai eksistē saiti starp viena reģistra vairāku tabulu ierakstiem (starp konkrētām instancēm);
6. tiesības iegūt informāciju, atlasītu pēc kāda kritērija, no viena reģistra vairākām tabulām, kas saistītas ar noteiktu relāciju;
7. tiesības iegūt informāciju no vairākiem reģistriem par vienu objektu pēc tā primārās atslēgas;
8. tiesības iegūt informāciju par saistības pastāvēšanu starp vairāku reģistru objektiem;
9. tiesības iegūt ierakstu kopu, kas atlasīta pēc lietotāja dotiem kritērijiem no vairāku reģistru vairākām tabulām, kas savā starpā ir saistītas.

Arī atbildes sniegšana ir diferencējama 4 līmeņos:

1. iegūt atbildi, vai ir atrasta meklētā informācija, vai nav;
2. iegūt atbildi, cik ierakstu ir atrasts;
3. iegūt objektu primārās atslēgas;
4. iegūt meklēto ierakstu.

Katrs no šiem līmeņiem dod dažādu informācijas daudzumu, pie tam ir gadījumi, kur šis lēciens starp blakus esošiem līmeņiem ir kvantitatīvs, un ir gadījumi, kur šis lēciens ir kvalitatīvs. Šeit kā piemēru varētu minēt sekojošus pieprasījumus:

- "Vai personai X pieder kāda automašīna?"
- "Cik automašīnas pieder personai X?"
- "Kāda(s) automašīna(s) pieder personai X?"
- "Vai personai X pieder automašīna Y?"

KS jābūt gatavam akceptēt dažādā veidā un formā izteiktus informācijas pieprasījumus. Kā galvenais KS darbības režīms tiek uzskatīts tiešsaistes pieslēgums, taču nedrīkstētu aizmirst arī par citiem pieprasījuma nodošanas veidiem – pieprasījums caur elektronisko pastu, pieprasījums uz kāda elektroniskās informācijas nesēja (piemēram, diskete), pieprasījums dokumenta veidā uz papīra, vai pat mutisks pieprasījums. Atbildes uz pieprasījumu var tikt dotas tādos pašos veidos kā pieprasījums, un lietotājs pats var izvēlēties veidu, kādā saņemt atbildi.

Pieprasījumu un atbildes forma var būt visai daudzveidīga. Populārākā sadarbības forma sākotnēji varētu būt pārlūkprogrammas lapa gan pieprasījumam, gan atbildei. Šai sadarbības formai ir iespējama visa daudzveidība, ko pašlaik jau nodrošina tīmekļa lietojumprogrammas. Tāpat populāra varētu būt speciālu procedūru (funkciju) lietošana, kur pati procedūra no parametru vērtībām nosaka pieprasījumu formu un atbildes formu uz vēlamo pieprasījumu.

Sadarbībai ar KS vēl varētu lietot sekojošas formas:

1. speciālas lietojumprogrammas, kas spēj strādāt ar KS;
2. aktīvie objekti, kas spēj strādāt ar KS un ir izmantojami klienta lietojumprogrammās;
3. fails ar noteiktā formātā pierakstītu pieprasījumu vai atbildes fails kādā formātā;
4. failu kopums (ieskaitot pat datubāzes) gan pieprasījumam, gan atbildēm;
5. papīra dokumenti norunātā formātā gan pieprasījumiem, gan atbildēm;
6. elektroniskais pasts, kas uzskatāms tikai par kāda iepriekš minēto veidu (3, 4, 5) modifikāciju.

Tiek uzskatīts, ka pieprasījumi no lietotāja var nākt gan dialoga režīmā no lietotāja-cilvēka, gan automātiskā režīmā, kur lietotājs-mašīna ir kāda lietojumprogramma uz lietotāja datora.

Bez tam jāparedz darbs gan sinhronajā režīmā (pieprasījums – gaidīšana – atbildes saņemšana), gan asinhronajā režīmā (pieprasījums– pieprasījuma apstrāde kaut kādā laika intervālā – ziņojums lietotājam par rezultāta pieejamību – rezultāta paņemšana), kas nodrošinātu efektīvāku gan lietotāja, gan KS darbu, it sevišķi lielu un sarežģītu pieprasījumu apstrādei.

Sadarbībā ar lietotāju jāņem vērā arī tādi aspekti, kā dažādais lietotāju sagatavotības līmenis, sazināšanās valoda, tekstu kodēšanas formāti, lietotāja datortehnikas, operāciju sistēmas un lietojumprogrammu iespējas, un ierobežojumi.

Tātad, galvenais uzdevums un tajā pašā laikā arī galvenā problēma, kuru jāatrisina KS, ir - kā apmierināt daudzos dažādos pieprasījumus, sniedzot apstrādātu informāciju no dažādiem, iespējams nesavietojamiem informācijas avotiem, un atgriezt rezultātu lietotājam vēlamā veidā un formā. Pašlaik viena no populārākajām sadarbības formām ir tīmekļa lietojumprogramma gan pieprasījumu uzdošanai, gan atbildes saņemšanai.

Pieaugot informācijas avotu skaitam, kas ir pieejami caur KS, ātri var nonākt pie informācijas pārbagātības, kurā nespēs orientēties pat KS administratori. Ir nepieciešams visus informācijas avotus un tajos ietverto informāciju klasificēt, ņemot vērā, ka informācijas avoti mainās.

KS iekšienē ir jābūt datu avotu repozitorijam, kurā formālā veidā ir aprakstīti avoti, to īpašības, tajos ietvertie dati un to īpašības. Repozitorijam ir jābūt elastīgam, viegli un ātri maināmam, lai izsekotu līdzi apkārtējās vides izmaiņām. Lai adekvāti reaģētu uz lietotāja pieprasījumiem, citām KS sastāvdaļām jāprot pašām dinamiski pieskaņoties izmaiņām šajā repozitorijā.

Arī lietotājam ir jāzina, kur un ko viņš var saņemt savu lietotāja tiesību ietvaros. Līdz ar to KS ir jāsniedz arī informācija par informāciju. Lietotājam saprotamā veidā un terminos ir jāapraksta potenciāli iegūstamā informācija un veidi, kā viņš to var pieprasīt. Bez tam, jācenšas daudzus pieprasījuma formulēšanas mehānismus cieši saistīt ar repozitoriju, tādējādi atvieglojot darbu lietotājam, kas reti lieto KS pakalpojumus.

KS jāspēj apmierināt pieprasījumus no daudziem datu avotiem. Līdz ar to repozitorijā ir jāapraksta arī avotu savstarpējā saistība un likumi, kā risināt dažādas nesaskaņas starp avotiem, piemēram, kā veikt datu konvertēšanu. Repozitorijam ir jābūt tādam, lai visus avotus KS varētu uzskatīt par vienu lielu datubāzi.

Finansu norēķinu sistēmai vajadzētu būt ieplānotai KS pieprasījumu izpildes mehānismā. Izpildot konkrētu pieprasījumu, tiek ne tikai izpildīts šis pieprasījums, bet arī automātiski aprēķināti patērētie resursi. Par resursu tiek uzskatīta informācijas pieprasījuma izpilde kādam reģistram. Piemēram, mērāmie lielumi ir izmantotais laiks, telpa, intelektuālā sarežģītība, īpašumtiesību un pakalpojuma kompensācija.

KS iekšienē katram resursam ir piešķirta cena, kas var dinamiski mainīties atkarībā no pieprasītā informācijas daudzuma, pieprasījuma laika, pieprasījuma un atbildes formas, pieprasījuma sarežģītības, reģistra, utt. Katra pieprasījuma cena tiek automātiski izrēķināta un saglabāta žurnālā, kas kalpo par pamatu finansu norēķiniem.

Līdzīgas problēmas, integrējot daudzus datu avotus un veidojot speciālas integrējošās sistēmas, ir risinātas arī [TAB+97, HMN+99, HGI+95, Sin98, Rob97].

2.2 Darba plūsmas procesu pārvaldība

Mūsdienās jaunas tehnoloģijas ielaužas arī valsts sektorā, ļaujot runāt par e-Valdību. Procesi savukārt ir svarīgākā e-Valdības komponente [Kar01]. Mēs esam konstatējuši, ka praktiski nav automatizētu procesu valsts institūcijās, kas organizē eksistējošo sistēmu sadarbību starp dažādām institūcijām un organizācijām. Dokumentu plūsma ir papīrbāzēta ar manuālo apstrādi vai caur elektronisko pastu. Vajadzētu ieviest automatizēto darba plūsmas pārvaldību, lai vismaz nodrošinātu ātrāku dokumentu apmaiņu un uzlabotu iedzīvotājiem sniegto servisu kvalitāti.

Ar darba plūsmas procesu pārvaldību parasti saprot "biznesa procesu pilnīgu vai daļēju automatizāciju, kuras darbības laikā dokumenti, informācija vai uzdevumi tiek pārsūtīti no viena dalībnieka citam, lai veiktu nepieciešamo darbību saskaņā ar procedurālajiem likumiem" [Fis00].

Ar darba plūsmas procesu pārvaldības sistēmu parasti saprot "sistēmu, kas definē, izveido un pārvalda darbu izpildi izmantojot programmatūru, kas darbojas uz vienas vai vairākām aparatūras vienībām, kas spēj interpretēt procesa definīciju, sazināties ar darba plūsmas dalībniekiem, un kas nepieciešamības gadījumā iesaista izpildē IT rīkus un lietojumprogrammas" [Fis00].

Pēdējos gados daudzi pētnieki un izstrādātāji velta savu uzmanību problēmai – kā organizēt sadarbību starp pastāvošajām sistēmām. Viens no populārākajiem

veidiem ir izmantot darbu plūsmas pārvaldību [Waria, Wfmc]. Tā rezultātā ir izstrādātas darbu plūsmas definēšanas valodas, kas var tikt uzskatītas par vienu no domēnspecifisko valodu apakšklasēm.

Darba plūsmas pārvaldības implementācijas līdzīgi kā programmēšanas valodas parasti balstās uz vienu fiksētu semantiku. Taču mēs esam ieinteresēti vairākās semantikās vienai uz tai pašai darbu plūsmai, piemēram, tradicionālā darba plūsmas pārvaldības semantika, statistisko datu uzskaites semantika, semantika plūsmas atkļūdošanai un simulācijai. Bez tam mums ir nepieciešams ne tikai aprakstošās valodas kompilators vai interpretators, bet arī citi atbalstoši rīki, sakarā ar arvien pieaugošajām prasībām pēc augstas kvalitātes programmatūras. Interesants ir arī jautājums par semantikas nomaiņu izpildes laikā kādām noteiktām plūsmā iesaistītajām un aktīvajām instancēm.

Mēs piedāvājam semantiku saistīt ar sintaktiskajiem elementiem, izmantojot semantiskos konektorus, kas dabīgā veidā sasaista eksistējošās sistēmas vienotā darba plūsmā, kā arī atļauj definēt un izpildīt dažādas semantikas vienlaicīgi. Reāli semantikas implementācija ir programma, ko var uzskatīt par rīku. Tālāk darbā tiks dots semantikas pieraksta princips, semantikas apraksta piemēri vienkāršai programmēšanas valodai un arī darba plūsmas definēšanas valodai. Piedāvātais princips ir noderīgs ne vien tradicionālām programmēšanas valodām, bet arī lielai domēnspecifisko valodu saimei.

2.3 Datu avoti un to izmantošana

Gan valsts, gan biznesa organizācijām ir nepieciešams pārvaldīt lielu informācijas daudzumu, kas tiek glabāts kādā formātā datubāzē vai failā. Viena no galvenajām problēmām informācijas pārvaldībā ir vājā saistība starp daudzajām datubāzēm un informatīvajām sistēmām, kā arī grūtības organizēt sadarbību starp dažādu organizāciju informatīvajām sistēmām.

Mūsdienās liela daļa jauno informācijas sistēmu vai servisu organizē komunicēšanos (saskarni) starp datu avotu un datu patērētāju (lietotāju), taču tajā pašā laikā datu avoti paliek tie paši vecie, kas pieder eksistējošai vai agrāk lietotai informācijas sistēmai. Rodas darbošanās ar ļoti neviendabīgiem datiem. Lai tiktu galā ar šīm problēmām, heterogēno informatīvo modeļu aprakstīšanā tiek izmantoti datu avotu meta-dati (datu struktūra, saturs, atribūti, utt.). Šī pieeja atbalsta dinamisku sistēmu izveidi, kā arī vieglāku un elastīgāku to uzturēšanu strauji mainīgajā pasaulē.

2.3.1 Datu avotu raksturojums

Datu avots ir informācijas resurspunkts, no kurienes mums ir iespējams iegūt kādu vēlamo informāciju. Lai varētu veidot kvalitatīvākus informācijas pārvaldības rīkus, ir jāaplūko problēmas, kuras rodas, darbojoties ar datu avotiem heterogēnajā informatīvajā vidē:

- **Dažāda juridiskā piederība.** Par datu avotu var kļūt jebkuras organizācijas informatīvā sistēma vai datubāze. Analoģiski var apskatīt juridisko piederību vienas organizācijas ietvaros starp tās struktūrvienībām. Tas nozīmē, ka datu avota attīstību un tā izmantošanas politiku nosaka ārēji spēki, kas var nepakļauties datu integrēšanas kopīgajai politikai.

- **Liels skaits.** Laika gaitā informācijas avotu skaits var pārsniegt vairākus desmitus vai pat simtus (tas iespējams arī vienas organizācijas ietvaros). Piemēram, Latvijā vien jau ir apzināti vairāki desmiti valsts reģistru un to skaits turpina pieaugt. Bez tam integrēšanās iespējama ne tikai organizācijas vai vienas valsts līmenī, bet arī starpvalstu vai arī starporganizāciju līmenī, kas teorētiski nozīmē gandrīz neierobežotu skaita pieaugumu.
- **Orientācija uz pamatmisijas izpildi.** Katrs informācijas avots sākotnēji ir izstrādāts, lai nodrošinātu konkrētas specifiskas funkcijas, kas jāveic savā organizācijā. Informatīvās sistēmas un datubāzes, kas tiek izmantotas organizācijā, ir izvēlētas, izstrādātas un optimizētas tieši šīs konkrētās organizācijas vajadzībām. IS var nebūt orientētas uz iespēju kādam citam saņemt informāciju, bet, ja tāda iespēja paredzēta, tad tā var būt ļoti specifiska un ar daudziem ierobežojumiem. Tas nozīmē, ka informācijas integrēšanas līdzekļiem pamatā jāpielāgojas datu avotiem, nevis otrādi.
- **Avota nozīmība un lielums.** Datu avoti var atšķirties pēc savas nozīmības un lieluma. Jo nozīmīgāka ir datubāze, jo lielākai ir jābūt sadarbībai ar to. Arī datubāzu lielums jāņem vērā, jo no tā ir atkarīgi datu apstrādes mehānismi.
- **Avotu kvalitāte un stabilitāte.** Informatīvās sistēmas var būt izstrādātas ar visdažādākajām tehnoloģijām un dažādos laikos. Atkarībā no ieguldītajiem resursiem tās var būt vairāk vai mazāk kvalitatīvas. Protams, ka ar modernu tehnoloģiju izstrādātu augstas kvalitātes IS un datubāzi saprasties būs vienkāršāk un stabilāk, nekā ar vecu un mazkvalitatīvu sistēmu. Tātad jābūt gatavam uz to, ka datu avoti var kļūt nestabili, kļūdaini strādājoši, vai reizēm pat vispār būt nepieejami.
- **Heterogēna izstrādes un ekspluatācijas vide.** IS programmatūrai var būt izmantota dažāda izstrādes vide, dažādas datubāzes, to darbināšanai lietotas dažādas operāciju sistēmas un datortehnika. Bez tam arī komunikāciju tīkli, piekļuve tiem un sadarbošanās protokoli var atšķirties.
- **Dažādi tehniskie datu formāti.** Informācija datu avotos var glabāties visdažādākajos tehniskajos formātos, ar to saprotot relāciju datubāzes, objektorientētās datubāzes, statiskas WEB lapas, dinamiskas WEB lapas, kas ģenerētas, izmantojot kādu iekšēju formātu, faili ar visdažādāko iekšējo struktūru.
- **Dažādi loģiskie datu formāti.** Konkrēta informācijas vienība vai loģiska informācijas vienību grupa var glabāties dažādos formātos, dublēties, var tikt dažādi kodēta, daļa no informācijas var iztrūkt (noklusētā vērtība dotajā datu avotā).
- **Informācijas pretrunīgums.** Informācija var būt pretrunīga gan viena datu avota ietvaros, gan starp dažādiem avotiem.
- **Avotu mainība un attīstība.** Katrs datu avots dzīvo samērā patstāvīgu un neatkarīgu dzīvi. Tas pilnveidojas, mainās, var likvidēties, var dzimt no jauna, var uz laiku pārtraukt savu darbību vai sadarbību ar citiem.

Pieminētie aspekti demonstrē datu avotu atšķirības problemātikas nopietnību. Tātad vide patiešām ir ļoti heterogēna un mainīga. Sadarbību ar tādiem datu avotiem var nodrošināt tikai dinamiska, pašadaptējoša programmatūra, kas vadās no datu avotu aprakstiem (datu avotu repozitorijs un meta modeļi) un konkrētās situācijas.

2.3.2 Informācija par informāciju

Datu avotus un tajos ietverto informāciju nepieciešams klasificēt, ņemot vērā, ka informācijas avoti nemitīgi mainās. Visiem avotiem, to īpašībām, tajos ietvertajiem datiem jābūt aprakstītiem un informācija jātur speciālā repozitorijā. Tas ir samērā tipisks risinājums šādās situācijās, kad ir jādarbojas dinamiskā, ātri mainīgā vidē, lai nepārtraucot sistēmas darbu, būtu iespējams mainīt tās darbību. Savukārt programmatūrai vai rīkiem, kuriem jānodrošina sistēmas darbs, jāprot adekvāti reaģēt uz notikušajām izmaiņām un nepieciešamības gadījumā pat pieskaņoties jaunajām prasībām automātiski.

Repozitorijam ir liela nozīme ne tikai pašas sistēmas darbā, bet arī sniedzot lietotājam informāciju par datu avotos pieejamo informāciju. No repozitorija lietotājam saprotamā veidā un terminos ir jāapraksta potenciāli iegūstamā informācija un veidi, kā viņš to var pieprasīt. Bez tam vēlama iespēja no informācijas par informāciju skatīšanās uzreiz pāriet uz informācijas pieprasīšanu, kas atvieglotu darbu retiem sistēmas lietotājiem.

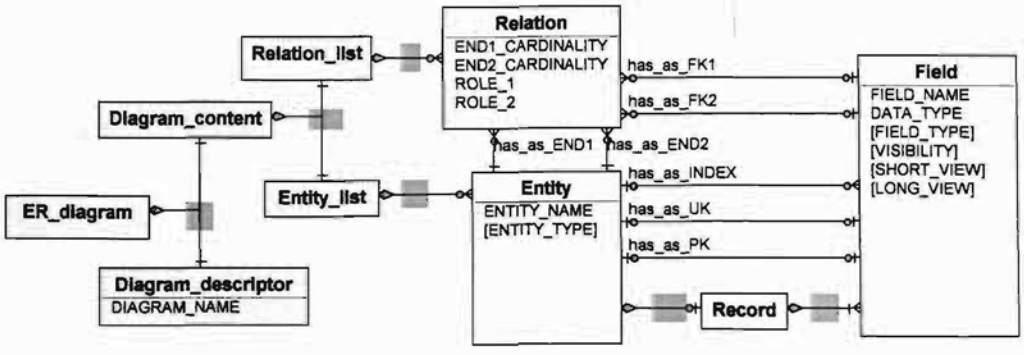
Repozitorijā jātur arī informācija par attiecībām starp daudzajiem datu avotiem un informācijas vienībām, kas tajās atrodamas. Citādi nav iespējama informācijas integrēšana un lietotājam pašam ir jāpārvietojas pa visiem datu avotiem, lai sameklētu nepieciešamo informāciju (piemēram, meklējot visu informāciju par personu X). Papildus jātur arī likumu bāze, kas nosaka, kā risināt atrastās pretrunas, kā integrēt informāciju, kā veikt datu transformāciju uz citiem formātiem, utt. Repozitorijam jānodrošina tāds serviss, ka uz visiem datu avotiem var skatīties kā uz vienu lielu datubāzi.

Datu avoti ir jāapraksta formāli, lai ar to varētu darboties rīki, kas nodrošina lietojumprogrammas ģenerāciju. Mēs galvenokārt apskatam divas iespējas datu avotu aprakstīšanai:

1. fiziskais datu modelis, kas precīzi apraksta konkrētu datubāzi,
2. loģiskais datu modelis, kas apraksta iedomātu virtuālu datubāzi.

2.3.3 Datu avotu aprakstīšana ar fizisko datu modeli

Fiziskā datu modeļa, t.i. reālas eksistējošas datubāzes aprakstīšanai mēs izmantojam ER diagrammas. ER diagrammu paveidi un to papildus modifikācijas var būt dažādas. Kā piemēru glabāšanai repozitorijā mēs piedāvājam ER diagrammu, ko apraksta Zīm. 3 attēlotais objektu modelis.



Zīm. 3 ER diagrammas konceptuālais objektu modelis

Kvadrātiekavās ietvertie atribūti ir ieviesti, lai ģenerētu labākas lietojumprogrammas. Vienkāršības pēc tiek pieņemts, ka primārās atslēgas un ārējās atslēgas tiek veidotas tikai no viena lauka. Vienkāršotajam modelim tiek pieņemts, ka indeksi arī tiek veidoti tikai no viena lauka bez jebkādu funkciju izmantošanas. Nav grūti šo vienkāršoto modeli papildināt, lai varētu izmantot kombinācijas no vairākiem laukiem un pielietot funkcijas gan primāro vai ārējo atslēgu veidošanai, gan arī indeksu veidošanai.

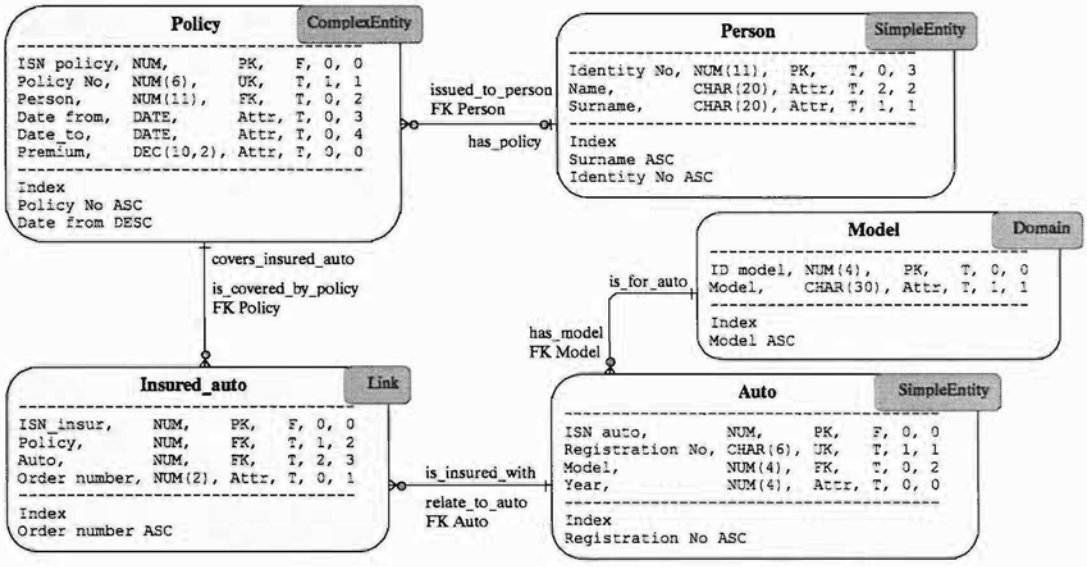
Tiek ieviesti vairāki jauni atribūti, kas nav tradicionālā ER diagrammā. Šie atribūti nodrošina lietojumprogrammu ģeneratoru ar papildu informāciju, lai uzģenerētu daudz ērtāku un pielietojamāku lietojumprogrammu.

- **Entity type** ir entītijas atribūts, kas nosaka tā tipu un ļauj ģenerēt lietojumprogrammas ekrānus ar specifisku informācijas izkārtojumu ar atbilstošiem datu un ekrāna kontroles līdzekļiem. Šis atribūts tiek noteikts automātiski un ir atkarīgs no entītiju savstarpējām attiecībām. Lietotājs automātiski noteikto tipu var koriģēt.
- **Field type** ir automātiski izrēķināms tabulas lauka atribūts. Ja lauks ir definēts kā primārā atslēga entītijai *Entity* ar relāciju *has as PK* (Zīm. 3), tad laukam ir tips ***PK***. Līdzīgi mēs definējam tipu ***UK*** (ar relāciju *has as UK*) unikālai atslēgai un tipu ***FK*** (ar relāciju *has as FK1* vai *has as FK2*) ārējai atslēgai. Pārējos gadījumos tips ir ***Attribute***.
- **Visibility** ir tabulas lauka īpašība, kas nosaka, vai informācija, kas asociējas ar šo lauku tiek vai netiek attēlota lietotājam. Laukiem ar tipu *UK*, *FK* un *Attribute* noklusētā vērtība atribūtam *visibility* ir TRUE (attēlot), bet tipam *PK* - FALSE (neattēlot).
- **ShortView** un **LongView** ir lauka atribūti, kas definē, kā entītijas ieraksts labākajā veidā tiek attēlots uz ekrāna. Piemēram, tiek uzrādīti rādāmie atribūti un to secība, t.i. ierakstam 0, ja atribūts nav jārāda, vai ierakstam kārtas numuru starp attēlojamiem entītijas atribūtiem.

Entity type ir svarīgs jēdziens mūsu lietojumprogrammas ģenerācijas ideoloģijai. Mēs definējam sekojošus entītijas tipus:

- **Domain** – saraksts ar standarta datu elementiem, kas ir atļautās atribūta vērtības vai objekta īpašība.
- **SimpleEntity** – vienkāršs objekts, kas tiek definēts ar atribūtu kopumu vai standarta datu elementiem.
- **ComplexEntity** – komplekss objekts, kas ir līdzīgs SimpleEntity, bet tas iekļauj sevī citus vienkāršus vai kompleksus objektus.
- **Link** – loģiska relācija starp vismaz diviem vienkāršiem vai kompleksiem objektiem.

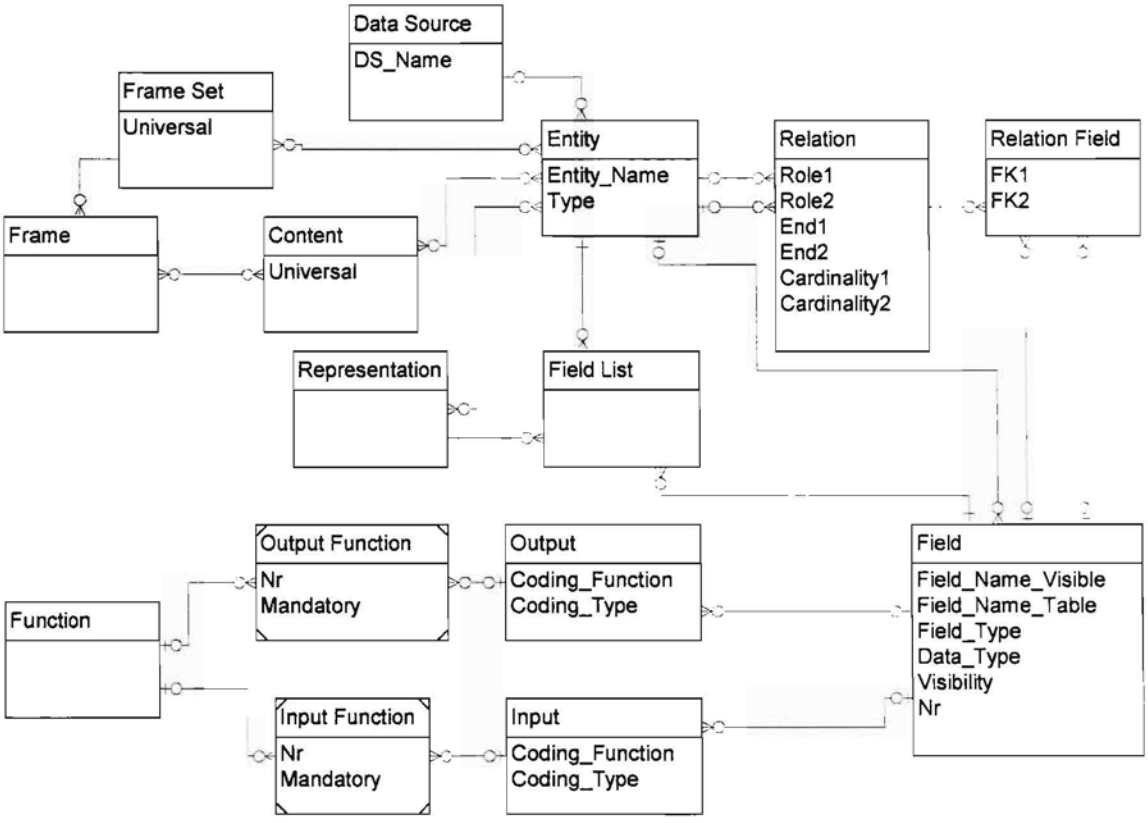
ER diagrammas piemērs ir dots Zīm. 4. Katra lauka atribūti ir doti sekojošā secībā – *field name*, *data type*, *field type*, *visibility* (T priekš TRUE, F priekš FALSE), *ShortView*, *LongView*.



Zīm. 4 Fiziskā datu modeļa piemērs

2.3.4 Datu avotu aprakstīšana ar loģisko datu modeli

Loģiskā datu modeļa, t.i. reālas eksistējošas datubāzes virtuāla skata aprakstīšanai mēs par pamatu arī izmantojam ER diagrammas. Papildus nepieciešams aprakstīt arī veidu, kā loģiskais skats uz datubāzi tiek saistīts ar fizisko modeli, t.i. datubāzes tabulām un relācijām starp tām. Visa šī informācija tiek glabāta repozitorijā, ko apraksta Zīm. 5 attēlotais objektu modelis.

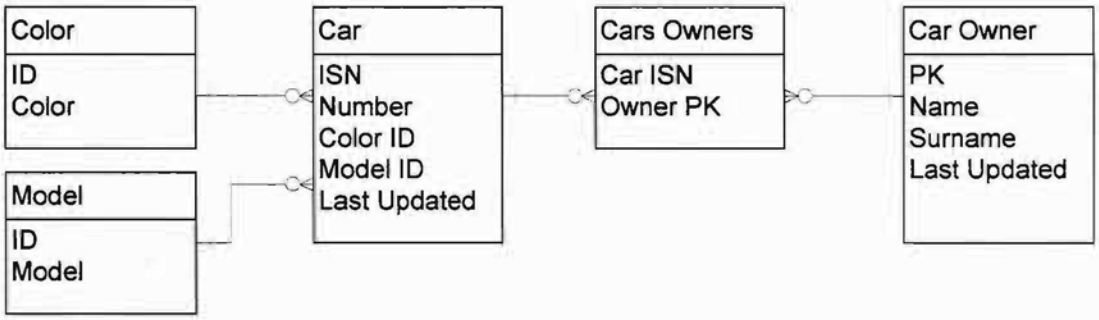


Zīm. 5 Repozitorija meta-modelis

Repozitorijs satur sekojošas galvenās daļas:

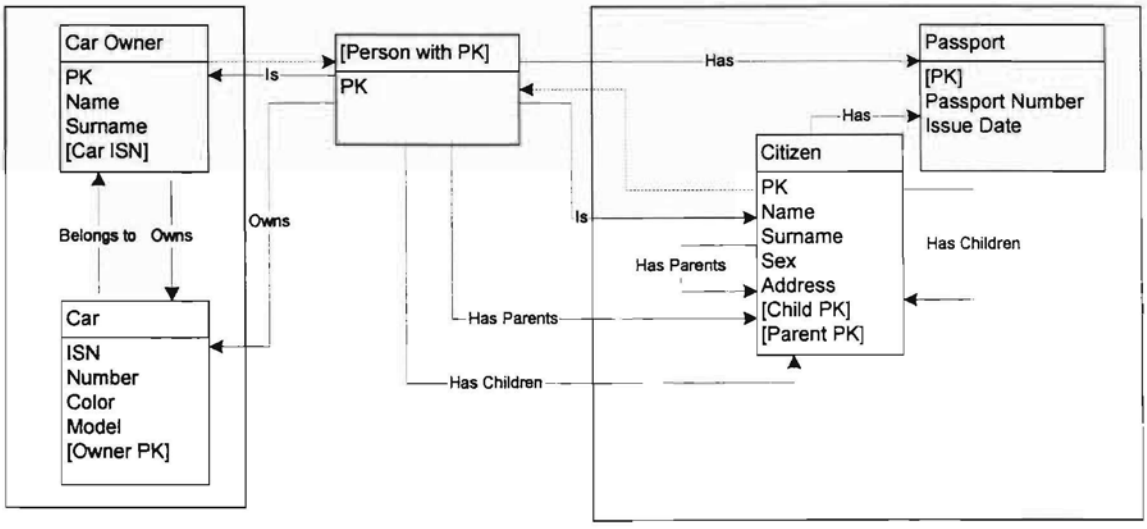
- Entītijas *Data Source*, *Entity*, *Field*, *Relation*, *Relation Field* satur datu avotu modeļus un informāciju par entītijām un relācijām.
- Entītijas *Function*, *Input*, *Input Function*, *Output*, *Output Function* satur informāciju par funkcijām, kas iegūst informāciju no datu avotiem un apraksta šo funkciju ieejas parametrus un rezultātu.
- Entītijas *Representation* un *Field List* satur informāciju par katras entītijas vizuālo reprezentāciju, t.i. kādi lauki un kādā secībā tiek attēloti. Piemēram, entītijai *Persona*, kas satur informāciju par personu, īsajā skatā ir redzami atribūti *PK*, *Vārds*, *Uzvārds*, bet garajā - *PK*, *Vārds*, *Uzvārds* un *Adrese*.
- Entītijas *Frame Set*, *Frame* un *Content* satur informāciju par visa ekrāna vizuālo noformējumu.

Lietojot loģisko datu modeli, mēs paši izvēlamies kādas entītijas un relācijas dotajam datu avotam mums ir svarīgas (loģiskā entītija var būt vairāku fiziskās datubāzes tabulu apvienojums). Entītijas un relācijas parasti tiek veidotas tuvu reālajai dzīvei. Vienam datu avotam var tikt saveidoti daudzi dažādi loģiskie datu modeļi atkarībā no lietotāja vēlmēm. Tādējādi tiek apslēptas visas datu avota tehniskās detaļas. Piemēram, fiziskajam modelim, kas attēlots Zīm. 6 , atbilst tikai divas entītijas *Car Owner* un *Car* loģiskajā modelī, kas attēlots Zīm. 7.



Zīm. 6 Fiziskais datu modelis loģiskā datu modeļa piemēram

Starp loģiskajiem datu modeļiem mēs varam definēt saites, tādējādi sasaistot kopā vairākus datu avotus. Lai saistītu kopā dažādus datu avotus, tiek izmantotas entītijas ar citu tipu. Šīs entītijas tiek lietotas kā pamatklases datu avotu entītijām un tās nepieder ne pie viena konkrēta datu avota. Piemēram, entītija *Person with PK* ir šāda pamatklase (Zīm. 7). Šī pamatklase satur tikai vienu lauku *PK* (Personas kods). Šis lauks ir primārā atslēga daudziem datu avotiem, kuriem ir entītija ar intuitīvi nojaušamo nozīmi *persona*. Ja jūs zināt personas kodu, tad jūs varat iegūt arī attiecīgo personas informāciju no atbilstošajiem datu avotiem. Piemēram, *Person with PK* var saistīt kopā informāciju starp entītijām *Citizen*, *Passport*, *Car*, *Car Owner* (Zīm. 7).



Zīm. 7 Piemērs datu avota loģiskajam datu modelim

Kvadrātiekavās ieliktie lauki ir neredzami, jo tiek izmantoti tikai meklēšanai. Nepārtrauktās līnijas ar bultām nozīmē sekojošo: ja jūs zināt informāciju no entītijas, kas ir bultas sākuma punktā, tad jūs varat iegūt saistīto informāciju no entītijas bultas otrajā galā. Pārtrauktā līnija ar bultu nosaka attiecību starp normālo entītiju un pamatklases entītiju.

2.3.5 Datu ieguve

Informācijas patērētājs ir jebkurš subjekts, kas vēlas un ir tiesīgs iegūt informāciju no integrētās informatīvās vides. Piemēram, patērētājs var būt gan fiziska persona, gan juridiska persona, gan automātisks process. Informācijas patērētāja tiesības nosaka

likumdošana, datu avota īpašnieks un uzturētājs, integrētās informatīvās vides uzturētājs.

Integrētās sistēmas galvenajam mērķim jābūt informācijas patērētāju vēlmju izpilde saskaņā ar piešķirtajām tiesībām. Līdz ar to jābūt gataviem saņemt informācijas pieprasījumus visdažādākajā veidā un formā, kā arī jāsniedz atbilde pieprasītājam ērtā veidā un formā.

Kā galvenais sadarbības veids mūsdienās būtu jāuzskata tiešsaistes režīms, piemēram, izdalītā līnija, iezvanpieeja, savienojums caur citiem informatīvajiem tīkliem (Internets, VNDPT vai citiem valsts, pasaules vai organizāciju tīkliem). Tajā pašā laikā nedrīkstētu aizmirst par citiem pieprasījuma veidiem – pieprasījums ar elektroniskā pasta palīdzību, pieprasījums uz kāda elektroniskā informācijas nesēja (piemēram, diskete), pieprasījums dokumenta veidā uz papīra, vai pat mutisks pieprasījums (klātienē vai pa telefonu). Arī atbilde var tikt sniegta visdažādākajos veidos.

Sadarbība jānodrošina gan sinhronajā režīmā (pieprasījums – gaidīšana – atbildes saņemšana), gan asinhronajā režīmā (pieprasījums – pieprasījuma apstrāde kādā laika intervālā – ziņojums lietotājam par rezultātu pieejamību vai lietotāja pieprasījums par pieprasījuma izpildi – rezultāta saņemšana). Nodrošinot abus sadarbības veidus, iespējams optimizēt sistēmu darbu, nodrošināt efektīgāku sadarbību it sevišķi lielu un sarežģītu pieprasījumu apstrādei.

Sadarbībā ar informācijas pieprasītāju jāņem vērā arī tādi aspekti, kā dažādais lietotāju sagatavotības līmenis, sazināšanās valoda, tekstu kodēšanas formāti, lietotāja datortehnikas, operāciju sistēmas un lietojumprogrammas iespējas un ierobežojumi.

No augstāk minētā arī izriet galvenais uzdevums integrējošai sistēmai (reizē arī galvenā problēma) – apmierināt daudzos pieprasījumus, sniedzot apstrādātu informāciju no dažādiem reizēm nesavietojamiem informācijas avotiem, un atgriezt rezultātu lietotājam vēlamā veidā un formā.

Ja mums ir dots fiziskais datu modelis, tad datu ieguve no datu avota tiek realizēta ar procedūru palīdzību, kas pēc dotā nosacījuma atgriež sarakstu ar atlasītajiem tabulas ierakstiem. Atlasīti tiek tikai tie lauki, kas nepieciešami rādīšanai. Šajā gadījumā rīkam pietiek izstrādāt pāris metodes, kas lasa no datu modeļa nepieciešamo informāciju un dinamiski ģenerē pieprasījumu datubāzei. Iespējams arī ģenerēt sākotnēji populārākos vai visus iespējamos pieprasījuma variantus, nokompilēt tos un saglabāt jau kā gatavus servisus.

Taču reālajā praksē daudz elastīgāks ir darbs ar loģiskajiem datu modeļiem. Šajā gadījumā tiek rakstītas speciālas datu ieguves funkcijas, kas nodrošina reālo datu paņemšanu no fiziskās datubāzes un šo datu translēšanu uz loģiskajā datu modelī noteikto formātu. Informācija par šīm funkcijām tiek glabāta repozitorijā, piemēram, funkcijas vārds, ieejas parametri ar pazīmi, vai tie ir obligāti, izejas parametri.

Eksperimentu rezultātā ar reāliem datu avotiem tika konstatēts, ka vienkāršākais veids ir funkcijas veidot tā, lai ieejas un izejas lauki ir tikai no vienas entītijas loģiskajā datu modelī (tai pašā laikā funkcija iespējams operē ar daudzām fiziskajām tabulām un relācijām). Reālajā dzīvē ar to parasti pietiek.

Mēs esam izdalījuši divus galvenos veidus, kā veidot datu modeļus un funkcijas darbam ar tiem.

1. Mums jau ir funkcijas, un mēs veidojam datu avota loģisko modeli atbilstoši šīm funkcijām.
2. Ja datu avots ir sistēma, kuru mēs uzturam un pārvaldām, tad iespējams izveidot funkcijas atbilstoši datu loģiskajam modelim. Tādā gadījumā labāk vispirms izveidot datu loģisko modeli un tikai pēc tam implementēt nepieciešamās funkcijas.

Ņemot vērā praktisko pieredzi, mēs iesakām veidot divu tipu funkcijas:

1. Funkcija dod objekta identifikatoru, kas identificē objektu datu avotā, pēc kāda meklēšanas kritērija. Piemēram, dod personas PK (personas kodu), padodot tās vārdu un uzvārdu. Atbilde parasti ir saraksts ar personu identifikatoriem (PK) saskaņā ar meklēšanas kritēriju.
2. Funkcija, saņemot datu objekta identifikatoru, dod informāciju par vienu objektu no datu avota. Piemēram, dod visu informāciju par personu, atbilstoši pieprasītajam personas kodam (PK).

Piemēram, mums varētu būt divas funkcijas entītijai *Iedzīvotājs*:

1) Ieejas dati: *Vārds*, *Uzvārds* (var būt daļēji uzrādīti). Izejas dati (saraksts): *PK*, *Vārds*, *Uzvārds* (pilns teksts)
2) Ieejas dati: *PK*. Izejas dati: *PK*, *Vārds*, *Uzvārds*, *Adrese*

Papildus ir arī divas funkcijas, lai iegūtu informāciju par iedzīvotāja vecākiem un bērniem:

3) Ieejas dati: *Vecāka_PK*. Izejas dati: *Bērna_PK*, *Vārds*, *Uzvārds*
4) Ieejas dati: *Bērna_PK*. Izejas dati: *Vecāka_PK*, *Vārds*, *Uzvārds*

3 Informācijas ieguve un attēlošana

Dotajā nodaļā detalizētāk ir apskatīta informācijas ieguve un attēlošana lietotājam. Īpašs akcents ir likts uz to, lai izveidotu vienkāršu mehānismu, kas apmierinātu gan dinamiskuma, gan programmatūras korektuma, gan arī lietotāja saskarnes prasības. Ideja balstās uz informācijas prezentēšanu dažāda tipa ekrānos, izmantojot noteiktus ekrāna objektus ar precīzi definētu atbilstību datu avotu aprakstošajam modelim. Piedāvātie principi ir izveidoti tā, lai varētu viegli izveidot dažādus rīkus, kas nodrošina darbu ar datu avotiem, to aprakstiem un veidotu lietotāja saskarni.

3.1 Pamatprincipi

Informāciju, ko attēlo vizuāli vai nosūta lietotājam kādā formātā, iegūst no datu avotiem un datu avotu aprakstiem, kas atrodas repozitorijā. Mēs pārsvarā orientējamies uz tādu rīku izveidi, kas gala rezultātā attēlo informāciju vizuāli. Informācijas vizualizācijai mēs apskatām divus veidus – tradicionālas lietojumprogrammas logs (piemēram, VisualBasic) vai pārlūkprogrammas lapa (piemēram, Internet Explorer). Tiek apskatīti arī abi gadījumi – gan, kad datu modelis apraksta fizisku datubāzi, gan arī loģisku datubāzi.

Pirmām kārtām mūsu mērķis ir piedāvāt lietotājam tehniku, kas nodrošina datubāzes pārlūka izveidi no fiziskā datu modeļa, kas ir aprakstīts ar ER diagrammu. Kas izstrādātājam jādara? Viņam ir jāizveido vienkārša ER diagramma eksistējošai vai plānojamajai datubāzei. Mēs pat nerūpējamies, vai tiek veikta rūpīga analīze un projektēšana, vai tiek izveidots daļējs pagaidu ER modelis. Izstrādātājs iegūst ātri ģenerētu datubāzes pārlūku, vienkāršu informācijas analīzes un filtrēšanas rīku, datu ievades un rediģēšanas rīku, prototipu daudz nopietnākai biznesa lietojumprogrammai, vienkāršu datubāzes testēšanas rīku. Tādējādi, kamēr tiek izstrādāta reālā sistēma, tiek iegūta robusta informatīvā sistēma.

Ģenerētās sistēmas kvalitāte un lietojamība galvenokārt ir atkarīga no ģenerēto ekrānu sistēmas. Ar terminu *ekrāns* šeit saprotam lietojumprogrammas logu, pārlūkprogrammas lapu vai pat pārlūkprogrammas lapas kadru (*frame*). Nodefinēsim dažus ekrānu standartus, kas nodrošina mums datu pārvaldību datubāzē.

Galvenais mūsu pieejas ģenerēšanas princips ir izveidot specifisku datu redaktoru vienai vai vairākām tabulām, kas saistītas ar relācijām. Mēs ģenerējam kopu ar saistītiem ekrāniem, kas nodrošina tiešu pāreju no viena ekrāna uz citu.

Mūsu sistēmas primārie objekti ir entītijas un relācijas starp tām. Mēs definējam dažus ekrāna tipus ar dažādu lietotāja saskarni un dažādu funkcionalitāti katrai entītijai vai relācijai. Piemēram, uz ekrāna mēs varam attēlot entītiju, saites uz citām entītijām (relācijas ER modelī), informāciju par saistītajām entītijām vai attēlot dažu relāciju saistītās entītijas. Ekrāni tiek ģenerēti katrai entītijai saskaņā ar tās tipu un katrai relācijai, ja ekrāna ģeneratora opcijas nenosaka citu uzvedību.

Izvēlne nodrošina piekļuvi jebkuram ģenerētajam ekrānam vai standartoperācijai, kas definēta katrai lietojumprogrammai. Ekrāni ir organizēti kādā vēlamā hierarhijā vieglākai orientācijai.

Katrs ekrāns sastāv no divām lielām kopām ar dažādiem ekrāna objektiem:

1. **Informatīvā grupa** – ekrāna objekti, kas attēlo informāciju no datubāzēm, un objekti, kas tiek ģenerēti no ER modeļa. Šī grupa pārsvarā sastāv no tabulas laukiem un relācijām starp entītijām. Svarīgākās informatīvās apakšgrupas ir:
    - **Field group** (ekrāna objekti, kas attēlo ieraksta redzamos laukus);
    - **Entity presentation group** (ekrāna objekti, kas attēlo vienu entītijas ierakstu vai sarakstu ar entītijas ierakstiem);
    - **Relationship presentation group** (ekrāna objekti, kas attēlo relācijas starp entītijām);
    - **Order button group** (radio pogu grupa, kas nosaka ierakstu attēlošanas secību).
2. **Vadības grupa** – ekrāna objekti, kas nodrošina papildus pārvaldību pār datiem, kas tiek glabāti datubāzē. To ģenerācija ir atkarīga no ekrāna tipa. Ir definētas sekojošas vadības apakšgrupas:
    - **Edit button group** (pogu grupa entītijas ieraksta rediģēšanai - *New*, *Edit*, *Save*, *Delete*, *Cancel* pogas);
    - **Locate button group** (pogu grupa vēlamā ieraksta atrašanai - *First*, *Next*, *Previous*, *Last*, *Find* pogas);
    - **Print button** (poga pašreizējā ieraksta vai ierakstu saraksta drukāšanai);
    - **OK button** (poga ekrāna atstāšanai);
    - **Control button group** (specifiskas pogas atkarībā no ekrāna tipa).

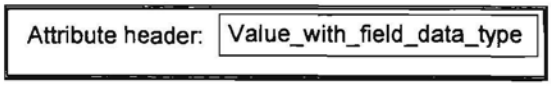
Mēs savā darbā apskatām tikai pašu ideju. Līdzīgi varētu ģenerēt HTML lapas objektus vai XML dokumenta fragmentus.

3.2 Ekrāna objektu attēlošana

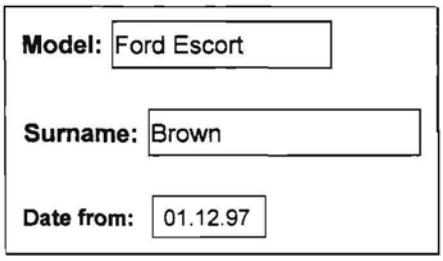
Turpmāk ir doti piemēri iespējamajai informācijas attēlošanai tradicionālās lietojumprogrammās. Piemēri WEB lietojumprogrammām tiks doti nākošajā nodaļā. Visi piemēri atbilst datu modelim, kas attēlots Zīm. 4.

3.2.1 Entītijas lauku attēlošana

Lauks ar tipu *Attribute* attēlojas uz ekrāna kā objektu grupa ***AttributeInfo*** (Zīm. 8). *Attribute header* ir tekstlodziņa tipa objekts, kura vērtība tiek ģenerēta no *FieldName*, un rediģēšanas lodziņa tipa objekts *Value_with_field_data_type* satur vērtību ar tipu, ko nosaka lauks *DataType*. Rediģējamā objekta *Value_with_field_data_type* garums ir atkarīgs no datu tipa, bet noteikti ir ierobežots ar kādu saprātīgu maksimālo garumu. Ja nepieciešams, šim objektam tiek nodrošināta ritināšanās iespēja. Piemēri iespējamajam izskatam doti Zīm. 9.

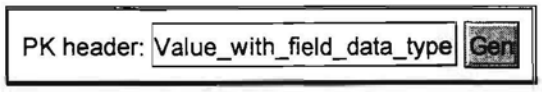


Zīm. 8 Ekrāna objektu grupa AttributeInfo

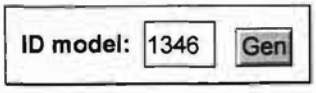


Zīm. 9 Piemēri ekrāna objektu grupai AttributeInfo

Lauks ar tipu *PK* (primārā atslēga) attēlojas par ekrāna objektu grupu ***PkInfo*** (Zīm. 10). Šī grupa ir līdzīga *AttributeInfo*, bet tas satur arī pogu ***Gen***. Lietotājs var manuāli ievadīt ieraksta primārās atslēgas vērtību vai uzģenerēt to automātiski, nospiežot pogu *Gen*. Atslēgas ģenerēšana ir atkarīga no izvēlētā ģenerēšanas likuma. Kad lietotājs atstāj objektu *Value_with_fiel_data_type*, tad sistēma kontrolē, vai vērtība ir unikāla. Piemērs iespējamajam izskatam dots Zīm. 11.

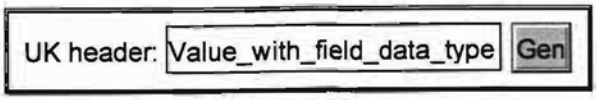


Zīm. 10 Ekrāna objektu grupa PkInfo

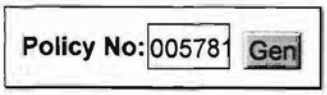


Zīm. 11 Piemērs ekrāna objektu grupai PkInfo

Lauks ar tipu *UK* (unikāla atslēga) attēlojas par ekrāna objektu grupu ***UkInfo*** (Zīm. 12). Šī grupa ir līdzīga ekrāna objektu grupai *PkInfo*. Piemērs iespējamajam izskatam dots Zīm. 13.



Zīm. 12 Ekrāna objektu grupa UkInfo



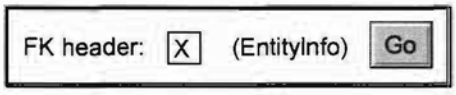
Zīm. 13 Piemērs ekrāna objektu grupai UkInfo

Lauks ar tipu *FK* (ārējā atslēga) attēlojas par ekrāna objektu grupu ***FkInfo*** (Zīm. 14). Šīs grupas saturs ir atkarīgs no ekrāna tipa, relācijas un saistītās entītijas tipa. *Fk header* ir tekstlodziņa tipa objekts ar vērtību, kas ģenerēta no *FieldName*.

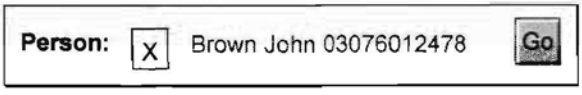
Izvēles rūtiņa ir neobligāts elements. Tas tiek ģenerēts, kad atbilstošai relācijai pretējā galā ir kardinalitāte 0..1, citādi (kardinalitāte ir 1) izvēles rūtiņa netiek ģenerēta. Mēs varam piešķirt tukšu vērtību ārējās atslēgas laukam, neatzīmējot izvēles rūtiņu.

EntityInfo ir cita ekrāna objektu grupa (skatīt 3.2.2 Entītijas attēlošana). Poga ***Go*** ir neobligāta un tiek ģenerēta atbilstoši ekrāna tipam (pogas pielietojums aprakstīts 3.2.5 Traversēšana pa ekrāniem).

Piemērs iespējamajam izskatam dots Zīm. 15.



Zīm. 14 Ekrāna objektu grupa FkInfo



Zīm. 15 Piemērs ekrāna objektu grupai FkInfo

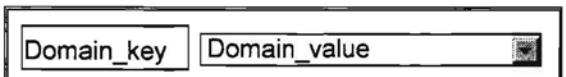
3.2.2 Entītijas attēlošana

Entītijas instance tiek attēlota ar ekrāna objektu grupu ***EntityInfo***. Definēsim dažus *EntityInfo* apakštipus.

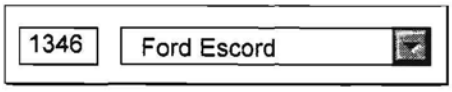
Entītija ar tipu *Domain* tiek attēlota ar ***DomainInfo*** (Zīm. 16). Objekts *Domain_value* ar tipu kombinētais lodziņš nodrošina domēna vērtības izvēli un attēlo izvēlēto vērtību. Šis elements ir obligāta grupas sastāvdaļa.

Rediģēšanas lodziņa tipa objekts *Domain_key* ir neobligāts. Tas tiek ģenerēts pēc sekojošiem likumiem. Vispirms, ja entītijai ir redzama unikālā atslēga *Domain_key*, ņem datu tipu no šī lauka. Citādi, ja entītijai ir redzams primārās atslēgas lauks, tad tiek ņemts datu tips no tā. Ja nav redzams ne unikālās, ne primārās atslēgas lauks, tad rediģēšanas lodziņš netiek ģenerēts.

Abi objekti vienmēr norāda uz vienu un to pašu tabulas ierakstu. *Domain_key* var tikt izmantots alternatīvai vērtības izvēlei, ja tiek ievadīta pieļaujama vērtība. *Domain_value* ir entītiju reprezentējošs teksts, kas atkarīgs no noklusētās entītijas instances teksta ģenerēšanas funkcijas. Piemērs *DomainInfo* iespējamajam izskatam dots Zīm. 17.

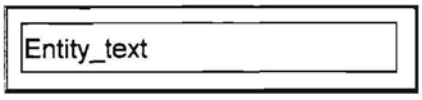


Zīm. 16 Ekrāna objektu grupa DomainInfo

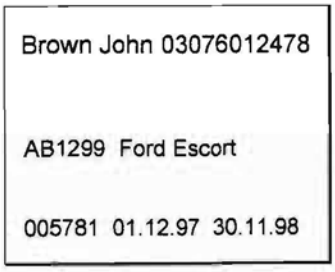


Zīm. 17 Piemērs ekrāna objektu grupai DomainInfo

Entītija ar tipu *SimpleEntity* vai *ComplexEntity* tiek attēlota ar ***EntityTextInfo*** (Zīm. 18). Objekts *Entity_text* ar tipu tekstlodziņš ir entītiju reprezentējošs teksts, kas atkarīgs no noklusētās entītijas instances teksta ģenerēšanas funkcijas. Piemēri iespējamajam izskatam doti Zīm. 19.

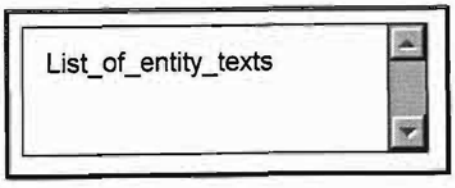


Zīm. 18 Ekrāna objektu grupa EntityTextInfo

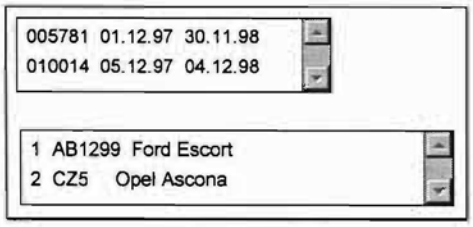


Zīm. 19 Piemērs ekrāna objektu grupai EntityTextInfo

Vairākas līdzīgas entītijas ar tipu *SimpleEntity* vai *ComplexEntity* tiek attēlotas ar ***EntityListInfo*** (Zīm. 20). Objekts *List_of_entity_texts* ar tipu sarakstlodziņš satur entītijas reprezentējošos tekstus, kas atkarīgi no noklusētās entītijas instances teksta ģenerēšanas funkcijas. *EntityListInfo* attēlo arī entītijas ar tipu *Link*, bet teksta ģenerēšanas funkcija var izslēgt lauku ar tipu *FK*. Piemēri iespējamajam izskatam doti Zīm. 21.



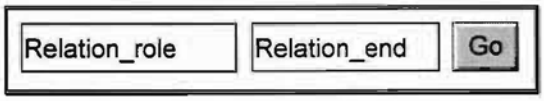
Zīm. 20 Ekrāna objektu grupa EntityListInfo



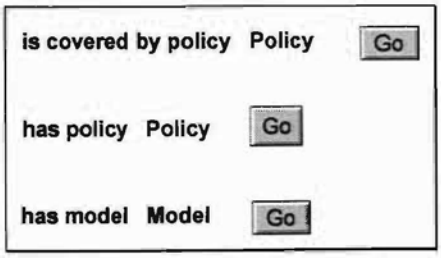
Zīm. 21 Piemēri ekrāna objektu grupai EntityListInfo

3.2.3 Relācijas attēlošana

Relācijas viens virziens tiek attēlots ar ***RelationshipEndInfo*** (Zīm. 22). Pieņemsim, ka mēs attēlojam relāciju no *Entity_1* uz *Entity_2* ar lomu *Role_1*. Tekstlodziņa tipa objekts *Relation_role* saturēs tekstu 'Role_1' un tekstlodziņa tipa objekts *Relation_end* saturēs tekstu 'Entity_2'. Poga ***Go*** nodrošina pārvietošanos uz ekrānu, kas reprezentē entītiju *Entity_2*. Piemēri iespējamajam izskatam doti Zīm. 23.



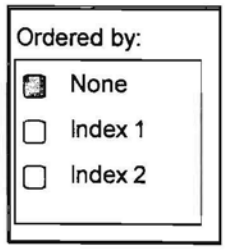
Zīm. 22 Ekrāna objektu grupa RelationshipEndInfo



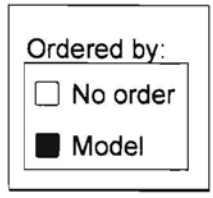
Zīm. 23 Piemēri ekrāna objektu grupai RelationshipEndInfo

3.2.4 Indeksu attēlošana

Ja entītijai ir vismaz viens indekss, tad visi indeksi tiek attēloti ar radio pogu grupu ***OrderButtonGroup*** (Zīm. 24). Pirmā poga *None* nodrošina iepriekš uzstādītās ierakstu kārtības noņemšanu. Katra nākamā poga atbilst kādam indeksam. Piemērs iespējamajam izskatam dots Zīm. 25.



Zīm. 24 Ekrāna objektu grupa OrderButtonGroup



Zīm. 25 Piemērs ekrāna objektu grupai OrderButtonGroup

3.2.5 Traversēšana pa ekrāniem

Traversēšana pa ekrāniem tiek nodrošināta ar pogas ***Go*** palīdzību. Šī poga parasti tiek pievienota ekrāna objektu grupai, kas reprezentē entītiju. Šī poga nodod kontroli uz ekrānu, kas pieder šai entītijai. Poga var strādāt divos režīmos – ar filtru vai bez tā. Ja ir izvēlēta filtra opcija, tad jaunatvērtajā ekrānā mēs tiksim klāt tikai tiem ierakstiem, kas loģiski ir saistīti ar ierakstu vai ierakstiem iepriekšējā ekrānā. Piemēram, mēs fiksējam kādu automašīnas modeli tabulā Model, tad tabulā Auto būs pieejamas tikai automašīnas ar fiksēto modeli.

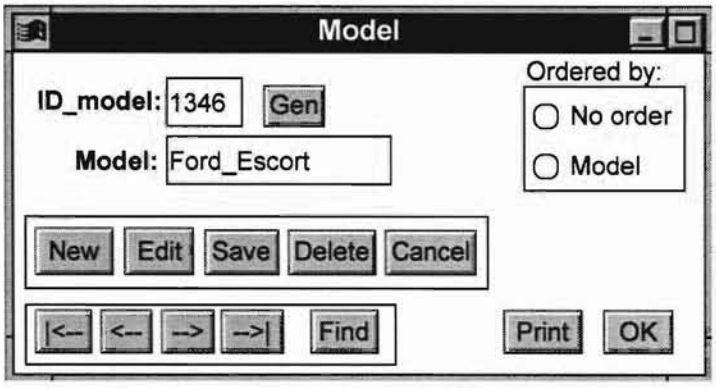
Piemēri iespējamajiem izskatiem un izmantošanas variantiem doti Zīm. 15 un Zīm. 23.

3.3 Ekrānu tipi

Ekrāna tipu projektējums ir atkarīgs no lietotāja vajadzībām. Mēs piedāvājam piemērus, kas varētu tikt uzskatīti par bāzes tipiem. Ekrānu piemēri atbilst ER modelim attēlotam Zīm. 4.

3.3.1 Ekrāna tips Simple entity view

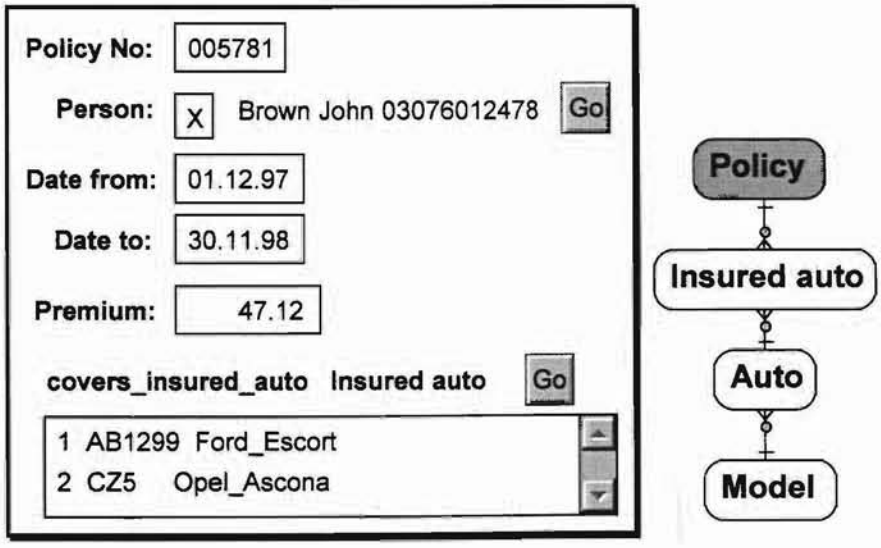
Dotais ekrāna tips var tikt lietots, lai attēlotu jebkuru entītiju (piemērs dots Zīm. 26). Informatīvā grupa satur visus redzamo lauku ekrāna objektu grupas un *OrderButtonGroup*, ja ir definēts kāds indekss. Vadības grupa satur *Edit button group*, *Locate button group*, pogu *Print* un pogu *OK*.



Zīm. 26 Ekrāns entītijai Model

3.3.2 Ekrāna tips Entity view extension with links

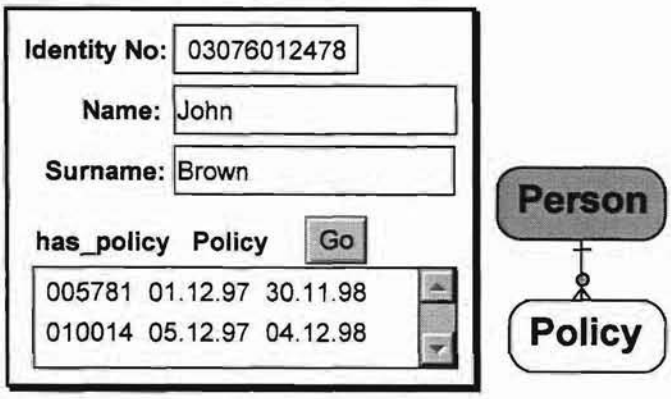
Dotais ekrāna paplašinājums var tikt pievienots *SimpleEntity* vai *ComplexEntity* entītiju ekrāniem. Visas entītijas, kas ir ar tipu *Link* un ir tieši saistītas caur relāciju ar doto entītiju, tiek attēlotas uz ekrāna. Attēlošana tiek nodrošināta ar ekrāna objektu grupām *RelationshipEndInfo* un *EntityListInfo*. Zīm. 27 satur ekrāna fragmentu entītijai Policy ar apdrošinātajām automašīnām šajā polisē. Automašīnu attēlošanai tiek izmantota LongView entītijas teksta ģenerēšanas funkcija.



Zīm. 27 Fragments ekrānam entītijai Policy un tai saistītajām entītijām

3.3.3 Ekrāna tips Entity view extension with relations

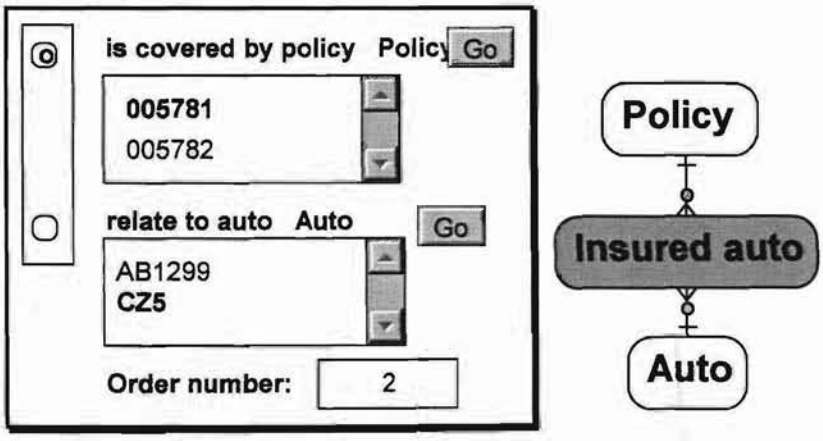
Dotais ekrāna paplašinājums var tikt pievienots *Domain*, *SimpleEntity* vai *ComplexEntity* entītiju ekrāniem. Mēs attēlojam visas relācijas, kurām pretējā galā ir ārējā atslēga (tas nozīmē, ka cita entītija referencējas uz doto entītiju ar ārējās atslēgas palīdzību), un arī tām atbilstošās entītijas. Attēlošanai tiek izmantotas ekrāna objektu grupas *RelationshipEndInfo*, *EntityTextInfo* vai *EntityListInfo*. Zīm. 28 fiksētajai personai tiek attēlotas visas polises (tiek izmantota LongView entītijas teksta ģenerēšanas funkcija).



Zīm. 28 Fragments ekrānam entītijai Person un tai saistītajām entītijām

3.3.4 Ekrāna tips Simple link view

Dotais ekrāns ir speciāls skats uz entītijām ar tipu *Link*, kas saista kopā tieši divas entītijas ar tipu *SimpleEntity* vai *ComplexEntity* (Zīm. 29). Entītijas teksta ģenerēšanas funkcija ShortView ir izmantota, lai attēlotu saistītās entītijas objektā *EntityListInfo*. Speciāla ekrāna objektu grupa *Control button group* nosaka, kurš ListBox tiek uzskatīts par galveno dotajā brīdī. Šajā gadījumā fiksētajai polisei 005781 visas apdrošinātās mašīnas tiek attēlotas otrā sarakstā ar pazīmi Auto. Otrajā sarakstā aktīvais elements ir auto CZ5 un tā kārtas numurs polisē ir 2.

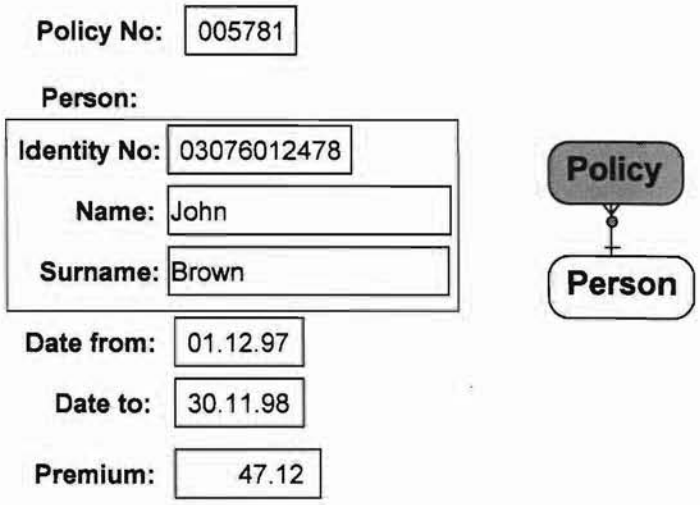


Zīm. 29 Fragments ekrānam entītijai Insured auto un tai saistītajām entītijām

3.3.5 Ekrāna tips Embedded entities view

Dotais ekrāns ir pielietojams tikai entītijām ar tipu *ComplexEntity*. Paņemsim attēlojumu *Simple entity view* kā bāzes attēlojumu šai entītijai. Visu ekrāna objektu grupu *FkInfo* vietā mēs iekļausim visus redzamos laukus no attiecīgās entītijas. Mēs varam iztēloties katru iekļauto entītiju kā apakšekrānu, kurā tiek attēlota šī entītija atbilstoši ar *Embedded entities view* kompleksajām entītijām vai *Simple entity view* vienkāršajām entītijām. Piemēram, Objektu grupa ar virsrakstu Person ir aizvietota ar

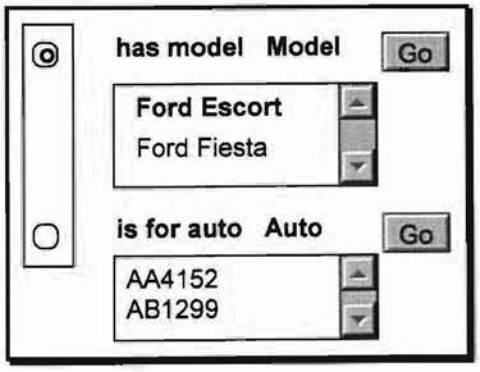
trīs ekrāna objektu grupām - *Identity_No*, *Name*, *Surname* no entītijas *Person* (Zīm. 30). Ģenerācijas procesā jāuzmanās no cikliskas iekļaušanas un jāaptur iekļaušana, sastopot ciklu, vai arī jāvadās pēc citas drošas stratēģijas.



Zīm. 30 Iekļauto entītiju skats

3.3.6 Ekrāna tips Relationship view

Dotais ekrāns nodrošina speciālu skatu uz relāciju un entītijām, ko tā saista. Saistītās entītijas ir reprezentētas ar *RelationshipEndInfo* un *EntityListInfo*. Galvenā entītija tiek izvēlēta ar radio pogu. Zīm. 31 attēlo ekrāna fragmentu relācijai *[Model] is_for_auto / has_model [Auto]* ar ShortView entītijas teksta ģenerēšanas funkciju. Izmantoto elementu princips ir līdzīgs *Simple link view*.



Zīm. 31 Fragments ekrānam relācijai starp entītijām Model un Auto

Līdz ar to mēs esam parādījuši principus, kā var organizēt lietotāja saskarni, balstoties uz informāciju repozitorijā un izmantojot vienkāršus saskarnes elementus. Pēc šo principu prezentēšanas līdz ar tīmekļa programmatūras izstrādes veidu attīstīšanos līdzīgi principi tika piedāvāti satura pārvaldības sistēmās (*content management system*). Tālākajos autora darbos šie principi jau ir pielietoti pārlūkprogrammas aplikāciju veidošanā. Tuvāka informācija pat to ir atrodama nākamajā nodaļā.

4 Informācijas apstrādes lietojumprogrammu izveide

Lietojumprogrammu ģenerācijas pamatideja ir izveidot sistēmu ar predefinētu lietotāja saskarni un funkcionalitāti. Ģenerētās sistēmas īpašības ir atkarīgas no ģeneratora. Mēs varam ģenerēt lietojumprogrammu visam dotajam datu modelim vai tikai atsevišķām lietojumprogrammas komponentēm. Ģenerācija un koda kompilācija varētu notikt gan pirms izpildes, gan arī dinamiski lietojumprogrammas izpildes laikā.

Par rīku var tikt uzskatīts gan lietojumprogrammu ģenerators, gan arī rezultāts – lietojumprogramma, kas strādā ar datu avotu un prezentē informāciju.

4.1 Statiska lietojumprogramma

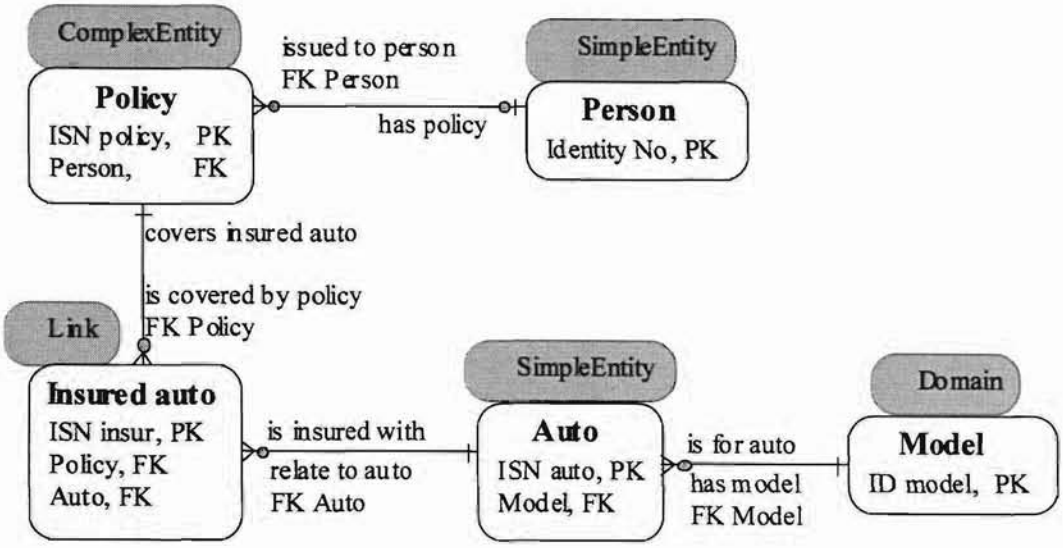
Statiska lietojumprogramma mūsu izpratnē ir lietojumprogramma, kas faktiski ir nemainīga tās izpildes laikā. Lietojumprogrammas kods tiek nokompilēts jau pirms izpildes un šajā gadījumā ģeneratoram vajadzētu saģenerēt visu vēlamo kodu uzreiz. Tālāk apskatītais piemērs ir domāts fiziskam datu modelim, t.i. mēs balstāmies uz pieejamās datubāzes ER modeli.

4.1.1 Algoritms entītijas tipa noteikšanai

Pirms mēs sākam ģenerēt lietojumprogrammu, ir nepieciešams noteikt katras entītijas tipu (Zīm. 32, kas ir vienkāršojums ER modelim, kas dots Zīm. 4). Algoritms visu entītiju tipu noteikšanai ir sekojošs:

1. Piešķiram visām entītijām tipu *Undefined.*
2. Skanējam visas entītijas un fiksējam tās, kurām nav lauku ar tipu FK (ārējā atslēga) un visi relāciju gali pie šīs entītijas ir ar kardinalitāti 1 vai 0..1. Mēs piešķiram šādām entītijām tipu *Domain* vai *SimpleEntity*. Pēc noklusēšanas tips ir *Domain.*
3. Skanējam visas entītijas ar tipu *Undefined* un fiksējam tās, kurām visi lauki ar tipu FK referencējas tikai uz entītijām ar tipu *Domain* un kurām visi atlikušie relāciju gali pie šīs entītijas ir ar kardinalitāti 1 vai 0..1. Mēs piešķiram šādām entītijām tipu *SimpleEntity.*
4. Skanējam visas entītijas ar tipu *Undefined* un fiksējam visas tās, uz kurām šajā brīdī referencējas ar FK lauka palīdzību kāda entītija ar tipu *SimpleEntity*, *ComplexEntity* vai *Undefined.* Entītija var referencēties pati uz sevi. Mēs piešķiram šādām entītijām tipu *ComplexEntity.*
5. Skanējam visas entītijas ar tipu *Undefined* un fiksējam visas tās, kurām ir vismaz divi lauki ar tipu FK, kas referencējas uz entītijām ar tipu *SimpleEntity* vai *ComplexEntity*. Mēs piešķiram šādām entītijām tipu *Link* vai *ComplexEntity*. Pēc noklusēšanas tips ir *Link.*
6. Visām entītijām ar tipu *Undefined* tiek piešķirts tips *ComplexEntity.*

Lietotājs pats nepieciešamības gadījumā koriģē tipus (2. un 5. solis) atbilstoši datubāzes semantikai un tam, kāda tipa ekrānus entītijai gribēs ģenerēt. Citas tipa izmaiņas nav pieļaujamas, jo tad ģenerators var uzģenerēt neatbilstošu lietojumprogrammu.



Zīm. 32 Entītijas tipa noteikšana: 2. solī Person un Model, 3. solī Auto, 4. solī Policy un 5. solī Insured auto

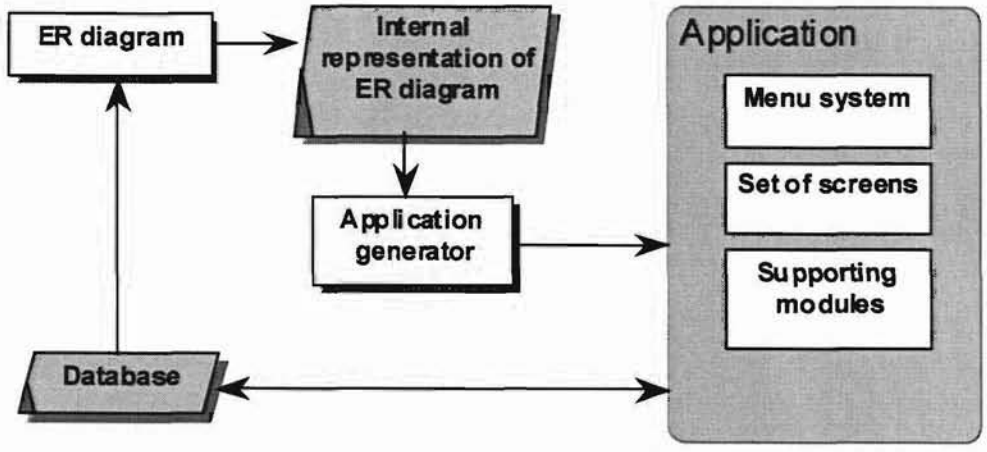
4.1.2 Lietojumprogrammas ģenerācijas shēma

Lietojumprogrammas ģenerācijas robusta shēma (Zīm. 33) varētu būt sekojoša:

- Ģenerēt lietojumprogrammas vārdu no ER diagrammas vārda.
- Ģenerēt ekrānus katrai entītijai (visus entītijas tipam atļautos ekrāna tipus):
  1. Ģenerēt ekrāna vārdu no entītijas vārda.
  2. Ģenerēt katru informatīvo apakšgrupu, kas nepieciešama dotajam ekrāna tipam. Grupas izvietojums (horizontāls, vertikāls, tabulārs vai cits) tiek izvēlēts atbilstoši vēlmēm.
     a. Ģenerēt visu lauku grupas dotajai entītijai.
     b. Ģenerēt informāciju par saistītajām entītijām, kas ir saistītas ar ārējās atslēgas palīdzību.
     c. Ģenerēt informāciju par saistītajām entītijām, kas ir saistītas ar relācijas palīdzību un dotajai entītijai tā nav ārējā atslēga.
     d. Ģenerēt sakārtojuma pogu grupu dotajai entītijai, ja eksistē kāds indekss.
  3. Ģenerēt visas vadības apakšgrupas, kas nepieciešamas dotajam ekrāna tipam.
- Ģenerēt ekrānus katrai relācijai ar visiem atļautajiem tipiem.
  1. Ģenerēt ekrāna vārdu no saistīto entītiju vārdiem un lomu vārdiem.
  2. Ģenerēt katru informatīvo apakšgrupu, kas nepieciešama dotajam ekrāna tipam (analoģiski augstāk jau aprakstītajam).
  3. Ģenerēt visas vadības apakšgrupas, kas nepieciešamas dotajam ekrāna tipam.
- Ģenerēt izvēlņu sistēmu, kas organizē ekrāna tipus. Dziļāko līmeņu izvēlnes tiek ģenerētas no entītijas vārda (ja pamatā ir entītija) vai no divu entītiju vārdiem un

lomas vārda (ja pamatā ir relācija). Izvēlne atver ekrānu ar atbilstošo tipu un entītiju/relāciju kā centrālo objektu. Pirms atvēršanas varētu būt objektu filtrējošs ekrāns.

- Ģenerēt papildus izvēlnes standarta operācijām.



Zīm. 33 Vienkāršota lietojumprogrammas ģenerēšanas shēma

4.2 Dinamiska lietojumprogramma

Ar dinamisku lietojumprogrammu mēs saprotam, ka lietojumprogrammas kods tiek ģenerēts lietojumprogrammas izpildes laikā. Visvienkāršāk šo principu demonstrēt un realizēt, veidojot tīmekļa lietojumprogrammas.

Tālāk demonstrētajā piemērā mēs operējam ar loģisko datu modeli, kas aptver vairākus datu avotus, t.i. mums var nebūt zināmas precīzas datu avotu implementācijas, jo ir pieejamas tikai funkcijas šī informācijas avota izmantošanai. Vienkāršības pēc tiek apskatīts tikai skatīšanās variants bez datu maiņas datu avotā. Tā kā dinamiskām lietojumprogrammām vēlamas formālas specifikācijas, tad piemēra aprakstā ir izmantoti formāli pierakstīšanas līdzekļi.

4.2.1 FrameSet jēdziens

Pamatideja dinamiskai datu pārlūkošanai daudzos datu avotos ir ģenerēt pārlūkprogrammas lapas ar predefinētu informācijas izklājumu un funkcionalitāti, ņemot datus no datu avotiem un ieliekot tos lapā.

Pārlūkprogrammas lapa sastāv no kadru kopas – *FrameSet*. FrameSet ir iepriekš definēta struktūra, t.i. ir noteikts kadru skaits, to izklājums un izmēri. Mēs varam definēt dažādus FrameSet pēc nepieciešamības, lai organizētu un attēlotu informāciju klientam vēlamajā veidā. FrameSet ir skats uz saistītu informāciju no viena vai vairākiem datu avotiem. Viens no kadriem ir galvenais - *MainFrame*. Informācija jebkurā citā kadrā ir loģiski saistīta ar datiem MainFrame. Kadrs var saturēt kontroles vadības līdzekļus, lai pārvaldītu informāciju citos kadros.

Kadra izklājums ir definēts ar likumu, ko sauksim par *Content*. Formāli runājot, Content ir formula vai funkcija: ***Content(frameEntity, filterExpr)***, kur *frameEntity* ir

jebkura entītija no datu avota meta-modeļa un *filterExpr* ir loģiska izteiksme, kas filtrē datus no atbilstošā datu avota.

Content definē:

1. kāda ir izklājuma struktūra un princips;
2. kādi dati no meta-modeļa un no aktuālā eksistējošā datu avota ir nepieciešami informācijas attēlošanai;
3. kādas aktuālās instances no definētās entītijas tiks paņemtas;
4. kādi kontroles līdzekļi tiks lietoti, lai pārvaldītu saturu citā kadrā vai arī lai atvērtu citu FrameSet;
5. kādas saistītās entītijas ir iesaistītas no tā paša vai citiem datu avotiem.

Ja mums ir definēti vairāki Content, tad mēs varam dinamiski pielietot jebkuru Content fiksētajam kadram un iegūt citu datu attēlojumu tiem pašiem *frameEntity* un *filterExpr*.

4.2.2 Kadra satura definēšana

Pieņemsim, ka Content ir funkcija ***Content(frameEntity, filterExpr)***. Noteiksim līdzekļus, ar kuru palīdzību var definēt Content.

Mēs ieviešam sekojošus datu tipus:

- **entity** - nosaka entītiju no meta-modeļa;
- **field** – nosaka entītijas lauku no meta-modeļa;
- **relation** – nosaka relāciju starp divām entītijām no meta-modeļa;
- **record** – nosaka aktuālos datus no datu avota priekš fiksētas entītijas instances;
- **value** – nosaka aktuālo vērtību laukam priekš fiksētas entītijas instances;
- **string** – nosaka simbolu virkni;
- **list** – nosaka sarakstu elementiem ar jebkuru citu atļautu datu tipu, mēs apzīmējam šādu tipu ar elementa tipa vārdu, kuram seko postfikss "List";
- **updateAction** – nosaka aktivitāti, kas maina kadra saturu;
- **navigateAction** – nosaka aktivitāti, kas pārnes aktīvo datu pārlūkošanu uz citu FrameSet;
- **sObject** – nosaka HTML objektu, kas satur attēlojamā objekta simbolisku vērtību;
- **aObject** – nosaka HTML objektu, kuram ir piešķirta kāda izpildāma aktivitāte;
- **fObject** – nosaka HTML objektu, kas ir noformēts attēlošanai;
- **frame** – nosaka kadru;
- **frameSet** – nosaka FrameSet;
- **view** – nosaka sarakstu ar laukiem, kas ir jāattēlo.

Pārrakstīsim Content kā funkciju ***Content(entity, expr(entity))***. Ieviesīsim dažas papildus funkcijas darbam ar meta-modeli un datu avotiem, kā arī metodes HTML lapas noformēšanai.

Funkcijas darbam ar meta-modeli:

1. ***SourceName(entity)→ string*** – atgriež datu avota vārdu, kuram entītija pieder;
2. ***EntityName(entity)→ string*** – atgriež entītijas vārdu;
3. ***RelationList(entity)→ relationList*** – atgriež visas tiešās relācijas no dotās entītijas uz citu entītiju (ieskaitot sevi pašu) no tā paša datu avota;
4. ***MetaRelationList(entity)→ relationList*** – atgriež visas netiešās relācijas no dotās entītijas uz citu entītiju no visiem pieejamajiem datu avotiem;
5. ***FieldList(entity)→fieldList*** – atgriež sarakstu ar visiem entītijas laukiem;
6. ***RelationName(relation)→ string*** – atgriež relācijas vārdu (lomu);
7. ***FieldName(field)→ string*** – atgriež lauka vārdu;
8. ***RelationEntity(relation))→ entity*** – atgriež entītiju relācijas otrajā galā.

Funkcijas darbam ar datu avotiem:

1. ***RecordList(entity, expr(entity))→ recordList*** – atgriež sarakstu ar entītijas instancēm (ierakstiem) atbilstoši filtrēšanas izteiksmei;
2. ***ValueList(record)→ valueList*** – atgriež sarakstu ar vērtībām entītijas instancei (ierakstam);
3. ***Value(value)→ string*** – atgriež lauka vērtību kā simbolu virkni.

Funkcijas darbam ar sarakstu:

1. ***List(element_1, element_2, ..., element_i)→ list_1*** – atgriež sarakstu no dotajiem elementiem un saraksta tips *list_1* ir atbilstošs elementa tipam;
2. ***IterateList(n%list_1, function(n%))→ list_2*** – atgriež sarakstu *list_2*, kas kā elementus satur rezultātus, pielietojot doto funkciju. Funkcija tiek izpildīta ar katru parametru *n%*, kas tiek ņemts no saraksta *list_1* apzīmēts ar identifikatoru *n%* (n ir unikāls vesels pozitīvs skaitlis) un saraksta tips *list_2* ir atbilstošs funkcijas atgriežamajam tipam;
3. ***Concatenate(list_1, list_2)→ list_3*** – atgriež konkatenāciju diviem sarakstiem ar to pašu elementu tipu.

Funkcijas HTML lapas noformēšanai:

1. ***SO(string)→ sObject*** – izveido sObject no simbolu virknes
2. ***StringListObject(stringList, separatorString)→ sObject*** – izveido sObject no saraksta ar simbolu virknēm atdalītas ar simbolu virkni *separatorString;*
3. ***Update(frame, entity, expr(entity), content)→ updateAction*** – aktivizē informācijas atjaunināšanu kadrā ar doto entītiju, filtra izteiksmi un izklājumu;

4. ***Navigate(frameSet , entity, expr(entity), content)→ navigateAction*** – veic pārvietošanos uz citu FrameSet un atjauno MainFrame ar doto entītiju, filtra izteiksmi un izklājumu;
5. ***Link(sObject, navigateAction, updateActionList)→ aObject*** – pārveido sObject uz aObject un piešķir tam navigācijas aktivitāti un kopu ar atjaunināšanas aktivitātēm. Parametri var būt tukši;
6. ***AObject(sObject)→ aObject*** – pārveido sObject uz aObject bez jebkādas aktivitātes;
7. ***FO(aObject)→fObject*** – pārveido aObject uz fObject bez jebkādas speciālas noformēšanas;
8. ***HorizontalTable(aObjectListList)→ fObject*** – izveido fObject no sarakstu saraksta un šis kadra objekts tiek attēlots kā tabula un iekšējie saraksti tiek izvietoti pa rindām;
9. ***VerticalTable(aObjectListList)→ fObject*** – izveido fObject no sarakstu saraksta un šis kadra objekts tiek attēlots kā tabula un iekšējie saraksti tiek izvietoti pa kolonām;
10. ***ListBox(aObjectList)→ fObject*** – izveido fObject no saraksta un šis kadra objekts tiek attēlots kā sarakstlodziņš;
11. ***Horizontal(fObjectList)→ fObject*** – izveido jaunu fObject, izvietojot doto sarakstu horizontāli;
12. ***Vertical(fObjectList)→ fObject*** – izveido jaunu fObject, izvietojot doto sarakstu vertikāli.

WEB lapā var tikt attēloti tikai kadra objekti ar tipu fObject.

4.2.3 Piemēri pārlūkprogrammas lapas struktūrai un funkcionalitātei

FrameSet un kadru projektēšana balstās uz šablonu pielietošanas principu. Ar nelielu pieredzi un šablonu bibliotēku vajadzīgie FrameSet un kadri var tikt izveidoti samērā ātri. Projektēšana sastāv no diviem soļiem – FrameSet struktūras plānošana un Content formulas izveide katram kadram. Mēs dodam dažus paraugus un idejas, kā WEB lapas varētu tikt plānotas. Tiek izmantotas augstākminētās funkcijas. Piemēriem ir izmantots loģiskais datu modelis, kas dots Zīm. 7.

- **Simple entity instance presentation in table**

Entītijas attēlošana notiek ar tabulas palīdzību. Pirmā kolona satur lauku vārdus, bet otrā – lauku vērtības. Rezultāts zemāk attēlotajai formulai, kas pielietota entītijai Persona, ir redzams Zīm. 34.

```
A(entity, record) = VerticalTable(List(A1, A2))
A1 = IterateList(1%FieldList(entity, view), AO(SO(FieldName(1%))))
A2 = IterateList(2%ValueList(record, view), AO(SO(Value(2%))))
```

PK	12121211111
Name	Andris
Surname	Kalns
Sex	M
Address	Riga, Liepu 1-12, LV-1000

Zīm. 34 Piemērs entītijas instances attēlošanai

- **Entity instance presentation as text**

Entītijas attēlošana notiek ar teksta palīdzību. Instances lauku vērtības tiek sakonkatenētas atbilstoši izvēlētajam skatam *view*. Rezultāts zemāk attēlotajai formulai, kas pielietota entītijai Persona, ir redzams Zīm. 35.

```
B(record) = FO(AO(SO(StringListObject(B1, " "))))
B1 = IterateList(3%ValueList(record, view), Value(3%))
```

12121211111 Andris Kalns M Riga, Liepu 1-12, LV-1000

Zīm. 35 Piemērs entītijas instances tekstiskai attēlošanai

- **Entity relations presentation in vertical list**

Entītijas relāciju attēlošana notiek ar vertikāla saraksta palīdzību. Katra relācija ir reprezentēta ar relācijas vārdu sakonkatenētu ar entītijas vārdu relācijas otrā galā. Rezultāts zemāk attēlotajai formulai, kas pielietota entītijai Persona, ir redzams Zīm. 36.

```
C(entity) = Vertical(IterateList(4%RelationList(entity), C1))
C1 = Horizontal(List(C2, FO(AO(SO(" "))), C3))
C2 = FO(AO(SO(RelationName(4%))))
C3 = FO(AO(SO(EntityName(RelationEntity(4%)))))
```

Has Passport
Has Parents Citizen
Has Children Citizen

Zīm. 36 Piemērs 1 relācijas attēlošanai

- **All relation presentation in table**

Entītijas visu relāciju attēlošana notiek ar saraksta palīdzību. Dati par visām relācijām (relācijas vārds, entītijas vārds un datu avots) tiek ielikti tabulā ar virsrakstiem. Rezultāts zemāk attēlotajai formulai, kas pielietota entītijai Persona, ir redzams Zīm. 37.

```
D(entity, expr(entity))=HorizontalTable(Concatenate(D1,D2))
D1 = AO(StringListObject("Relation", "Entity name", "Data source"))
D2 = IterateList(5%MetaRelationList(entity),List(D3, D4, D5))
D3 = AO(SO(RelationName(5%)))
D4 = AO(SO(EntityName(RelationEntity(5%))))
D5 = AO(SO(SourceName(RelationEntity(5%))))
```

Relation	Entity name	Data source
Is	Citizen	Register of Residents
Has	Passport	Register of Residents
Has Parents	Citizen	Register of Residents
Has Children	Citizen	Register of Residents
Is	Car Owner	Register of Motor vehicles
Owns	Car	Register of Motor vehicles

Zīm. 37 Piemērs 2 relācijas attēlošanai

4.2.4 FrameSet definēšanas piemērs

Apskatīsim, kā var tikt izveidots FrameSet. Pieņemsim, ka ir FrameSet *FRS_1* ar četriem kadriem – *FR_1, FR_2, FR_3, FR_4.*

- FR_1 tiek izmantots, lai attēlotu entītijas instanču sarakstu,
- FR_2 – lai parādītu detaļas fiksētai instancei no FR_1,
- FR_3 – lai attēlotu sarakstu ar visām relācijām uz citām entītijām visos datu avotos,
- FR_4 – lai parādītu detaļas citai saistītai entītijai no kadra FR_2 vai FR_4.

Rezultāts zemāk attēlotajai formulai, kas pielietota entītijai Persona, ir redzams Zīm. 38.

- Content formula E() kadram FR_4 (no FR_4 mēs varam atjaunināt visus kadrus iekš FRS_1)

```
E(entity, expr(entity)) = Vertical(E1, E5)
E1 = Horizontal(List(FO(E2), FO(AO(SO(" "))),
     FO(AO(SO(SourceName(entity))))))
E2 = Link(SO(EntityName(entity)), E3, E4)
E3 = Navigate("FRS_1", entity, expr(entity), "" )
E4 = List(Clear("FR_2"), Update("FR_3", entity, expr(entity), """),
     Clear(FR_4))
E5 = Vertical(IteateList(6%RecordList(entity, expr(entity)),
     A(entity, 6%)))
```

- Content formula F() un G() katram FR_2 (no FR_2 mēs varam atjaunināt šo pašu kadru vai atjaunināt kadru FR_4)

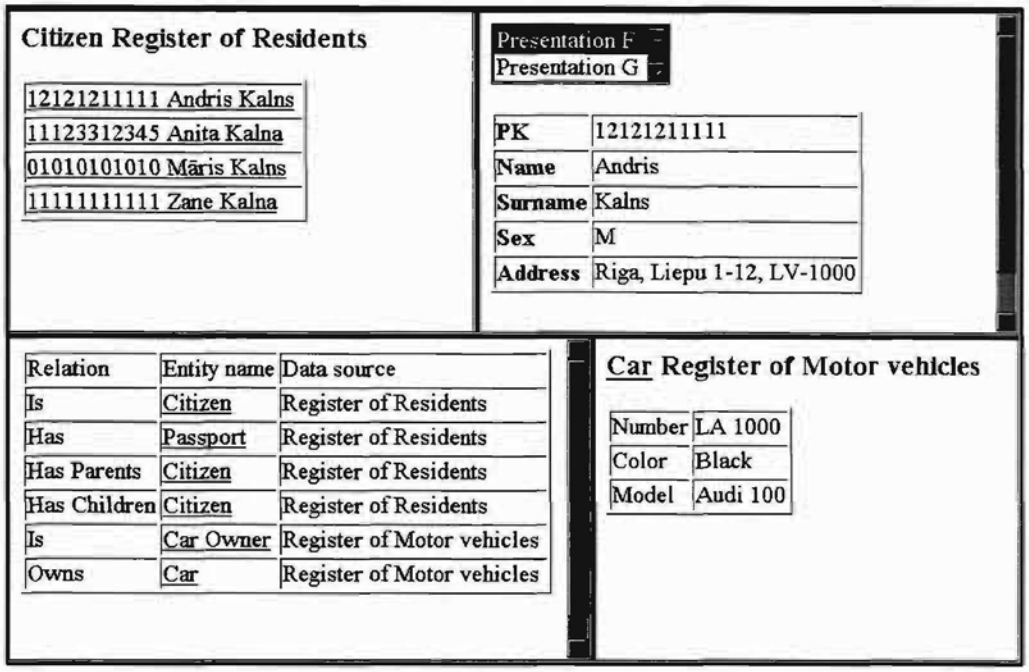
```
F(entity, expr(entity)) = Vertical(H, FO(AO(SO(" "))), E5)

G(entity, expr(entity))  = Vertical(H, FO(AO(SO(" "))),
      Vertical(E5, G1))

G1 = C(entity), where C3 is substitute with G2 in all places (we
      have added the action)

G2 = FO(Link(SO(EntityName(RelationEntity(4%))), NULL, G3))

G3 = List(Update("FR_4", RelationEntity(4%),
      expr(RelationEntity(4%)), "E"))

H = ListBox(List(Link("Presentation F", NULL, H1),
      Link("Presentation G", NULL, H2)))

H1 = Update("FR_2", entity, expr(entity), "F")

H2 = Update("FR_2", entity, expr(entity), "G")
```

- Content formula I() kadram FR_1 (no FR_1 mēs varam atjaunināt kadrus FR_2, FR_3, FR_4)

```
I(entity, expr(entity))  = Vertical(I1, I2)

I1 = Horizontal(List(FO(EntityName(entity)), FO(AO(SO(" "))),
      FO(AO(SO(SourceName(entity))))))

I2 = HorizontalTable(IterateList(7%RecordList,Link(B(7%), NULL,
      I3))

I3 = List(Update("FR_2", entity, expr(entity) and expr(7%), "F"),

Update("FR_3", entity, expr(entity) and expr(7%), ""),
      Clear("FR_4"))
```

- Content formula J() kadram FR_3 (no FR_3 mēs varam atjaunināt kadru FR_4)

```
J(entity, expr(entity))  = D(entity, expr(entity)), where D4 is
      substitute with J1 in all places (we have add the action)

J1 = Link(SO(EntityName(RelationEntity(5%))), NULL, J2)

J2 = List(Update("FR_4", RelationEntity(5%),
      expr(RelationEntity(5%)), "E"))
```



Zīm. 38 Piemērs pārlūkprogrammas lapai

5 Rīka darbības semantika un tās implementācija

Dotajā nodaļā ir apskatīta rīka darbības semantikas pierakstīšanas veids un tās iespējamā implementācija, kā arī tiek definēts daudzvalodu interpretators (MLI), kas nodrošina programmas izpildi, saņemot arī atbilstošās programmēšanas valodas sintaksi un nepieciešamo semantiku.

Daudzvalodu interpretators var kalpot par pamatu lielākajai daļai mums nepieciešamo rīku izveidei. Galvenā uzmanība ir veltīta mūsu daudzvalodu interpretatora semantikas (uzvedības) uzdošanai, kā arī šīs semantikas pārvēršanai par izpildāmu kodu.

Semantika tiek definēta kā kompozīcija no daudziem semantiskajiem aspektiem, ņemot vērā valodas pragmatiku. Semantisko aspektu apraksti, tos apvienojot, tiek transformēti par semantiskām funkcijām, lai izpildi varētu veikt, traversējot iekšējo reprezentāciju un izsaucot semantiskās funkcijas, analoģiski *Visitor pattern* principam.

Semantikas pierakstā tiek izmantotas abstraktas komponentes, kas tiek saistītas meta līmenī ar konektoriem. Savukārt šo komponenšu un konektoru implementācija var būt ļoti dažāda.

Darbā doti piemēri semantikas definēšanai gan parastai, gan specifiskai izpildei.

5.1 Problēmas nostādne

Līdz ar straujo un plašo domēnspecifisko valodu pielietojumu palielināšanos ir pieaugušas šo valodu implementēšanas un uzturēšanas problēmas [ITSE99]. Mēs nevaram iztikt tikai ar kompilatoru, translatoru, interpretatoru vai kādu citu samērā standartisku rīku, kas nodrošina darbu ar doto valodu. Ir pieaugusi arī nepieciešamība pēc papildus rīkiem, kas palīdzētu izpildīt strauji kāpjošās programmatūras kvalitātes prasības.

Diemžēl valodas formālie semantikas apraksti (specifikācijas) pamazām zaudē savas agrāk stiprās un pārliecinošās pozīcijas galvenokārt dēļ savas vājās spējas risināt praktiskas problēmas [Sch96, Sch97, Lou97, Paa95]. Protams, ir izņēmumi, bet tie ir risinājumi ļoti konkrētai problēmai.

Viens no risinājumiem, lai uzlabotu formālu semantiku izmantošanu, ir *Rīku-orientēta pieeja semantikām* (*Tool-oriented approach to semantics*), kas pamazām gūst atbalstu [HK00]. Semantikas, valodas un rīki lēnām un stabili attālinās viens no otra. Katra virziena pārstāvji risina savas iekšējās problēmas savā apgabalā un maz uztraucas par citiem virzieniem. *Rīku-orientēta pieeja semantikām* mēģina satuvināt šos trīs virzienus vadoties pēc principa – izveidot semantiskās definīcijas daudz lietojamākas, lai varētu no tām ģenerēt uz valodu bāzētus rīkus, cik vien iespējams daudz.

Datoru pielietošana un to izmantošanas vide kļūst arvien dinamiskāka. Kā mēs varam formāli pierakstīt tādus semantiskos aspektus kā dalītā datu apstrāde, paralēlā skaitļošana, sadarbspēja un citus? Varbūt ir laiks veltīt vairāk pūles interpretatoriem (un tiem piemērotākiem semantikas pierakstiem), kas kļūst arvien populārāki un samazina daudzus trūkumus, izmantojot tikai kompilatorus?

Daudzus gadus semantiku veidotāji meklē ceļus, kā sadalīt valodu atkalizmantojamās komponentēs, un joprojām nav skaidrs, kāda metodoloģija ir vislabākā praksē – formāla pieeja vai daļēji formāla pieeja? Un kas pateiks, kura klasiskā formālo semantiku paradigma ir vislabākā? Bet varbūt jāmeklē jaunas paradigmas vai esošo kombinācija?

Šajā nodaļā tiek dots ieskats vienā no iespējamajiem veidiem, kā iepriekš minētās problēmas varētu risināt, atbalstot ideju par *Rīku-orientētu pieeju semantikām.*

5.2 Sintakse un semantika - savstarpēji sadarbojošos komponenšu kopa

5.2.1 Programmēšanas valodas aprakstīšana

Programmēšanas valoda ir mākslīgs līdzeklis, kā cilvēkam komunicēties ar datoru. Ar programmēšanas valodas palīdzību tiek pierakstīti algoritmi problēmu risināšanai un šo pierakstu dators spēj saprast un izpildīt.

Līdzīgi kā dabīgā cilvēku valoda arī programmēšanas valoda sastāv no trīs komponentēm jeb aspektiem: *sintakses*, *semantikas* un *pragmatikas* [Pag81, SK95]. Sintakse nosaka virspusējo (redzamo) valodas formu. Semantika nosaka valodas pamata jēgu. Savukārt pragmatika nosaka valodas praktisko lietošanu. Valodas sintakse var tikt formalizēta ar gramatikas palīdzību. Valodas semantika ar lielākām vai mazākām sekmēm arī var tikt formalizēta. Abi šie formālismi kopā nosaka formālo programmēšanas valodas specifikāciju.

Formālismi sintakses pierakstīšanai ir pietiekoši labi attīstīti. Skanēšanas un parsēšanas teorijas, atribūtu analīze nodrošina līdzekļus ne tikai līdzekļus sintaktiskās analīzes veikšanai, bet var arī uzkonstruēt visu kompilatoru. Teorija primāri balstās uz regulārām izteiksmēm un bezkonteksta gramatikām (atribūtu gramatikas).

Diemžēl atribūtu gramatiku lietošana nopietnu kompilatoru vai citu rīku izveidē nav kļuvusi par ierastu praksi. Iespējams, ka tas ir saistīts ar sarežģītību izteikt visu valodas semantiku kā atribūtu gramatiku un producēt labi nooptimizētu gala kodu (runa ir par lielu valodu un/vai netriviālu semantiku). Daudzus specifiskus rīkus ir ļoti grūti vai pat neiespējami ģenerēt.

Semantikas definēšanas metodes parasti iedala trīs lielās grupās: *operacionālā semantika*, *denotacionālā semantika* un *aksiomātiskā semantika* [SK95, Gun92, Pag81]. Pēdējā laikā klasisko formālismu kritika pieaug, jo tiek konstatētas problēmas to pielietošanai praksē. Kā rezultāts ir jaunu formālismu rašanās, piemēram, *darbības semantika* (*action semantics*) jau daudz vairāk atbilst lietotāju prasībām (atgādina denotacionālo semantiku, bet lieto operacionālās semantikas jēdzienus).

Ja apvieno pārsvarā valdošos lietotāju viedokļus, tad

1. ir labs apmierinājums, tiekot galā ar programmēšanas valodas sintaksi;
2. vēl ir daudz jāpiestrādā semantikas pierakstīšanas jomā, jo formālismi ir grūti aptverami un pielietojami praksē;
3. valodas pragmatiskie aspekti maz ir pētīti, un ir neskaidra to ietekme uz valodas formālismu.

Galvenā problēma, lietojot programmēšanas valodu formālās specifikācijas praksē, ir, ka specifikācijas kļūst ļoti sarežģītas, neskaidra to pārvaldība, reizēm nespēja izteikt visas vajadzības, un galu galā – kas pārbaudīs un pierādīs specifikācijas korektību?

Apskatot labāko pieredzi kompilatoru būvē, var teikt, ka lielākā daļa komerciālo kompilatoru, interpretatoru vai citu rīku, kas balstās uz programmēšanas valodu, ir izstrādāta, nelietojot vispār nekādu formālismu. Tikai pirmajās fāzēs (skanēšana un parsēšana) reizēm tiek lietots kāds formālisms [Lou97]. Visi citi formālismi pārsvarā tiek izstrādāti un lietoti datorzinātnieku kopienā pētnieciskos un zinātniskos nolūkos.

5.2.2 Sintakses un semantikas sadalīšana sastāvdaļās

Bieži vien, risinot lielas un sarežģītas problēmas, mēs lietojam *skaldi un valdi* principu. Lielā problēma tiek sadalīta mazākās apakšproblēmās, kuras mēs jau varam pārvaldīt daudz vieglāk. Kompilatoru būves teorija lieto šo principu un sadala kompilāciju vairākās fāzēs. Formālās specifikācijas atbalsta leksisko analīzi un parsēšanu, bet neatbalsta tālākās fāzes, piemēram, optimizāciju.

Apskatīsim valodas aprakstu vēlreiz, mēģināsim dalīt to mazākās daļās un skatīties, vai nevar kaut ko no tā iegūt. Tradicionāli pirmais lēmums ir atdalīt sintaksi no semantikas, savukārt semantika tiek sadalīta divās daļās: statiskā semantika un dinamiskā (izpildes laika) semantika.

Vai sintaksi un semantiku var atdalīt pilnīgi? Vai abas var tikt reprezentētas ar funkcionējošiem objektiem? Kas ir ieejā un izejā tādiem objektiem? Kāda ir saskarne starp tiem? Vai mēs varam izveidot vairākus semantiskos objektus vienai sintaksei? Vai var izveidot semantisku objektu, kas ir pielietojams vairākiem sintaktiskiem objektiem? Vai mēs varam kombinēt kopā vienu sintaktisko objektu ar vairākiem semantiskajiem objektiem vienlaicīgi, tādējādi iegūstot jaunu specifisku semantiku? Vai šīs prasības ir tikai utopija, vai arī vismaz daļēji sasniedzams mērķis? Šie jautājumi bija motivējošais spēks, kas noveda pie tālāk aprakstītās metodoloģijas.

Lai sasniegtu izvirzīto mērķi, mēs sintaksi un semantiku dalām tālāk mazākās sastāvdaļās, izdalām atkalizmantojamās komponentes un piedāvājam mehānismu, kā to visu salikt kopā vienotā strādājošā specifikācijā.

Sintaktiskās komponentes ir daudzmaz redzamas uzreiz: bāzes elementi (piemēram, terminālu vai neterminālu simboli, ja tiek izmantota klasiskā programmu parsēšana, t.i. virsotnes sintaktiskā izveduma kokā), kas savienoti ar kādām relācijām (piemēram, šķautnes sintaktiskā izveduma kokā).

Daudz neskaidrāka situācija ir semantikas sadalīšanā. Mēs uzskatām, ka programmēšanas valodas pragmatika ir ļoti maz izmantota un ka viens no virzieniem semantikas sadalīšanā varētu tikt balstīts tieši uz pragmatiku. No šī viedokļa skatoties, semantika tālāk ir jādala balstoties uz tās *semantiskajiem aspektiem.*

Semantiskais aspekts ir loģiska semantikas sastāvdaļa, kas nosaka kādu samērā elementāru semantisku īpašību un kuru var izteikt ar praktiskā lietošanā saprotamiem jēdzieniem. Lūk, daži semantisko aspektu piemēri:

1. programmas vadības plūsmas pārvaldība (piemēram, cikli, zarošanās);
2. komandu vai izteiksmju izpilde (piemēram, bāzes funkcija – divu skaitļu saskaitīšana, vērtības piešķiršana mainīgajam);

3. simbolu apstrāde (piemēram, konstantes, mainīgie);
4. kāda specifiska, skaista, īpaši noformēta programmas drukāšana;
5. dinamiska statistikas vākšana (piemēram, funkciju izsaukšanas uzskaitīšana);
6. simboliskā izpilde;
7. specifiska programmas instrumentēšana.

Tālāks solis semantiskajā sadalīšanā ir nodalīt tās semantiskās daļas, kuru aprakstam var izmantot jau praksē populārus un viegli saprotamus līdzekļus. Tas nozīmē, ka semantiskā aspekta atsevišķas daļas varam aplūkot kā melnās kastes, kuru funkcionalitāte ir samērā skaidra un kurām var pat formāli aprakstīt to korektumu.

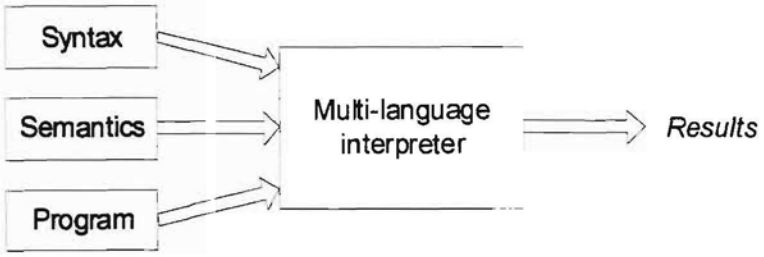
Faktiski šīs melnās kastes ir abstraktais datu tips. Piemēram, *Steks*, *Rinda*, *Binārais koks*, *Kopa*, *Vārdnīca* ar tās abstraktajām operācijām vai *Simbolu tabula* (*Symbol table*), kas ir samērā definēta datu struktūra ar operācijām kompilatoru teorijā.

5.3 Daudzvalodu interpretatora jēdziens

Daudzvalodu interpretators (MLI no Multi-Language Interpreter) ir rīks (programma), kas kā ieejas parametrus saņem valodas sintaksi, valodas semantiku, šajā valodā uzrakstītu programmu un veic darbības, kas noteiktas padotajā programmā saskaņā ar padoto semantiku. Daudzvalodu interpretatora jēdziens tika ieviests [AAB96a un AAB96b]. Vārds *interpretators* tika izvēlēts tāpēc, ka atbilstoši uzdotajai semantikai tieša darbība tiek veikta uzreiz. Atkarībā no padotās semantikas MLI var "spēlēt" interpretatora, kompilatora, translatora vai kāda cita rīka lomu.

5.3.1 MLI darbības pamatprincipi

MLI ideja ir redzama Zīm. 39. Koncepcija paredz, ka vienai sintaksei var tikt sagatavotas daudzas semantikas, un vienu un to pašu semantiku var izmantot arī dažādām sintaksēm. Protams, ka sintakses un semantikas apraksts ir jāpārvērš datoram saprotamā formā – izpildāmā kodā.



Zīm. 39 Daudzvalodu interpetatora jēdziens

MLI tika ieviests rīku, kas balstās uz kādu valodu, ērtākai veidošanai un nav domāts esošo populāro programmēšanas valodu kompilatoru vai interpretatoru aizstāšanai. Tā pamatmērķis ir ļaut izveidot papildus rīkus esošajām valodām, veidot visus nepieciešamos rīkus jaunām valodām (ieskaitot kompilatoru, interpretatoru vai

translatoru) un uzturēt atkalizmantojamu komponenšu bibliotēku, kas ļautu visus rīkus izveidot pēc vienotiem principiem.

MLI var interpretēt (bet faktiski iespējams kompilēt, translēt vai veikt citu darbību) daudzas *virtuālas valodas*, ko nosaka saņemtā valodas sintakse un semantika. No MLI skatu punkta katru programmēšanas valodu nosaka pārītis <syn, sem>, kur *syn* ir programmēšanas valodas sintakse un *sem* ir programmēšanas valodas semantika. Nav izslēgts, ka patvaļīgi izvēlēta sintakse *syn* un semantika *sem* nav savietojamas, taču MLI var mēģināt izpildīt programmu atbilstoši dotajai specifikācijai. Viens no svarīgākajiem uzdevumiem ir izvēlēties sintakses un semantikas reprezentācijas veidus un to savstarpējās saistības definēšanu, lai pēc iespējas minimizētu nesaprātīgu sintakses un semantikas pārīšu izmantošanu.

Sākotnējo realizācijas ideju var izteikt sekojoši:

- Sintakse pēc iespējas tiek nodalītas no semantikas;
- Sintaksi nosaka gramatika;
- Balstoties uz gramatiku, ģenerē rīku, kas spēj no programmas uzražot programmas sintaktiskā izveduma koku;
- Semantiku nosaka semantisko likumu kopa, un katru likumu piekārto kādam gramatikas simbolam (piemēram, terminālim vai neterminālim). Piekārtošanu veic simbolam nevis gramatikas izveduma likumam;
- Semantiskos likumus translē uz izpildāmām komandām, tiek izveidots semantiskais objekts, kas nodrošina to reālu izpildi;
- Eksistē papildus bibliotēkas, kas nodrošina sintaktiskā izveduma koka apstaigāšanu vai kas ļauj vienkāršot semantikas likuma pierakstus ar gatavām semantiskām konstrukcijām (piemēram, izpildes laika atmiņas objektu pārvaldība);
- Sintaktiskā izveduma koks tiek apstaigāts un tiek izpildīti mezglam atbilstošie semantiskie likumi.

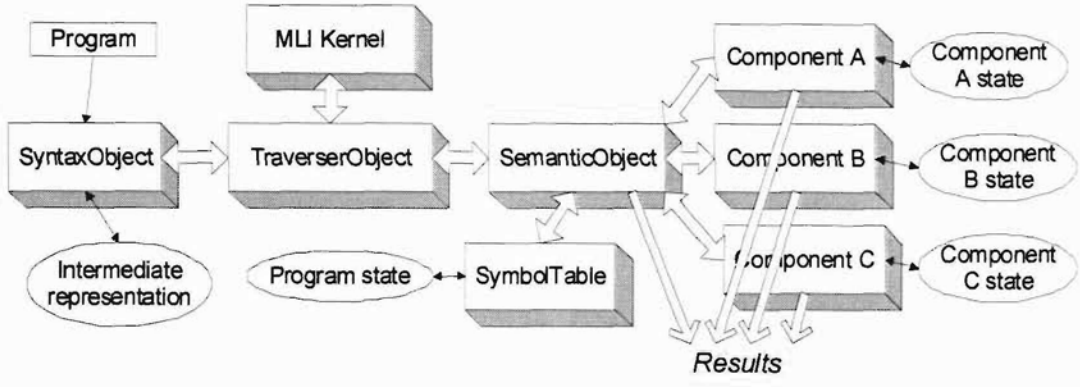
Konceptuāli runājot, izpildes laikā mēs parsējam ieejas valodu atbilstoši gramatikai, būvējam sintaktiskā izveduma koku, apstaigājam koku atbilstoši vēlamajai stratēģijai un izpildām mezgliem piekārtotos semantiskos likumus.

Pieeja ideoloģiski ir ļoti līdzīga principam *uzbūvē koku, saglabā to un veic nepieciešamās manipulācijas, to apstaigājot* (*build a tree, save a parse and traverse it* [Cla99]). Analoģiska pieeja tiek ieteikta arī [GHJV95], izmantojot *Visitor pattern* (papildus ieteicams izmantot arī *Composite pattern* un *Interpreter pattern*). Galvenās atšķirības ir semantisko likumu izveidošanas ideoloģijā.

MLI jēdziens jau sākotnēji balstās uz principu, ka mēs varam sagatavot vairākas semantikas vienai un tai pašai sintaksei, kā arī izmantot vienu semantiku vairākām valodām ar dažādu sintaksi. Sintakses un semantikas aprakstus (specifikācijas) ir nepieciešams pārveidot datoram izpildāmā formā pirms MLI sāk darbu.

Tehniskās realizācijas arhitektūras var būt dažādas. Viena no iespējamajām arhitektūrām ir dota Zīm. 40. Sintaksi šajā gadījumā reprezentē objekts SyntaxObject, bet semantiku – vairāki objekti: TraverserObject, SemanticObject, SymbolTable, Component (mūsu piemērā komponentes A, B un C). To visu kopā saista MLI

centrālā komponente MLI Kernel, kas nodrošina sākotnējo sintakses un semantikas objektu sasaisti.



Zīm. 40 MLI darbības laika arhitektūra

Katrai no minētajām komponentēm varētu būt daudzas implementācijas - ar atšķirīgu semantisko uzdevumu vai atšķirīgu fizisko implementāciju. Līdz ar to paveras iespēja kombinēt sintakses un semantikas dažādos veidos gan pēc būtības, gan pēc implementācijas. Protams, ka uzreiz rodas jautājums par sintakses un semantikas saderību, lai nebūtu nesakarīga interpretācija.

Nonākšana no sintakses un semantikas aprakstiem uz darbināmām komponentēm var būt gan statiska (pirms programmas izpildes), gan arī dinamiska (programmas izpildes laikā). Dinamiskā ģenerācija ir daudz grūtāka, jo visiem ģenerācijas etapiem ir jābūt automātiskiem.

Vēlākā laikā mērķi attiecībā uz MLI izveidi tika koriģēti. Tika nolemts veidot dinamisku rīku veidošanas tehnoloģiju un izstrādes vidi, tā vietā, lai uzrakstītu vienu universālu MLI. Uzsvars tika pārvietots uz ideoloģijas izstrādi, kā izveidot atkalizmantojamas sintakses un semantikas uzdošanas komponentes un kā no tām *salikt kopā* nepieciešamo rīku. Ar rīku šajā gadījumā tiek saprasta programma, kas strādā ar jebkādu strukturētu informāciju tai skaitā arī programmām, kas uzrakstīta kādā programmēšanas valodā.

5.3.2 Izgudrot, aizņemties vai adaptēt?

Izgudrot velosipēdu vēlreiz nav īpaši prātīgi. Saprātīgāk šķiet ir uzprojektēt un sakomplektēt sev tādu velosipēdu no pieejamajām detaļām, kas atbilst tieši mūsu vajadzībām. Mēģināsim apskatīties un apkopot formālismus, paradigmas, idejas, kas varētu noderēt mūsu problēmu risināšanā, veidojot MLI konstruēšanas tehnoloģiju un vidi.

Pirmām kārtām mūs interesē formālismi, kas apraksta valodas sintaksi un ļauj būvēt rīkus programmas iekšējās reprezentācijas iegūšanai. Tipiskām programmēšanas valodām šī problēmu klase mums nepieciešamajā līmenī jāuzskata par atrisinātu. Taču mums jāņem vērā, ka eksistē daudzas citas valodu klases, kurām tiek izmantotas netradicionālas gramatikas un programmas iekšējā reprezentācija ir nevis koks, bet, piemēram, grafs. Kā piemēru var minēt diagrammatisko vizuālo valodu klasi [FNT+97] (pārstāvju piemēri: Petri tīkli, ER diagrammas, UML klašu diagrammas, Stāvokļu diagrammas).

Nākošā interesējošā formālismu grupa ir atribūtu gramatikas (AG), jo pirmajā acu uzmetienā šķiet, ka AG idejiski ir tuvs mūsu metodei. Ir samērā grūti reālā praksē pielietot klasiskās atribūtu gramatikas [Knu68], taču eksistē daudzi uzlaboti varianti, kas ir daudz pielietojamāki. AG organizacionālās paradigmas, kas visvairāk dod vielu pārdomām, kā labāk saistīt valodas sintaksi ar semantiku, ir *Attribution paradigms*, *Structured AG*, *Modular AG*, *Object-oriented AG* [Paa95].

Kastens (Kasten) un Vaits (Wait) savos darbos [KW94] apgalvo, ka AG nepopularitāte slēpjas tās stilā, traucējot modulāro dekompozīciju un specifikāciju atkalizmantošanu. Viņi piedāvā savus uzlabojumus AG, tādus kā: attālinātu pieeju atribūtiem, mantojamību, atribūtu moduļus. Tās pašas problēmas ir risinātas, veidojot vienotu federālu vidi LISA [MLAŽ98, MŽLA99, MLAŽ00]. Bez tam mums vajadzētu domāt arī par dalīto skaitļošanu reālā laikā (piemērs ir *Communicating Timed AG* [MT98]).

Jau tika pieminēta popularitāti iegūstoša metode *parse and traverse* [Cla99], kas nozīmē izveidot programmas iekšējo reprezentāciju, apstaigāt to un veikt atbilstošus skaitļojumus katrā mezglā. Šajā pašā virzienā attīstās arī jaunu programmatūras izstrādes paraugu (*Design Patterns* [GHJV95]) izveide. Kā piemēri ir minami *Visitor pattern* paveidi vai uzlabojumi: [OW99] apraksta valodu, kas ļauj specificēt apstaigāšanu, [Küh97] piedāvā *Translator pattern* valodu procesēšanas rīku veidošanai. Jāņem vērā, ka var eksistēt arī netradicionālas apstaigāšanas stratēģijas [BM99].

Ar doto metodi cieši ir saistīti pētījumi arī tādos novirzienos kā *Adaptive programming*, *Component collaboration*, *Aspec-oriented programming*, u.c. [ML98]. Bieži daudzas problēmas tiek risinātas ar specializētu rīku izveidi [Slo97].

Un beidzot jāatzīmē, ka vairāki semantiku formālismi daudzas lietas līdz galam formāli neapraksta un izmanto jēdzienus vai abstraktus datu tipus, kas lasītājam ir intuitīvi saprotami. Jēdzienam *abstrakts datu tips* (ADT) eksistē daudzas definīcijas un interpretācijas [Tuc97].

Mēs pieturēsimies pie sekojošas definīcijas: Abstrakts datu tips ir kolekcija ar datu tipu un vērtību definīcijām un operācijām ar šīm definīcijām, un kas uzvedas kā primitīvs datu tips. Šī programmatūras izstrādes pieeja sadala problēmu komponentēs, identificējot to publisko interfeisu un privāto implementāciju. Netriviāls, bet tomēr tipisks ADT piemērs kompilatoru teorijā ir Simbolu tabula (*Symbol table*) [ASU86, FL88, SK95].

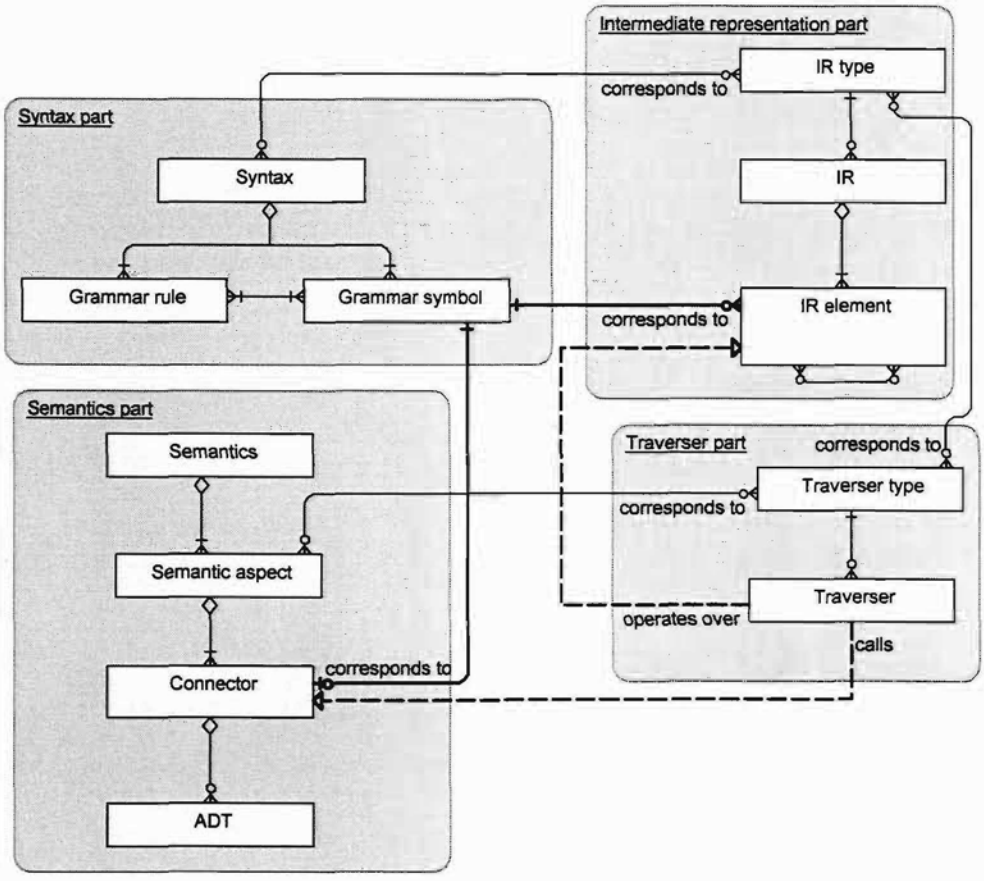
5.4 Sintakse un semantika daudzvalodu interpretatoram

Programmēšanas valoda ir mākslīgs līdzeklis, kā cilvēkam sazināties ar datoru, fiksēt algoritmus un zināšanas. Līdzīgi dabīgajai cilvēku valodai programmēšanas valoda sastāv no sintakses, semantikas un pragmatikas [Pag81, SK95]. Mūsu problemātikai ir būtiskas pilnīgi visas sastāvdaļas. Parasti mazāk skartā pragmatika (nosaka valodas praktisko lietošanu) mums ir būtisks pamats semantiku definēšanai.

MLI kontekstā sintakse un semantika ir jāaplūko definēšanas un izpildes stadijās. Mērķis ir izpildes laika komponentes, kas ir brīvi maināmas un spēj sadarboties savā starpā. Lai pie tām nonāktu, jāaplūko sintakses un semantikas definēšanas principi un pārejas etapi (koda ģenerācija) uz izpildes laika komponentēm.

Mūsu pamatprincips ir sadalīt sintaksi un semantiku mazās sastāvdaļās un ar pietiekoši vienkāršu un saprotamu principu savietot tās kopā. Skatoties no populārāko formālismu viedokļa, mūsu pieejai vistuvāk ir atribūtu gramatikas: definēšanas fāzei tuvs ir princips *Semantic aspect = Module*, bet izpildes fāzei - *Nonterminal = Procedure* [Paa95]. Bez tam definēšanai intensīvāk tiek izmantota valodas pragmatika ar semantikas dalījumu pa semantiskajiem aspektiem.

Zīm. 41 dots tālāk aprakstītajos principos iesaistīto jēdzienu savstarpējās attiecības.



Zīm. 41 Izpildē iesaistīto komponenšu attiecības

5.4.1 MLI sintakse

Formālismi, kas tiek galā ar valodas sintaktisko aspektu, ir ļoti labi attīstīti. Skanēšanas, parsēšanas teorija un atribūtu analīze nodrošina mūs ne tikai ar līdzekļiem veikt sintaktisko analīzi, bet arī uzģenerēt pat visu kompilatoru. Tādi jēdzieni vai rīki, kā *finite automata, regular expression, context-free grammar, attribute grammar (AG), Backus-Naur form (BNF), extended BNF (EBNF), Lex (also Flex), Yacc (also Bison), PCCTS* ir labi zināmi un akceptēti datorzinātņu sabiedrībā.

Mēs atbalstām, ka tiek izvēlēts vēlamais formālisms un tam jau gatavie ģeneratori. Pamatuzdevums būtībā ir uzģenerēt programmu, kas dotajā sintaksē uzrakstītu programmu spēj transformēt uz iekšējo reprezentāciju (*intermediate representation* - IR). Vienīgi nepieciešams pie ģenerētā koda pievienot funkciju

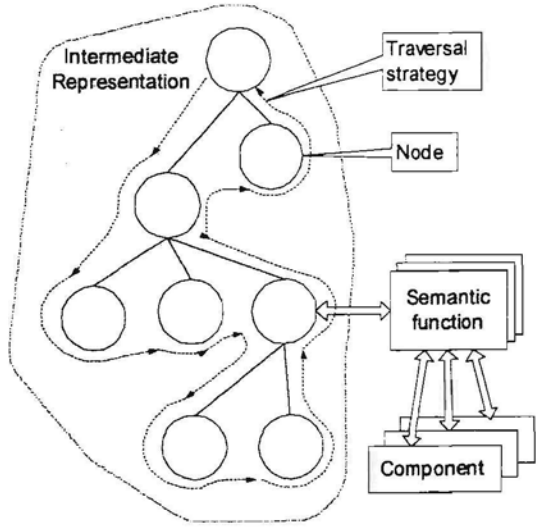
bibliotēku, kas spēj manipulēt ar iegūto IR un nokompilēt visu kodu. Iegūstam SyntaxObject (Zīm. 40), kas spēj iegūt IR un nodrošina manipulēšanu ar to (būtībā pietiek tikai ar visu atribūtu nolasīšanas iespēju, ja nav nepieciešama koka tālāka transformācija).

Jāatgādina, ka bez ierastajām imperatīvajām valodām eksistē arī dažādi citi valodu tipi, piemēram, diagrammatiskās valodas (Petri tīkli, ER diagrammas, Stāvokļu diagrammas, vizuālās programmēšanas valodas), kurām ir citi formālismi (*SR Grammars*, *Reserved Graph Grammar*) un apstrādes veidi [FNT+97, ZZ97]. Mūsu ideoloģija no tā principiāli nemainās.

5.4.2 MLI semantika

Semantikas izpildei izvēlētais princips *parse and traverse* nosaka, ka divas svarīgākās lietas ir traversēšanas stratēģija un semantiskās funkcijas, kas jāizpilda apmeklējot mezglu (Zīm. 42). Līdz ar to centrālo vietu ieņem abstrakta komponente Traverser (TraverserObject Zīm. 40), kas pārvalda mezglu apmeklēšanas kārtību, nodrošina semantiskās funkcijas ar informāciju no IR un kopumā ir galvenais MLI motors. Tai pat laikā mēs atļaujam semantiskajās funkcijās ielikt komandas, kas liek Traversētājam mainīt tekošo atrašanās vietu (Visitor pattern tā ir problēma, ko grūti risināt [Vis01]).

Otra būtiskākā komponente ir SemanticObject (Zīm. 40), kas satur visas nepieciešamās semantiskās funkcijas un nodrošina izpildes vidi. Programmas izpildes laikā sintakses un semantikas attiecības atainotas Zīm. 42.



Zīm. 42 Sintakses un semantikas atbilstība izpildes laikā

Semantiskajām funkcijām jābūt pēc iespējas mazām, uzrakstītām kādā meta-valodā, izmantojot augsta līmeņa izteiksmes līdzekļus, lai tās būtu samērā viegli saprotamas un pārbaudāmas. Šo principu ievērot palīdz abstraktu komponenšu lietošana, kuru semantisko pielietojumu bieži vien var saprast bez papildus skaidrojumiem. Zīm. 42 abstraktajām komponentēm jau ir konkrēta implementācija Component. Lūk, šis apgalvojums par ADT izmantošanu var izraisīt iebildumus formālo semantiku

aizstāvju vidū, jo abstraktās komponentes varētu nebūt matemātiski precīzi aprakstītas. Tai pat laikā daudzās formālajās semantikās tomēr tiek lietoti tādi pašsaprotami jēdzieni, kā Stack vai Symbol table.

Ieviesīsim jēdzienu sintaktiskais elements, kas ir gramatikas simbols ar vārdu (piemēram, netermināļa simbols ar vārdu vai termināļa simbols ar vārdu). Ņemot vērā, ka ir dažādi sintaktisko elementu tipi un dažādas to apmeklēšanas stratēģijas, mēs vēlamies atšķirt šīs dažādās elementu apmeklēšanas. Tam mēs ieviešam traversēšanas aspekta jēdzienu. Piemēram, mēs varam atsevišķi apstrādāt ierašanos mezglā no vecāka mezgla (*PreVisit*) un ierašanos mezglā no bērna mezgla (*PostVisit*). Tādējādi mums ir iespēja izveidot semantiskās funkcijas, balstoties ne tikai uz sintaktiskā elementa vārdu, bet arī uz ierašanos šajā elementā (traversēšanas aspektu) (Zīm. 42).

Semantiku kaut kādā nozīmē var uzskatīt arī par semantisko funkciju (koda gabalu) kopu. Izpildāmās semantikas būtiskāko daļu var attēlot ar matricu (Tabula 1). Tajā redzam, ka sintakses elementam atkarībā no *traversing aspect* ir piekārtots kāds izpildāmais kods (□) vai tā vispār nav (λ). *n* ir atkarīgs no sintakses lieluma (piemēram, termināļu un netermināļu kopsumma) un *m* no traversēšanas stratēģijas sarežģītuma (parasti m ir 1..3). Eksperimenti rāda, ka tabulā lielākā daļa ir tukšas funkcijas (λ).

Tabula 1 Sintakses elementu un semantikas funkciju atbilstības matrica

Syntax Element (SE)	**Traversing Aspect (TA)**				
	TA_1	TA_2	TA_3	...	TA_m
SE_1	□	□	λ	...	λ
SE_2	λ	λ	□	...	λ
SE_3	λ	λ	λ	...	□
...	...	...	...	...	λ
SE_n	□	λ	λ	...	λ

Semantisko funkciju identificēšana notiek pēc sintakses elementa un traversēšanas aspekta. Tehniskās realizācijas var būt ļoti dažādas, bet vēlams, lai funkcijas izsaukums būtu ar konstantu sarežģītības kārtu, pašas funkcijas izpildi neņemot vērā.

Un nu ir nonākts pie svarīgākās problēmas - kā atsevišķajām semantiskajām funkcijām nodrošināt korektu sadarbību un vai tās vispār ir atkalizmantojamas? Tālāk ir izskaidrota ideja, kā definēt semantiku un nonākt līdz augstāk apskatītajai matricai, t.i. no semantikas apraksta ģenerēt izpildāmo semantiku.

5.5 Semantiskie aspekti un abstraktās komponentes

5.5.1 Semantiskie aspekti

Praksē programmēšanas valodas lietotājiem pārsvarā tiek pasniegtas, izmantojot valodas pragmatiku, t.i. ar piemēru palīdzību tiek parādīts, kā atsevišķas valodas konstrukcijas tiek lietotas (piemēram, mainīgā (*variable*) definēšana, vērtības piešķiršana mainīgajam, zarošanās konstrukcija, cikla konstrukcija, funkcijas

lietošana). Mēs šīs atsevišķās valodas konstrukcijas saucam par valodas semantiskajiem aspektiem.

Rezultātā mūsu izvēlētā semantikas specificēšanas metode ir definēt semantiku kā kopu ar savstarpēji saistītiem semantiskajiem aspektiem. Lūk, daži piemēri tipiskākām imperatīvo programmēšanas valodu semantisko aspektu grupām un to pārstāvjiem: komandu vai izteiksmju izpilde (mainīgā definēšana, vērtības piešķiršana mainīgajam, aritmētiskas izteiksmes izpilde), programmas simbolu apstrāde (mainīgais, konstante, tips), programmas kontroles plūsmas pārvaldība (cikls ar skaitītāju, nosacījuma cikls, nosacījuma zarošanās), vides pārvaldība (programmas simbolu redzamības apgabals). Savukārt mazāk tradicionālām semantikām var minēt sekojošas semantisko aspektu grupas: skaista izdruka, dinamiska programmas izpildes statistikas vākšana, simboliskā izpilde, specifiska programmas instrumentēšana.

Semantiskā aspekta definēšanai mēs savukārt izvēlamies operacionālo pieeju, t.i. nosakām, kas datoram ir jādara, lai realizētu doto semantisko aspektu.

5.5.2 Abstraktie datu tipi un abstraktās komponentes

Semantikas pieraksts ir veidots ar domu, ka pārsvarā viss reālais darbs tiek veikts ar abstrakto datu tipu (ADT) palīdzību. Pārējās semantikas pieraksta komponentes pārsvarā organizē ADT korektu sadarbību. Varam iedomāties, ka ADT ir komponente, kura tiek uzskatīta par melno kasti, bet kurai ir definēts interfeiss, t.i. kādas operācijas mēs varam veikt un kādus rezultātus mēs dabūsim. Piemēram, ADT var būt *Stack* (steks ir elementu saraksts, kurā elementu var pielikt vai noņemt tikai no viena saraksta gala), kuram ir definētas operācijas: *newStack* (izveidot steku), *push* (ielikt elementu), *pop* (izņemt elementu), *top* (apskatīties pēdējo ielikto elementu), *deleteStack* (iznīcināt steku).

Lietojot saprotamus vai augsta abstrakcijas līmeņa ADT, mēs varam noslēpt daudzas implementācijas detaļas, kuras semantikas veidotājam vai lasītājam varētu traucēt koncentrēties uz semantiskā aspekta loģiku un būtību. Tikai nepieciešamības gadījumā var skatīties ADT specifikāciju vai pat implementāciju. Mūsu gadījumā tipiski ADT piemēri ir *Stack*, *Queue*, *Dictionary*, *Symbol table* (kompilatoru veidošanas teorijā [ASU86, FL88] un formālajās semantikās [SK95]).

Tā kā izmantotās komponentes var būt visai sarežģītas (*E-mail*, *Graph visualization*, *Distributed communication*, *Transaction manager*, u.c.) un tiem varētu nebūt pat neformāls standarts, tad ADT vietā var lietot arī terminu *abstraktā komponente*.

Rīka izveidošanas noslēguma fāzē tiek izvēlēta visatbilstošākā ADT implementācija dotajam uzdevumam. Kā zināms, vienu un to pašu ADT var implementēt ļoti dažādos veidos atkarībā no praktiskā uzdevuma. Piemēram, laba operāciju izpildes algoritmu sarežģītības kārta, vai vienkārša, bet ļoti droša un korekta implementācija, vai algoritmi izmanto pēc iespējas maz atmiņas, vai ADT ir izstrādāts nepieciešamajā tehnoloģijā ar prasīto reālo interfeisu.

Ar šo pieeju rīku var izstrādāt iteratīvi – sākumā tiek izmantotas ADT implementācijas, kas ir funkcionāli vājākas, realizācijā vienkāršas, ātri implementējamas, bet neefektīvas rīka darbības laikā. Vēlāk implementācijas tiek uzlabotas. Piemēram, Stack (steks) var tikt realizēts tabulā nepārtrauktā atmiņā, saistītā atmiņā ar norāžu palīdzību, vai kā citādi. *Dictionary* (vārdnīca) var tikt

implementēta ar sakārtota saraksta palīdzību, ar binārā meklēšanas koka palīdzību, ar *hash* tabulu palīdzību, utt.

Jāatzīmē, ka ADT izdalīšana atsevišķi atvieglo izveidot rīku, kas strādā dalītā neviendabīgā skaitļošanas vidē. Tas nozīmē, ka ADT var būt kā dalīts objekts (*distributed object*) uz kāda tīklā esoša datora un sadarbība tiek organizēta caur kādu vēlamo tehnoloģiju un tīkla protokolu. Bez tam arī viena konkrēta ADT implementācija var sastāvēt no daudziem objektiem, kas atrodas tīklā uz daudziem datoriem, vai ADT operācija pati izlemj, cik daudz un kādus pieejamos tīkla resursus iesaistīt uzdevuma veikšanā.

ADT specifikācijas var tikt uzrakstītas ļoti formālā veidā. Vienkāršākiem ADT (piemēram, stekam, rindai vai vārdnīcai) var atrast formālas specifikācijas, kuru korektums pat ir pierādīts. Tāpat no formālajām specifikācijām var būt jau uzģenerētas korektas implementācijas.

Izvēlēto ADT specifikācijas formalizācijas pakāpe un implementācijas korektums ir atkarīgs no ADT sarežģītības, veicamā uzdevuma kritiskuma un mūsu vēlmes visu formalizēt līdz galam. Jāpiezīmē, ka praksē reti ir gadījumi, kad rīku izstrādātāji censtos pierādīt sava rīka korektumu, jo lietotāju parasti apmierina fakts, ka rīks ir lietojams viņa uzdevumu veikšanai.

Ir ieteicams semantisko aspektu aprakstīt ar meta-valodas palīdzību, pat ja jūs to netranslēsiet automātiski. Tad var translēt vai vienkārši ar roku pārrakstīt to mērķa programmēšanas valodā, izvēlēties atbilstošo implementāciju abstraktajām komponentēm, lietot nepieciešamo interfeisu, sadarbības protokolu un izpildes vidi. Vēl vairāk, abstrakto komponenšu instances var tikt apskatītas arī kā attālināti dalītie objekti heterogēnā skaitļošanas tīklā.

5.5.3 Piemēri abstraktajām komponentēm

Daļa no abstraktām komponentēm un to operācijām komentārus neprasa, piemēram, Stack (createStack, push, pop, top), Queue (createQueue, enqueue, dequeue, first). Daļa no komponentēm ir specifiskas, kurām operācijas ir nojaušamas daļēji, piemēram, E-mail (prepare, send, receive, open).

Veidojot daudzvalodu interpretatora prototipus, mēs izstrādājām mūsu tehnoloģijai piemērotu *Symbol table* variantu – MOMS (saīsinājums no *Memory Object Management System*). Tabulās (Tabula 2, Tabula 3, Tabula 4, Tabula 5 un Tabula 6) ir doti svarīgākie datu tipi (jēdzieni), ar ko operē MOMS un būtiskākās operācijas priekšstatam par šo abstrakto objektu (šajā darbā netiek aprakstīta visa MOMS ideoloģija un specifikācija). Interfeisa aprakstīšanai ir atstāta valodas C sintakse.

Praksē izrādījās, ka MOMS nodrošina ne tikai pamatprasības imperatīvu programmēšanas valodu implementācijai [DJ90], bet arī tika izmantots komerciālā lietojumprogrammā Mosaik (Sietec consulting GmbH Co. OHG, grafisks CASE rīks biznesa modelēšanai) kā galvenā objektorientētā datu krātuve.

Tabula 2 MOMS tipi

Type	Description
Constructor	Handle to an object type description
Value	Handle to a byte stream that contains an object value
Reference	Handle to a memory object
MemoryMap	Main central object that organizes other objects (other MemoryMap also)
IdentifDict	Dictionary of all identifiers (variables, constants, etc.)
TypesDict	Dictionary of all types (basic types and user defined)
FunctDict	Dictionary of all functions (basic operators, basic functions, user defined functions)
NamesTable	Compact storing of all strings
ConstrTable	Compact storing of all constructor descriptions
ValuesTable	Storing and managing of all object values
MemoryBlock	Organizing scope visibility in all dictionaries and providing memory management
...Others	Stack, queue, collection, etc.

Tabula 3 Sistēmas inicializācija un globālās operācijas

Operation	Description
MemoryMap **initialize**(Uint memoryCount)	Creates MOMS with internal parallel but related memories (MOMS)
Uint **switchMemoryTo**(Uint memNum)	We can exploit only the specific internal memory
Uint **getCurrentMemory**(void)	Returns the number of the actual memory
defineCountOfBaseTypes(Uint countOfBaseTypes)	Defines a count of the basic types of MOMS

defineBaseType(char* typeName, Uint type, Uint typeSize)	Defines base types. This interface is in C and depends on previously defined types. Some examples:
	defineBaseType("long_", LONG_, sizeof(long_));
	defineBaseType("boolean_", BOOLEAN_, sizeof(boolean_));
	defineBaseType("date_", DATE_, sizeof(date_))

defineBaseFunction(char* langFunctName, char* internalName, Uint returnType, int paramCount, ...)	Defines the base operations and functions. This interface is in C and depends on previously defined types. Some examples: defineBaseFunction("+", "PLUS", LONG_, 2, LONG_, LONG_); defineBaseFunction("day", "day", LONG_, 1, DATE_)
prepareProgramEnv(Uchar scope)	Prepares a new MemoryBlock, defines the scope (visibility) of previously defined variables, types, functions
releaseProgramEnv(void)	Releases a current MemoryBlock and all related memory in other objects (dictionaries, tables) and restores a previously defined MemoryBlock
defineAutomaticMemSwitching(Uint firstMemNum, Uint lastMemNum)	Provides for automatic switching in various functions. For instance, we look up the variable in a local memory and then in a global memory (if the variable is not founded yet).
... Others	

Tabula 4 Lietotāja definēto datu tipu definēšana

Operation	Description
ConstrPtr **createConstrArray**(Uint minIndex, Uint maxIndex, ConstrPtr ptrToElemConstr)	Defines an array type with the given dimensions and element types. Here and in other functions we can use any previously defined (or partly defined) data type.
ConstrPtr **createConstrFunct**(ConstrPtr ptrToReturnConstr, ConstrPtr ptrToParamConstr)	Defines a function type with the given parameters and return type.
ConstrPtr **createConstrName**(char* aName, ConstrPtr trToSubConstr)	Assigns a user-defined name for the given type.
ConstrPtr **createConstrPointer**(ConstrPtr ptrToSubConstr)	Defines a pointer type to the given type.
ConstrPtr **createConstrProduct**(ConstrPtr ptrToSubConstr1, ConstrPtr ptrToSubConstr2)	Creates a production of two types (establishes some relation between them). It is useful to construct a serious data structure.
ConstrPtr **createConstrRecord**(ConstrPtr ptrToSubConstr)	Defines a record data type (a set of pairs {name, type}).
... Other constructors	For instance, base data type constructors
constrArraySetMinIndex(ConstrPtr ptrToConstr, Uint minIndex)	Modifies the type description (attributes).
Uint **constrArrayGetMinIndex**(ConstrPtr ptrToConstr)	Provides details about data type attributes.
... Others	

Tabula 5 Operācijas ar mainīgajiem un līdzīgiem objektiem

Operation	Description
createVar(char* aName, ConstrPtr ptrToConstr)	Creates a variable with the given name and type.
createVar(char* aName, char* typeName)	Creates a variable with the given name and type name.
createLiteral(char* aName, ConstrPtr ptrToConstr)	Creates a literal (constant) with the given name and type.
Ref **createRef**(ConstrPtr ptrToConstr)	Creates an object without a name with the given type, for instance, internal loop counter, return value of function.
ConstrPtr **getConstrRef**(char* typeName)	Returns pointer to type with the given type name.
createSynonym(char* aName, Ref aRef)	Creates another reference by name to the existing object.
... Other	

Tabula 6 Operācijas ar vērtību

Operation	Description
putValue(Ref aRef, char* aValue)	Sets a new value for the object.
char* **getValue**(Ref aRef)	Returns a value for the object.
char* **createDynamicValue**(ConstrPtr ptrToConstr)	Provides dynamic memory allocation for the object given by type.
deleteDynamicValue(char* aValue)	Releases dynamically allocated memory.
setValueProtectionOn(char* aName)	Protects a value of the given object against modification, for instance, protects constants.
setValueProtectionOff(char* aName)	Takes off a value protection.
gotoArrayElementConstr(Ref& aRef, Sint index)	Sets a virtual mark to element constructor and to a given array element value. aRef is modified, it refers to the array element.
gotoNameConstr(Ref& aRef)	Moves the virtual mark to the name constructor.
gotoPointerConstr(Ref& aRef)	Moves the virtual mark to the pointer subconstructor and to the start of value.
gotoProductionLeftConstr(Ref& aRef)	Moves the virtual mark to the left subconstructor and to the start of the corresponding value.
gotoProductionRightConstr(Ref& aRef)	Moves the virtual mark to the right subconstructor and to the start of the corresponding value.
gotoRecordConstr(Ref& aRef)	Moves the virtual mark to the start of record.
gotoNameInList(Ref& aRef, char* aName)	Moves to the appropriate type and value (list of named types linked by productions) E.g., search a field in the user-defined structure.
... Other	

Svarīga komponente ir Traverser (TraverserObject Zīm. 40). Tās pamatuzdevums ir nodrošināt nepieciešamās traversēšanas stratēģijas realizāciju, iespēju mainīt tekošo programmas izpildes punktu un sadarbību ar sintakses objektu (informācija par IR mezgliem, IR modificēšanas līdzekļi).

Tālāk dotajos piemēros ir izmantota *depth-first left-to-right* traversēšanas stratēģija (Zīm. 43). Šajā traversēšanā ir trīs apmeklēšanas (*visit*) aspekti: *Visit* (koka lapām, t.i. terminālļiem (*terminals*)) un *PreVisit*, *PostVisit* (pārējiem koka mezgliem, t.i. neterminālļiem (*nonterminals*)). Semantisko funkciju definēšanā ir izmantotas sekojošas funkcijas: *NodeValue()* atgriež pašreizējā terminālļa vai neterminālļa vērtību (vērtība no tekošā IR mezgla), *goSiblForw(aName)* un *goSiblBackw(aName)* nodrošina pašreizējā mezgla maiņu uz mezglu ar vārdu *aName*, meklējot to starp kaimiņiem, ejot uz priekšu vai atpakaļ.

```
Traverse(node P)
   if IsLeaf(P)
      Visit(P)
   PreVisit(P)
   for each child Q of P,
   in order, do
      Traverse(Q)
   PostVisit(P)
```

Zīm. 43 Koka apstaigāšanas algoritma piemērs

Komponenšu interfeisus (SyntaxObject, TraverserObject, SemanticObject Zīm. 40) var aprakstīt arī ar domēnspecifiskas valodas palīdzību. Tad iekšējās reprezentācijas iegūšanu, manipulēšanu ar to, kā arī darbu ar simbolu tabulu iespējams kompilēt saistīti, veicot augsta līmeņa optimizāciju un korektuma pārbaudi. Piemērs šādai sistēmai MAGIC ir dots [Eng99].

Arī traversēšanas stratēģijas aprakstīšanai var izveidot domēnspecifisku valodu. Tas ir aktuāli, ja stratēģija ir netriviāla un ir atkarīga no sintakses elementiem un programmas stāvokļa. Traversēšanas problēmas un iespējamo risinājumu ar *Visitor pattern* palīdzību piedāvā [OW99]. Traversēšanas stratēģija varētu būt implementēta pēc iespējas neatkarīga no sintakses un organizēta ar paraugu palīdzību [Vis01]. Savukārt līdzās ierastākām traversēšanas stratēģijām var būt arī mazāk tradicionālas, piemēram, stratēģija programmas reversai izpildei [BM99].

5.5.4 Semantiskā aspekta definēšana

Semantiskos aspektus ērtāk ir definēt ar grafisko diagrammu palīdzību (piemēram, Zīm. 44 - Zīm. 48). To pašu uzreiz var uzrakstīt arī tikai tekstuālā formā kā meta-programmu vai izpildāmu kodu mērķa programmēšanas valodā. Diagramma satur semantikas aspektam būtiskos sintaktiskos elementus kā grafiskus simbolus, kas var būt dažādi dažādām sintaktisko elementu klasēm (tālāk darbā ir vairāki piemēri).

Ja mums ir būtiska secība, kādā ir jāapmeklē semantikas aspektā iesaistītie sintaktiskie elementi, tad traversēšanas secību attēlojam ar bultiņu palīdzību.

Darbības, kas datoram ir jāveic, apmeklējot aspekta mezglu, sauksim par *semantic action*. Faktiski *semantic action* ir semantiskās funkcijas analogs, vienīgi to pieraksta meta-kodā un veiktās darbības attiecas tikai uz šo aspektu. *Semantic action* tiek attēlots kā kastīte ar meta-kodu, kas ir pievienota sintakses elementam. Diagrammā jāspēj atšķirt, kādam Traversēšanas aspektam katrs *semantic action* piekārtots.

Semantiskajā aspektā definē visu abstrakto datu tipu instances, kas nepieciešamas tā izpildē. Mēs tam izmantojam bloku ar atslēgas vārdiem IMPORT GLOBAL.

Aspekta uztveramībai var izmantot arī papildus grafiskos elementus, kas atvieglo uztveramību, bet faktiski nav nepieciešami reālā aspekta izpildei. Vēlāk apskatītajos piemēros mums tas ir *Other aspects*, kas signalizē, ka visticamāk mēs sagaidām, ka būs vēl kompozīcija ar citiem semantiskajiem aspektiem.

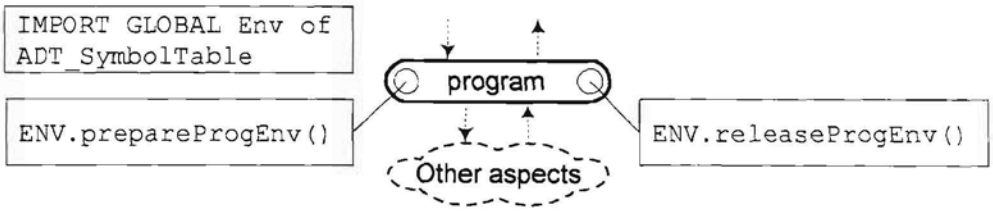
5.5.5 Semantisko aspektu piemēri

Apskatīsim dažus semantisko aspektu piemērus (Zīm. 44, Zīm. 45, Zīm. 46, Zīm. 47 un Zīm. 48), kas ir piemērojami vienkāršai imperatīvai programmēšanas valodai Pam (Tabula 7). Pam ir adaptēts no [Pag81], kur var atrast arī vairākas šīs pašas valodas formālās specifikācijas kādā klasiskā formālismā. Pam ir līdzīgs programmēšanas valodai Algol un nodrošina specifiskus aritmētiskus skaitļojumus ar veseliem skaitļiem. Valoda satur kontekstatkarīgus likumus, piemēram, mainīgais tiek izveidots pirmajā lietošanas reizē, kad tas vai nu tiek izmantots vērtības ielasīšanas komandā, vai arī tas atrodas vienādības zīmes kreisajā pusē piešķiršanas operācijā.

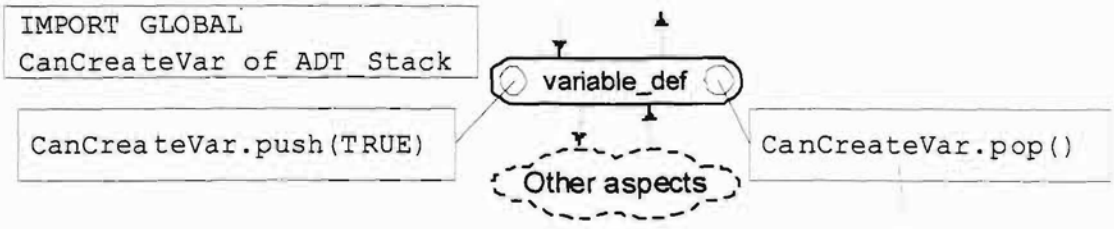
Termināļi ir apzīmēti ar taisnstūra kastītēm, bet netermināļi ar noapaļotajām kastītēm. Neterminālim kreisais aplītis atbilst *PreVisit semantic action*, labais - *PostVisit semantic action*, bet terminālim piekārtota *Visit semantic action*.

Tabula 7 Gramatika valodai PAM

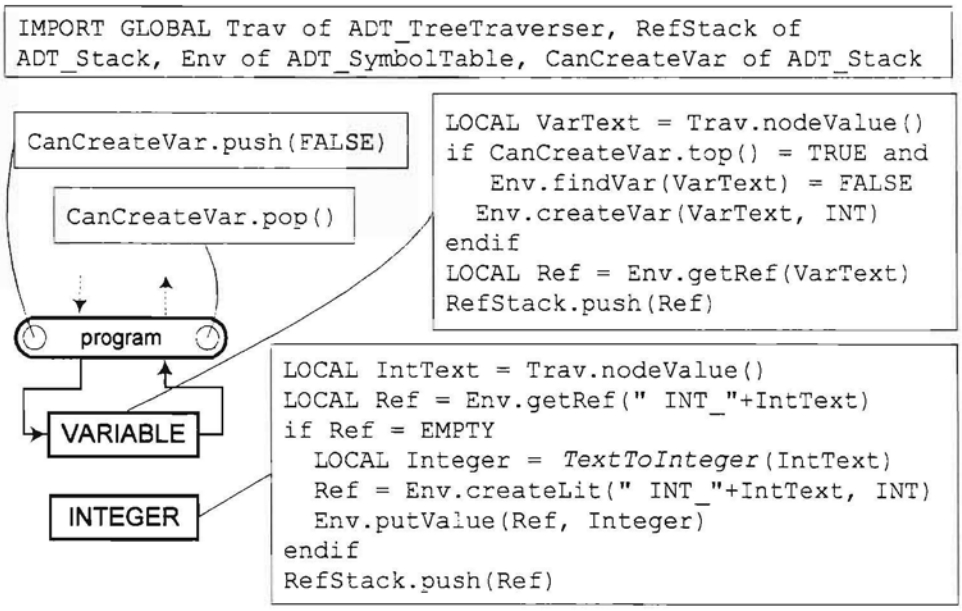
```
program                 ▯ series
series                  ▯ statement | series ; statement
statement               ▯ input_statement | output_statement |
                            assignment_statement | conditional_statement |
                          definite_loop | indefinite_loop
input_statement         ▯ READ variable_list
output_statement        ▯ WRITE variable_list
variable_list           ▯ VARIABLE | variable_list , VARIABLE
assignment_statement    ▯ VARIABLE ASSIGN expression
conditional_statement   ▯ IF comparison THEN series  FI |
                            IF comparison THEN series ELSE series FI
definite_loop           ▯ TO expression DO series END
indefinite_loop         ▯ WHILE comparison DO series END
comparison              ▯ expression RELATION expression
expression              ▯ term | expression WEAK_OPERATOR term
term                    ▯ element | term STRONG_OPERATOR element
element                 ▯ CONSTANT | VARIABLE |
                            LEFT_BRACKET expression RIGHT_BRACKET
```



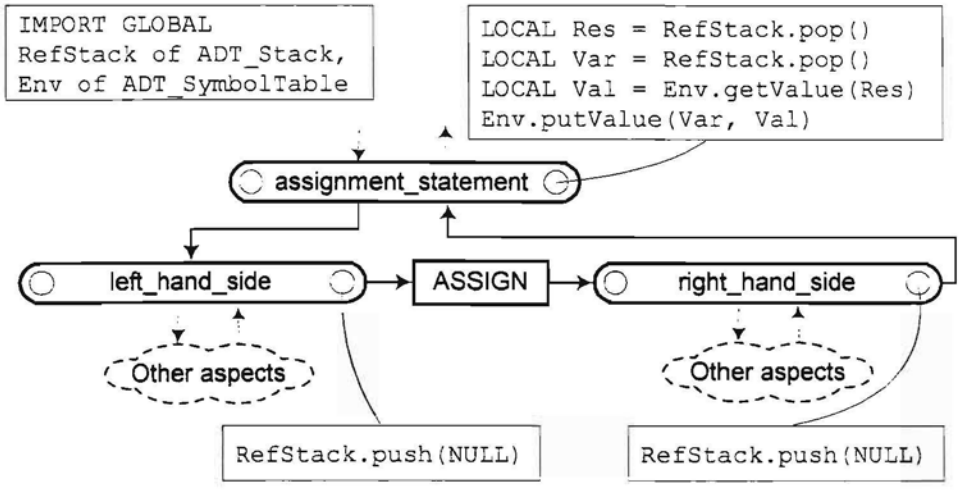
Zīm. 44 Semantiskais aspekts PROGRAM. Tas sagatavo programmas izpildes vidi, lai pārvaldītu mainīgos, konstantes un citus tiem pielīdzināmus objektus, un veiktu operācijas ar tiem. Izpildes beigās vide tiek iznīcināta.



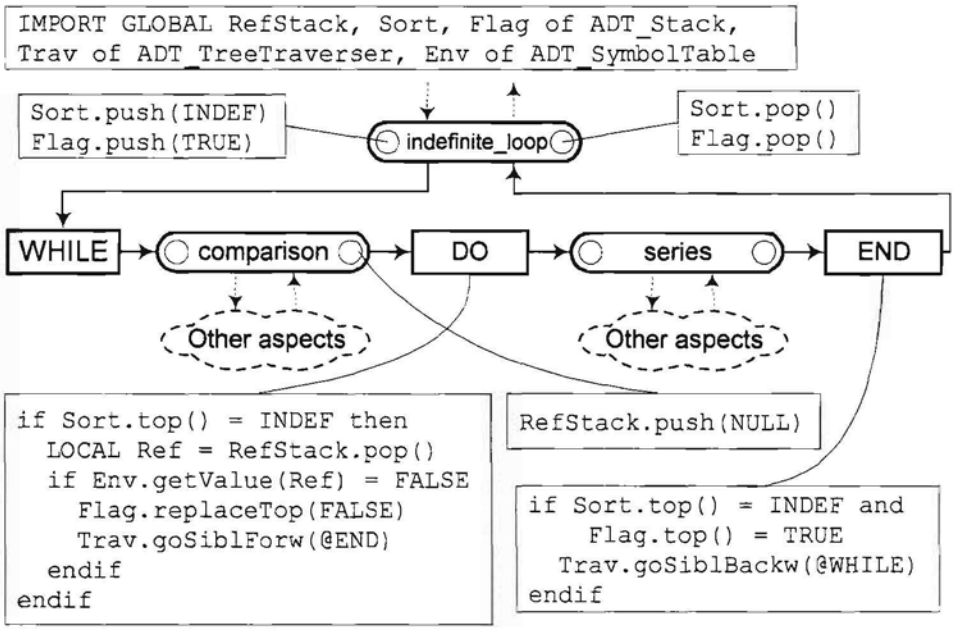
Zīm. 45 Semantiskais aspekts VARIABLE DEFINITION. Tas nodrošina mainīgā izveidošanu



Zīm. 46 Semantiskais aspekts ELEMENT. Tas nodrošina referenču ielikšanu stekā katram mainīgajam, kas tiek sastapts traversēšanā. Pēc noklusēšanas mainīgā izveidošana ir aizliegta. Trav nodrošina IR pašreizējā mezgla vērtību iegūšanu.



Zīm. 47 Semantiskais aspekts ASSIGNMENT. Tas ņem no steka referenci uz mainīgo, referenci uz vērtību un piešķir vērtību mainīgajam. Referenču izņemšana tiek simulēta.



Zīm. 48 Semantiskais aspekts INDEFINITE LOOP. Tas "iet caur *series*" un atgriežas uz WHILE tiklīdz *comparision* uzstāda NULL referenci vai referenci ar vērtību FALSE.

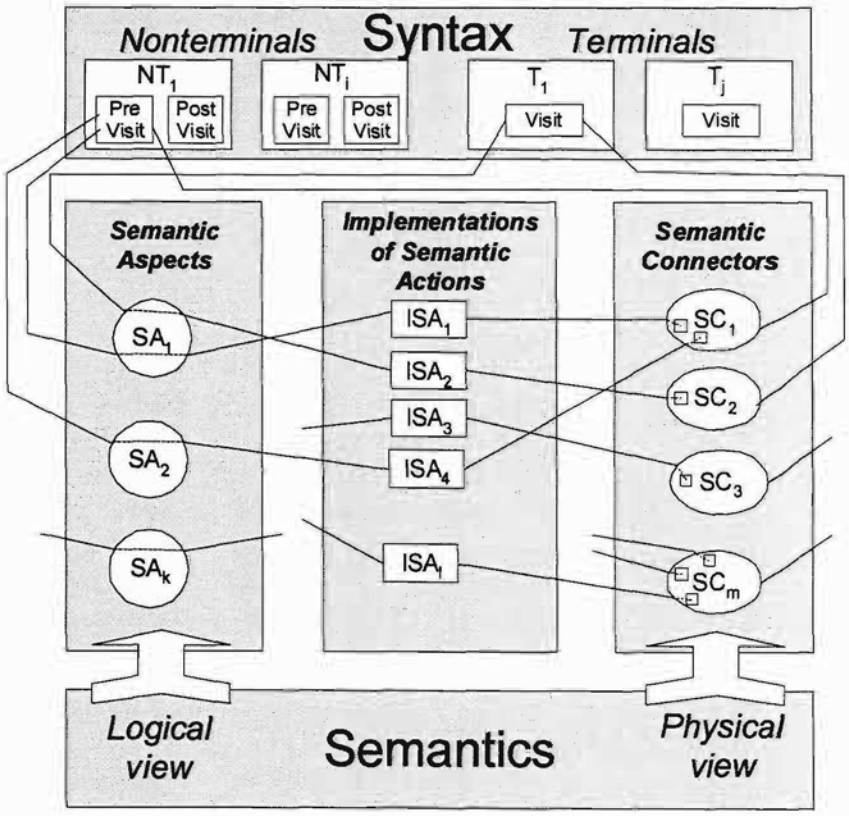
5.6 No semantiskajiem aspektiem uz meta-semantikām un izpildāmām semantikām

5.6.1 Meta-semantika

Meta-semantika ir meta-programma, kas apraksta no kādām komponentēm semantika sastāv un kā tās savā starpā saistās. Meta-semantikas konceptuālā shēma ir attēlota Zīm. 49.

No definēšanas viedokļa skatoties (loģiskais skats), semantiku kopumā veido traversēšanas stratēģija un semantisko aspektu kopa. Semantiskie aspekti savukārt sastāv no *semantic actions*, kas jāveic, apmeklējot atbilstošo iekšējās reprezentācijas IR mezglu. Skatoties no izpildes viedokļa (fiziskais skats), apmeklējot vienu mezglu, mums, iespējams, jāizpilda daudzas *semantic actions*, nodrošinot sadarbību starp abstraktajām komponentēm vēlamajā vidē. Atliek problēma, kā to visu sasaistīt kopā.

Lai nodrošinātu minētās problēmas atrisināšanu, tiek ieviests *semantic connector* jēdziens. Šis jēdziens ir adoptēts no jēdziena *Grey-box connector* līdzīgā problēmā [AGB00]: kā savienot iepriekš izveidotas komponentes dalītā un neviendabīgā vidē? *Grey-box connector* ir meta-programma, kas izveido konkrētu komunicēšanās savienojumu starp komponentēm, t.i. no meta-programmas var ģenerēt izpildāmu kodu komunikācijām un vides adaptācijai, lai nodrošinātu vēlamo savienojumu starp komponentēm.



Zīm. 49 Meta-semantikas konceptuālā shēma. Piemēram, SA_1 iesaista neterminālu NT_1 un terminālu T_1, un SC_1 tiek izpildīts *previsiting* laikā. SC_1 ir kompozīcija no ISA_1 un ISA_4

5.6.2 Semantikas izveide

Semantikas izveide mums vēlamajai fiksētai sintaksei norisinās vairākos etapos:

1. izvēlamies jau predefinētus vai definējam jaunus mums nepieciešamos semantiskos aspektus;
2. pārsaucam sintakses elementus semantiskajos aspektos atbilstoši fiksētajai sintaksei;
3. pārsaucam abstrakto komponenšu instances, lai nodrošinātu semantisko aspektu sadarbību;
4. veidojam kompozīciju no semantiskajiem aspektiem;
5. specificējam izpildes vidi un translējam meta-semantikas kodu uz kompilējamu programmu;
6. kompilējam semantiku uz izpildāmu kodu.

- **Semantisko aspektu izvēle (1.etaps)**

Ja mums ir jau agrāk izveidota semantisko aspektu bibliotēka, tad mēģinām atrast sev piemērotos, tādējādi tos atkalizmantojot. Traversēšanas stratēģija arī ir jāņem vērā.

- **Pieskaņošanās sintakses elementiem (2.etaps)**

Pieskaņošanās sintaksei notiek, pārsaucot semantiskajos aspektos sintakses elementus atbilstoši mērķa valodas sintaksei (*rename with*) vai izveidojam vairākus līdzīgus sintakses elementus (*dublicate to*):

rename <name> <traversing aspect> *with* <target name> <target traversing aspect>
duplicate <name> <traversing aspect > *to*
<target name> <target visiting aspect> [, <target name> <target visiting aspect> ...]

Piemēram,

rename left_hand_side PostVisit *with* VARIABLE Visit ;
duplicate COUNTABLENODE Visit *to* VARIABLE Visit, assignment_statement PostVisit

- **Abstrakto komponenšu instanču pārsaukšana (3.etaps)**

Komponentēm instanču saskaņošana nepieciešama, lai organizētu sadarbību starp daudziem semantiskajiem aspektiem, jo nav tiešas datu apmaiņas starp *semantic actions*. Programmas stāvoklis tiek fiksēts ar komponenšu palīdzību. Vispirms tiek nolemts, kādas komponenšu instances ir nepieciešamas kopīgai sadarbībai un tad aizvietojam semantiskajā aspektā iekļautās instances ar izvēlētajiem vārdiem:

replace <name> *with* <target name>

Piemēram,

replace RefStack *with* DataStack ;
replace Sort *with* LoopSortStack

- **Semantisko aspektu kompozīcija (4.etaps)**

Semantisko aspektu apvienošanas jeb kompozīcijas mērķis ir apvienot vairākus semantiskos aspektus kopā vienotam darbam, iegūstot daudz komplicētāku semantisko aspektu. Gala mērķis ir iegūt semantisko aspektu, kas apraksta faktiski visu vēlamo semantiku. Kompozīcijā tiek ņemti vienādi nosauktās *semantic actions* un to implementācijas meta-kodi tiek apvienoti.

Kodu apvienošanas principi varētu būt dažādi, piemēram, secīgi (pieliek vienam galā otru), paralēli (koda fragmenti var tikt izpildīti neatkarīgi un vienlaicīgi), brīvi (kods tiek apvienots, paredzot, ka lietotājs to var modificēt pēc sava prāta). Bez tam varētu būt apvienošanas principa precizēšana labāku rezultātu sasniegšanai, piemēram, kādu *semantic action* ignorēt vai apvienošanas principu pielietot pretēji semantisko aspektu secībai. Bez tam iespējams, ka vēl ir jāatrisina lokālo mainīgo vienādu vārdu konflikti.

Jāņem vērā, ka var rasties dažādu semantisko aspektu implementāciju neatbilstības kopīgam darbam. Piemēram, *Element* ieliek stekā vērtību *Ref. Assign* izņem *Ref* un izmanto. Arī *SimbolicAssign* izņem *Ref* un izmanto. Ja abus vērtību *Ref* patērējošos aspektu *Assign* un *SimbolicAssign* sakombinē, tad rodas konflikts, jo viens patērē *Ref*, bet otram tā vairs nav.

compose aspect <<new SA>> (<refined SA>) [[*append* | *parallel* |*free*] (<refined SA>) ...] , where
<refined SA> = <<old SA>> [*ignore* <name> <trav aspect> [, <name> <trav aspect>]] | [*reverse*]

Rezultātā mēs iegūstam meta semantiku. Meta semantikas koda piemērs ir dots Tabula 8.

Tabula 8 Piemērs meta-semantikai

```
compatible with
  ir_type ParseTree
  traverser_type ParseTreeTraverser

syntax elements (program, expression, VARIABLE, ...)
semantic actions (<PROGRAM> program PreVisit { ENV.prepareProgEnv()},
                  <PROGRAM> program PostVisit {...}, ...)

global Trav of ADT_TreeTraverser
global Env of ADT_SymbolTable
create DataStack, OperatorStack, CanCreateVar, LoopSortStack,
    LoopCounterStack, LoopFlagStack, IfFlagStack of ADT_Stack
create InputFile, OutputFile of ADT_FILE

compose aspect <A1>     // composes semantic aspects from aspects given above
  (<PROGRAM>)           // semantic aspects PROGRAM remains the same
append (<ELEMENT>
  replace RefStack with DataStack // replaces stack for collaborating work
  rename INTEGER Visit with CONSTANT Visit)   // renames nonterminal according to PAM syntax
append (<ASSIGNMENT>
  replace RefStack with DataStack
  rename left_hand_side PostVisit with VARIABLE Visit,
         right_hand_side PostVisit with expression PostVisit
  ignore left_hand_side PostVisit)      // ignore pushing of NULL reference
append (<INDEFINITE LOOP>
  replace RefStack with DataStack,
    Sort with LoopSortStack, Flag with LoopFlagStack)
append (<VARIABLE DEFINITION>
  rename variable_def PreVisit with assign_statement PreVisit,
    variable_def PostVisit with ASSIGN Visit)
end compose aspect

compose aspect <A2>
(<A1>
  ignore expression PostVisit, comparision PostVisit)
  append ( ...
  /* Others aspect are appended such as <TYPE AND OPERATOR>, <INPUT>,
<OUTPUT>, <BASE BYNARY OPERATION>, <DEFINITE LOOP>, <CONDITIONAL STATEMENT> */
  ...)
end compose aspect
```

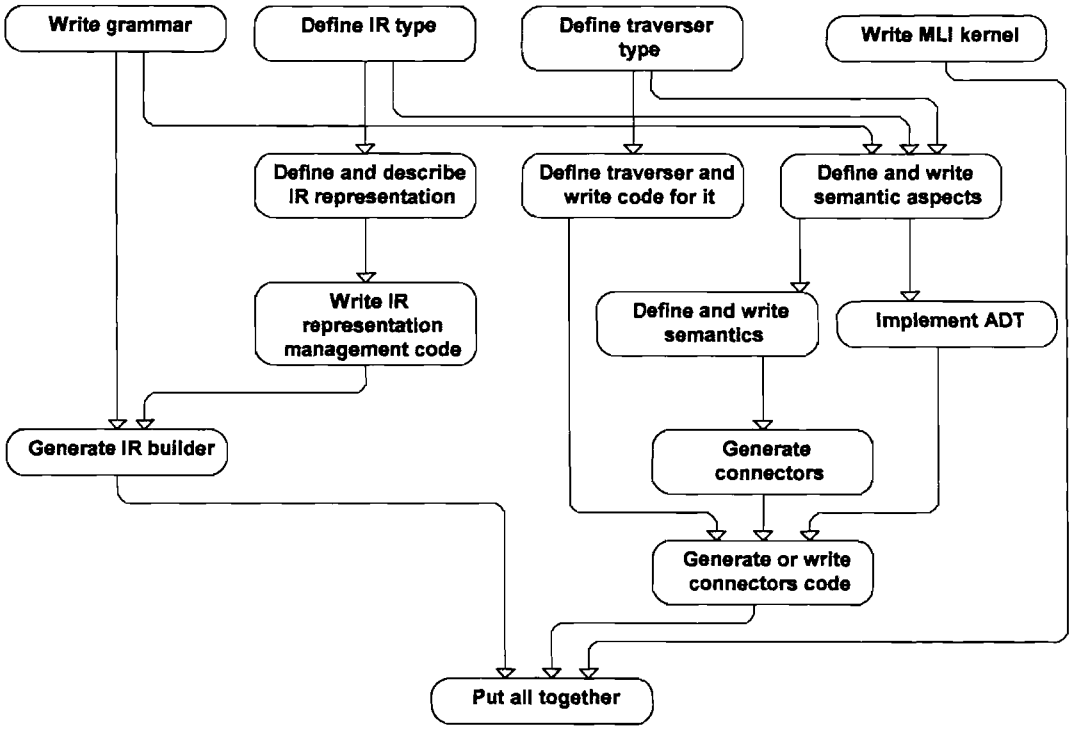
- **Translēšana uz mērķa programmēšanas valodu (5.etaps)**

Konkrētas semantikas programmas kodu iegūst no meta-semantikas, abstraktos objektus aizstājot ar nepieciešamo implementācijas tipu (piemēram, *Stack in linked memory*), norādot mērķa valodu (piemēram, C++), izpildes vidi (piemēram, Unix), komunikāciju veidu (piemēram, CORBA), semantikas objekta tipu (piemēram, DLL) un translējot konektorus, ņemot vērā iepriekš minēto specifiku, utt. Pāreja no meta-semantikas uz kompilējamu kodu varētu būt automātiska vismaz populārākajiem gadījumiem.

- **Semantikas objekta iegūšana**

Kompilējot iegūto programmu, iegūstam izpildāmu kodu, kas nodrošina semantikas izpildi, t.i. satur kopu ar funkcijām, kas tiek izsauktas traversējot IR. Objekts pats veido norādīto abstrakto objektu instances vai arī dinamiski saistās ar eksistējošām instancēm, izmantojot norādītos komunikāciju veidus.

Vispārējā procesa shēma, lai iegūtu visu rīku, kas balstās uz doto valodu (sintaksi un semantiku) ir attēlota Zīm. 50.



Zīm. 50 Piemērs rīka izveides soļiem

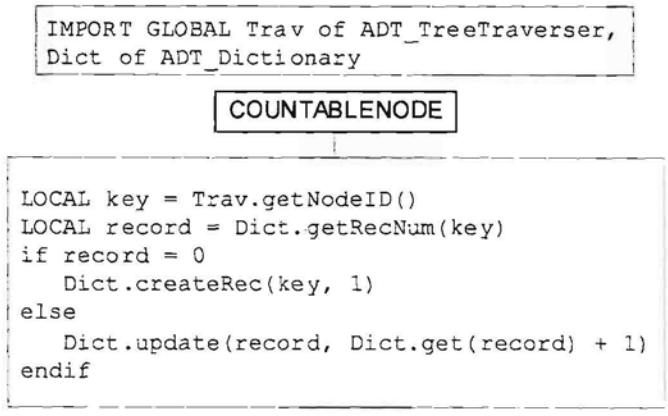
5.7 Piemēri alternatīviem semantiskajiem aspektiem

Šajā apakšnodaļā dosim īsu ieskatu, kā var veidot citas semantikas.

Izmantojot papildus semantikas, mēs papildinām jau apskatīto esošo semantiku ar jaunām īpašībām un iegūstam specifisku rīku, kas darbojas ar programmām uzrakstītām dotajā valodā. Mēs apskatīsim divus vienkāršus piemērus, kā var papildināt parasto izpildi – statistikas vākšanu par katru mums nepieciešamo programmas punktu un mainīgo simbolisko vērtību uzkrāšanu (faktiski abi gadījumi ir programmas instrumentēšana programmatūras testēšanas terminos).

5.7.1 Programmas punktu apmeklēšanas uzskaite

Mūsu mērķis ir uzskaitīt katra mums vēlamā programmas punkta apmeklējuma reizi. Tas nozīmē, ka ir jāpieliek skaitītāji šajos punktos. Vispirms uzrakstam semantisko aspektu NODE COUNTER (Zīm. 51). Tam ir nepieciešama abstraktā komponente Dictionary, kurā mēs varam saglabāt, lasīt un mainīt ierakstus <key, value>.



Zīm. 51 Semantiskais aspekts NODE COUNTER

Veidojot abstrakto semantiku, mums ir jāpievieno šis semantiskais aspekts pārējiem. Piemēram, uzskaitām visu mainīgo lietošanas un piešķiršanas operāciju skaitu:

duplicate countable_node Visit *to* VARIABLE Visit, assignment_statement PostVisit

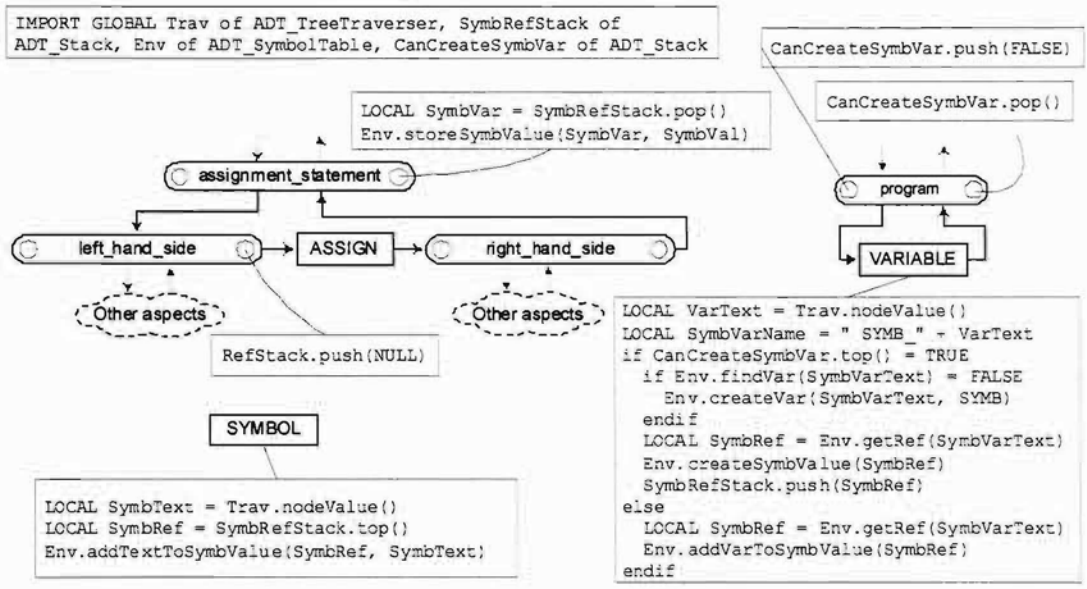
Varētu izveidot vēl vienu aspektu, kas citā vārdnīcā uzskaita katra konkrētā mainīgā izmantošanu (vārdnīcā par atslēgu tiek lietots mainīgā vārds). Savukārt šo semantisko aspektu vēl mazliet modificējot varētu uzskaitīt arī mainīgā izmantošanas biežumu pēc konteksta (*defined, modified, referenced, released*).

Ja apskatām tikai programmu analīzes un instrumentēšanas sfēru, tad mūsu pieeja ir līdzīga Wyong sistēmai (balstās uz kompilatoru ģenerēšanas sistēmu Eli un programmu instrumentēšanas sistēmu ATOM), jo specifiskās operācijas ir pievienotas sintaktiskajiem elementiem. Šādā veidā var tikt iegūts specifisks rīks ar papildus semantiku [Slo97].

5.7.2 Simbolisko vērtību uzkrāšana mainīgajiem

Otrajā piemērā mēs nodrošinām simbolisko vērtību fiksēšanu visiem mainīgajiem (Zīm. 52). Lai to nodrošinātu, komponentē Symbol table vajadzētu izveidot papildus iebūvēto datu tipu efektīvai simbolisko vērtību glabāšanai un papildus operācijas darbam ar to.

Mūsu izveidotajā MOMS šādas papildus operācijas ir izveidotas. Operācija *createSymbValue* izveido norādi uz simbolisko vērtību. Ar operācijām *addTextToSymbValue* un *addVarToSymbValue* mēs veidojam simbolisko vērtību, traversējot visus mezglus mūsu izvēlētajā apakškokā un saglabājot gan visus nepieciešamos programmas simbolus, gan arī mainīgo simboliskās vērtības. Pašās beigās ar operāciju *storeSymbValue* mēs iegaumējam uzkrāto vērtību.



Zīm. 52 Semantiskais aspekts SYMBOLIC VALUES

5.7.3 Piemērs darba plūsmas pārvaldības semantikai

Lai demonstrētu nākošo piemēru, mēs izmantojam vienkāršu darba plūsmas definēšanas valodu, kas aprakstīta ar BNF (Tabula 9). Ir ieviesti divi tipi ar vispārīgiem uzdevuma tipiem, kas apraksta darbu – universāls (*Universal_stm*) un specifisks (*Specific_stm*).

Tabula 9 BNF fragments vienkāršai darba plūsmas definēšanas valodai

```
workflow        -> series
series          -> statement | series ; statement
statement       -> generic_stm | cond_stm
cond_stm        -> IF compar THEN series ELSE series FI
compar          -> expr relation expr
expr            -> const | var
generic_stm     -> universal_stm | specific_stm
universal_stm   -> u_stm_type name
u_stm_type      -> DCOM | CORBA | WEBSERVICE | MANUAL
specific_stm    -> s_stm_type name
```

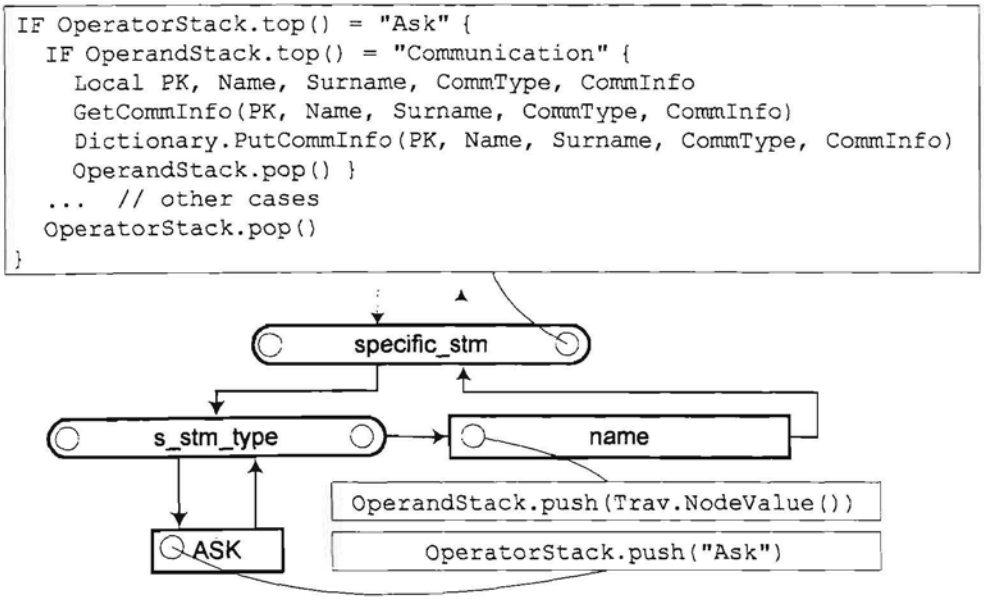
Universal_stm tiek lietots, lai sadarbotos ar ārējām lietojumprogrammām. Tas apraksta kādu no komunikācijas tipiem: DCOM, CORBA, WEBSERVICE (visi trīs tipi nozīmē automātisku uzdevumu apstrādi) un MANUAL (operāciju veic cilvēks).

Specific_stm tiek lietots, lai komunicētos ar personu – parasti ar iedzīvotāju, kas izmanto noteikto servisu. Ir divi tipi specifiskajiem uzdevumiem: ASK saņem informāciju no personas, ANSW sūta informāciju personai.

Apskatīsim nelielu vienkāršu darbu plūsmas specifikāciju:

```
WEBSERVICE Application_writing_and_submitting
ASK Communication
DCOM Application_data_control_and_update
IF Is_data_control_and_updating_successful = True THEN
   DCOM Printing_of_passport
   ANSW Positive_answer
   MANUAL Passport_handing_out
ELSE
   ANSW Negative_answer
FI
```

Darba plūsmas mērķis ir izdot jaunu pasi personai. Darba plūsmai ir sekojošas aktivitātes – persona aizpilda pieteikuma formu un iesniedz to ierēdnim. Pieteikums var būt uz papīra vai WEB tipa lietojumprogramma. Ierēdnis vai lietojumprogramma jautā personai komunicēšanās tipu un adresi un ieraksta to darba plūsmas izpildes vidē. Tad ierēdnis pārbauda iedzīvotāja aizpildītās formas pareizību ar datiem no Iedzīvotāju reģistra. Ja dati ir korekti, tad pase tiek izdota un izsniegta personai. Citādi iedzīvotājam tiek nosūtīta negatīva atbilde. Piemērs vienam no semantiskajiem aspektiem šai darba plūsmai ir dots Zīm. 53.



Zīm. 53 Semantiskais aspekts SPECIFIC STATEMENT ASK

5.8 Secinājumi par semantikas pierakstu

Šī metode tika izveidota ar mērķi samazināt plaisu starp praktiķiem (rīku veidotājiem) un teorētiķiem (semantikas formālo specifikāciju veidotājiem). Mūsu pieredze rāda, ka būtiskākais metodes ieguvums ir iespēja, ka abstraktās komponentes vai abstraktie datu tipi realizē lielāko semantikas daļu, jo šādā veidā ir vieglāk uztvert visu semantikas implementāciju.

Otrais mūsu metodes ieguvums ir būtiska sintakses un semantikas atdalīšana vienai no otras. Tas ļauj mums kombinēt dažādas sintakses un semantikas un atrast vispiemērotāko semantiku dotajai sintaksei. Tādējādi, ja mēs esam uzrakstījuši sintaksi jaunai valodai, tad mēs salīdzinoši īsā laikā varam izveidot vairākas semantikas šai sintaksei.

Mūsu pieeja atļauj arī dinamiski mainīt semantikas interpretatora darbības laikā, t.i. aizvietot vienu semantiku ar otru vai izpildīt vairākas semantikas vienlaicīgi. Ir iespējams reducēt programmas sintaktisko izveduma koku (piemēram, izmetot mezglus ar tukšiem konektoriem) vai optimizēt to (piemēram, koka restrukturēšana statiski, t.i. pirms izpildes, vai arī dinamiski izpildes laikā, ņemot vērā izpildes statistiku).

Šobrīd rīku veidošanas vai semantikas ģenerēšanas vide nav pilnīgi izveidota. Mūsu pieredze rāda, ka rīki var tikt izveidoti arī bez būtiskām investīcijām, piemēram, lietojot Lex/YACC kā ģeneratoru, lai izveidotu sintaktisko objektu, kas ražo programmas iekšējo reprezentāciju. Nav grūti izveidot arī objektu Traverser un vienkāršu SymbolTable implementāciju. Un kā pēdējais darbs ir semantisko aspektu izveide, balstoties uz mūsu metodoloģiju, un kompozīciju veidošana no tiem. Tādējādi tiek iegūti konektori, kas var tikt uzrakstīti kādā populārā programmēšanas valodā (tiek izlaista meta-valodas lietošana un tās translācija). Abstrakto komponenšu izmantošana ir atkarīga no izvēlētās mērķa semantikas.

Mēs uzskatām, ka nopietnu rīku izveide prasa samērā universālu *Symbol table* implementāciju. Mūsu *Symbol table* realizācija – MOMS ir pielietojama ne tikai imperatīvo programmēšanas valodu implementācijai. MOMS var kalpot arī kā objekt-orientēta datubāze cita tipa lietojumprogrammām.

Mūsu metodes vājās vietas ir meklējamas semantisko aspektu kompozīcijas fāzē. Šobrīd mēs vēl neesam izanalizējusi visus riskus, kas var sekmēt bezjēdzīgas vai kļūdainas semantikas iegūšanu. Problēma nav triviāla, un tā ir analoģiska problēmai par atbilstošas sadarbības organizēšanu starp objektiem vai komponentēm objektorientētajā programmēšanā. Lielākā daļa *vārdu konfliktu* var tikt izslēgta automātiski, taču daudz grūtāk ir organizēt sadarbību starp kopīgajām komponentēm semantiskajos aspektos. Problēma atrisinās, ja semantisko aspektu darbība tiek organizēta pēc iespējas neatkarīgi vienai no otras.

Cita bieži sastopama problēma ir, ka valodas gramatika mēdz nebūt bezkonteksta (arī iepriekš apskatītais piemērs bija šāds). Šādā situācijā mums ir jāievieš papildus karodziņi, lai atcerētos sintaktiskā elementa kontekstu katrā konkrētajā situācijā. Lai samazinātu problēmas, ļoti ieteicams valodai veidot bezkonteksta gramatiku.

Nobeigums

Darbā ir apkopoti rezultāti, kurus autors viens vai kopā ar kolēģiem ir ieguvis laika posmā no 1994.gada līdz 2003.gadam. Kā būtiskākos jauninājumus mēs uzskatām sekojošas lietas.

1. Jau 1994.gadā tika formulēts daudzvalodu interpretatora (MLI) jēdziens, kuram dinamiski var piesaistīt izpildāmās valodas sintaksi un semantiku. Faktiski valodai var piesaistīt vairākas semantikas vienlaicīgi un programmas darbības laikā semantikas ir iespējams mainīt. Savukārt viena un tā pati semantika var tikt pielietota daudzām valodām. MLI atbalsta principu – veidot pēc iespējas daudz rīku, kas balstās uz valodas gramatiku.
2. Ir izveidota iespējamā MLI arhitektūra un darbības pamatprincipi. MLI svarīgākās komponentes ir Sintakses objekts, kas pārvērš programmu iekšējā attēlojumā, Semantikas objekts, kas reāli izpilda programmu un traversētājs, kas visas komponentes sasaista kopā. Programmas izpildes princips idejiski ir tuvs *Visitor Pattern* principam.
3. Ir izveidots jauns semantikas pieraksta princips, izmantojot semantiskos aspektus, kas tiek pierakstīti ar grafisku apzīmējumu un meta-valodas palīdzību. Semantikas aspektu kompozīcija veido visu semantiku. Semantikas aspektu specifikācijas ir atkārtoti izmantojamas jaunu semantiku veidošanā.
4. Ir izveidota saistība starp loģisko semantikas skatu, kas ir ērtāks un saprotamāks semantikas veidotājam, un fizisko semantikas skatu, kas ir izmantojams valodas implementēšanā un atbalsta MLI ideoloģiju.
5. Ir aprakstīti sintakses un semantikas objektu izveides soļi, kas var tikt veikti gan manuāli, gan automātiski. Tādā veidā mēs iegūstam MLI piesaistāmas komponentes – valodas sintaksi un semantiku.
6. Ir izveidota domēnspecifiska valoda meta-semantikas pierakstīšanai mērķa valodai. Ar tās palīdzību tiek noteikti likumi, kā semantiskos aspektus var sasaistīt kopā.
7. Semantikas pierakstā ļoti būtiska loma ir atvēlēta abstrakto datu tipu izmantošanai, t.i. lielāko semantikas jēgu apraksta šie datu tipi un to definētās funkcijas. Tādējādi semantikas pieraksts kļūst formālāks un tiek izmantoti līdzekļi, kuru korektumu var pierādīt. Otrs būtiskais ieguvums ir krasa semantikas lasāmības un saprotamības uzlabošana, kā arī pastāv dažādās implementēšanas iespējas atkarībā no specifiskajām vajadzībām.
8. Ir izveidota specifikācija atmiņas objektu vadības sistēmai (MOMS), kas būtiski atvieglo MLI semantikas implementēšanu.
9. Pētot domēnspecifiskās valodas un to implementēšanas iespējas, ir konstatēts, ka MLI izmantošana ir viena no alternatīvām, kā var ātri sasniegt rezultātu. MLI, darbojoties ar konkrētu fiksētu semantiku, kļūst par rīku ar nepieciešamo funkcionalitāti.
10. Darbā ir dota vispārēja rīka arhitektūra, kas ir piemērota lielai nepieciešamo rīku grupai.

11. Autors, izskatot dažādu rīku un sistēmu integrēšanas problēmas un klasiskos risinājums, ir izveidojis papildus kritērijus kvalitatīvākai rīku integrēšanai. Izvēloties MLI par realizācijas pamatu, šie kritēriji lielākoties tiek izpildīti.

12. Lielāka uzmanība ir veltīta specifiskai rīku klasei – lietojumu ģeneratori darbam ar datubāzēm. Ir aprakstītas daudzas idejas, kas atvieglo rīka funkcionalitāti izteikt ar domēnspecifiskas valodas palīdzību. Visam pamatā ir datu modelis, kas apraksta fizisko datubāzi vai loģisko skatu uz datu avotu ar pieļaujamajām datu manipulācijas iespējām. Lietojumu ģenerācija balstās uz šo datu modeli, pieejamajiem datu apstrādes līdzekļiem un saņemtajiem datiem. Darbošanās princips ir tuvs pēdējos gados populārajām satura vadības sistēmām tīmekļa lietojumprogrammās.

13. Darbā ir parādīts, kā var ģenerēt vienkāršas lietojumprogrammas, balstoties tikai uz ER diagrammām, kas mazliet papildinātas ar ģenerācijai būtisku informāciju. Jaunievedums ir entītijas tipu ieviešana, kuru noteikšanai ir izstrādāts automātiski izpildāms algoritms ar iespēju lietotājam pielabot automātiski noteikto tipu. Ir ieviesti arī skata un redzamības atribūti.

14. Ir izveidots entītiju, tās atribūtu un relāciju attēlošanas princips. Autors piedāvā arī komplektu ar iespējamiem saskarnes ekrānu formātiem, starp kuriem ir arī oriģināli citur nesastapti principi. Saskarnes veidošana ir aprakstāma ar domēnspecifiskas valodas palīdzību.

15. Ir izveidoti principi, kā var pārvietoties no viena saskarnes ekrāna uz citu, gan ar datu filtrēšanu, gan bez tās. Princips sevi ir labi attaisnojis, integrējot daudzas informatīvās sistēmas.

16. Ir dotas lietojumprogrammu ģenerācijas shēmas gan statiskai lietojumprogrammai, gan dinamiskai lietojumprogrammai. Saskarnes ģenerācijai ir izveidota domēnspecifiska valoda, kas nodrošina datu paņemšanu no dažādiem datu avotiem, to pārvēršanu vienotā informācijā un attēlošanu lietotājam HTML lapas veidā.

17. Ir izveidots repozitorija formāts, ar kura palīdzību var aprakstīt datu avotus un funkcijas darbam ar datiem. Izdomāts oriģināls entītiju saistīšanas princips starp datu avotiem un nodrošināta iespēja darboties ar daudziem datu avotiem, kā ar vienu lielu datubāzi.

18. Reālu projektu ietvaros, kuru pamatuzdevums bija integrēt informatīvās sistēmas starp daudzām organizācijām un pat valstīm, tika definēts Komunikāciju servera (KS) jēdziens. KS nodrošina lietotājam vienu resurspunktu nepieciešamās informācijas saņemšanai no daudziem datu avotiem. Tika noteikta KS pamatfunkcionalitāte.

19. Ir izveidoti svarīgākie datu saņemšanas tiesību apraksti, darbojoties ar daudziem datu avotiem, lai ievērotu datu aizsardzības prasības un dažādos datu avotu īpašnieku un likumdošanas noteiktos ierobežojumus.

20. KS un MLI idejas ir izmantojamas arī, veidojot globālus darba vadības procesus, kas notiek starp daudzu organizāciju informatīvajām sistēmām. Ir doti piemēri, kā to var realizēt.

Daļa no rezultātiem ir pārbaudīta arī praksē.

1. Ir izveidots MLI prototips vienkāršai imperatīvai programmēšanas valodai, darbinot programmas gan ar tradicionālu valodas semantiku, gan netradicionālām semantikām.
2. Visvairāk un tiešāk daļa no idejām ir izmantota Komunikāciju servera 1.versijas izveidē. Rezultāts, ņemot vērā tā implementācijai patērētos resursus un iegūto lietojumprogrammas funkcionalitāti, vērtējams kā ļoti labs.
3. Ir izveidota atmiņas objektu vadības sistēma MOMS, kas ir neatkarīga MLI komponente un kas ir izmantota arī komerciālā produktā kā objektorientēta datubāze.
4. Dažas biznesa informatīvās sistēmas ir izveidotas pēc apskatītās statiskās lietojumprogrammas ģenerācijas principa, tiesa, kodu nevis ģenerējot, bet sakopējot nepieciešamos programmas fragmentus "ar roku".

Tālākie darbības virzieni varētu būt sekojoši.

1. Jaunu MLI versiju izveide.
2. MLI semantikas pieraksta valodas uzlabošana.
3. Meta semantikas veidošanas principu tālāka pētīšana, analīze un vājo vietu uzlabošana.
4. Specifisku semantiku izstrāde loģiskajā līmenī plašam programmēšanas valodu lokam, lai tos varētu izmantot MLI darbināšanā.
5. Reālu rīku izveide, balstoties uz MLI principiem vai citiem šajā darbā apskatītajiem principiem. Attiecībā uz rīku būvi ir divi galvenie stratēģiskie attīstības virzieni. Viens virziens ir veidot dažādus rīkus, kas nepieciešami tieši dinamiskai datu apstrādei. Otrs virziens saistās ar dažādu kvalitātes rīku izveidi jebkurai programmēšanas valodai – gan klasiskai, gan domēnspecifiskai.

Literatūras saraksts

Publikācijas recenzējamos starptautiskos zinātniskos izdevumos

[AAB96b] V. Arnicane, G. Arnicans, and J. Bicevskis. **Multilanguage interpreter**. In H.-M. Haav and B. Thalheim, editors, *Proceedings of the Second International Baltic Workshop on Databases and Information Systems (DB&IS '96)*, Volume 2: Technology Track, pages 173-174. Tampere University of Technology Press, 1996.

[ABK00a] G. Arnicans, J. Bicevskis and G. Karnitis. **The Unified Megasystem of Latvian Registers: Development of a Communication Server – the First Results and Conclusion**. In abstracts of papers of *4rd International Conference "Information Technologies and Telecommunications in the Baltic States"*, pages 163-168. Riga, 2000.

[ABK99] G. Arnicans, J. Bicevskis and G. Karnitis. **The Concept of Setting Up a Communication Server**. In abstracts of papers of *3rd International Conference "Information Technologies and Telecommunications in the Baltic States"*, pages 48-57. Riga, 1999.

[ABKK02] G. Arnicans, J. Bicevskis, E. Karnitis, and G. Karnitis. **Smart Integrated Mega-system as a Basis for e-Governance**. *Proceedings D of the 5th International Multi-Conference Information Society IS'2002*, pages 197-201. Ljubljana, Slovenia, 2002.

[AK00a] G. Arnicans and G. Karnitis. **Heterogeneous Database Browsing in WWW Based on Meta Model of Data Sources**. In Albertas Caplinskas, editor, Databases & Information Systems, *Proceedings of the 4th IEEE International Baltic Workshop*, Volume 1, pages 174-187. Vilnius "Technika", 2000.

[AK00b] G. Arnicans and G. Karnitis. **Heterogeneous Database Browsing in WWW Based on Meta Model of Data Sources**. In Janis Barzdins and Albertas Caplinskas, editors, Databases and Information Systems, *Fourth International Baltic Workshop, BalticDB&IS 2000 Vilnius, Lithuania, May 1-5, 2000, Selected Papers*, pages 167-178. Kluwer Academic Publishers, 2000.

[AK02a] G. Arnicans and G. Karnitis. **Semantics for Managing Systems in Heterogeneous and Distributed Environment**. In Hele-Mai Haav and Ahto Kalja, editors, Databases and Information Systems, *Proceedings of the Fifth International Baltic Conference BalticDB&IS 2002*, Volume 1, pages 51-62. Tallinn, 2002.

[AK02b] G. Arnicans and G. Karnitis. **Semantics for Managing Systems in Heterogeneous and Distributed Environment**. In Hele-Mai Haav and Ahto Kalja, editors, Databases and Information Systems II, *Fifth International Baltic Conference, BalticDB&IS'2002 Tallinn, Estonia, June 3-6, 2002, Selected Papers*, pages 149-160. Kluwer Academic Publishers, 2002.

[Arn98] G. Arnicans. **Application generation for the simple database browser based on the ER diagram**. In Jānis Bārzdiņš, editor, Databases and Information Systems, *Proceedings of the Third International Baltic Workshop*, Volume 1, pages 198-209. Rīga, 1998.

Citas publikācijas, referāti un raksti

[AAB96a] V. Arnicane, G. Arnicans, and J. Bicevskis. **Multilanguage interpreter**. Submitted paper to *the Second International Baltic Workshop on Databases and Information Systems (DB&IS '96), Tallinn*, pages 14, 1996.

[ABK00b] G. Arnicans, J. Bicevskis and G. Karnitis. **Development of a Communication Server: First Results and Conclusions**. *Baltic IT Review*, 17(2):29-32, 2000.

[ABKK01] G. Arnicans, J. Bicevskis, G. Karnitis, and E. Karnitis. **The Mega-system: integration of National information systems. Conceptual and Methodological Baselines**. In Latvian Academic Library Grey Literature database, http://159.148.58.74/greydoc/megasystem_baselines/mega_base.doc, pages 16, 2001.

[Arn02] G. Arnicāns. **Domēnspecifisko valodu izmantošanas iespējas**. Referāts, *Latvijas Universitātes Zinātniskā Konference 2002*, slaidi 41, 2002.

[Arn03a] G. Arnicans. **Description of Semantics and Code Generation Possibilities for a Multi-language Interpreter**. Akceptēts iespiešanai *Latvijas Universitātes zinātniskie raksti. Sējums 669. Datorzinātne un informācijas tehnoloģijas.*, lapas 22, 2003.

[Arn03b] G. Arnicāns. **Information Processing Tools and Environments**. Referāts, *Latvijas Universitātes Zinātniskā Konference 2003*, lapas 9, 2003.

Citu autoru darbi

[AGB00] U. Aßmann, T. Genßler, and H. Bär. **Meta-programming Grey-box Connectors**. Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS 33), pp.300-311, 2000.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. **Compilers: Principles, Technigues, and Tools**. Addison-Wesley, 1986.

[BG96] T. Bergin and R. Gibson, editors. **History of programming languages**, Addison-Wesley, 1996.

[BM99] B. Biswas and R. Mall. **Reverse Execution of Programs**. ACM SIGPLAN Notices, 34(4):61-69, April 2000.

[Bro96] Frederick P. Brooks. **LANGUAGE DESIGN AS DESIGN**. In Bergin and Gibson [BG96], pp.4-16.

[Cla99] C. Clark. **Build a Tree – Save a Parse**. ACM SIGPLAN Notices, 34(4):19-24, April 2000.

[DJ90] Herbert L. Dershem, Michael J.Jipping. **Programming Languages: Structures and Models**, Wadsworth Publishing Company, USA, 1990, 413 p.

[DK98] A. van Deursen and P. Klint. **Little languages: Little maintenance?** *Journal of Software Maintenance*, 10:75-92, 1998.

[DKV00] A. Deursen, P. Klint, and J. Visser. **Domain-Specific Languages: An Annotated Bibliography**. ACM SIGPLAN Notices, 35(6):26-36, June 2000.

[Eng99] Dawson R. Engler. **Interface Compilation: Steps toward Compiling Program Interfaces as Languages**. In DSL-99 [ITSE99], pp.387-400.

[Fis00] Layna Fischer, editor. **The Workflow Handbook 2001**, Published in association with the Workflow Management Coalition, 2000

[FL88] Charles N. Fisher, and Richard J. LeBlanc, Jr. **Crafting A Compiler**. Benjamin-Cummings, 1988.

[FNT+97] F. Ferrucci, F. Napolitano, G. Tortora, M. Tucci, and G. Vitiello. **An Interpreter for Diagrammatic Languages Based on SR Grammars**. Proceedings of the 1997 IEEE Symposium on Visual Languages (VL '97), pages 292-299, 1997.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlisides. **Design Patterns: Elements of Reusable Software**, pages 331-334. Addison-Wesley, 1995.

[Gun92] Carl A. Gunter. ***Semantics of Programming Languages: Structures and Techniques***. Cambridge MIT Press, 1992.

[HGI+95] Hammer J, Garcia-Molina H, Ireland K, Papakonstantinou Y, Ullman J, Widom J. **Information translation, mediation, and Mosaic-based browsing in the TSIMMIS system**. *Proceedings of ACM SIGMOD International Conference on Management of Data*, 1995, Project Demonstration.

[HK00] J. Heering and P. Klint. **Semantics of Programming Languages: A Tool-Oriented Approach**. ACM SIGPLAN Notices, 35(3):39-48, March 2000.

[HMN+99] Haas L. M, Miller R. J, Niswonger B, Tork Roth M, Schwarz P. M, Wimmers E. L. **Transforming Heterogeneous Data with Database Middleware: Beyond Integration**. Data Engineering Bulletin, 1999

[HOT00] W. Harrison, H. Ossher, and P. Tarr. **Software Engineering Tools and Environments: A Roadmap**. Proceedings of the conference on The future of Software engineering (ICSE '00), pp.261-277, 2000.

[How82] W. E. Howden. **Contemporary Software Development Environments**. Communications of the ACM, 25(5):318-329, May 1982.

[ITSE99] Special issue on domain-specific languages. IEEE Transactions on Software Engineering, 25(3), May/June1999.

[Kam90] Samuel N. Kamin. **Programming Languages: An Interpreter-Based Approach**. Addison-Wesley, 1990.

[Kar01] E. Karnitis. **E-Government: An Innovative Model of Governance in the Information Society**. *Baltic IT&T Review*, 1, 2001

[Kin95] W. Kinnersley, ed., **The Language List**. 1995. http://wuarchive.wustl.edu/doc/misc/lang-list.txt

[Knu68] D. E. Knuth. **Semantics of context-free languages**. *Mathematical Systems Theory*, 2(2):127-145, 1968. Errata 5(1):95-96, 1971.

[Küh97] T. Kühne. **The Translator Pattern – External Functionality with Homomorphic Mappings**. Proceedings of the Tools-23: Technology of Object-Oriented Languages and Systems, pp. 48-59, 1997.

[KW94] U. Kastens and W. M. Wait. **Modularity and reusability in attribute grammars**. *Acta Informatica 31,* pages 601-627,1994.

[Lou97] Kenneth C. Louden. **Compilers and Interpreters**. In Tucker [Tuc97], pp.2120-2147.

[ML98] M. Mezini and K. Lieberherr. **Adaptive Plug-and-Play Components for Evolutionary Software Development**. SIGPLAN Notices, 33(10):97-116, 1998. Proceedings of the 1998 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA '98).

[MLAŽ00] M. Mernik, M. Lenič, E. Avdičauševič, and V. Žumer. **Compiler/Interpreter Generator System LISA.** *Proceedings of the 33rd Hawaii International Conference on System Sciences - 2000*, pp.10, 2000.

[MLAŽ98] M. Mernik, M. Lenič, E. Avdičauševič, and V. Žumer. **The Template and Multiple Inheritance Approach into Attribute Grammars.** *Proceedings of the 1998 International Conference on Computer Languages*, pp.102-110, 1998.

[MoT98a] **The Latvian national program "Informatics"**, Ministry of Transportation, 1998, 211 pp.

[MoT98b] **The Latvian national program "Informatics" (summary)**, Ministry of Transportation, 1998, 60 pp.

[MT98] T. Matsuzaki and T. Tokuda. **CTAG Software Generator Model for Constructing Network Applications.** *Proceedings of the Asia Pacific Software Engineering Conference*, pp.120-127, 1998.

[MŽLA99] M. Mernik, V. Žumer, M. Lenič, and E. Avdičauševič. **Implementation of Multiple Grammar Inheritance in the tool LISA.** *ACM SIGPLAN Notices*, 34(6):68-75, June 1999.

[OW99] J. Ovlinger and M. Wand. **A Language for Specifying Recursive Traversals of Object Structures.** SIGPLAN Notices, 34(10):70-81, 1999. Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA '99).

[Paa95] J. Paakki. **Attribute Grammar Paradigms – A High-Level Methodology in Language Implementation.** ACM Computing Surveys, 27(2):196-255, June 1995.

[Pag81] Frank G. Pagan. **Formal Specification of Programming Languages: A Panoramic Primer.** Prentice-Hall, 1981.

[R98a] **The Baltic States Government Data Transmission Network: Conceptual and Methodological Considerations**, Riga, 1998, 11 pp.

[R98b] **The Baltic States Government Data Communications Network. Feasibility Study for a Data Networking Concept to Improve the Interchange of Information Among the Baltic States**, Riga, 1998, 83 pp.

[R98c] **The Integrated State Significance Information System (Megasystem): Conceptual and Methodological Considerations**, Riga, 1998, 16 pp.

[Rei96] S. P. Reiss. **Software Tools and Environments.** ACM Computing Surveys, Vol. 28, N0. 1, March 1996.

[RLV+96] T. H. Romer, D. Lee, G. M. Voelker, A. Wolman, W. A. Wong, J. L. Baer, B. N. Bershad, and H. M. Levy. **The Structure and Performance of Interpreters.** ASPLOS VII, USA, 1996

[Rob97] P. Robertson. **Integrating Legacy Systems with Modern Corporate Applications.** *Communications of the ACM*, 40(5):39-46, May 1997.

[Sal98] P. H. Salus, editor. **Handbook of Programming Languages Vol. III: Little Languages and Tools.** Macmillan Technical Publishing, 1998.

[Sam96] Jean E. Sammet. **FROM HOPL TO HOPL-II (1978-1993): 15 Years of Programming Language Development.** In Bergin and Gibson [BG96], pp.16-23.

[Sch96] D. A. Schmidt. **On the need for a popular formal semantics.** *ACM Computing Surveys*, 28(4es), 1996. Elektronic supplement: Strategic Directions in Computing Research.

[Sch97] David A. Schmidt. **Programming Language Semantics**. In Tucker [Tuc97], pp.2237-2254.

[Sin98] N. Singh. **Unifying Heterogeneous Information Models**. *Communications of the ACM*, 41(5):37-44, May 1998.

[SK95] K. Slonneger and B. L. Kurtz. **Formal Syntax and semantics of Programming Languages: A Laboratory Based Approach**. Addison-Wesly, 1995.

[Slo97] A. M. Sloane. **Generating Dynamic Program Analysis Tools**. Proceedings of the Autralian Software Engineering Conference (ASWEC'97), pp.166-173, 1997.

[TAB+97] Tomasic A, Amouroux R, Bonnet P, Kapitskaia O, Naacke H, and Raschid L. **The distributed information search component (disco) and the World Wide Web**. *Proceedings of ACM SIGMOD International Conference on Management of Data*, Tuscon, Arizona, 1997, Prototype Demonstration.

[Tuc97] Allen B. Tucker, editor. The computer science and engineering handbook. CRC Press, 1997.

[Vis01] J. Visser. **Visitor Combination and Traversal Control**. SIGPLAN Notices, 36(11):270-282, 2001. Proceedings of the 1998 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA '01).

[Waria] **Workflow/BPR Tools Vendors**, http://www.waria.com/databases/wfvendors-A-L.htm

[Wfmc] **Workflow Standards and Associated Documents**, http://www.wfmc.org/standards/docs/Stds_diagram.pdf

[ZZ97] D.-Q. Zhang and K. Zhang. **Reserved Graph Grammar: A Specification Tool For Diagrammatic VPLs**. Proceedings of the 1997 IEEE Symposium on Visual Languages (VL '97), pages 292-299, 1997.