ŽILINSKÁ UNIVERZITA
FAKULTA RIADENIA A INFORMATIKY

CASE nástroj pre simulačnú knižnicu ABAsim

Diplomová práca

Študijný program:	Aplikovaná informatika
Školiteľ:	Ing. Peter Jankovič, PhD.
Pracovisko:	Katedra matematických metód a operačnej analýzy

Žilina, Apríl 2015 Bc. Tomáš Hakel

Abstrakt

Táto práca sa zaoberá návrhom a implementáciou CASE nástroja pre tvorbu simulačných modelov. Nástroj poskytuje niekoľko typov diagramov pre popis simulačných modelov na rôznej úrovni detailu. Výstupom nástroja je generovaný zdrojový kód, ktorý je možné upravovať dodatočným generovaním sekcií modelu.

Kľúčové slová: simulácia, agent, CASE nástroj, generátor kódu

Abstract

This thesis deals with the design and implementation a CASE tool for creating simulation models. The tool provides several diagram types for describing simulation models at various levels of detail. The output of the tool is generated source code which can be updated by additional generation of model sections.

Keywords: simulation, agent, CASE tool, code generator

2. Obsah

3. Ciele diplomovej práce

Cieľom diplomovej práce je vytvorenie CASE (Computer-aided software engineering) nástroja pre simulačnú knižnicu ABAsim (Agent-Based Architecture of simulation model). Vytvorený nástroj umožní pohodlne vytvoriť a neskôr doplniť model simulačných agentov, definovať správanie agentov pomocou Petriho sieťe a generovať príslušný programový kód.

Jednotlivé časti simulačného modelu sú modelované rôznym spôsobom podľa úrovne detailu, preto je potrebných niekoľko typov diagramov (napríklad editor hierarchie agentov pre modelovanie štruktúry modelu na najvyššej úrovni a editor ABAgrafov pre modelovanie správania jednotlivých agentov).

Keďže tvorba simulačného modelu nie je jednoduchý proces je predpoklad, že tvorca bude meniť alebo rozširovať už vytvorenú časť simulačného modelu. Tomuto by malo byť prispôsobené aj generovanie kódu. Ak tvorca simulačného modelu upravil vygenerovaný kód a následne niečo doplnil do jedného z diagramov simulačného modelu, potom pri opätovnom generovaní kódu by mal byť pôvodný kód len rozšírený o zmeny v diagrame bez toho, aby bol poškodený kód, ktorý doň tvorca pripísal.

4. Architektúra ABAsim

ABAsim (Agent Based Architecture of simulation models) je architektúra pre tvorbu agentovo orientovaných simulačných modelov, ktorá bola vyvinutá na Fakulte riadenia a informatiky Žilinskej univerzity. Pre pojem agent existuje niekoľko definícií. Jednou z nich je napríklad: "Agent je zapuzdrený počítačový systém zasadený do nejakého prostredia, ktorý v ňom pružne a autonómne pôsobí so zámerom plnenia daného cieľa" (Jennings a Wooldridge, 2000). ABAsim poskytuje tri typy prostriedkov pre modelovanie prvkov skúmaného systému: riadiacich agentov, dynamických agentov a entity.

Riadiaci agenti (managing agents) tvoria hierarchickú štruktúru. Predstavujú riadiace a rozhodovacie prvky systému. Riadiaci agenti sú vytvorení na začiatku simulačného behu a existujú v modeli nepretržite až do jeho ukončenia. Majú spoločný cieľ (napríklad obslúženie zákazníkov) a navzájom spolupracujú na dosiahnutí tohto cieľa.

Dynamickí agenti (dynamic agents) sú primárne určení pre reprezentáciu autonómnych objektov modelovaného systému, ktoré disponujú istou mierou vlastnej inteligencie a sú schopní samostatne riešiť pridelené úlohy. Môžu byť vytváraní a rušení počas behu simulačného modelu. Netvoria žiadnu pevnú štruktúru. Každý dynamický agent je vždy pod správou niektorého z riadiacich agentov, ktorý mu prideľuje úlohy.

Entity predstavujú prvky skúmaného systému, ktoré buď nedisponujú inteligenciou, alebo je ich riadenie ponechané na riadiacich agentoch.

V architektúre ABAsim samotný agent nepredstavuje atomický prvok. Je zložený z interných komponentov, ktoré zabezpečujú vykonávanie základných funkcií agenta. Komponenty môžu byť niekoľkých typov: Manažér, ktorý riadi správanie agenta, Poštár ktorý sprostredkuje komunikáciu medzi agentami, Správca času, ktorý riadi synchronizáciu simulačného času a Asistent, ktorý môže plniť rôzne funkcie.

Asistentov môžeme rozdeliť z viacerých hľadísk.

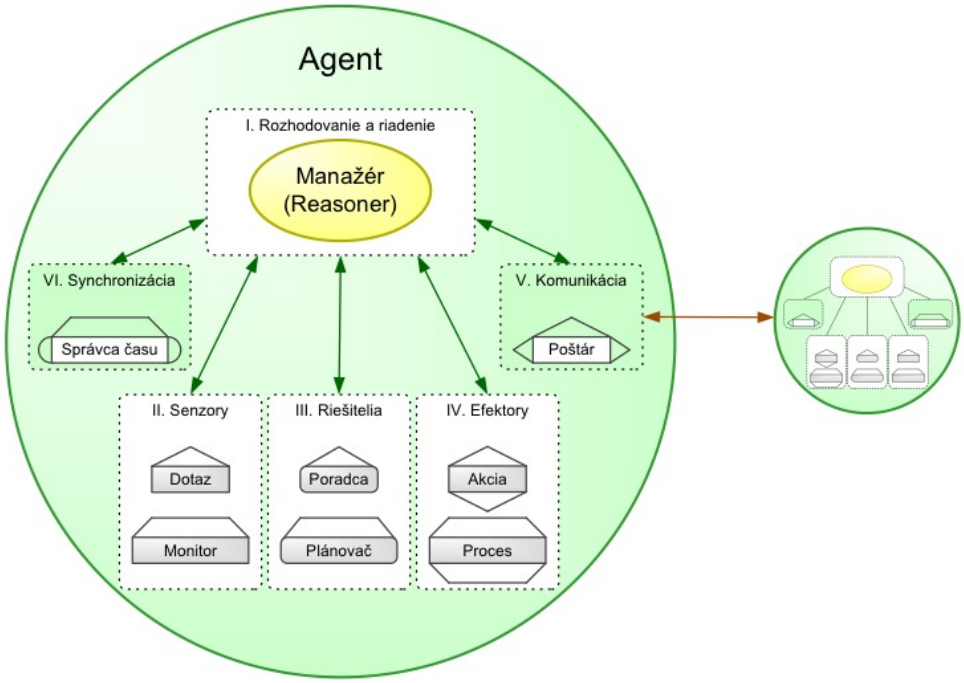
Z hľadiska funkcie asistentov:

1. Senzory (dotaz a monitor) zabezpečujú sprístupnenie informácií o stave prostredia,
2. Riešitelia (poradca a plánovač) podporujú rozhodovanie manažéra tým, že mu poskytujú návrhy riešenia problémov,
3. Efektory (akcia a proces) vykonávajú rozhodnutia manažéra o zmene stavu systému. Žiadne iné komponenty by nemali meniť stav systému.

Z hľadiska trvania činnosti asistentov:

1. Okamžití asistenti (dotaz, poradca, akcia) vykonajú svoju činnosť okamžite bez zmeny simulačného času,

2. Kontinuálni asistenti (monitor, plánovač, proces) vykonajú činnosť, ktorá trvá nenulový simulačný čas.



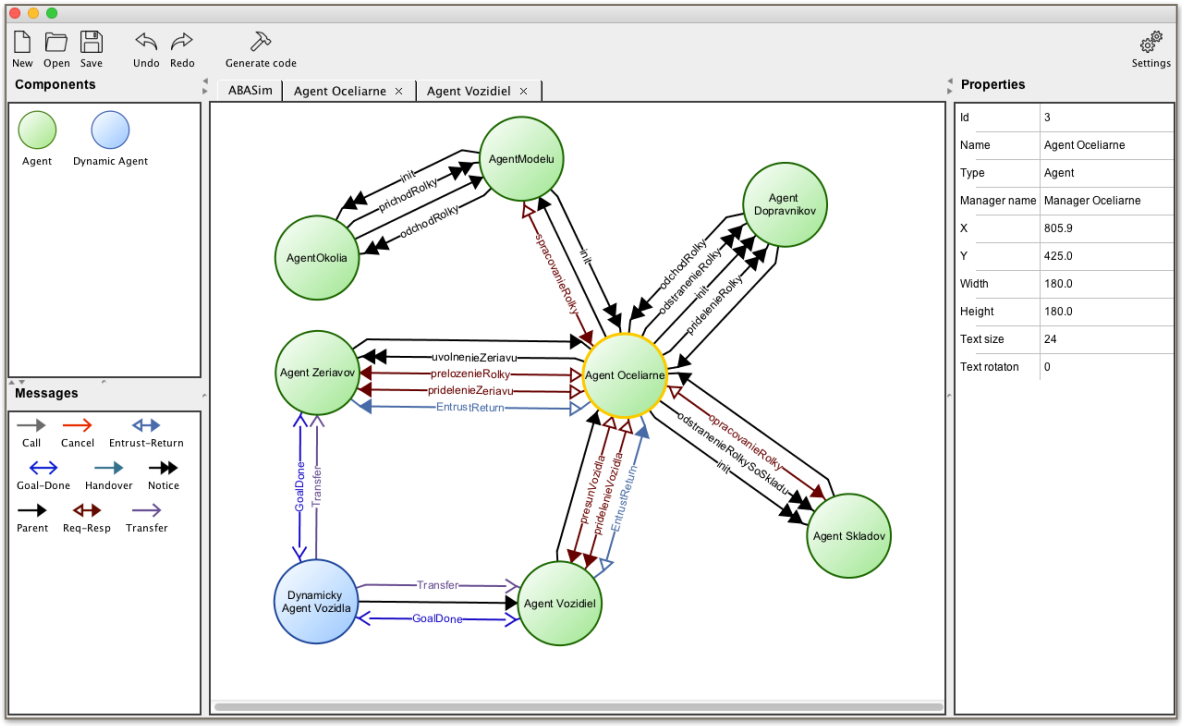
Obr. 4.1: Komponenty agenta

Agenti, ako aj komponenty v rámci agenta, spolu komunikujú zasielaním správ. Každý agent obsahuje poštovú schránku, do ktorej sú doručované správy, odkiaľ si ich agenti vyberú a doručia danému internému komponentu.

Popis správania agenta (implementácia komponentu manažér) môže byť vyjadrený Petriho sieťou alebo realizovaný programom implementujúcim reakcie agenta na doručenie správ s konkrétnym kódom od konkrétneho odosielateľa (agent môže reagovať rôzne na správu s tým istým kódom podľa toho, od ktorého odosielateľa bola doručená).

5. CASE nástroj ABAbuilder

ABAbuilder je CASE (computer-aided software engineering) nástrojom, ktorého úlohou je zjednodušenie tvorby simulačných modelov.



Obr. 5.1: Nástroj ABAbuilder

Aktuálny stav implementácie simulačného modelu bez použitia CASE nástroja je taký, že je použité simulačné jadro (softvérová knižnica) a celý simulačný model je implementovaný napísaním kódu. Pomerne veľké množstvo kódu je konfigurácia simulačného jadra (registrácia správ, deklarácia identifikárorov, definícia hierarchie agentov), ktorú je nutné napísať, ale nijako neprispieva k implementácii špecifík simulačného modelu. Bolo by teda vhodné tvorbu tejto konfigurácie automatizovať.

CASE nástroj ABAbuilder, ktorý som vytvoril, zjednodušuje tvorbu simulačných modelov tým, že umožňuje simulačné modely vytvárať vizuálne v grafickom editore a poskytuje možnosť generovania kódu. Zahŕňa tiež editor ABAgrafu (typ Petriho siete) a generuje súbory, ktoré sú spracované interpreterom ABAgrafov za behu simulácie.

Nástroj ABAbuilder generuje kódy simulačných modelov pre jazyky Java a C#. Keďže bola architektúra ABAsim inplementovaná iba pre jazyk Delphi vytvoril som simulačné jadrá pre jazyky Java a C#.

Model simulačného modelu

Model aplikácie tvorí štruktúra objektov reprezentujúcich prvky simulačného modelu. Ich atribúty môžu patriť do jednej z dvoch skupín:

- atribúty, ktoré budú použité pri generovaní kódu simulačného modelu, ktoré majú priamy vplyv na beh simulácie,
- atribúty, ktoré využíva iba nástroj ABAbuilder. Väčšinou sa jedná o údaje reprezentujúce grafické prvky (veľkosť, pozícia, poradie hrany, ...), ale aj interné identifikátory komponentov.

Simulačné jadro využíva na identifikáciu jednotlivých komponentov celočíselný identifikátor. Tento identifikátor ale nie je možné použiť aj pre identifikáciu komponentov modelu nástroja, pretože je používateľovi umožnené ho meniť. To by spôsobovalo problémy, keby bol tento identifikátor použitý ako kľúč (hash) v štruktúre, v ktorej sú uložené údaje patriace k danému komponentu a tiež pri opätovnom generovaní kódu. Keďže som potreboval jedinečný identifikátor komponentov a prepisovať hodnoty identifikátora v celej aplikácii pri každej zmene atribútu identifikátora by zrejme nebolo praktické, rozhodol som sa pre každý objekt vygenerovať dva identifikátory: simId a internalId. simId je identifikátor využívaný na identifikáciu komponentov a správ za behu simulácie a internalId je identifikátor, ktorý sa používa pri tvorbe modelu a generovaní kódu.

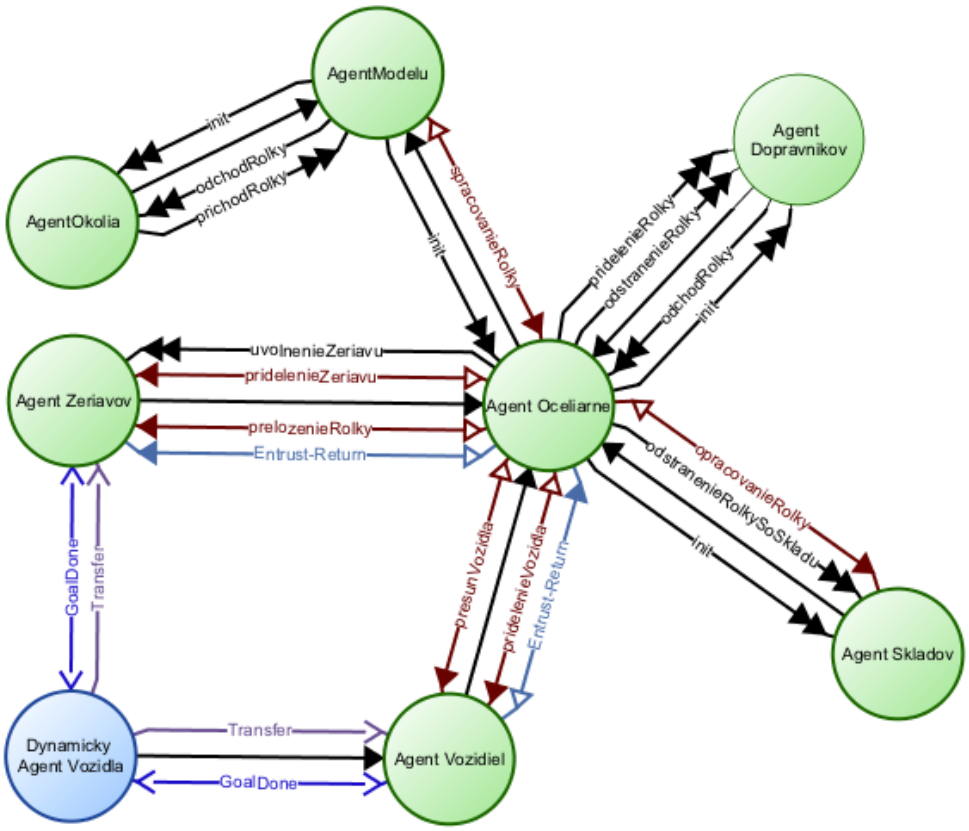
6. Editácia modelu

Nástroj využíva niekoľko typov diagramov, ktoré popisujú simulačný model na rôznej úrovni detailu. Tvorca modelu môže pridať nový prvok tým, že vyberie prvok požadovaného typu v paneli s komponentami v ľavej časti nástroja a klikne myšou na

prázdne miesto diagramu v prípade pridávania komponentu, alebo spojí dva komponenty ťahaním myši v prípade pridávania hrany. V paneli s atribútami potom nastaví požadované hodnoty atribútov komponentov.

Diagram hierarchie agentov

Popisuje simulačný model na najvyššej úrovni. Definuje jednotlivých agentov, ich hierarchiu a správy, ktoré si posielajú. Agent môže byť dvoch typov: riadiaci agent a dynamický agent. To akého je agent typu determinuje, aké správy vie odoslať a prijať.



Obr. 6.1: Diagram hierarchie agentov

Agenti

Diagram hierarchie agentov poskytuje dva typy agentov:

Riadiaci agenti tvoria pevnú štruktúru, ktorá je definovaná pri spustení simulácie a počas behu simulácie sa už nemení. Riadiaci agent modeluje špecifickú časť systému, pričom

môže modelovanie určitých častí delegovať podriadeným agentom. V diagrame je reprezentovaný elipsou zelenej farby.

Dynamickí agenti reprezentujú entity simulačného modelu, ktoré sa vedia do istej miery sami rozhodovať a riešiť problémy. V diagrame je reprezentovaný elipsou modrej farby.

Obr. 6.2: Typy agentov

Hrany, ktoré spájajú agentov môžu byť dvoch typov: správy a vzťahy.

Parent je hrana, ktorá špecifikuje vzťah nadradeného agenta a tým definuje hierarchickú štruktúru simulačného modelu. Je jedinou hranou diagramu, ktorá nie je správou. Je niekoľko obmedzení spôsobu, akým môžu byť agenti prepojený hranou parent. Nástroj používateľovi nedovolí prepojiť agentov hranou parent neprípustným spôsobom. Agenti tvoria strom (acyklický graf). Agent na vrchole hierarchie, nazývaný boss, jediný nemá predka. Pri generovaní kódu je v prípade, že používateľ nedefinoval hierarchiu s jedným agentom bez predka, zobrazené upozornenie. Nástroj nedovolí spojiť dvoch agentov vzťahom typu parent, ak by bola porušená jedna z podmienok:

- agent má najviac jedného predka
- nemôže existovať orientovaný cyklus z hrán typu parent

Správy

Správy sú spôsobom, akým spolu agenti komunikujú. Je niekoľko typov správ, ktoré si medzi sebou môžu agenti posielať, každý pre špecifický účel.

Obr. 6.3: Typy správ diagramu hierarchie agentov

Notice (oznámenie) je základným typom správy, na ktorú sa neočakáva žiadna odpoveď. Agenti ju používajú na informovanie iných agentov o vzniknutej situácii, prípadne ich žiadajú o vykonanie činností, o ktorých uskutočnení nepotrebujú byť informovaní.

Request (žiadosť) je správa obsahujúca určitú požiadavku, pričom odosielateľ očakáva na ňu od adresáta odpoveď vo forme správy typu Response.

Response (odozva) je správa, ktorá reprezentuje odpoveď na správu typu Request, pričom môže byť doručená iba jej odosielateľovi.

Call (telefonát) predstavuje možnosť okamžitej komunikácie medzi agentami. Správa je prijímajúcemu agentovi doručená okamžite, mimo štandardného postupu doručovania správ.

Goal (cieľ) je správa, ktorú využívajú riadiaci agenti na priradenie cieľov dynamickým agentom. Dynamickí agenti sú povinní plniť priradené ciele, no môžu sledovať aj vlastné lokálne ciele, ak tieto nie sú v rozpore s cieľom priradeným riadiacim agentom. Po splnení cieľa dynamický agent informuje o tejto skutočnosti riadiaceho agenta zaslaním správy typu Done.

Done (hotovo) je správa, ktorú zasiela dynamický agent svojmu riadiacemu agentovi v okamihu splnenia priradeného cieľa. Riadiaci agent na túto správu zvyčajne reaguje zadaním nového cieľa danému dynamickému agentovi (zaslaním správy typu Goal).

Transfer (presun) je správa, ktorú dynamický agent zasiela riadiacemu agentovi, aby mu signalizoval, že sa práve snaží opustiť oblasť jeho pôsobnosti. Riadiaci agent po prijatí tejto správy zvyčajne identifikuje riadiaceho agenta, ktorý je zodpovedný za oblasť, do ktorej dynamický agent vstupuje a odovzdá mu ho.

Cancel (zrušenie) je správa, ktorú riadiaci agent využíva na zrušenie v minulosti zadaných cieľov dynamickým agentom.

Handover (odovzdanie) je správa, ktorú využíva riadiaci agent na presunutie dynamického agenta do pôsobnosti iného riadiaceho agenta. Prijímajúci riadiaci agent sa stáva novým riadiacim agentom daného dynamického agenta.

Entrust (zverenie) je správa, ktorou riadiaci agent dočasne odovzdáva spravovanie daného dynamického agenta inému riadiacemu agentovi. Očakáva sa, že dynamický agent po vykonaní definovaných úkonov opätovne prejde do pôsobnosti odosielateľa tejto správy. Dočasné odovzdávanie dynamických agentov je možné vykonávať aj viacnásobne.

Return (návrat) je správa využívaná pri návrate spravovania dynamického agenta pôvodnému riadiacemu agentovi, ktorý spravovanie tohto dynamického agenta dočasne odovzdal inému riadiacemu agentovi správou typu Entrust.

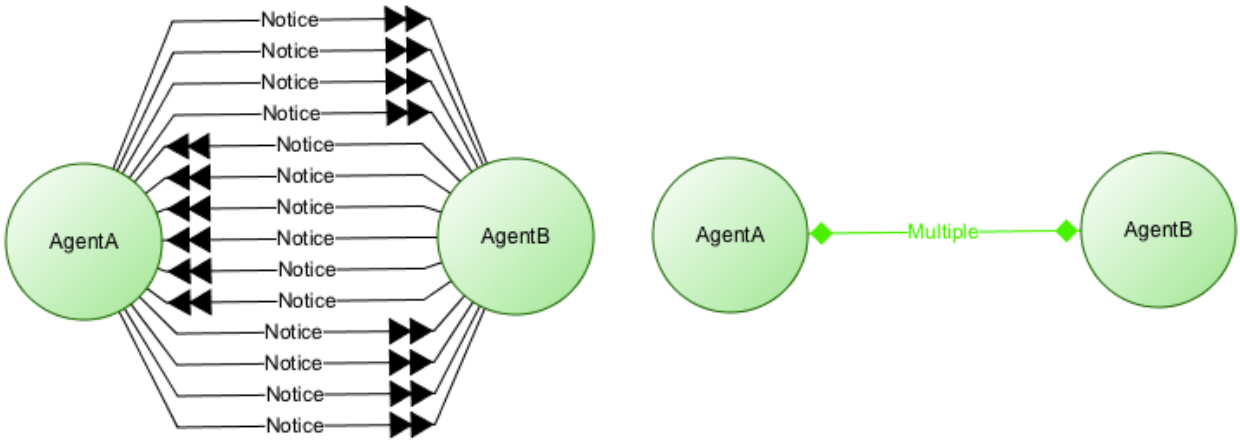
Typy správy, ktoré môžu byť odoslané, sú obmedzené typom agenta adresáta a agenta odosielateľa.

- Riadiaci agent môže poslať inému riadiacemu agentovi správy typu call, entrust, return, handover, notice, request a response.
- Riadiaci agent môže poslať dynamickému agentovi správy typu call, cancel, goal a notice.
- Dynamický agent môže poslať riadiacemu agentovi správy typu call, done, notice a transfer.
- Dynamický agent môže poslať dynamickému agentovi správy typu call a notice.

Nástroj nedovolí spojiť dvoch agentov správou, ktorá nie je validná.

Jedným z atribútov agenta, ktorý môže tvorca modelu nastaviť v paneli s atribútami, je atribút "Use ABAgraph", ktorý vyjadruje, či je správanie daného agenta popísané ABAgrafom. Je dvojhodnotový, preto je preň v paneli použitý checkbox. Od hodnoty tohto atribútu okrem iného závisí aj to, ktorý diagram sa zobrazí po dvojkliku na agenta (diagram komponentov agenta alebo diagram ABAgrafu).

Medzi dvoma komponentami môže byť správ pomerne veľa, preto je pre prehľadnosť možné skryť správy posielané medzi dvoma komponentami dvojklikom na jednu z hrán spájajúcich tieto komponenty. Dvojklikom na hranu "Multiple" sa opäť rozbalia.



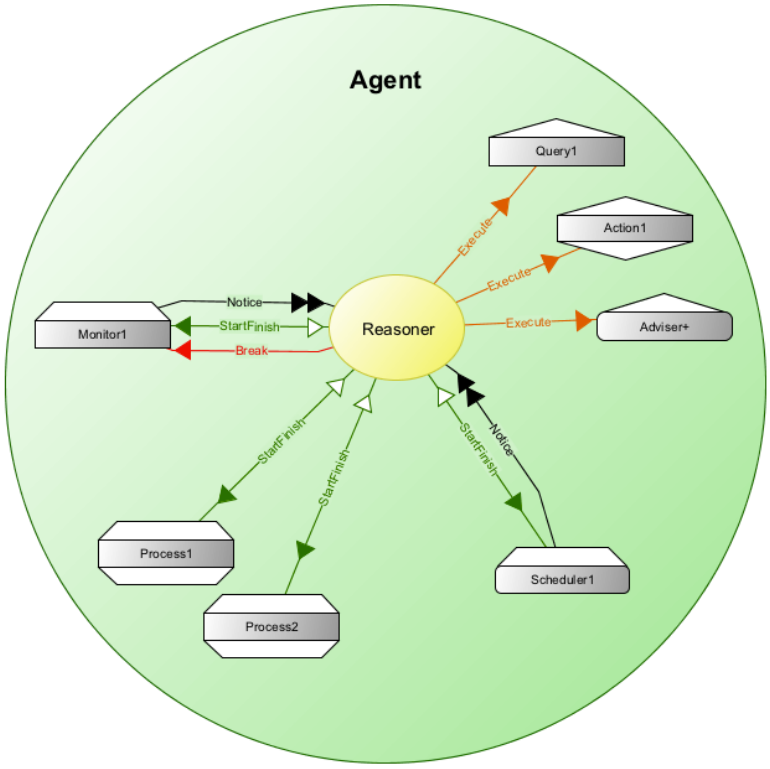
Obr. 6.4: Skrytie správ posielaných medzi dvoma agentami

Diagram komponentov agenta

Diagram modeluje vnútornú štruktúru agenta.

Skladá sa z komponentov agenta, ktorými sú manažér a asistenti, a správ, ktoré si posielajú. Asistenti komunikujú výhradne s manažérom. Nemajú dovolené posielať si správy navzájom ani komunikovať s inými agentami. Nástroj ABAbuilder preto nedovolí vytvorenie hrany medzi dvoma asistentami.

Tento diagram je možné otvoriť v diagrame komponentov agenta dvojklikom na agenta, ak je nastavené, že tento agent nepoužíva na popis svojho správania ABAgraf. To potom znamená, že popis správania agenta nebude vygenerovaný graficky nástrojom ABAbuilder, ale používateľ ho napíše ako kód pripísaním do vygenerovaného zdrojového kódu, ktorý je výstupom nástroja.

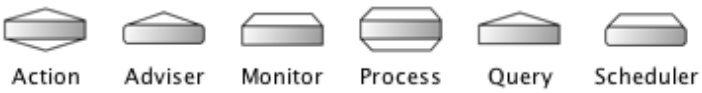


Obr. 6.5: Diagram komponentov agenta

Manažér (reasoner) je v diagrame vytvorený implicitne pri vytvorení agenta. Používateľ ho nemôže zmazať ani vytvoriť ďalšieho. To znamená, že agent má vždy práve jedného manažéra. Jeho úlohou je riadenie agenta. Využíva pri tom ostatné komponenty agenta - asistentov (agent obsahuje aj ďalšie implicitné komponenty, ktoré sú ale pre tvorcu simulačného modelu skryté).

Asistenti

Asistenti sú dvoch typov: kontinuálny a okamžitý. Sú od seba graficky rozlíšený tým, že horná (pri procese a akcii aj dolná) hrana kontinuálnych asistentov je plochá, zatiaľ čo pri okamžitých asistentoch tvorí trojuholník.



Obr. 6.6: Typy asistentov

Sú tri druhy kontinuálnych asistentov:

Proces (process) je komponentom, ktorý vykonáva rozhodnutia manažéra o zmene stavu systému. Po spustení autonómne mení hodnoty stavových premenných.

Plánovač (scheduler) je komponentom, ktorý podporuje rozhodovanie manažéra tým, že mu poskytuje návrhy riešenia problémov. Plánovač v predstihu pre istý časový interval vytvára plány riešení problémov, ktoré sú založené na predikcii vzniku situácií vyžadujúcich rozhodnutie. V zvolených časových okamihoch vykonáva svoju funkciu aj bez iniciatívy manažéra.

Monitor (monitor) je komponentom, ktorý sprístupňuje informácie o stave prostredia. Vykonáva dlhodobejšie snímanie istej časti prostredia a sprostredkuje manažérovi tie informácie, ktoré sú pre neho významné.

Sú tri druhy okamžitých asistentov:

Akcia (action) je komponentom, ktorý vykonáva rozhodnutia manažéra o zmene stavu systému. Zabezpečuje okamžitú zmenu stavu.

Poradca (adviser) je komponentom, ktorý podporuje rozhodovanie manažéra tým, že mu poskytuje návrhy riešenia problémov. Poradca je pasívny komponent, ktorý iba na pokyn manažéra okamžite navrhne spôsob riešenia problému. Poradcom je obvykle optimalizačný algoritmus alebo človek.

Dotaz (query) je komponentom, ktorý sprístupňuje informácie o stave prostredia. Dotaz sprostredkuje informáciu na pokyn manažéra okamžite.

Pri vytvorení kontinuálneho asistenta sú medzi ním a manažérom automaticky vytvorené správy start a finish a pri vytvorení okamžitého asistenta je medzi ním a manažérom automaticky vytvorená správa execute - jediná správa, ktorú vie okamžitý asistent spracovať.

Správy

Podobne ako v diagrame hierarche agentov môžu byť správy posielané v rámci agenta niekoľkých typov.

Obr. 6.7: Typy správ diagramu komponentov agenta

Start (štart) je správa, po ktorej prijatí začne kontinuálny asistent, ktorý je jej adresátom, svoju činnosť.

Break (zrušenie) je správa, ktorou môže manažér ovplyvniť beh kontinuálneho asistenta, ktorý je po prijatí tejto správy ukončený. Tento typ správy sa používa, keď už nemá zmysel, aby kontinuálny asistent dokončil svoju činnosť.

Execute (vykonaj) je špecifickým typom správy určenej výhradne pre okamžitého asistenta, ktorý okamžite vykoná zodpovedajúcu činnosť (identifikovanú kódom a parametrami správy) a okamžite správu vráti manažérovi s vyznačeným výsledkom svojej činnosti. Správy tohto typu obchádzajú mechanizmus doručovania prostredníctvom poštovej schránky agenta a sú doručované priamo a okamžite, čo zvyšuje efektívnosť činnosti agenta. Tento postup je možné zvoliť nakoľko adresátom tohto typu správ môžu byť výlučne okamžití asistenti, vykonávanie ktorých trvá nulový simulačný čas.

Finish (koniec) je správa, ktorú každý z kontinuálnych asistentov povinne pošle svojmu manažérovi v okamihu ukončenia svojej činnosti.

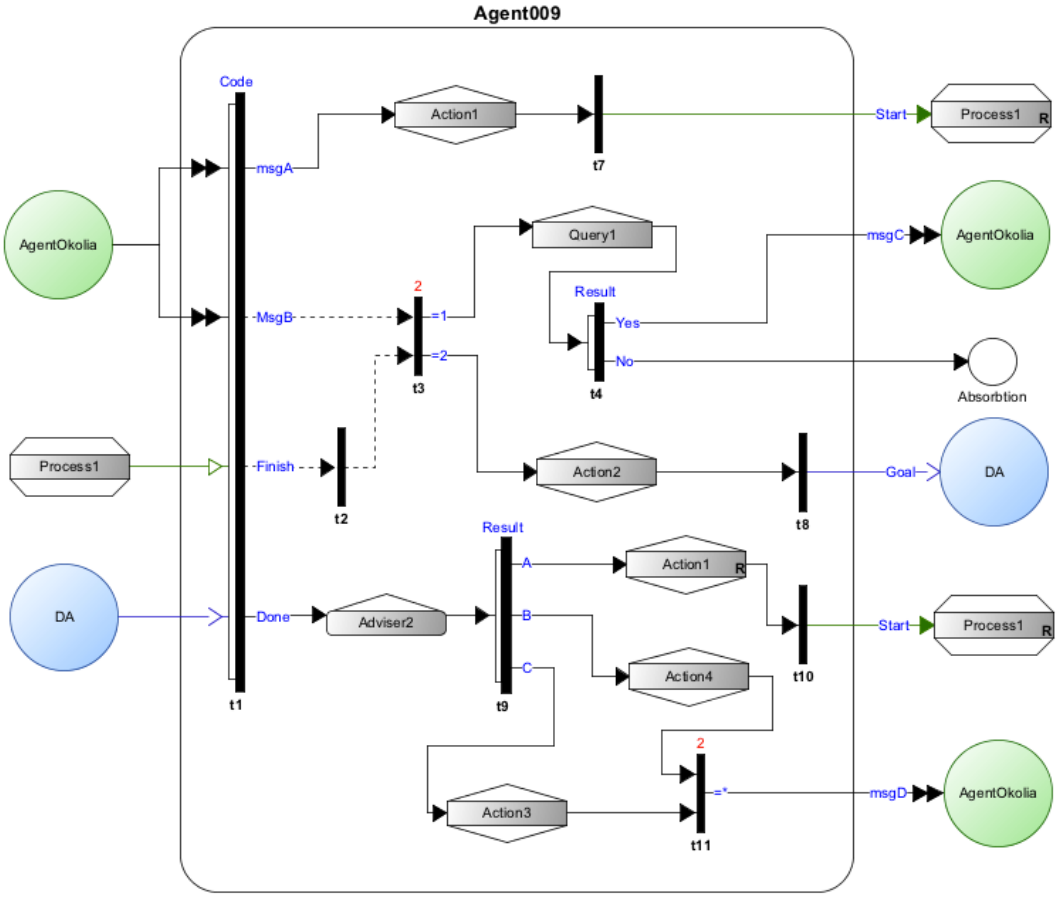
Notice (oznámenie) sú správy, ktoré môžu byť posielané kedykoľvek v priebehu činnosti kontinuálneho asistenta. Asistent ich využíva na informovanie manažéra o dôležitých skutočnostiach vyžadujúcich jeho reakciu.

Hold (zdrž) je jedinou správou, ktorá môže mať požadovaný čas doručenia rôzny od aktuálneho simulačného času. Pomocou tohto typu správy je v architektúre ABAsim realizovaný posun simulačného času. Správu zasiela kontinuálny asistent vždy iba sám sebe.

Diagram ABAgrafu

ABAgraf je modifikovaná Petriho sieť popisujúca správanie agenta.

Diagram je možné otvoriť v diagrame komponentov agenta dvojklikom na agenta ak je nastavené, že agent používa na popis svojho správania ABAgraf.



Obr. 6.8: Diagram ABAgrafu

Správanie agenta je definované spôsobom, akým reaguje na správy. Množina správ, ktoré príjíma ABAgraf je daná atribútmi hrán vychádzajúcich zo vstupných miest ABAgrafu. Môže sa jednať o tie isté správy, ktoré boli definované v diagrame hierarchie agentov a v takom prípade pri generovaní kódu s nimi musí nástroj pracovať, ako by sa jednalo o tú istú hranu. Toto je realizované párovaním hrán podľa ich názvu. Okrem toho je používateľ vizuálne notifikovaný o tom, že došlo k spárovaniu tým, že z tabuľky s atribútmi zmiznú niektoré riadky, ktoré už nemajú zmysel pre spárovanú hranu.

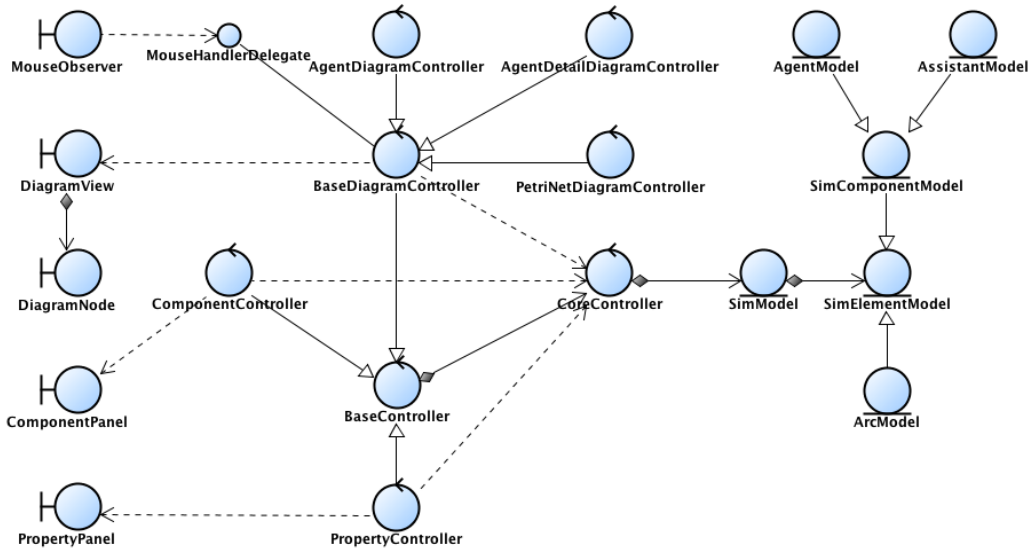
Nechcel som, aby bol tvorca modelu príliš obmedzený spôsobom, akým nástroj pracuje so spárovanými hranami a teda nepožadujem, aby boli všetky správy spárované. To umožnuje, aby boli správy definované v kóde a súčasne spracované interpreterom ABAgrafu. Môžu nastať tri prípady:

- Správa existuje aj v diagrame hierarchie agentov aj v ABAgrafe. V tom prípade sú príslušné hrany spárované a nástroj zohľadnuje to, že hrany reprezentujú tú istú správu.
- Správa existuje iba v diagrame hierarchie agentov.
- Správa existuje iba v ABAgrafe.

V aktuálnej verzii nástroja je pri práci so spárovanými hranami jedno obmedzenie. Hrany musia byť definované v správnom poradí - najskôr v diagrame hierarchie agentov, až potom v ABAgrafe.

7. Architektúra aplikácie

Aplikácia je postavená na architektúre MVC (Model View Controller). Jednotlivé triedy sú rozdelené do týchto troch vrstiev.



Obr. 7.1: Zjednodušený diagram architektúry aplikácie ABAbuilder

Model

Vrstva model obsahuje triedy reprezentujúce prvky simulačného modelu (agentov, asistentov, správy a vzťahy medzi nimi). Je zodpovedná za serializáciu modelu. Vrstva definuje pre jednotlivé prvky rozhranie, s ktorým pracuje zvyšok aplikácie. Toto rozhranie je implementované viacerými triedami, ktoré sú troch typov:

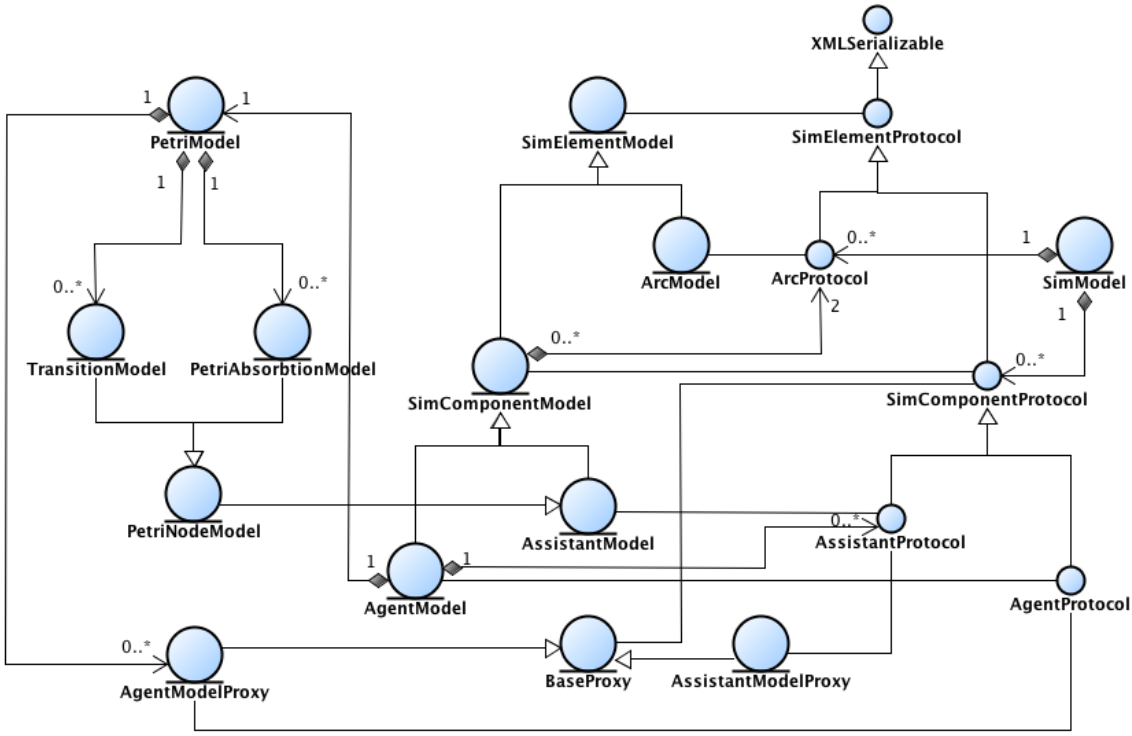
Komponent model je priamou reprezentáciou komponentu. Sú v ňom uložené všetky informácie o danom komponente a niektorých v danom kontexte podstatných vzťahoch s ostatnými komponentami.

Komponent proxy je nepriamou reprezentáciou komponentu. Je použitý v prípade, že jeden komponent simulačného modelu má byť v diagrame zobrazený na niekoľkých miestach súčasne. V tom prípade jednotlivé uzly v diagrame síce reprezentujú jeden komponent, ale môžu byť prepojené rôznymi hranami. Je implementáciou návrhového vzoru dekorátor. Väčšinu operácií trieda deleguje svojmu originálu, ale niektoré špecifické operácie (napríklad umiestnenie v diagrame) vykonáva iným spôsobom.

Null model nereprezentuje žiadny komponent. Všetky implementované operácie sú prázdne. Je použitý ako model grafických prvkov, ktoré nemajú reprezentáciu v simulačnom modeli, napríklad: kontrolné uzly na hranách pri vytváraní novej hrany pred spojením s cieľovým uzlom je cieľom dočasný uzol, ktorého modelom je null komponent. Dôvodom, prečo táto trieda existuje je to, že k zobravovačom vrstvy view má byť priradený ich model. Alternatívou by bolo namiesto triedy null object použiť jednoducho null pointer, ale rozhodol som sa pre toto riešenie, pretože vyžadovať, aby mal každý grafický prvok model znamená, že:

1. jednoduchšie sa s triedou pracuje, pretože nie je potrebné pred každou operáciou zisťovať, či model existuje,

2. väčšia odolonosť voči chybám, pretože je oddelený valídny stav, pri ktorom má čisto grafický komponent ako model null object od chybného stavu, pri ktorom nemá zobrazovač komponentu simulačného modelu vytvorený model.



Obr. 7.2: Zjednodušený diagram tried vrstvy model

Do vrstvy model patrí aj niekoľko tried, ktoré nereprezentujú komponenty, ale väčšiu časť modelu, ktorú zoskupujú do celku. Obsahujú referencie na komponenty, ktoré spravujú. Ich úlohou je zjednodušenie práce s modelom. Vrstva controller k jednotlivým komponentom pristupuje cez tieto triedy.

SimModel je trieda, ktorá reprezentuje model simulačného modelu. Obsahuje modely agentov, asistentov a správ. Je implementovaná ako singleton. Umožňuje vyhľadať komponenty podľa ich interného aj simulačného ID.

ViewModel je trieda, ktorá obsahuje zobrazovače, v štruktúre ktorej sa dajú jednoducho vyhľadať. Je implementovaná ako singelton. Komponent vrstvy model nemá referenciu na zobrazovače vrstvy view, pretože by ich nemal priamo meniť. Zmeny modelu sú väčšinou spôsobené vstupom používateľa, ale aby bolo možné niektoré operácie automatizovať je potrebné, aby vedela vrstva controller vyhľadať zobrazovač podľa jeho modelu. Toto sprostredkuje práve trieda ViewModel.

PetriModel je trieda, ktorá reprezentuje Petriho sieť jedného agenta. Je vytvorená pre každého agenta, ktorý používa ABAgraf. Obsahuje komponenty, ktoré sú špecifické pre ABAgraf, ktorými sú napríklad prechody a absorbcie, ale neobsahuje komponenty, ktoré sú používané aj inými diagramami, napríklad hrany a asistenti. Tie sú uložené v triede SimModel, aby bolo možné použiť tú istú implementáciu controllera pre všetky diagramy.

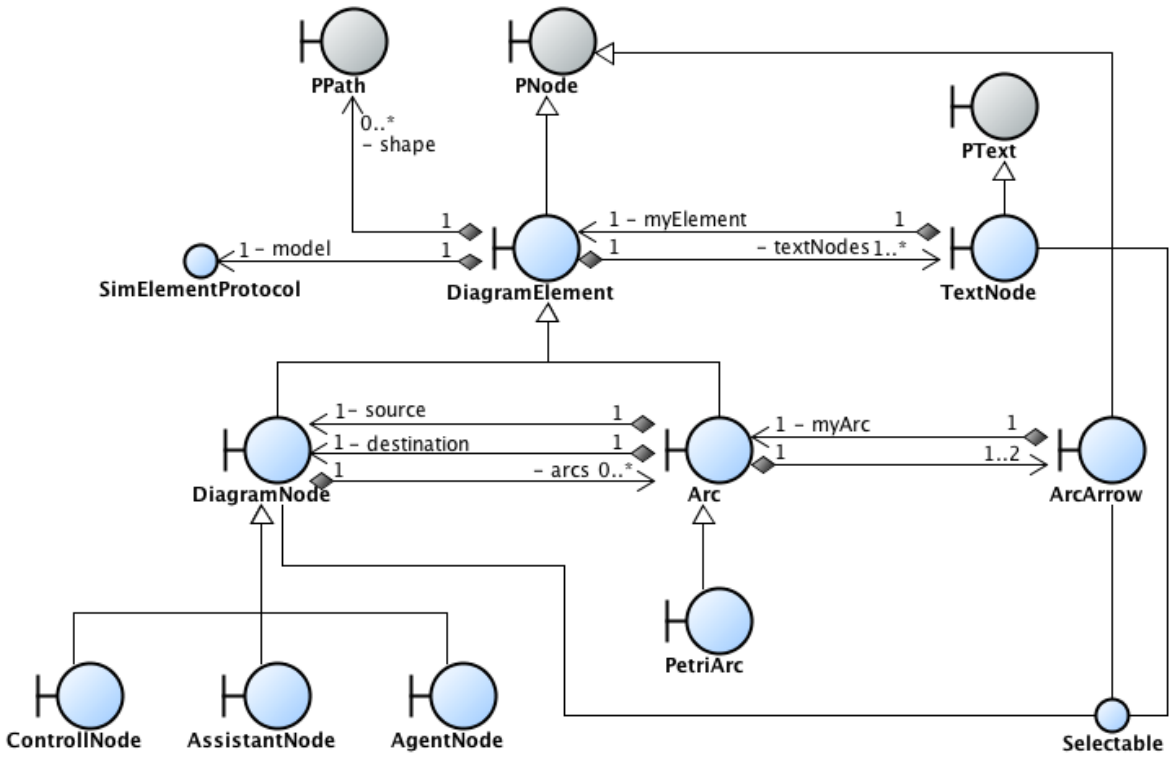
View

Vrstva view by sa dala rozdeliť na dve časti: Triedy pre vykresľovanie diagramov a triedy reprezentujúce ostatné časti grafického rozhrania (panely, tabuľky, menu, tlačidlá).

Zobrazovače komponentov

Diagram je tvorený plátnom a zobrazovačmi komponentov simulačného modelu. Pre vykresľovanie používam grafickú knižnicu Piccolo2D. Zobrazovače sú potomkami triedy DiagramElement, ktorá je potomkom triedy PNode z grafickej knižnice. Úlohou zobrazovača je nakresliť grafickú reprezentáciu komponentu na plátno.

Zobrazovovač môže implementovať rozhranie Selectable. V takom prípade používateľ naň môže kliknúť myšou, čo spôsobí zvýraznenie zobrazovača a jeho atribúty sú zobrazené v paneli s atribútmi.



Obr. 7.3: Diagram tried vrstvy view (sivé uzly sú triedy externej knižnice)

Zobrazovače sú dvoch typov: uzly a hrany.

Uzly sú dvojrozmerné zobrazovače, ktoré v diagrame zobrazujú agentov, asistentov a miesta a prechody Petriho siete. Sú inštanciami triedy DiagramNode a jej potomkov. Tvar uzlu závisí od komponentu, ktorý reprezentuje.

Hrany sú jednorozmerné zobrazovače, ktoré spájajú uzly a v diagrame zobrazujú posielané správy, vzťah definujúci hierarchu agentov a hrany Petriho siete. Sú inštanciami triedy Arc a jej potomkov. Hrana je zobrazená ako čiara nejakej farby, ktorá závisí od správy alebo vzťahu, ktorý reprezentuje. Môže obsahovať šípky pri cieľovom a zdrojovom uzle. Všetky hrany majú šípku pri cieľovom uzle. Pri zdrojovom uzle majú šípku iba správy, pre ktoré sa očakáva odpoveď.

Panel s atribútmi

V pravej časti aplikácie je panel zobrazujúci atribúty vybraných komponentov v tabuľke. Tabuľka má dva stĺpce: názov atribútu a hodnota atribútu. Šírka stĺpcov sa nastavuje automaticky podľa obsahu. Hodnoty atribútov môžu byť rôznych typov a podľa toho je použitý vhodný grafický komponent v stĺpci s hodnotami atribútov.

- Nápis (Label) je použitý v prípade, že atribút slúži len ako informácia pre používateľa, ktorý ho nemá dovolené meniť (napríklad typ komponentu).
- Textové pole (text field) je použité v prípade, že atribút je typu text alebo číslo a používateľovi je dovolené ho meniť (napríklad názov komponentu).
- Zaškrtávacie pole (checkbox) je použité v prípade, že atribút popisuje existenciu daného znaku (napríklad či agent používa ABAgraf).
- Zoznam (combobox) je použitý v prípade, že používateľ môže vybrať hodnotu atribútu len z obmedzenej množiny hodnôt (napríklad spôsob poslania správy).
- Tlačidlo (button) je použité v prípade, že komponent vie vykonať nejakú akciu, nie je použité pre konkrétny atribút (napríklad otočenie smerovania hrany).

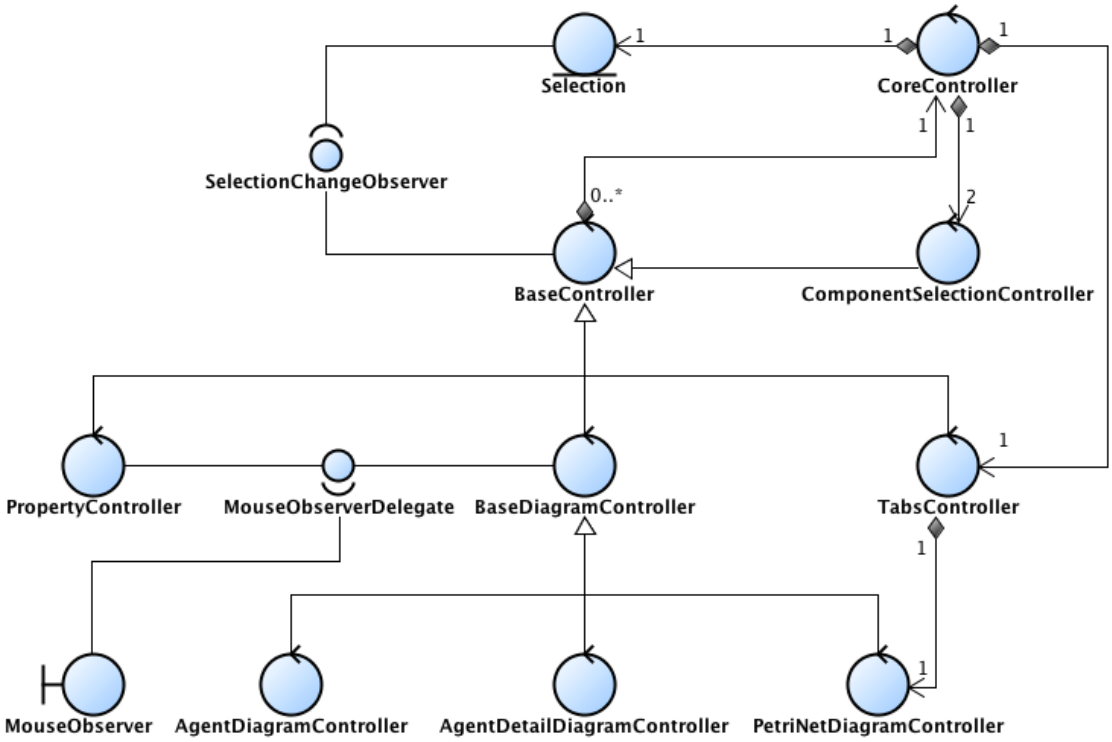
Hodnoty atribútov v tabuľke sa menia dynamicky podľa aktuálneho stavu komponentu, ktorý popisujú. Napríklad pri presúvaní komponentu ťahaním myši sú jeho súradnice priebežne aktualizované v tabuľke s atribútmi. Pri úprave názvu agenta je súčasne v tabuľke aktualizovaný aj názov manažéra, ktorý je možné určiť na základe názvu agenta (používateľ ho môže zmeniť tým, že priamo vyplní hodnotu tohto atribútu, čo však väčšinou nie je potrebné a preto je to automatizované). Počas zmeny názvu je názov súčasne menený aj na komponente v diagrame.

V niektorých prípadoch je pri upravovaní atribútu zmenený počet riadkov tabuľky. Napríklad používaťeľ môže vyberať ako spôsob poslania správy typ, pre ktorý nemá zmysel definovať názov správy (start, finish, ...).

Využívam vlastnú implementáciu tabuľky, keďže dosiahnutie vyššie popísanej funkcionality použitím štandardných komponentov Javy by nebolo praktické.

Controller

Vrstva controller je riadiacou vrstvou, ktorá preberá vstupy od vrstvy view, nastavuje hodnoty modelu. Je zodpovedná za validáciu vstupov od používateľa. Základnou triedou tejto vrstvy je trieda BaseController, od ktorej je odvodených niekoľko tried, ktorých úlohou je riadiť jednotlivé časti aplikácie.



Obr. 7.4: Zjednodušený diagram tried vrstvy controller

PropertyController riadi panel s atribútmi v pravej časti aplikácie. Zobrazuje údaje o aktuálne vybraných komponentoch aplikácie. Zobrazené údaje sú kontextovo senzitívne, teda jednotlivé položky závisia od aktuálneho výberu, ktorý ale nie je daný len typom vybraného komponentu, ale aj stavom tohto komponentu a jeho susedných komponentov. Napríklad zobrazené atribúty hrany ABAgrafu sa môžu líšiť v závislosti od toho, aké komponenty spája (môžu nimi byť prechody a miesta, naviac je rozdiel, či sa jedná o vstupné, interné alebo externé miesta) a stavu správy, ktorá môže už byť definovaná, napríklad aj v inom diagrame, alebo novovytvorená.

TabsController riadi záložky v hornej časti aplikácie. Prvá záložka vždy obsahuje diagram hierarchie agentov. Ostatné záložky obsahujú diagramy ABAgrafov. Úlohou tohto controllera je zabezpečiť, aby bol v strednej časti aplikácie zobrazený správny diagram a notifikovať príslušné časti aplikácie o zatvorení a prepnutí záložiek. Počas behu aplikácie je vytvorený najviac jeden PetriNetDiagramController, ktorý pracuje s práve aktívnym diagramom ABAgrafu. Úlohou TabsController-a je pri zmene stavu záložiek o tom poskytnúť informácie PetriNetDiagramController-u, ktorému je predaný model agenta, ktorého ABAgraf je práve zobrazovaný.

ComponentSelectionController riadi panel s výberom komponentov v ľavej časti aplikácie. Aplikácia má tieto controllery dva: jeden pre výber uzlov a jeden pre výber hrán. Toto rozdelenie je len pre uľahčenie orientácie používateľa. Skladá sa z tlačidiel, ktoré môžu byť v jednom z dvoch stavov: aktívne a neaktívne. V danom momente môže byť aktívne len jedno tlačidlo v obidvoch diagramoch. Pri zmene stavu ktoréhokoľvek tlačidla sú o tom prostredníctvom SelectionChangeObserver-a notifikované ostatné controllery, ktoré na to adekvátne zareagujú.

BaseDiagramController riadi diagram v strednej časti aplikácie. Má niekoľko potomkov pre jednotlivé typy diagramov. Sú nimi AgentDiagramController pre riadenie diagramu hierarchie agentov, AgentDetailDiagramController pre riadenie diagramu komponentov agenta a PetriNetDiagramController pre riadenie ABAgrafu. Obsahuje všeobecné operácie pre prácu s diagramom ako napríklad: vytvorenie nového uzla, presun uzla, vymazanie uzla, vytvorenie novej hrany, prepojenie hrany s uzlom, zrušenie hrany... Potomkovia triedy tieto operácie upravujú tak, aby vyhovovali potrebám daného diagramu.

SelectionChangeObserver je rozhranie, ktoré implementujú a využívajú všetky vyššie uvedené controllery. Je využité pri implementácii návrhového vzoru observer, prostredníctvom ktorého sa controllery navzájom notifikujú o zmene stavu selekcie komponentov. Napríklad v prípade, že je ComponentSelectionController-y správ vybraná

správa "notice" a používateľ vyberie komponent "Dynamic Agent" v ComponentSelectionController-y komponentov, tak o tejto udalosti selection kontroler komponentov notifikuje všetky kontrolery, ktoré na to zareagujú zrušením svojho výberu.

CoreController sprostredkuje komunikáciu medzi ostatnými controllermi a poskytuje im služby (napríklad clipboard, práca s modelom).

8. Editor ABAgrafov

ABAgraf je modifikovanou Petriho sieťou. Štandardná Petriho sieť sa skladá z množín miest, prechodov, hrán, značiek a pravidiel. ABAgraf rozširuje Petriho sieť pridaním viacerých typov miest a prechodov, pridaním parametrov hrán a modifikáciou niektorých pravidiel.

Značky reprezentujú správy doručené agentovi. ABAgraf popisuje spôsob ich spracovania.

Nástroj ABAbuilder poskytuje možnosť vybrať si, či konkrétny agent používa ABAgraf alebo nie. Ak nie, je potrebné napísať implementáciu manažéra vo forme kódu, v opačnom prípade je správanie manažéra definované ABAgrafom. Výhodou využitia ABAgrafu je hlavne to, že je prehľadnejší ako zdrojový kód a v prípade použitia CASE nástroja je jeho tvorba podstatne jednoduchšia.

ABAgraf je zložený z niekoľkých typov uzlov a hrán. Základné rozdelenie uzlov je na miesta a prechody. To je zhodné so štandardnou Petriho sieťou. Na rozdiel od štandardnej Petriho siete sú miesta a prechody ABAgrafu ďalej rozdelené do podtypov, pričom sú dva typy prechodov: jednoduchý a rozhodovací. Miesta môžu byť rozdelené na externé a interné, pričom sa dajú ešte ďalej rozdeliť podľa komponentu, ktorý reprezentujú.

Miesto reprezentuje komponent simulačného modelu, ktorý vie spracovať, alebo ktorému môže byť doručená správa. Pri spracovaní správy interpreterom ABAgrafu sa správa vždy nachádza v jednom z miest.

Úlohou prechodu je presun správ medzi miestami. Prechod má ľubovolný počet vstupných a výstupných hrán. Po splnení podmienok je prechod aktivovaný a správy sú presunuté zo vstupných miest na výstupné.

Jednoduchý prechod je prechod, ktorý je aktivovaný, keď je na jeho vstupných miestach požadovaný počet hrán. Je preto potrebný mechanizmus na určenie smerovania správ. Toto je realizované tým, že na výstupnej hrane je možné definovať podmienku, ktorá definuje spôsob poslania správy. Jedna správa môže byť poslaná aj po viacerých hranách, preto sú možné dva spôsoby jej zaslania: preposlanie správy a poslanie jej kópie, čo je tiež definované v podmienke hrany. Podmienka sa skladá z dvoch častí: spôsobu poslania správy a identifikátora zdroja, z ktorého správa prichádza. Napríklad podmienka pre poslanie správy, ktorá prišla z prechodu s identifikátorom 9 je v prípade, že sa odosiela originál "=9" a v prípade, že sa odosiela kópia "+9". Pokiaľ nezáleží na tom, ktorá značka je preposlaná, je tiež možné namiesto identifikátora zdrojového prechodu do podmienky napísať hviezdičku, teda "=*" alebo "+*".

V diagrame je reprezentovaný čiernym obdĺžnikom. Pod ním je identifikátor prechodu (napríklad: t1, t2, ...) a nad ním je počet správ požadovaných pre aktiváciu prechodu (v prípade, že je potrebných viac než jedna správa, inak nie je počet zobrazený).

Rozhodovací prechod je prechod, ktorým je implementované vetvenie. Výstupné miesto, na ktoré je správa poslaná je vybraté podľa podmienky, ktorá je vyhodnotená podľa hodnoty atribútov správy. Svrávy ABAsim majú niekoľko preddefinovaných atribútov, ktoré používa rozhodovací prechod. Jedným z jeho parametrov je parameter správy, podľa ktorého má rozhodovať. Na jeho výstupných hranách sú definnované podmienky, ktoré určujú smerovanie správy.

V diagrame je reprezentovaný čiernym obdĺžnikom s menším bielym obdĺžnikom na ľavo od čierneho. Nad prechodom je rozhodovací parameter a pod ním jeho identifikátor.

Interné miesto reprezentuje instantných asistentov agenta. Instantný asistenti majú metódu execute, ktorej parametrom je správa. Pri vykonaní execute môžu nastavovať

parametre správy, ktoré sú používané rozhodovacími prechodmi. Musí mať práve jednu vstupnú a práve jednu výstupnú hranu.

Externé miesto môže reprezentovať agenta, kontinuálneho asistenta alebo absorbciu. Nemôže mať súčasne vstupné a výstupné hrany, ale ich počet je ľubovolný. Môžu byť rozdelené na vstupné a výstupné externé miesta, podľa toho, aké hrany majú. Reprezentujú miesta, na ktorých vznikajú a zanikajú značky siete. Vznik značky znamená doručenie správy od komponentu, na ktorom táto značka vznikla. Zánik značky znamená odoslanie správy komponentu, v ktorom značka zanikla.

Absorbcia je špeciálnym externým výstupným miestom ABAgrafu, ktoré nereprezentuje komponent simulačného modelu. Správy doručené absorbcii nie sú poslané ďalej. Jej úlohou je dealokácia správ.

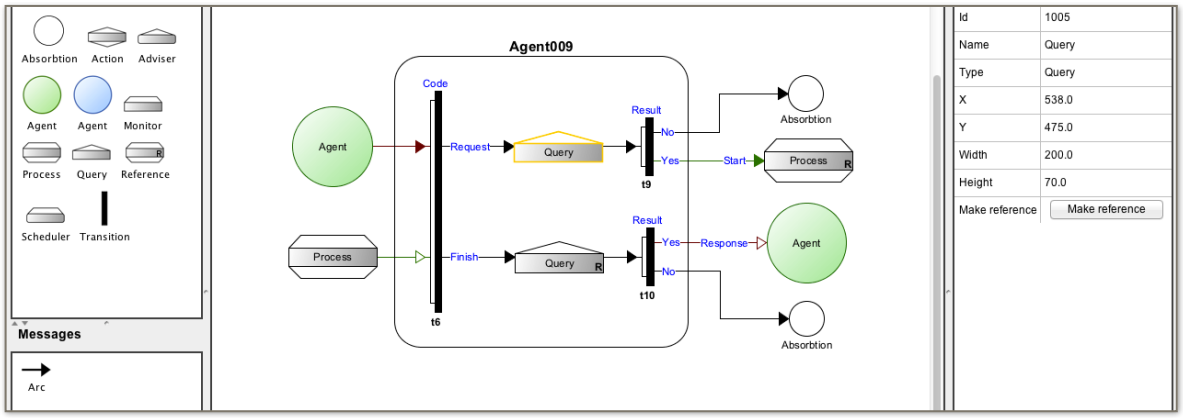
Hrany spájajú miesta a prechody. Na rozdiel od štandardnej Petriho siete, kde sú miesta prepojené výlučne s prechodmi a prechody výlučne s miestami, ABAgraf povoľuje prepojenie dvoch prechodov hrán, keďže miesta reprezentujú asistentov agenta, ktorý vykonávajú zadanú úlohu a medzi dvoma prechodmi nemusí byť potrebné vykonať nejakú operáciu. Interpreter ABAgrafu v tomto prípade vytvorí fiktívne miesto medzi prechodmi, ktoré ale nie je zobrazené v diagrame.

Hrany ABAgrafu môžu mať niekoľko parametrov, ktoré závisia od komponentov, ktoré spájajú. Napríklad pri vetvení rozhodovacieho prechodu má atribút s hodnotou, ktorá je porovnaná so zodpovedajúcim atribútom správy a v prípade zhody je správa poslaná po tejto hrane. V prípade, že je hrana prepojená s výstupným miestom obsahuje atribút, podľa ktorého je nastavený kód správy pri jej odoslaní.

Referencie komponentov

Diagram ABAgrafu nakreslený v nástroji ABAbuilder nie je len obrázkom, ktorý pekne vyzerá. Je zdrojom dát pre generátor kódu simulačného modelu. To znamená, že pre každé miesto vytvorené v diagrame je vygenerovaná trieda komponentu simulačného modelu. Musí ale byť možné použiť jeden konkrétny komponent na viacerých miestach ABAgrafu

(napríklad dotaz na dĺžku frontu - očividne nie je prípustné požadovať, aby sa tento komponent mohol v diagrame nachádzať len jeden krát, na druhej strane nemôže byť pre každý výskyt tohto komponentu vytvorená samostatná trieda). Ako riešenie tohto problému som navrhol použitie referencií na komponenty. Každé miesto môže byť z tohto hľadiska dvoch typov: komponent alebo referencia na komponent. Pri editácii diagramu sa s obidvoma typmi komponentov pracuje rovnako, rozdiel je len pri ich vytváraní a pri generovaní kódu. Dôvodom existencie referencií je to, aby mohol byť jeden komponent na viacerých miestach súčasne, pričom každá inštancia daného komponentu môže byť prepojená rôznymi hranami, ktoré sú na sebe nezávislé.



Obr. 8.1: Referencie v editore ABAgrafu

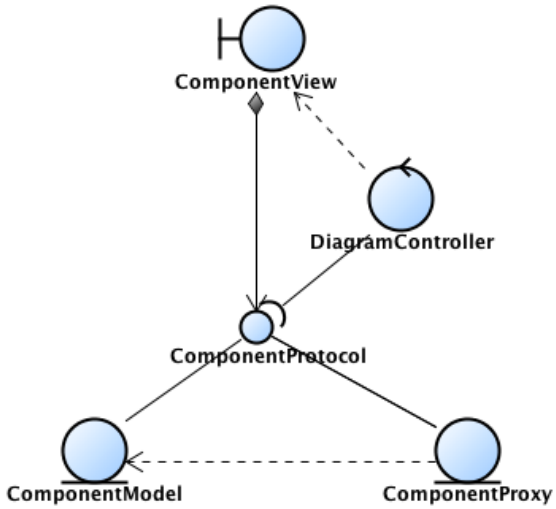
Všetci agenti v ABAgrafe sú takisto referenciami na komponenty v diagrame hierarchie agentov.

Graficky sú v diagrame referencie asistentov agenta odlíšené znakom "R" v pravom dolnom rohu. Pre agentov nemá zmysel rozlišovať, či sa jedná o referenciu, keďže ABAgraf neobsahuje originály agentov, preto sú zobrazení rovnako ako v diagrame hierarchie agentov. Pre uľachčenie orientácie v diagrame je po dvojkliku myšou na referenciu asistenta zvýraznený jeho originál (v prípade, že originál nie je aktuálne zobrazený, urobí sa scroll plátna).

Referencie komponentov som implementoval použitím návrhového vzoru dekorátor. Dôvodom je to, že je pomerne veľký rozdiel v tom, či je komponent referenciou alebo

originálom, ale chcel som, aby s nimi bolo možné pracovať rovnakým spôsobom. Model referencie komponentu je proxy objektom. To znamená, že veľkú časť operácií, ktoré má vykonať referencia jednoducho deleguje originálu. Referencia a originál implementujú to isté rozhranie a zvyšok aplikácie pracuje výhradne s týmto rozhraním (až na prípad vytvárania inštancií). Celá implementácia referencií je zapúzdrená v proxy triedach.

Referencie sa od originálu líšia spracovaním operácií, ktoré menia grafické zobrazenie referencie (x, y súradnice v diagrame) a hranami, ktorými sú prepojené s ostatnými komponentami.



Obr. 8.2: Implementácia referencií

Na obrázku je znázornená implementácia referencií komponentov. Je niekoľko druhov komponentov, preto každý uzol v obrázku, okrem DiagramController, reprezentuje hierarchiu tried.

ComponentModel je modelom originálu komponentu. Obsahuje základné údaje o komponente. Neobsahuje všetky hrany, ktoré patria k tomuto komponentu. Niektoré z nich sú uložené v zástupcoch komponentu podľa toho, ku ktorému uzlu v diagrame patria.

ComponentProxy je modelom referencie komponentu. Jedným z jeho atribútov je jeho originál. Má tiež zoznam hrán, ktorý je nezávislý na hranách originálu.

ComponentProtocol je rozhranie, ktoré implementujú ComponentModel a ComponentProxy. Zvyšok aplikácie pracuje s týmto rozhraním.

Implementácia

Pre implementáciu editora ABAgrafov je využitá architektúra MVC. ABAgraf popisuje správanie agenta, preto je pri vytvorení diagramu controlleru predaná referencia na agenta z diagramu hierarchie agentov, ktorého popisuje. Controller je potomkom Controlleru, ktorý je použitý aj pri implementácii ostatných diagramov rozšírený o funkcie špecifické pre ABAgraf.

Miesta

Implementácia jedného miesta diagramu sa skladá z dvoch častí: modelu a zobrazovača. Modely a zobrazovače miest ABAgrafu sú s výnimkou miesta absorbcia, totožné s ostatnými diagramami. Model absorbcie je null object.

Prechody

Prechody sú takisto tvorené modelom a zobrazovačom. Prechod môže mať niekoľko vstupných a niekoľko výstupných hrán. Tieto hrany sú na zobrazovači rovnomerne rozložené pozdĺž prechodu a sú automaticky usporiadené podľa súradníc komponentu na opačnej strane hrany tak, aby sa predišlo prekríženiu hrán. Výšku prechodu je možné upraviť v paneli s atribútmi, ale pre urýchlenie práce je tiež možné zväčšiť výšku prechodu na dvojnásobok dvojklikom myši a zmenšiť výšku prechodu na polovicu dvojklikom myši a podržaním tlačidla alt.

Prechod typu code je na rozdiel od ostatných prechodov zaujímavý tým, že jednotlivé vstupné hrany tvoria páry s výstupnými hranami.

Pre uľahčenie práce používateľa je pre prechod typu code implementovaných niekoľko funkcií. Pri pridávaní novej hrany nástroj vo väčšine prípadov nedovolí spojiť dve miesta hrán. Výnimkou je, ak je zdrojom hrany externé miesto. Namiesto toho, aby nástroj vypísal

správu o tom, že takéto prepojenie nie je povolené, prepojí externé miesto s prechodom typu code jednou hranou a súčasne vytvorí druhú hranu, ktorou prepojí prechod s cieľovým uzlom alebo prechodom.

Pri vyplňovaní niektorých atribútov jednej z hrán prechodu typu code je súčasne nastavený iný atribút spárovanej hrany na opačnej strane prechodu, ktoré vyplýva z nastavenia prvej hrany.

Pri odstraňovaní jednej z hrán prechodu typu code je odstránená aj s ňou spárovaná hrana na opačnej strane prechodu.

Pri automatickom usporiadaní hrán sa predchádza prekrižovaniu hrán len na strane zdroja. Hrany na opačnej strane prechodu zostávajú spárované a je teda na používateľovi, aby vhodným usporiadaním cieľových komponentov zabezpečil dobrú čitateľnosť diagramu.

Hrany

Hrany ABAgrafu majú niekoľko atribútov, ktoré popisujú ich stav, ale počet a typy týchto atribútov závisia od typov komponentov, ktoré spájajú a toho, či je daný komponent zdrojom alebo cieľom. Pri spájaní komponentov hranou nástroj overuje, či je toto prepojenie povolené. Pre uľahčenie práce je možné priamo prepojiť vstupné externé miesto s interným miestom. V takom prípade sú automaticky vytvorené dve hrany: prvá hrana medzi vstupným miestom a prechodom typu code a druhá hrana medzi prechodom typu code a interným miestom.

Ohraničenie

Vnútorná časť ABAgrafu (interné miesta a prechody) je označená ohraničením, ktoré automaticky mení veľkosť pri pridávaní a posúvaní komponentov. Okrem toho, že mení vlastnú veľkosť, tiež posúva externé miesta tak, aby ich vzdialenosť od okraja ohraničenia zostala nezmenená, čím zabezpečí, že tvorca simulačného modelu nebude musieť pri rozširovaní diagramu posúvať potenciálne veľké množstvo komponentov.

9. Interpreter ABAgrafov

Pri uložení modelu je každý ABAgraf exportovaný do XML súboru. Sú v ňom uložené všetky informácie o ABAgrafe vrátane grafických údajov. Súbor je použitý pri opätovnom načítaní ABAgrafu. Okrem toho je vytvorený XML súbor použitý aj pre interpreter ABAgrafov.

Pre interpreter ABAgrafov som navrhol nový formát, pretože používaný formát pre jazyk Delphi bol pomerne starý a bolo by dobré ho aktualizovať. Okrem toho by preň bolo nutné napísať parser. Keďže ale používam XML, je možné použiť už existujúci. Nový formát tiež popisuje sieť všeobecnejšie a je ľahšie rozšíriteľný.

Navrhnutý formát má ako koreňový element značku ABAgraph a obsahuje tri sekcie: Places, Transitions a Arcs. Každá z nich obsahuje zoznam prvkov daného typu.

Sekcia Places obsahuje miesta ABAgrafu všetkých typov (vstupné, interné a výstupné). Jednotlivé miesta sú reprezentované značkou "Place". Miesta ABAgrafu reprezentujú komponenty simulačného modelu (interné miesta - instantní asistenti, externé miesta - agenti a kontinuálni asistenti). Okrem toho sú vytvorené fiktívne miesta v prípade priameho spojenia dvoch prechodov. Atribúty miesta sú zapísané do xml súboru, pričom dané miesto môže obsahovať len niektoré atribúty - tie ktoré majú zmysel v danom kontexte. Nasleduje popis jednotlivých atribútov miest:

- **PlaceId** je unikátne ID uzla Petriho siete. Nie je vyplnený pre fiktívne miesta, ktoré sú v programe identifikované práve tým, že tento identifikátor má hodnotu "nedefinovaný".
- **ComponentId** je ID komponentu simulačného modelu, podľa ktorého je za behu simulácie interpreterom Petriho sietí daný komponent vyhľadaný. Podobne ako pri atribúte PlaceId, nie je v prípade fiktívneho miesta vyplnený.
- **Bounds** je čiarkami oddelená grafická reprezentácia komponentu. Prvé dve hodnoty sú x a y súradnice komponentu nasledované jeho šírkou a výškou.

- **Source** obsahuje identifikátor prechodu, ktorý predchádza tomuto miestu. Vstupné externé miesta ABAgrafu tento atribút nemajú vyplnený.
- **Destination** obsahuje identifikátor prechodu, ktorý nasleduje za týmto miestom. Výstupné externé miesta nemajú tento atribút vyplnený.
- **TransitionCase** je podmienkou, podľa ktorej sa rozhodne, či a akým spôsobom je správa poslaná na toto miesto. Formát tejto podmienky závisí od typu prechodu.
    - V prípade, že sa jedná o jednoduchý prechod je podmienka zložená zo spôsobu poslania správy ("=" pre poslanie originálu správy a "+" pre poslanie kópie) a identifikátora zdrojového prechodu (alebo "*" v prípade, že správa môže byť zobraná z ľubovolného).
    - V prípade prechodu typu code je podmienka bodkočiarkou oddelená dvojica kód správy a identifikátor odosielateľa ("messageCode;senderId"). Nie je teda uložený iba kód správy, ale aby bola zjednodušná identifikácia spôsobu spracovania správy tým, že sú do podmienky doplnené dodatočné informácie o odosielateľovi tejto správy.
    - Pri ostatných typoch prechodov podmienka obsahuje hodnotu príslušného atribútu, podľa ktorého sa pri prechode rozhoduje.
- **PostType** obsahuje spôsob poslania správy (Notice, Request, Response, ...).
- **PostCode** obsahuje kód posielanej správy. Jedná sa o kód, ktorý správe nastaví interpreter ABAgrafu, keď správa dorazí do externého miesta. V prípade, že sa jedná o jednu so systémových správ (to je dané atribútom PostType), nie je tento atribút vyplnený.

Sekcia Transitions obsahuje všetky prechody ABAgrafu. Jednotlivé prechody sú definované značkou "Transition" a môžu obsahovať tieto atribúty:

- **SimId** je identifikátor prechodu, s ktorým pracuje simulačné jadro. Používteľ nástroja ho môže meniť.
- **InternalId** je identifikátor prechodu, s ktorým pracuje nástroj ABAbuilder. Používateľovi nie je umožnené ho zmeniť.

- **Condition** obsahuje hodnotu definujúci typ prechodu. V prípade, že má tento atribút hodnotu "None" jedná sa o jednoduchý prechod. V opačnom prípade je prechod rozhodovacím prechodom typu "Code", "Result" alebo "Sender".
- **RequiredTokens** je vyplnený iba pre jednoduché prechody. Obsahuje počet značiek potrebných pre aktiváciu prechodu.
- **Bounds** obsahuje grafickú reprezentáciu komponentu, podobne ako v sekcii miest.

Sekcia Arcs obsahuje všetky hrany ABAgrafu. Jednotlivé hrany sú definované značkou "Arc" a môžu obsahovať tieto atribúty:

- **SimId** je identifikátorom správy, ktorú táto hrana reprezentuje, ktorá je využívaná za behu simulácie.
- **InternalId** je identifikátor,, ktorý používa nástroj pre rozlišovanie hrán. Používateľ ho, na rozdiel od atribútu SimId, nemôže meniť.
- **Source** je identifikátorom zdrojového prechodu alebo miesta.
- **Destination** je identifikátorom cieľového prechodu alebo miesta.
- **PostType** obsahuje spôsob poslania správy (Notice, Request, Response, ...).
- **Condition** obsahuje podmienku pre prechod. Na rozdiel od podmienky pre prechody v sekcii miesta, obsahuje podmienka pri hrane v prípade prechodu typu code textovú reprezentáciu kódu správy.
- **Code** obsahuje textovú reprezentáciu kódu odosielanej správy. Atribút je vyplnený pre hrany, ktorých cieľovým miestom je externé výstupné miesto. Táto textová reprezentácia kódu správy je využitá generátorom kódu pri generovaní identifikátorov kódov správ, ktoré využíva tvorca modelu (t.j. nepracuje priamo s číselnou reprezentáciou kódu správy).
- **SourceCode** je číselnou reprezentáciou kódu správy na vstupnom uzle hrany.
- **DestinationCode** je číselnou reprezentáciou kódu správy na výstupnom uzle hrany.
- **SourceIndex** je grafickou informáciou pre nástroj o poradí hrany na prechode, ktorý je zdrojovým uzlom hrany.

- **DestinationIndex** je grafickou informáciou pre nástroj o poradí hrany na prechode, ktorý je cieľovým uzlom hrany.
- **LineOffset** je grafickou informáciou pre nástroj o umiestnení hrany.

Príklad formáu ABAgrafu zapísaného ako XML:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<ABAgraph>
    <Places>
        <Place>
            <PlaceId>placeId</PlaceId>
            <ComponentId>componentId</ComponentId>
            <Bounds>x,y,w,h</Bounds>
            <Source>transitionId</Source>
            <Destination>transitionId</Destination>
            <TransitionCase>value</TransitionCase >
            <PostType>notice/finish/request</PostType>
            <PostCode>postCode</PostCode>
        </Place>
        <Place> ... </Place>
    </Places>
    <Transitions>
        <Transition>
            <SimId>id</SimId>
            <InternalId>iid</InternalId>
            <Condition>Code/Result/Sender/None</Condition>
```

<RequiredTokens>*2*</RequiredTokens>

<Bounds>*x,y,w,h*</Bounds>

</Transition>

<Transition> ... </Transition>

</Transitions>

<Arcs>

<Arc>

<SimId>*simId*</SimId>

<InternalId>*iid*</InternalId>

<Source>*transitionId/placeId*</Source>

<Destination>*transitionId/placeId*</Destination>

<PostType>*Notice/Finish/Request*</PostType>

<Condition>*0/1/2*</Condition>

<Code>*outputArcCode*</Code>

<SourceCode>*1024*</SourceCode>

<DestinationCode>*1024*</DestinationCode>

<SourceIndex>*0/1/2*</SourceIndex>

<DestinationIndex>*0/1/2*</DestinationIndex>

<LineOffset>*0.5*</LineOffset>

</Arc>

<Arc> ... </Arc>

</Arcs>

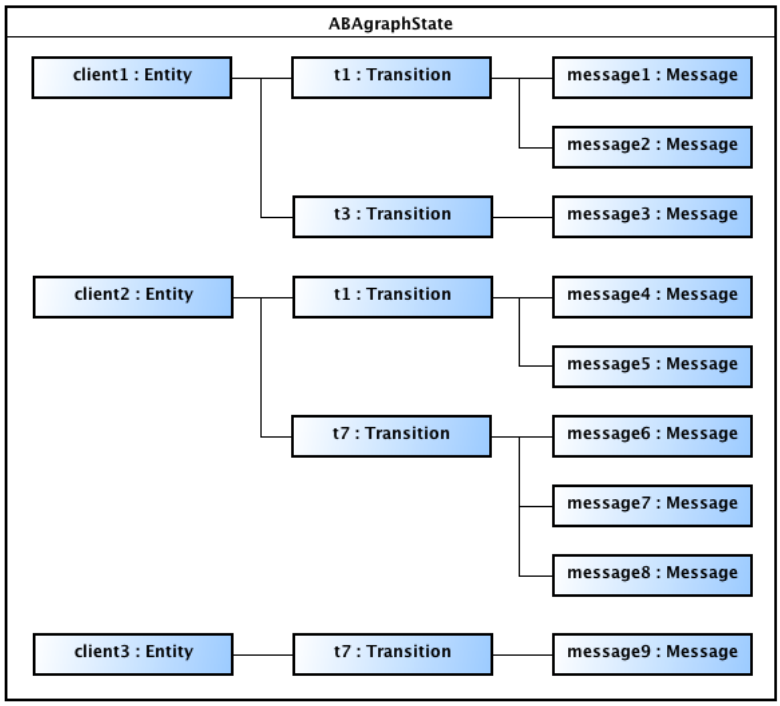
</ABAgraph>

Interpreter ABAgrafu je vytvorený pri spustení simulácie pri vytvorení agenta pre všetkých agentov, pre ktorých tvorca modelu rozhodol, že ich správanie bude definované ABAgrafom. Pri prijatí správy manažér agenta túto správu predá interpreteru ABAgrafu, ktorý ju spracuje.

Spracovanie správy závisí od toho, ku ktorej entite táto správa patrí. Napríklad pri simulácii obslužného systému môže entita čakať na priradenie niekoľkých zdrojov obsluhy. O priradení zdroja sa agent dozvie doručením správy od správcu tohto zdroja. V prípade, že má čakajúcich entít niekoľko, nestačí mu informácia o tom, že prijal správu o priradení zdroja od požadovaných správcov zdrojov, pretože potrebuje vedieť, ktorej entite bol tento zdroj priradený.

Štandardná správa simulačnej knižnice obsahuje atribút servedObject, ktorým tvorca simulačného modelu môže nastaviť entitu, s ktorou táto správa súvisí.

Interpreter ABAgrafov potom podľa tohto atribútu zaregistruje entitu ako svojho klienta. Aktuálny stav ABAgrafu je daný správami, ktoré aktuálne spracováva. Správy sú rozdelené podľa toho, v akej časti ABAgrafu sa práve nachádzajú a klienta, ku ktorému sú priradené.



Obr 9.1: Príklad štruktúry stavu ABAgrafu

Interpreter ABAgrafov je zložený niekoľkých tried:

- **PetriNet** je trieda, ktorá riadi vykonávanie spracovania správy. Udržuje stav siete vrátane informácii o klientoch a nespracovaných prechodoch.
- **PetriNetLoader** je trieda, ktorej úlohou je načítanie siete zo súboru.
- **Token** je trieda, ktorá reprezentuje značku, ktorá sa posúva po sieti. Obsahuje spracovávanú správu a dodatočné informácie, ktoré sú podstatné pre spracovanie správy ABAgrafom.
- **PetriNode** je abstraktná trieda, ktorá reprezentuje uzol siete. Je nadtriedou tried Place a Transition. Definuje rozhranie pre prijatie značky.
- **Place** je trieda, ktorá reprezentuje miesto siete. Pri prijatí značky závisí spôsob jej spracovania od typu miesta:
  - Externé vstupné miesta pri prijatí značky túto značku jednoducho prepošlú na ich výstupný prechod. Reprezentujú agentov, alebo kontinuálnych asistentov simulačného modelu.
  - Interné miesta majú práve jeden vstupný prechod a práve jeden výstupný prechod. Môžu byť dvoch typov. Prvým typom sú miesta, ktoré reprezentujú okamžitých asistentov, ktoré pri prijatí značky nechajú asistentov spracovať správu (využitím správy execute). Následne značku umiestnia na výstupný prechod. Druhým typom interných miest sú fiktívne miesta. Fiktívne miesta nemajú priradeného asistenta. Ich jedinou úlohou je presun značky medzi dvoma prechodmi.
  - Externé výstupné miesta zakončujú spracovanie správy. Pri prijatí značky z nej vyberú správu, ktorej nastavia príslušné parametre (kód, adresát) a odošlú ju definovaným spôsobom odoslania (notice, request, ...). Tieto miesta reprezentujú agentov alebo kontinuálnych asistentov. Okrem toho existuje špeciálny typ externého výstupného miesta: absorbcia. Absorbcia zruší správu bez jej odoslania.
- **Transition** je trieda, ktorá reprezentuje prechod siete. Pri prijatí značky ju okamžite nespracuje. Namiesto toho sa zaregistruje ako nespracovaný prechod. Spracovanie prechodu nastáva, až keď k tomu dá pokyn riadiaci algoritmus spracovania ABAgrafu. Najskôr sa zistí, či môže byť daný prechod aktivovaný. To závisí na počte požadovaných

značiek na vstupe prechodu, ktorý môže byť väčší ako jedna iba v prípade jednoduchých prechodov. Spôsob uskutočnenia samotnej aktivácie prechodu závisí od toho, či sa jedná o jednoduchý alebo o rozhodovací prechod. V prípade rozhodovacieho prechodu je značka umiestnená na miesto vybrané zo zoznamu výstupných miest prechodu podľa parametrov správy a nastavení výstupných miest. V prípade jednoduchého prechodu môže byť na vstupných miestach prechodu značiek niekoľko. Na výstupné miesta sú umiestnené originály alebo kópie značiek podľa nastavení výstupných miest. Aktivácia prechodu môže spôsobiť zaradenie iných prechodov medzi nespracované.

Pri požiadavke o spracovanie správy ju najskôr interpreter obalí do značky (token). Značku následne vloží na vstupné miesto, ktoré vyberie podľa odosielateľa správy. To spôsobí umiestnenie značky na príslušný prechod. Pri prijatí značky prechodom je prechod zaradený medzi nespracované prechody. Následne začína spracovanie nespracovaných prechodov, ktoré prebieha, až kým nie sú všetky prechody spracované.

10. Grafická knižnica

Vytvoril som niekoľko prototypov nástroja, ktoré obsahovali obmedzenú verziu diagramu hierarchie agentov, pričom som použil viacero technológií, aby som porovnal náročnosť ich implementácie a ich vhodnosť pre implementáciu nástroja pre editáciu diagramov. Pár prototypov bolo postavených na veľkých frameworkoch pre tvorbu modelov a editáciu diagramov, akými sú napríklad GEF (Graphical modeling framevork), EMF (Eclipse modeling framework), ktoré už síce mali veľa vecí hotových, ale za cenu menšej flexibility a zložitej konfigurácie. Ďalší prototyp, na druhej strane, používal štandardnú grafickú knižnicu Javy, ktorá ale poskytuje len možnosť kresliť základné útvary (čiary, obdĺžniky, elipsy, ...). Pokročilejšia funkcionalita, akou je napríklad hierarchická štruktúra grafických prvkov, animácie, detekcia kolízií a podobne, ktoré by boli užitočné pre implementáciu nástroja, by bolo potrebné doprogramovať.

Ďalej som vyskúšal niekoľko grafických knižníc, ktoré poskytujú aj ďalšie možnosti, ktoré chýbajú v štandardnej knižnici Javy, ale na rozdiel od GEF a podobných frameworkov sú flexibilnejšie a nevyžadujú štúdium tak veľkého množstva informácií o technológii na to,

aby sa dali používať. Z nich som sa nakoniec rozhodol použiť knižnicu Piccolo2D - aj preto, že som našiel niekoľko aplikácií, editorov diagramov a nástrojov na prácu s grafmi, ktoré túto knižnicu používajú.

Piccolo2D je vektorová grafická knižnica navrhnutá pre tvorbu ZUI (Zoomable graphical interface). Poskytuje funkcie: Kreslenie jednoduchých grafických prvkov (elipsy, obdĺžniky, polgóny, ...), animácie, vnáranie grafických prvkov, zoom a scroll plátna.

11. Generovanie kódu

Transformácia simulačného modelu

Konečným výstupom nástroja ABAbuilder je zdrojový kód simulačného modelu. Tento kód nie je generovaný z diagramov simulačného modelu priamo, ale ako výsledok procesu zloženého z viacerých krokov, pri ktorých sa vytvára nový model z modelu predchádzajúceho kroku.

Diagramy simulačného modelu sú grafickou reprezentáciou simulačného modelu, s ktorou používateľ pracuje priamo. Je uložený ako dátová štruktúra zložená zo zoznamov a hašovacích tabuliek jednotlivých prvkov (agentov, asistentov, správ, ...) a vzťahov medzi nimi. Nie je triviálne ani praktické z tohto modelu priamo generovať zdrojový kód.

Podľa diagramov simulačného modelu je najskôr vytvorený abstraktný model zdrojového kódu, abstraktný syntaktický strom, ktorý je následne použitý pre generovanie kódu konkrétneho programovacieho jazyka.

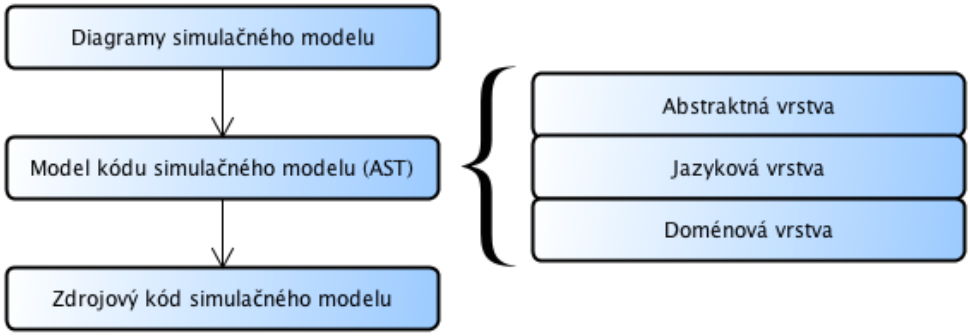
Výstupom nástroja sú triedy obsahujúce kód, ktorý je implementáciou simulačného modelu.

Dôvodom, prečo z diagramov simulačného modelu nie je zdrojový kód generovaný priamo, je potreba väčšej flexibility. Kód má byť generovaný do viacerých programovacích jazykov a v budúcnosti môžu pribudnúť ďalšie. Rozšírenie o dodatočný programovací jazyk by malo byť čo najjednoduchšie. Preto je ako medzikrok vytvorený abstraktný model zdrojového kódu, modelujúci spoločnú časť pre všetky programovacie

jazyky, implementovaná len na jednom mieste. Špecifickú časť konkrétneho programovacieho jazyka je možné jednoducho vymeniť.

Generovanie kódu simulačného modelu je teda zložené z niekoľkých krokov, ktorých stav je daný modelom, ktorý bol vytvorený z modelu predchádzajúceho kroku. V tejto časti sa zaoberám modelom zdrojového kódu simulačného modelu - syntaktickým stromom a spôsobom, ako generuje zdrojový kód.

Pre tento model som navrhol tri vrstvy: abstraktnú, jazykovú a doménovú.



Obr 11.1: Proces generovania kódu simulačného modelu

Abstraktná vrstva

Táto vrstva modeluje abstraktný model objektovo orientovaného programovacieho jazyka.

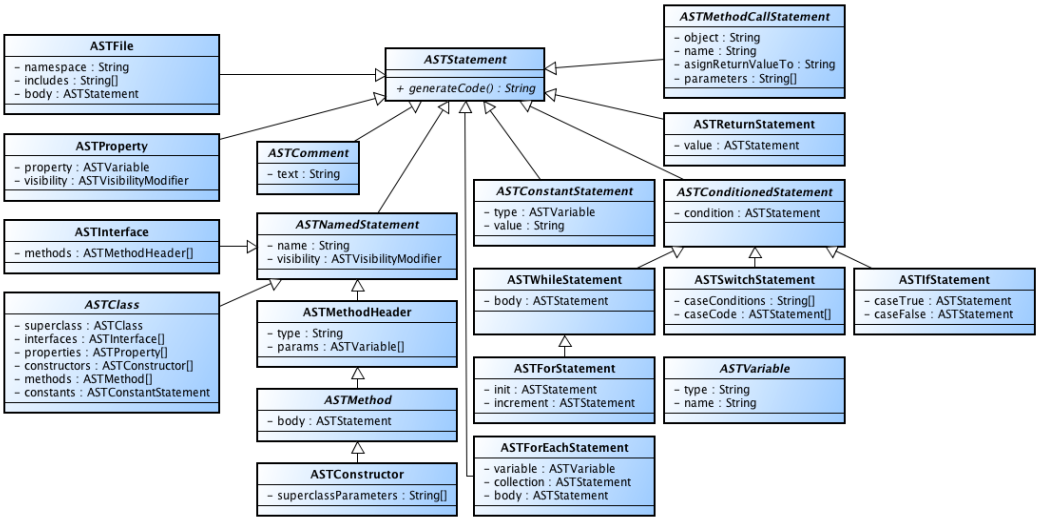
Modeluje všeobecnú štruktúru a vzťahy programovacieho jazyka. Tvorí abstraktný syntaktický strom (AST - abstract syntax tree). Uzly sú prvkami programovacieho jazyka. Uzly tejto vrstvy obsahujú dáta (napríklad názov triedy) a referencie na súvisiace prvky (napríklad zoznam metód triedy), ale nedefinujú konkrétnu syntax jazyka. Vrstva je nezávislá aj na programovacom jazyku, aj na doméne (doménou môže byť simulačný model ABAsim alebo akýkoľvek konkrétny kód, ktorého kód by mohol byť generovaný).

Táto vrstva je kolekciou abstraktných tried, ktoré definujú objektové prvky jazyka (triedy, metódy, rozhrania, atribúty tried, ...) a prvky štrukturovaného programovania, ktoré sú

používané pri implementácii metód a konštruktorov tried (výrazy if, switch, for, while, operátory, volania metód, vytváranie inštancií tried, ...).

Všetky triedy tejto vrstvy modelujúce výrazy programovacieho jazyka sú potomkami triedy ASTStatement. Sú implementované ako návrhový vzor composite. To znamená, že sa daný výraz môže skladať z ďalších výrazov. To z akých výrazov môže byť zložený a aká je ich štruktúra je obmedzené tým, o aký druh výrazu sa jedná. Napríklad výraz while má dva podvýrazy: podmienku a telo. Samotná podmienka je ľubovolným výrazom a telo je postupnosťou výrazov.

Keďže sú tieto triedy abstraktné, nie je ich možné použiť priamo. Okrem tried modelujúcich výrazy programovacieho jazyka je v abstraktnej vrstve a abstraktná factory, ktorá poskytuje rozhranie pre vytvorenie jednotlivých tried abstraktnej vrstvy. Implementácia všetkých jej metód je implementáciou jazykovej vrstvy generátora kódu.



Obr 11.2: Zjednodušený diagram tried abstraktnej vrstvy generátora kódu

Pri generovaní zdrojového kódu je potrebné rozlišovať, či sa jedná o výraz alebo o príkaz.

ASTStatement má niekoľko metód pre formátovanie a rozlíšenie príkazov. Kód porgramovacieho jazyka je postupnosťou príkazov. Spôsob ukončenia príkazov je v rôznych programovacích jazykoch rôzny. Najčastejšie je ním bodkočiarka (Java, C#) alebo

nový riadok (Python, Swift) alebo ukončenie bloku kódu (príkazy if, for, ...) v prípade bloku kódu uzatváracou zloženou zátvorkovu "}" (Java, C#) alebo kľúčovým slovom "end" (Delphi) alebo odsadením (Python).

Príklad príkazu:

```
id = message.sender().id();
```

Príklad výrazu:

```
id = message.sender().id()
```

Jediný rozdiel je v tomto prípade bodkočiarka na konci. Pri generovaní kódu je potrebné rozlišovať, či sa jedná o výraz alebo príkaz, pretože zatiaľ čo každý príkaz je výrazom, výraz nemusí byť príkazom. Okrem toho môže byť výraz zložený z viacerých podvýrazov. V predchádzajúcom príklade je aj message.sender() výrazom, ale pri generovaní kódu nemôže byť ukončený bodkočiarkou, pretože nasleduje volanie metódy id(), pri ktorom je na druhej strane bodkočiarka vyžadovaná (poznámka: v niektorých jazykoch nie sú príkazy ukončené bodkočiarkou. Tu sa jedná len o ilustračný príklad).

Okrem toho sú aj iné prípady, kedy nie je výraz ukončený bodkočiarkou, napríklad v podmienke if, while alebo switch.

```
if (message.sender().id() == Id.myAgent)
```

Abstraktná vrstva generátora kódu má pre riešenie týchto typov problémov metódy definované v triede ASTstatement, ktoré plnia dve úlohy: Nastavujú, akým spôsobom sa bude generovať kód a generujú kód podľa týchto nastavení. Metódy teda tvoria dvojice preto, aby bolo možné urobiť nastavenie spôsobu generovania aj na inej úrovni ako samotné generovanie zdrojového kódu.

- metóda end(): definuje príkaz tým, že nastavuje príznak, či má byť pri generovaní kódu výraz vygenerovaný ako príkaz (vo väčine programovacích jazykov to znamená pridanie bodkočiarky na koniec). Túto metódu využíva predovšetkým doménová vrstva.
- metóda terminate(): zabezpečuje generovanie formátovania, komentárov a kódu ukončujúceho výraz. Ak sa jedná o príkaz, čo je dané príznakom, ktorý môže nastaviť metóda end(), vygeneruje jeho ukončenie. Túto metódu využíva predovšetkým jazyková vrstva.
- metóda nl(): nastavuje príznak pre formátovanie. Určuje, či má výraz na svojom začiatku vygenerovať nový riadok. Túto metódu využíva predovšetkým doménová vrstva. Jej návratovou hodnotou je výraz, ktorý modifikuje a je ju možné použiť viacnásobne. To znamená, že výraz statement.nl().nl().nl() má po vygenerovaní kódu tri znaky nového riadku na začiatku.
- metóda begin(): zabezpečuje generovanie formátovania a komentárov na začiatku výrazu. Túto metódu využíva predovšetkým jazyková vrstva.
- metódy beginBlock() a endBlock(): generujú formátovaný kód začiatku a konca blokov kódu (zložené zátvorky "{" a "}" v jazykoch C# a Java, "begin" a "end" v jazyku Delphi).
- atribút openBraceOnNewLine(): nastavuje príznak, ktorý špecifikuje formátovanie blokov kódu. Jedná sa o statický atribút - stačí ho zavolať len na jednom mieste pri konfigurácii generátora. Rozhoduje o tom, či je začiatok bloku kódu generovaný na novom riadku.

Okrem toho môžu byť v kóde generované komentáre. Samotný komentár je síce tiež potomkom ASTStatement, takže je možné s ním pracovať ako s akýmkoľvek iným výrazom, ale okrem toho je možné v záujme zjednodušenia a sprehľadnenia implementácie komentár pripojiť k inštancii triedy ASTStatement metódou addComment().

Komentár môže byť vygenerovaný buď pred príkazom, alebo za príkazom (toto je využité pri generovoaní metadát doménovej vrstvy):

```
// komentár pred príkazom
x = a.b();

x = a.b(); // komentár za príkazom
```

Jazyková vrstva

Jazková vrstva je konkrétnou implementáciou abstraktnej vrstvy pre daný programovací jazyk (Java, C#, ...).

Nepotrebuje poznať nič o doméne ani štruktúre programovacieho jazyka. Úlohou vrstvy je len doplniť syntax jednotlivých štruktúr programovacieho jazyka.

Triedy sú podtriedami absraktných tried abstraktnej vrstvy. Popisujú syntax konkrétneho programovacieho jazyka. (Napríklad potomok triedy ASTClass popíše aj ako vyzerá hlavička triedy - kľúčové slová a ich poradie.) Implementujú abstraktné metódy (generateCode() alebo konkrétnejšie (generateHeader()), podľa typu výrazu), pričom využijú model abstraktnej vrstvy a pridajú syntax špecifickú pre daný programovací jazyk.

Príklad generovania hlavičky triedy

Hlavička triedy by mala vyzerať nasledovne:

C#: [visibility] class [Name] : [Superclass], [Interface1], [Interface2]

Java: [visibility] class [Name] extends [Superclass] implements [Interface1], [Interface2]

Dáta v hranatých zátvorkách sú súčasťou modelu abstraktnej vrstvy, jazyková vrstva k nim pristupuje volaním getter metód. Triedy implementujúce jazykovú vrstvu by mali doplniť podčiarknutú časť - špecifickú syntax programovacieho jazyka.

Generovanie hlavičky triedy pre jazyk C# je možné urobiť nasledovne:

```
generateHeader(indent: int): String
begin
    String header = visibilityModifier()
    header += " class " + name()
    if hasSuperclass()
        header += " : " + superclass().name()
    for interface in interfaces()
        header += ", " + interface.name()
    return header
end
```

Poznámka: jedná sa o zjednodušený pseudokód pre ilustráciu spôsobu generovania kódu. Obsahuje chybu (ak trieda nemá vyplnenú nadtriedu, nie sú správne vyplnené čiarky a dvojbodka), ktorá ale nie je podstatná pre ilustráciu princípu a kód je s ňou podstatne prehľadnejší.

Pre pridanie ďalšieho programovacieho jazyka je potrebné vytvoriť dodatočnú jazykovú vrstvu. Táto vrstva je podstatne jednoduchšia než abstraktná a doménová, čo vyhovuje požiadavke, aby bolo rozšírenie generátora o nový programovací jazyk čo najjednoduchšie.

Doménová vrstva

Práca sa zaoberá tvorbou simulačných modelov ABAsim. Z hľadiska generovania kódu je toto doména - špecifická časť genenerátora modelujúca štruktúru zdrojového kódu simulačného modelu architektúry ABAsim.

Na rozdiel od doménovej vrstvy, abstraktná vrstva a jazykové vrstvy sú nezavislé na doméne a môžu byť použité pre generovanie kódu z akejkoľvek oblasti (je samozrejme

nutné pre danú oblasť implementovať doménovú vrstvu). Táto vrstva implementuje generátory tried domény - simulačného modelu (napríklad generátory kódu agentov, asistentov).

Program, ktorý je implementáciou simulačného modelu ABAsim sa skladá z niekoľkých typov súborov. Pre každý typ súboru je v doménovej vrstve vytvorená trieda - generátor kódu daného typu súboru. Niektoré typy súborov majú niektoré časti spoločné (manažér aj kontinuálny asistent vedia spracovať správu), preto som na zdieľanie týchto častí využil dedičnosť.

Všetky triedy generátorov kódu doménovej vrstvy sú potomkami triedy BaseGenerátor, ktorá definuje základné funkcie generátora a rozhranie, ktoré jej potomkovia implementujú, aby bolo možné využiť polymorfizmus.

Typy tried, ktoré by mali byť výstupom generovania kódu nástrojom ABAbuilder vyplývajú z domény, triedy tvoriace simulačný model.

Typ Agent: pre každého agenta, ktorý je pridaný do diagramu, je vygenerovaná samostatná trieda. Pre každú triedu je vytvorená inštancia generátora pre generovanie agentov AgentGenerator.

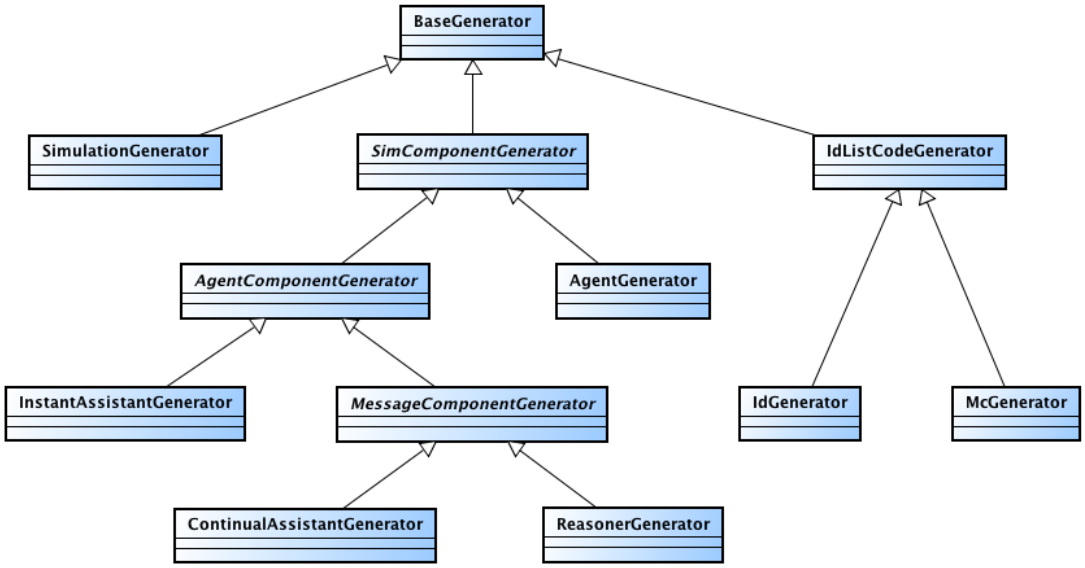
Typ Manažér: manažér je komponentom agenta, ktorý riadi jeho činnosť. Používateľ ho nedefinuje priamo v diagrame, je vytvorený implicitne pre každého agenta. Pre každý model manažéra je vytvorený samostatný generátor pre generovanie manažéra ReasonerGenerator. Generátor je potomkom triedy MessageComponetnGenerator, ktorá implementuje generovanie kódu spracovania doručenej správy.

Typ Kontinuálny asistent: pre každého kontinuálneho asistenta (proces, plánovač a monitor) je vytvorená inštancia generátora kódu ContinualAssistantGenerator. Generátor je ako pri type manažér, je podtriedou generátora MessageComponetnGenerator.

Typ Instantný asistent: pre každého instantného asistenta (akcia, dotaz a poradca) je vytvorená inštancia generátora kódu InstantAssistantGenerator.

Typ Simulácia: na rozdiel od predchádzajúcich typov tried je vytvorená len jedna inštancia generátora SimulationGenerator, nezávisle od toho, aký veľký je simulačný model.

Typ Identifikátor: Podobne ako pri type Simulácia, je vždy vygenerovaný rovnaký počet tried identifikátorov. V tomto prípade sú dva: IdGenerator pre generovanie identifikátorov agentov a komponentov agenta (manažér a asistenti) a trieda McGenerator pre generovanie kódov správ.



Obr 11.3: Hierarchia tried generátorov kódu doménovej vrstvy

Pre prepojenie jazykovej a doménovej vrstvy je použitý návrhový vzor factory. Doménová vrstva pracuje len s rozhraniami definovanými abstraktnou vrstvou. Konkrétne implementácie tried jazykovej vrstvy sú doménovej vrstve poskytnuté prostredníctvom návrhového vzora factory. To znamená, že doménová vrstva je úplne nezávislá na programovacom jazyku.

Implementácia

Generátor kódu bol ako zvyšok nástroja implementovaný v jazyku Java. Jednotlivé vrstvy generátora sú rozdelené do balíčkov. Abstraktná vrstva je v balíčku ast.base, Doménová vrstva v ast.abasim a Jazyková vrstva má pre každý programovací jazyk vlastný balíček (Java - ast.java, C# - ast.cs). Okrem toho som vytvoril balíček ast.defaultLang pre zjednodušenie implementácie niektorých programovacích jazykov.

Syntax mnohých moderných programovacích jazykov je založená na jazykoch C a C++, takže zdieľa veľa spoločných prvkov. V mnohých prípadoch sú dokonca identické (príkaz if, for, bodková notácia volania funkcie, ...).

Balíček ast.defaultLang poskytuje triedy, ktoré takýto jazyk modelujú. Pre jazyky ako Java a C#, ktorých syntax bola inšpirovaná jazykmi C/C++, je potom implementácia zjednodušená tým, že je jazyková vrstva daného jazyka implementovaná vytvorením potomkov tried balíčka ast.defaultLang, v ktorom je už veľká časť jazyka implementovaná. Špecifické časti jazyka sú doplnené prekrytím poskytnutých metód. Napríklad pre cyklus for each by to vyzeralo nasledovne:

Pre generovanie cyklu for each je v balíčku ast.defaultLang trieda DefaultForEachStatement, ktorá je potomkom triedy ASTForEachStatement balíčka ast.base.

Trieda DefaultForEachStatement by bez zmeny vygenerovala tento cyklus nasledovne:

```
for ([Type] [value] in [container]) {
        [body]
}
```

Výrazy v hranatých zátvorkách (Type, value, container, body) sú parametrami triedy abstraktnej vrstvy ASTForEachStatement.

Toto však nie je korektné ani pre Javu ani pre C#. Java používa namiesto kľúčového slova in dvojbodku a C# namiesto kľúčového slova for kľúčové slovo foreach. Trieda ASTDefaultForEachStatement má metódy foreach() a in(), ktorých prekrytím môže byť výsledný kód jednoducho upravený.

Takže triedy, ktoré generujú korektný výraz foreach pre Javu a C# vyzerajú nasledovne:

```
class JavaForEachStatement extends DefaultForEachStatement {
        // Constructor...
        @Override
        public String in() {
                return ":";
        }
}

class CSForEachStatement extends DefaultForEachStatement {
        // Constructor...
        @Override
        public String foreach() {
                return "foreach";
        }
}
```

Problém súčasnej zmeny kódu a modelu:

Nástroj ABAbuilder vygeneruje veľkú časť kódu simulačného modelu a používateľ môže potom tento kód ďalej upravovať. Pri upravovaní však môže zistiť, že je potrebné urobiť ďalšie zmeny v modeli (úpravou diagramov). Keď tieto zmeny urobí a nechá kód pre daný súbor znovu vygenerovať, tak by zrejme nebolo vhodné, aby bol kód súboru vygenerovaný nanovo ako pri pôvodnom generovaní a teda zmazal všetky zmeny, ktoré doň používateľ napísal. Tento kód by mal zostať nezmenený.

Je niekoľko možností ako to zabezpečiť:

1. Prvou možnosťou je parsovať celý zdrojový kód súborov a vytvoriť syntaktický strom (AST - abstract syntax tree), v ktorom sa identifikujú zmenené miesta porovnaním so syntaktickým stromom, ktorý je vygenerovaný na základe diagramov simulačného

modelu. Tieto identifikované zmenené miesta sa adekvátne upravia, aby bolo možné vygenerovať zdrojový kód.

Parsovanie kódu a vytvorenie syntaktického stromu nie je triviálny problém. Je na to potrebné buď napísať vlastný parser, alebo využiť existujúcu knižnicu (akou je napríklad ANTLR).

Výhodou tohto prístupu je, že keď už je abstraktný syntaktický strom vybudovaný, tak sa doň zmeny aplikujú podstatne jednoduchšie, ako do obyčajného textového súboru, pretože sú od seba oddelené operácie parsovania, zmeny modelu a generovania kódu.

Nevýhodou je potreba vytvoriť abstraktný syntaktický strom, pričom každý programovací jazyk je potrebné parsovať špecifickým spôsobom - to znamená, že pre každý jazyk je treba implementovať samostatný parser.

2. Druhá možnosť je pracovať priamo s textovým súborom a hľadať rozdiely. V tomto prípade by sa pracovalo s dvoma zdrojovými súbormi: súbor, do ktorého používateľ dopísal vlastný kód a nový súbor s kódom vygenerovaným podľa diagramov simulačného modelu. Tento samozrejme neobsahuje text, ktorý do druhého súboru pripísal používateľ, ale obsahuje naviac časti kódu, ktoré boli vygenerované na základe úprav, ktoré boli urobené v diagramoch od posledného generovania kódu. Prvým krokom teda je vygenerovanie kódu súboru podľa modelu, čo je pomerne jednoduché, keďže sa syntaktický strom vytvorí priamo z diagramu simulačného modelu (nie je treba parsovať zdrojový kód). Druhým krokom je nájdenie rozdielov v týchto dvoch súboroch, ktoré budú následne použité pri spojení súborov do jedného, ktorý bude obsahovať aj používateľský kód, aj vygenerovaný kód. Toto spojenie by teda malo fungovať podobne ako spojenie súborov nástrojov na správu zdrojových kódov (git, mercurial, ...).

Výhodou tohto prístupu je to, že je nezávislý na programovacom jazyku (môže obsahovať akýkoľvek obsah). Nevýhodou je to, že sa so samotným textom náročnejšie pracuje, keďže nie je vytvorený model kódu, tak je text spracovaný porovnávaním a manuálnym skladaním reťazcov (substring, regulárne výrazy). Ďalší problém, ktorý je známy každému, kto pracoval s git-om alebo podobným systémom, sú konflikty. Keby bol použitý rovnaký mechanizmus riešenia konfliktov ako používa git, tak niektoré časti kódu by mohli byť neurčité a bolo by potrebné, aby ich riešil používateľ. Bolo by samozrejme lepšie, aby

program usúdil sám, kam daná sekcia kódu patrí, ale na to by potreboval mať znalosť o tom, čo dané časti vygenerovaného kódu znamenajú. Túto znalosť ale pri tomto prístupe nemá. Na to by bolo potrebné parsovať upravený zdrojový kód.

3. Treťou možnosťou, ktorú som sa rozhodol použiť, je kombinácia obidvoch uvedených spôsobov, pričom sú využité informácie o doméne. V git-e rieši konflikty programátor, pretože git nevie, ako má výsledný zdrojový kód vyzerať - v súbore môže byť čokoľvek. Na rozdiel od toho je kód, s ktorým pracuje nástroj ABAbuilder pomerne špecifický: jeho výstupom je kód simulačného modelu ABAsim a túto informáciu o zdrojovom kóde môže použiť na to, aby dokázal urobiť spojenie súborov automaticky. Pri generovaní kódu doň doplní metadáta, ktoré popisujú o aký kód sa jedná. Parsovanie súboru je len obmedzené, implementované tak, aby bolo nezávislé na programovacom jazyku. S kódom upraveného súboru sa pracuje po blokoch, ktoré sú špecifikované len metadátami - nie je podstatné, či bol použitý programovací jazyk Java, C# alebo PHP. Vygenerovaný kód je nahradený po blokoch, ktoré sú generované z abstraktného syntaktického stromu, ktorý je vybudovaný podľa modelu. Vyhol som sa teda potrebe parsovania zdrojového kódu programovacích jazykov, ale napriek tomu je možné podľa diagramov upravovať zdrojový kód, aj keď doň používateľ nástroja doplní vlastný kód spôsobom, ktorý je nezávislý na programovacom jazyku.

Kvôli tomu, aby bolo čo najjednoduchšie rozšíriť nástroj o podporu generovania kódu ďalšieho programovacieho jazyka, som sa snažil vyhnúť tomu, aby bolo potrebné parsovať zdrojový kód programovacieho jazyka, keďže toto by bolo nutné implemenntovať pre každý zvlášť.

Problém nastáva jedine pri zmene nadtriedy, napríklad zmena z potomka triedy Process na potomka triedy Scheduler. Pre tento prípad musí trieda vedieť parsovať svoju hlavičku.

Problém premenovania komponentu

Po premenovaní komponentu v diagrame nastáva problém pri viacnásobnom generovaní kódu s vyhľadaním správneho komponentu. Komponent nie je možné nájsť podľa názvu súboru, pretože názov súboru je odvodený od názvu triedy, ktorý bol zmenený. Model by

si mohol pamätať pôvodný názov súboru, ale v tom prípade by boli údaje popisujúce súbor uložené mimo samotného súboru, čo by mohlo spôsobiť problémy. Rozhodol som sa rozlišovať súbory podľa údajov, ktoré sú v nich zapísané tak, aby boli nezávislé na implementácii triedy.

To znamená rozlišovať súbory podľa identifikátorov zapísaných v metadátach, nie podľa ich názvov.

Metadáta

Metadáta majú formát json (čiarkami oddelená množina dvojíc kľúč="hodnota", napríklad: key1="value1", key2="value2")

Vygenerované triedy obsahujú kód (metódy, atribúty, konštruktory a konštanty), ktorý môže byť z hľadiska generátora troch typov: nemodifikovateľný, modifikovateľný a používateľský.

Jednotlivé typy kódu sú dané metadátami, ktoré generátor kódu vygeneruje ako komentáre.

Tieto komentáre začínajú identifikátorom, ktorý špecifikuje, že sa jedná o metadáta, za ktorým nasledujú samotné metadáta - informácie pre generátor (a do istej miery aj pre používateľa).

Ako identifikátor metadát som zvolil retazec "meta!". Začiatok nemodifikovateľného kódu je označený metadátami, ktoré obsahujú: tag="begin", a koniec nemodifikovateľného kódu je označený ako: tag="end". Časť ohraničená metadátami s hodnotami tag="begin" a tag="end" tvorí blok kódu, ktorý môže obsahovať ľubovolný počet metód, atribútov alebo konštánt.

Metadáta modifikovateľného kódu sú nad hlavičkou časti kódu, ktorú popisujú (metóda, trieda).

Nemodifikovateľný kód by nemal byť menený používateľom. V prípade pregenerovania súboru bude všetko, čo používateľ do tejto sekcie pridal prepísané novou verziou nemodifikovateľného kódu. Obsah tejto sekcie závisí od typu generovaného súboru. Napríklad u agentov obsahuje inicializáciu - vytvorenie manažéra a asistentov a registráciu

správ, ktoré vie daný agent spracovať. Tieto veci sú dané modelom a keby ich používateľ akokoľvek zmenil, tak by už nezodpovedali tomu, ako to bolo pôvodne definované v modeli.

Napríklad:

```
//meta! tag="begin", userInfo="Generated code: do not modify"
private void init(){
        new MyManager(Id.myManager, self.mySim, self);
        new MyProcess(Id.myProcess, self.mySim, self);
        new MyAction(Id.myAction, self.mySim, self);
        addOwnMessage(Mc.messageA);
        addOwnMessage(Mc.messageB);
}
//meta! tag="end"
```

Modifikovateľný kód môže používateľ zmeniť, ale len obmedzeným spôsobom. Tieto obmedzenia vyplývajú napríklad z toho, že je potrebné nejakým spôsobom prepojiť nemodifikovateľný kód s používateľským. Hlavičky metód modifikovateľného kódu tvoria rozhranie pre nemodifikovateľný kód, ktorý tieto metódy používa.

Používateľ by nemal meniť hlavičky metód označených ako modifikovateľné, ale môže doplniť telo metód.

Kód je označený ako modifikovateľný prostredníctvom metadát v komentári nad hlavičkou metódy alebo triedy.

Napríklad:

```
//meta! id="1", sender="2"
processNotice(message: MessageForm): void
begin
    // user code
end
```

Samotná trieda je tiež modifikovateľným kódom, to znamená, že používateľ môže upravovať jej telo ľubovoľne (mal by ale samozrejme zohľadniť to, že telo triedy môže obsahovať kód ľubovoľného typu - aj modifikovateľný aj nemodifikovateľný), ale úprava hlavičky triedy je obmedzená tým, že používateľ nemôže zmeniť názov triedy a nadtriedu triedy, pretože táto časť je určená modelom a keby súbor pregeneroval, tak by boli tieto zmeny prepísané hodnotami modelu. Môže však ľubovolne pridávať rozhrania.

Pri opätovnom generovaní modifikovateľného kódu sú hlavným problémom rozdiely medzi modelom vytvoreným parsovaním metadát zdrojového kódu a modelom vytvoreným z diagramov simulačného modelu.

Tieto rozdiely môžu byť niekoľkých typov, podľa toho aké zmeny používateľ v modeli urobil.

V prípade, že používateľ do diagramu pridal nový komponent, je potrebné do kódu pridať novú sekciu. Je vygenerovaná nová metóda s prázdnym telom, do ktorého používateľ napíše kód, ktorý spracuje danú správu.

V prípade, že používateľ zmenil komponent v diagrame (napríklad ho premenoval), je potrebné túto zmenu aplikovať do správnej sekcie kódu. To znamená zmenu hlavičky príslušnej metódy a/alebo jej metadát bez zmeny jej tela. To tiež znamená, že nie je možné identifikovať triedy podľa ich názvu, keďže ten môže používateľ zmeniť. Namiesto toho sú pred hlavičku triedy vygenerované metadáta, ktoré obsahujú jej interné ID a tým ju jednoznačne identifikujú.

Prípad, že používateľ z diagramu odstránil komponent, by sa dal riešiť odstránením príslušnej sekcie kódu, ale v tomto prípade existuje možnosť, že telo metódy obsahuje kód

ktorý používateľ nechce zmazať. Preto som sa rozhodol nechať túto metódu v modifikovateľnej sekcii nezmenenú. Je však odstránená v nemodifikovateľnej časti. Okrem toho sú metadáta metódy zmenené. Pôvodné údaje o type správy, identifikátore metódy a odosielateľovi správy sú nahradené informáciou pre používateľa: userInfo="Removed from model".

Implementácia aktualizácie kódu podľa zmeneného modelu sa skladá z niekoľkých krokov:

Najskôr je vytvorený model zdrojového kódu podľa metadát, ktoré obsahuje. Toto je nezávislé na programovacom jazyku. Tento model obsahuje informácie o tom, kde sa v zdrojovom kóde nachádzajú jednotlivé sekcie modifikovateľného a nemodifikovateľného kódu.

Súčasne je vytvorený abstraktný syntaktický strom podľa diagramov simulačného modelu, ktorý obsahuje požadovanú štruktúru triedy, ale neobsahuje používateľský kód.

Nasleduje spárovanie metód do jednej z troch skupín: nová, odstránená a zmenená (metódy môžu patriť aj do skupiny "nezmenené", ale tie nie sú upravované, preto sú pri spracovaní ignorované). Do ktorej skupiny metóda patrí je rozlíšené porovnávaním metadát, ktoré boli získané parsovaním kódu a modelu metód abstraktného syntaktického stromu. Identita metódy je daná interným ID, ktoré je vygenerované pri vytvorení daného prvku a potom už nie je menené. Sú vytvorené dve množiny modelov metód. Tieto množiny síce reprezentujú tie isté metódy, ale jedná sa o rôzne druhy nekompatibilných modelov. Prvý zoznam obsahuje uzly AST, zatiaľ čo druhý je tvorený metadátami zdrojového kódu. Z obidvoch zoznamov však nie je problém získať interné ID metód a podľa nich metódy rozdeliť.

Ak sa metóda nachádza v množine AST, ale v množine metadát nie, tak sa jedná o novopridanú metódu, ktorú je potrebné vygenerovať kompletne (metadáta, hlavička a telo).

Ak sa metóda nenachádza v množine AST, ale v množine metadát áno, tak sa jedná o metódu reprezentujúcu správu, ktorá bola odstránená z diagramu. Jej spracovanie teda končí (aby používateľ neprišiel o kód, ktorý do nej napísal).

Ak sa metóda nachádza aj v množine AST aj v množine metadát a aspoň jeden z ich parametrov je odlišný (napríklad názov), tak sa jedná o zmenenú metódu, ktorú je potrebné vyhľadať v zdrojovom kóde a prepísať jej hlavičku a metadáta kódom vygenerovaným uzlom syntaktického stromu reprezentujúcim túto metódu.

Nakoniec sú nájdené zmeny aplikované do zdrojového kódu.

Používateľský kód je všetko ostatné, čo používateľ nástroja ABAbuilder doplní do vygenerovaného kódu mimo sekcii označených ako modifikovateľný a nemodifikovateľný kód. Po pregenerovaní súboru tento kód zostáva nezmenený.

Používateľský kód nemá priradené žiadne metadáta.

Inicializácia komponentov podľa modelu je síce v nemodifikovateľnej sekcii, ale nemusí tam byť výhradne. Je možné pridať akúkoľvek ďalšiu inicializáciu do konštruktora, aj nových komponentov simulačného modelu, ale tieto nebudú pridané do modelu nástroja ABAbuilder. Bolo by dobré, aby po pridaní komponentu jeho pripísaním do kódu bol tento komponent pridaný aj do modelu nástroja, ale túto funkcionalitu nástroj v aktuálnej verzii neposkytuje.

Generovanie kódu agenta:

Agent môže byť buď potomkom riadiaceho agenta alebo potomkom dynamického agenta. V obidvoch prípadoch je použitý ten istý typ generátora. V používateľskej sekcii vygeneruje konštruktor agenta, do ktorého môže používateľ dopísať vlastnú inicializáciu agenta. Okrem toho tento konštruktor obsahuje inicializáciu nadtriedy a volanie metódy init(). Metóda init je vygenerovaná v nemodifikovateľnej sekcii, preto by ju používateľ nemal meniť. Obsahuje registráciu správ, ktoré vie agent spracovať a sú v nej vytvorené interné komponenty agenta vrátane manažéra.

Generovanie kódu manažéra:

Generovanie kódu manažéra agenta môže prebiehať dvomi spôsobmi podľa toho, či používateľ vybral možnosť, že má byť správanie agenta riadené ABAgrafom alebo nie.

V obidvoch prípadoch je štandardne vygenerovaná trieda, ktorej predkom je trieda Manager alebo DynamicManager zo simulačného jadra a konštruktor s príkazmi inicializácie nadtriedy a volaním metódy init, ak je to potrebné.

V prípade, že je použitý ABAgraf je v tele metódy init vygenerovaný kód pre načítanie ABAgrafu z xml súboru a v tele processMesage spracovanie správy ABAgrafom. Obidve metódy patria do nemodifikovateľnej sekcie triedy. Do používateľskej sekcie je vygenerovaná konštanta s cestou, na ktorej manažér nájde súbor s ABAgrafom. Používateľ ju môže zmeniť pokiaľ mu nevyhovuje implicitná cesta vygenerovaná generátorom.

Používaťeľ v tomto prípade nemusí robiť žiadne ďalšie zmeny v tejto triede. Namiesto toho kód implementujúci reakciu manažéra na prijaté správy vytvorí ABAgraf v grafickom editore.

V prípade, že agent nepoužíva ABAgraf, je v nemodifikovateľnej sekcii vygenerovaná metóda processMessage a v nej volania metód modifikovateľnej sekcie. Metóda, ktorá má byť zavolaná je vybraná podľa kódu a odosielateľa správy, ak je to potrebné rozlíšiť. Okrem toho sú v modifikovateľnej sekcii vygenerované metódy pre spracovanie doručenej správy a dodatočná metóda processOther(), v ktorej môže používateľ spracovať správy, ktoré definoval v kóde (t.j. neboli vygenerované generátorom kódu).

V prípade viacnásobného generovania kódu manažéra sú metódy init a processMessage prepísané aktuálnou verziou a sú upravené hlavičky metód spracujúcich správy.

Môže sa stať, že prijaté správy môžu mať rovnaký kód, ale líšia sa odosielateľom, pritom spôsob spracovania týchto správ je odlišný (napríklad správa s kódom Mc.finish môže byť prijatá od dvoch kontinuálnych asistentov - ProcessA a ProcessB). Tento problém som vyriešil jednoducho tým, že názov vygenerovanej metódy je daný kódom aj odosielateľom. V prípade, že správu s daným kódom manažér príjma len od jedného komponentu, je pre jej spracovanie vygenerovaná metóda s názvom "process[identifikátor kódu správy]" napríklad processNotice(). Ak môže správu s daným kódom manažér prijať od viacerých

odosielateľov je vygenerovaná metóda s názvom "process[identifikátor kódu správy] [odosielateľ]" napríklad processFinishProcessA().

Príklad triedy manažéra, ktorý nepoužíva ABAgraf:

```
//meta! id=5
class ManagerXXX begin
    //meta! id="1", sender="ProcessA"
    processFinishProcessA(message: MessageForm)
    begin
        // user code
    end

    //meta! id="2", sender="SchedulerB"
    processFinishSchedulerB(message: MessageForm)
    begin
        // user code
    end

    //meta! id="3", sender="AgentABC"
    processNoticeXXX(message: MessageForm)
    begin
        // user code
    end
```

```
        //meta! userInfo="Process mesages defined in code", id="0"
        processNoticeXXX(message: MessageForm)
        begin
            switch (message.code)
                // user code
        end

        //meta! tag="begin", userInfo="Generated code: do not modify"
        processMessage(message: MessageForm)
        begin
            switch (message.code)
                case Mc.finish:
                    switch (message.sender.id)
                        case Id.processA: processFinishProcessA(message)
                        case Id.SchedulerB: processFinishSchedulerB(message)
                case Mc.noticeXXX: processNoticeXXX(message)
                default: processOther(message)
        end
        //meta! tag="end"
end
```

Generovanie kódu triedy simulácie:

Trieda simulácie vytvára hierarchiu agentov. Vytvorenie jednotlivých inštancií agentov musí byť urobené v správnom poradí - v opačnom prípade by program nebolo možné skompilovať, pretože parameter konštruktora agenta je jeho predok, ktorý už musí byť

vytvorený. Ako prvé je vygenerované vytvorenie inštancie agenta na vrchole hierarchie (boss), potom jeho potomkov, potom potomkov jeho potomkov...

Vytvorené inštancie uloží do atribútov triedy.

Generovanie kódu triedy správy:

Okrem tried implementujúcich komponenty simulačného modelu, generátor kódu vytvorí aj triedu pre správy, ktoré si komponenty posielajú. Používateľ si do nej doplní vlastné atribúty.

Trieda obsahuje iba používateľský kód, teda nepodporuje viacnásobné generovanie kódu, ktoré ale ani nepotrebuje. Úlohou triedy je iba zjednodušenie práce používateľa tým, že mu poskytne šablónu správy, ktorá má už definovaného požadovaného predka a prekrýva potrebné metódy.

12. Uloženie modelu

Model je uložený vo formáte XML. Súbor má ako koreňový element značku "ABAsim", ktorá má dve sekcie: "Agents" a "Messages".

V sekcii agents sú uložené údaje o všetkých agentoch, ako aj údaje o interných komponentoch agenta. Každý agent má svoje interné komponenty vnorené v sekcii Components. Samotná sekcia Components by mohla byť aj na úrovni Agents a Messages, ale to by znamenalo, že by sa v nej musel pri načítavaní súboru vyhľadať. Aby boli súvisiace komponenty pokope, rozhodol som sa vytvoriť vnorenú sekciu Components pre každého agenta.

To znamená, že diagram hierarchie agentov a diagramy komponentov agenta sú uložené v jednom súbore, ale pre každú Petriho sieť je vytvorený samostatný súbor. Dôvodom je to, že nástroj vygeneruje z diagramu hierarchie agentov a diagramu komponentov agenta kód a teda nie je dôvod na to, aby boli v samostatných súboroch. XML súbory pre ABAgrafy načítava interpreter Petriho sietí za behu simulácie, preto sú uložené samostatne.

V sekcii Messages sú uložené všetky hrany, ktoré spájajú komponenty. Pre identifikáciu komponentov, ku ktorým patria, využívajú značky source a destination, ktoré obsahujú interné ID daného komponentu.

Príklad formátu súboru:

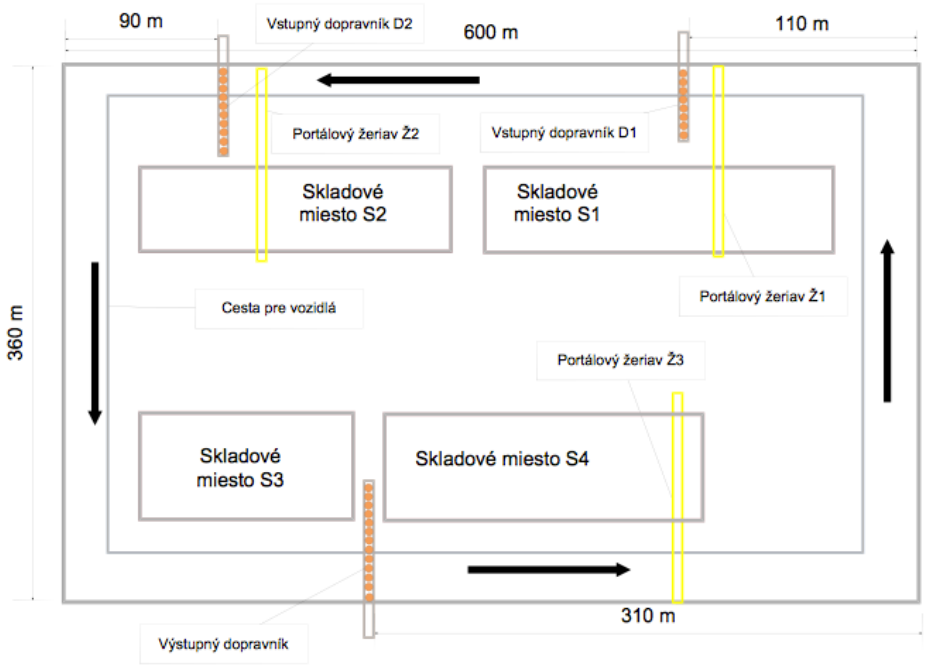
```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<ABAsim>
    <Agents>
        <Agent>
            <simId>1</simId>
            <internalId>1</internalId>
            <name>MyAgent</name>
            <bounds>336,226,180,180</bounds>
            <reasoner>MyManager</reasoner>
            <reasonerBounds>74,25,32,25</reasonerBounds>
            <usePetriNet>false</usePetriNet>
            <Components>
                <Action>
                    <simId>1001</simId>
                    <internalId>7</internalId>
                    <name>MyAction</name>
                    <bounds>73,123,32,15</bounds>
                </Action>
                <Process> … </Process>
            </Components>
        </Agent>
```

```
            <Agent> … </Agent>
        </Agents>
        <Messages>
            <Message>
                <simId>1</simId>
                <internalId>17</internalId>
                <name>MyNotice</name>
                <type>Notice</type>
                <source>1</source>
                <destination>12</destination>
            </Message>
            <Message> … </Message>
        </Messages>
</ABAsim>
```

13. Overenie riešenia

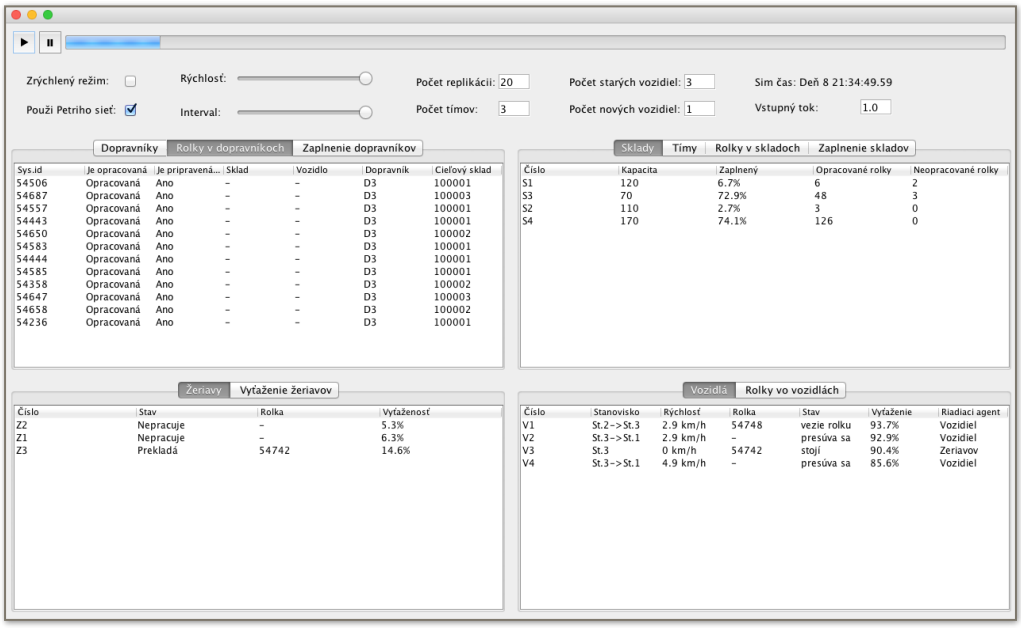
Interpreter ABAgrafov vygenerovaných vytvoreným nástrojom som overil na simulačnom modeli oceliarne. Oceliareň funguje nasledovne:

Do oceliarne vchádzajú rolky ocele po dvoch vstupných dopravníkoch, z ktorých sú žeriavmi prekladané na skladové miesta alebo na vozidlá. V skladových miestach sú rolky opracované tímami pracovníkov, ktorí sa presúvajú medzi jednotlivými skladovými miestami. Vozidlá sa presúvajú okolo haly v stanovenom smere a presúvajú rolky medzi skladovými miestami. Po opracovaní sú rolky žeriavom presunuté na výstupný dopravník. V skladových miestach pri výstupnom dopravníku sú rolky ocele pripravované na expedovanie. Počet tímov pracovníkov a počet vozidiel sú parametrami modelu.

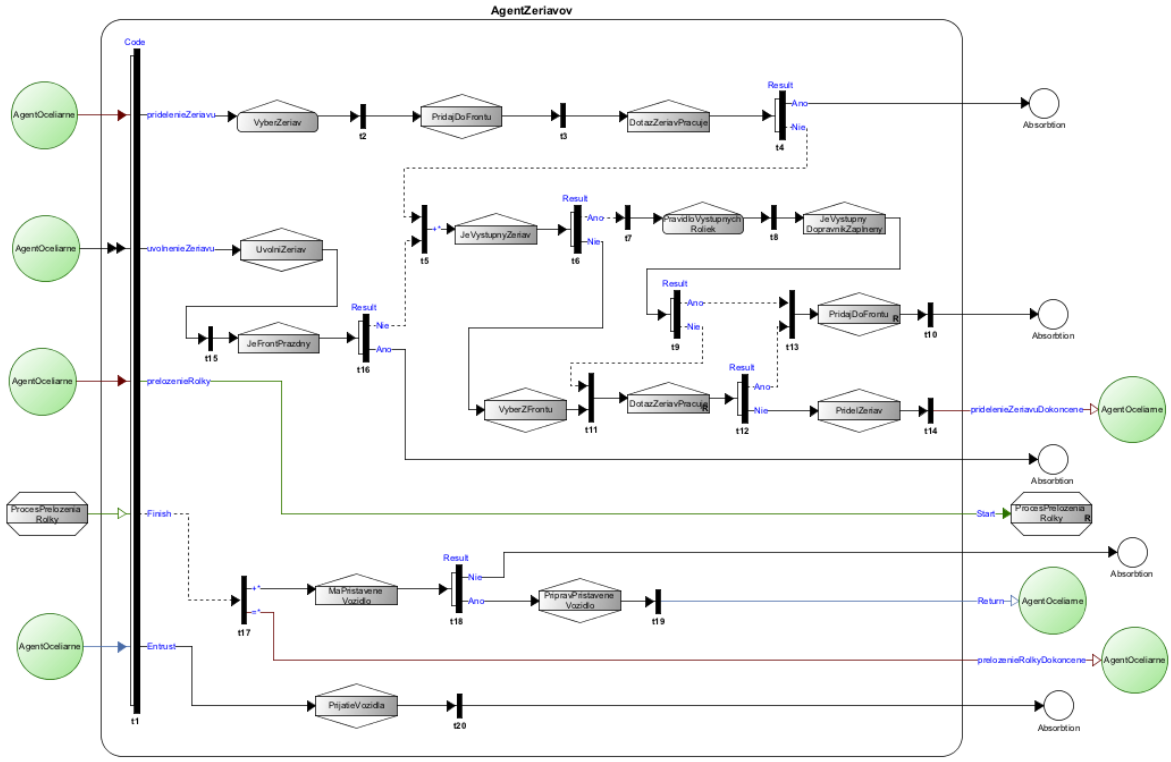


Obr. 13.1: Schéma modelu oceliarne

Simulačný model oceliarne som vypracoval dvoma spôsobmi: s použitím interpretera ABAgrafov a priamou implementáciou manažérov agentov kódom. Pri spustení simulácie je možné vybrať, či má byť použité spracovanie interpreterom ABAgrafov (checkbox "Použi Petriho sieť").



Obr. 13.2: Simulácia modelu oceliarne



Obr. 13.3: ABAgraf jedného z agentov modelu oceliarne

Pre obidva prípady som simuláciu spustil s 1000 replikáciami a dĺžkou simulačného behu jeden rok. V tabuľke sú porovnané výsledky simulácie.

	S použitím ABAgrafu	Bez použitia ABAgrafu	Rozdiel
Vyťaženie vozidiel	86.78%	86.76%	0.02
Vyťaženie žeriavu 1	6.16%	6.16%	0.00
Vyťaženie žeriavu 2	5.29%	5.29%	0.00
Vyťaženie žeriavu 3	14.16%	14.16%	0.00
Zaplnenie skladu 1	10.31%	10.27%	0.04
Zaplnenie skladu 2	8.48%	8.44%	0.04
Zaplnenie skladu 3	24.93%	24.42%	0.51
Zaplnenie skladu 4	59.58%	58.29%	1.29

Tab. 13.1: Výsledky simulácie modelu oceliarne

Ako je možné vidieť v tabuľke 13.1, výsledky simulačných behov sa výrazne nelíšia. S toho vyplýva, že interpreter ABAgrafov funguje správne.

Nástroj bol poskytnutý desiatkam študentov fakulty riadenia a informatiky v rámci výučby predmetu Diskrétna simulácia, ktorý ho použili pre tvorbu svojich semestrálnych prác, čo je ďalším overením funkčnosti nástroja.

14. Záver

Vytvoril som CASE nástroj pre tvorbu simulačných modelov architektúry ABAsim. Nástroj obsahuje editory diagramov pre definovanie hierarchickej štruktúry agentov a správ, ktoré si agenti posielajú, ako aj editory pre definíciu internej implementácie agenta. Nástroj podporuje viacnásobné generovanie kódu do zdrojových súborov, ktoré môže tvorca simulačného modelu manuálne upravovať a súčasne pokračovať v modelovaní v grafickom editore.

Vtvoril som simulačné jadrá pre Javu a C#, ktoré sú kompatibilné s vytvoreným nástrojom a vypracoval k nim metodickú príručku a niekoľko ukážkových simulačných modelov.

15. Zoznam skratiek

ABAsim	agent-based architecture of simulation model
ANTLR	another tool for language recognition
AST	abstract syntax tree
CASE	computer-aided software engineering
EMF	eclipse modeling framework
GEF	graphical modeling framevork
XML	extensible markup language
ZUI	zoomable user interface

16. Zoznam tabuliek

17. Zoznam obrázkov

18. Zoznam použitej literatúry

ADAMKO, N. 2013. Agentovo orientovaná simulácia zložitých obslužných systémov. Žilina. Habilitačná práca. Žilinská univerzita.

HOLÚBEK, M. 2011. Metodika vývoja simulačných modelov rozsiahlych obslužných systémov. Žilina. Dizertačná práca. Žilinská univerzita

KOCIFAJ, M. 2014. Modelovanie činnosti autonómnych mobilných prostriedkov v komplexnom stochastickom prostredí dopravných terminálov. Žilina. Dizertačná práca. Žilinská univerzita