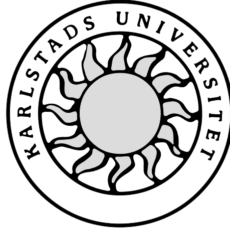


Datavetenskap

Peter Labraaten

Stefan Larsson

GNU Emacs som semantisk editor med stöd för kontraktsprogrammering

---

D-uppsats

2004:04

GNU Emacs som semantisk editor med stöd för kontraktsprogrammering

Peter Labraaten

Stefan Larsson

© 2004 Peter Labraaten, Stefan Larsson och Karlstads universitet

Denna uppsats är skriven som en del av det arbete som krävs för att erhålla en magisterexamen i datavetenskap. Allt material i denna uppsats, vilket inte är vårt eget, har blivit tydligt identifierat och inget material är inkluderat som tidigare använts för erhållande av annan examen.

Peter Labraaten

Stefan Larsson

Godkänd, 2004-06-17

Handledare: Donald F. Ross

Examinator: Thijs Jan Holleboom

Sammanfattning

I denna uppsats beskrivs och motiveras utvecklingen av en prototyp som ger stöd vid kontraktsprogrammering i textredigeraren GNU Emacs. Prototypen ger kontraktsstöd vid programmering i C++ genom visning av kontraktsinformation, infogning av förvillkor, analys av förvillkorskontroller samt parsning av förvillkor. Stödet är i första hand tänkt att underlätta inlärning av kontraktsprogrammering genom att förenkla kontraktshanteringen men även genom att ge kontrakt en tydligare roll i mjukvaruutveckling. Eftersom kontrakts-programmering är uppsatsens grund ges en grundläggande genomgång av denna metod där bland annat kontrakts olika delar behandlas. Vikten av att använda kontrakt för att beskriva mjukvaras semantik under mjukvaruutveckling diskuteras också. Den utvecklade prototypens funktionalitet och den använda miljön GNU Emacs utvärderas och förslag på framtida arbete presenteras.

GNU Emacs as a Semantic Editor with Support for Contract Programming

Abstract

This dissertation describes and motivates the development of a prototype that provides support for contract programming in the GNU Emacs editor. The contract programming support is designed for the C++ programming language and consists of the displaying of contract information, insertion of preconditions, analysis of precondition tests and parsing of preconditions. The support is primarily intended to ease the learning of programming with contracts by simplifying the management of contracts but also to give contracts a more defined role in software development. Since contract programming is the key concept of the dissertation, there is a basic review of the method which includes a description of the different elements of contracts. The importance of using contracts to describe the semantics of software during development will also be discussed. The functionality of the developed prototype along with the GNU Emacs environment will be evaluated and suggestions for future work presented.

Innehållsförteckning

1 Inledning ... 1

2 Bakgrund ... 5

2.1 Tidigare arbete ... 5

2.2 Syfte ... 5

2.3 Mål ... 6

2.4 Semantik och kontrakt ... 6

2.5 Befintliga verktyg ... 7

2.5.1 iContract
2.5.2 Harmonia
2.5.3 Summering

2.6 GNU Emacs ... 9

3 Kontraktsprogrammering ... 11

3.1 Inledning ... 11

3.2 Terminologi ... 12

3.2.1 Modul
3.2.2 Klient
3.2.3 Leverantör
3.2.4 Funktion

3.3 Semantik och kontrakt ... 13

3.4 Ett kontrakts delar ... 14

3.4.1 Förvillkor
3.4.2 Eftervillkor
3.4.3 Invarianter

3.5 Mjukvarukvalitet ... 16

3.6 Användning av kontrakt vid mjukvaruutveckling ... 19

3.7 Sammanfattning ... 21

4 Framtagning av prototyp ... 23

4.1 Inledning ... 23

4.2 GNU Emacs ... 24

4.2.1 Textredigeraren Emacs
4.2.2 Emacs Lisp
4.2.3 Minor Modes
4.2.4 CEDET

4.3 Funktionalitet ... 28
4.4 Prototyp ... 29
4.4.1 Överblick
4.4.2 Timer
4.4.3 Kontroll av anrop
4.4.4 Inhämtning av information
4.4.5 Visning av information
4.4.6 Exekverbara förvillkor
4.4.7 Kodanalys
4.4.8 Infogning av förvillkor
4.5 Sammanfattning ... 41
5 Utvärdering av prototyp ... 43
5.1 Inledning ... 43
5.2 Demonstration av prototyp ... 43
5.2.1 Visning av kontrakt
5.2.2 Infogning av förvillkor
5.2.3 Markering av förvillkorskontroll
5.2.4 Alternativ visning av kontrakt
5.2.5 Parsning av förvillkor i leverantörskod
5.2.6 Inställningar via inställningspanelen
5.3 Utvärdering av funktionalitet ... 51
5.3.1 Timer
5.3.2 Kontroll av anrop
5.3.3 Inhämtning av information
5.3.4 Visning av information
5.3.5 Exekverbara förvillkor
5.3.6 Kodanalys
5.3.7 Infogning av förvillkor
5.4 Utvärdering av GNU Emacs ... 56
5.5 Sammanfattning ... 58
6 Slutsats ... 59
6.1 Sammanfattning ... 59
6.2 Framtida arbete ... 60
6.3 Slutord ... 61
Referenser ... 63
Webreferenser ... 63
A Grammatik för exekverbara förvillkor ... 65
B Användarmanual ... 67
B.1 Att utföra kommandon i GNU Emacs ... 67
B.2 Att ladda stödet ... 67
B.3 Att slå på och av stödet ... 68
B.4 Utformning av kontrakt ... 69

B.5 Visning av kontrakt för funktioner ... 70
B.6 Infogning av kontroll av förvillkor ... 73
B.7 Att spåra om en funktion uppfyller sitt förvillkor ... 74
B.8 Parsning av förvillkor ... 75
B.9 Inställningspanel ... 77
B.10 Standardvärden för inställningar ... 79

C Systembeskrivning ... 81

C.1 GNU Emacs ... 81
C.2 CEDET ... 81
C.2.1 Parser
C.2.2 Symboltabell
C.3 Översikt ... 84
C.4 Beskrivning ... 84
C.4.1 contract-mode.el
C.4.2 contract-parse.el

D Källkod ... 87

D.1 contract-load.el ... 87
D.2 contract-mode.el ... 87
D.3 contract-parse.el ... 104
D.4 contract-utils.el ... 109

Figurförteckning

Figur 1.1: Översikt av prototyp. 2
Figur 3.1: Illustration av använda termer. 12
Figur 3.2: Användning av defensiv programmering för att säkerställa korrekthet. 17
Figur 3.3: Användning av kontrakt för att förenkla moduler. 18
Figur 3.4: Illustration av kontrakts möjliga påverkan. 20
Figur 4.1: Emacs uppdelat i två fönster. 25
Figur 4.2: Här kan ses att C++ major mode och Abbrev minor mode är aktiva. 26
Figur 4.3: Kommandot som sparar till fil interagerar med användaren. 26
Figur 4.4: Grundfunktionalitet 30
Figur 4.5: Positioner som leder till anropsdetektering. 31
Figur 4.6: Markörpositioner som leder till parsning av förvillkor. 34
Figur 4.7: Trädrepresentation av kod. 37
Figur 5.1: Visning av kontraktsinformation. 44
Figur 5.2: Infogning av förvillkor. 45
Figur 5.3: Markering av förvillkorskontroll. 47
Figur 5.4: Alternativt visningssätt för kontraktsinformation. 48
Figur 5.5: Syntaktiskt korrekt exekverbart uttryck. 49
Figur 5.6: Ej exekverbart förvillkorsuttryck. 49
Figur 5.7: Inställning av Contract-Mode med hjälp av inställningspanel. 51
Figur 5.8: Visning av kontrakt för parameterfunktion. 53
Figur B.1: Emacs mode line anger att Contract-mode är påslaget (Contr). 68
Figur B.2: Exempel där kontrakt visas med en tooltip ruta. 71
Figur B.3: Exempel där kontrakt visas i minibuffern. 72
Figur B.4: Positioner som resulterar i anropsdetektering. 72
Figur B.5: Infogning av test om förvillkor är uppfyllt. 74
Figur B.6: Närmast föregående kontroll av förvillkor markeras. 75
Figur B.7: Markörpositioner som leder till parsning av förvillkor. 76

Figur B.8: Syntaktiskt korrekt exekverbart uttryck. 76
Figur B.9: Ej exekverbart förvillkorsuttryck. 77
Figur B.10: Inställningspanel för Contract-Mode... 78

1 Inledning

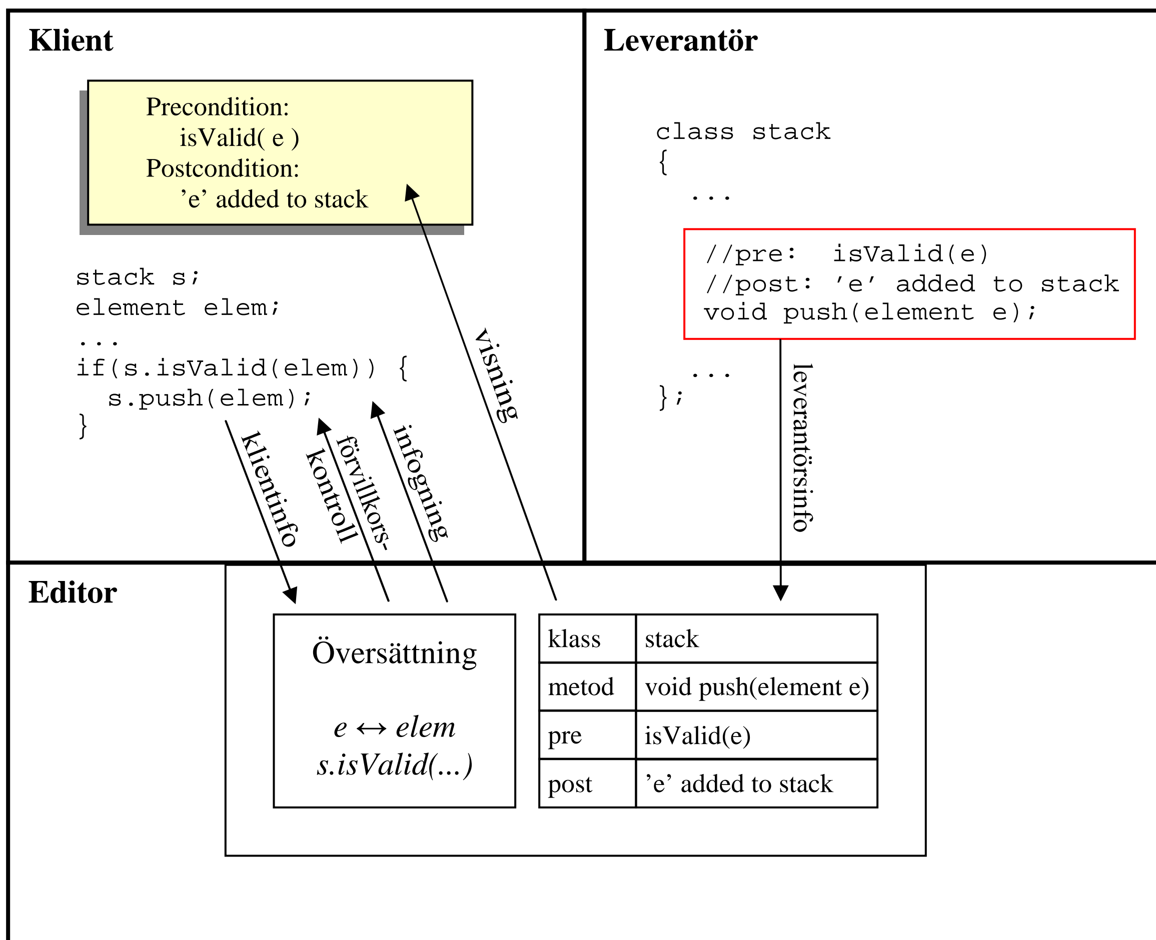
För att kunna utveckla mjukvara med hög kvalitet är det viktigt att beskriva mjukvarans semantik. De moduler som utgör ett mjukvarusystem bör ha en tydlig semantisk beskrivning för att undvika felaktig användning som kan leda till fel i systemet. Då mjukvaruutveckling oftast involverar ett flertal personer är semantiska beskrivningar viktigt för att undvika att olika tolkningar görs. För att öka livslängden hos mjukvarusystem sker också en kontinuerlig utveckling och uppdatering av system som är i bruk. Detta ställer också höga krav på dokumenteringen av mjukvarans semantik eftersom lång tid kan ha passerat eller att uppdatering inte utförs av orginalutvecklarna.

En metod för att beskriva modulers semantik är kontraktsprogrammering [3]. Genom användning av kontrakt, där leverantörmodulers och klientmodulers åtaganden och fördelar specificeras, tillhandahåller kontraktsprogrammering ett sätt att beskriva modulers semantik samtidigt som modulernas kodkomplexitet minskas. De kontrakt som används vid kontraktsprogrammering består av följande delar:

- Förvillkor, som är villkor klienter måste uppfylla innan de anropar en metod.
- Eftervillkor, som beskriver resultatet av en metod då förvillkoret var uppfyllt vid anrop.
- Invarianter, som beskriver en moduls tillstånd efter exekvering av en medlemsmetod.

Det finns idag editorer som tillhandahåller olika former av stöd vid programmering. Exempel på stöd vid programmering är syntax highlighting, integrerade hjälpsystem och kodkomplettering. Det finns dock ingen känd editor som ger stöd vid användning av kontraktsprogrammering.

I denna uppsats kommer en prototyp för kontraktprogrammeringstöd i programspråket C++ att tas fram. Den editor som kommer att användas som bas för prototypen är GNU Emacs. Prototypen skall bland annat ge stöd under programmering genom att visa kontraktsinformation, genom möjlighet till infogning av kontrakt samt genom analys av utförda kontraktskontroller. I Figur 1.1 illustreras grundtanken med den prototyp som skall tas fram.



Figur 1.1: Översikt av prototyp.

Uppsatsen behandlar även kontraktsprogrammering och en del av dess aspekter i samband med utvecklingsarbete.

För att ge en överblick över uppsatsens innehåll ges här en kort beskrivning av dess disposition. Efter detta inledande kapitel kommer ett bakgrundskapitel där uppgiften beskrivs och motiveras mer detaljerat. I kapitlet tas även tidigare arbete och befintliga semantiska verktyg upp för att ge en bakgrund till resterande kapitel. Efter bakgrundskapitlet kommer ett kapitel som ger en grundlig beskrivning av kontraktsprogrammering. I samband med denna beskrivning behandlas även betydelsen av att använda kontrakt för att beskriva mjukvaras semantik. I det fjärde kapitlet behandlas framtagningen av prototypen. Först i detta kapitel beskrivs GNU Emacs, som är den editor prototypen skall användas med, mer ingående. Sedan beskrivs prototypens funktionalitet detaljerat samt vilka lösningar som har arbetats fram. Efter kapitel fyra, som beskriver framtagningen av prototypen, ges i det femte kapitlet en

utvärdering där resultatet presenteras och utvärderas. GNU Emacs, som är hemmiljön för prototypen, utvärderas även i detta kapitel. I det sjätte och sista kapitlet sammanfattas uppsatsen och slutsatser presenteras. Det ges också förslag och tankar för eventuellt framtida arbete.

2 Bakgrund

Detta kapitel avser att ge en bakgrund till arbetet. Inledningsvis beskrivs anledningen till att utveckla en semantisk editor vilket följs av två avsnitt som behandlar arbetets syfte och mål. Sedan följer ett avsnitt som översiktligt beskriver relationen mellan ett programs semantik och kontrakt som är en central del i arbetet. Efterföljande avsnitt diskuterar befintliga verktyg som har likheter med detta arbete och det avslutande avsnittet beskriver textredigeraren GNU Emacs som är den grundläggande plattformen för arbetet.

2.1 Tidigare arbete

Under flera år har det på Karlstads universitet forskats kring utvecklingsmodeller för att höja kvalitén hos mjukvara genom att förbättra den semantiska beskrivningen. Dessa modeller bygger på användning av kontrakt för att specificera mjukvarumodulers semantiska egenskaper.

Vid datavetenskapsutbildningen på Karlstads universitet används kontraktsprogrammering som en del av programmeringsundervisningen. För att underlätta den praktiska användningen av kontraktsprogrammering vid undervisningen har det vid två tidigare tillfällen genomförts D-uppsatser [2][6] som behandlat utvecklingen av utvecklingsmiljöer med stöd för kontraktsprogrammering. Dessa uppsatser använde sig av den befintliga utvecklingsmiljön NetBeans IDE [24] som modifierades för att underlätta kontraktsprogrammering genom att utnyttja befintlig funktionalitet. På grund av att NetBeans IDE är en utvecklingsmiljö för Java, som inte är det primära undervisningsspråket på universitetet, har användningen begränsats. Det har också visat sig att den tänkta funktionaliteten hos utvecklingsmiljön medfört att prestandan blivit otillräcklig [6].

Denna uppsats kommer att behandla utvecklingen av en prototyp där hänsyn tas till begränsningar i den tidigare utvecklingsmiljön.

2.2 Syfte

Syftet med uppsatsen är att undersöka vilka möjligheter som finns till att utöka den befintliga textredigeraren GNU Emacs så att den ger stöd vid kontraktsprogrammering. En väl fungerande textredigerare med stöd för kontraktsprogrammering kan användas för att

underlätta undervisningen av programmering med kontrakt. Stödet skall i denna uppsats vara anpassat till programspråket C++ men tanken är att stöd för andra språk skall vara enkelt att lägga till.

Stödet som skall ges är i första hand visning av kontrakt vid programmering. Visningen bör ske genom att en textruta visas vid lämpligt tillfälle. Det skall också vara möjligt att infoga tester av de aktuella förvillkoren i källkoden. Det kommer också att undersökas vilka möjligheter som finns till att analysera programkod och avgöra om kontrakt har uppfyllts eller ej.

För att tillägget skall vara möjlig att använda i praktiken är det även viktigt att uppnå en tillfredställande prestanda.

2.3 Mål

Målet med uppsatsen är att utveckla en prototyp av en tilläggsmodul som tillhandahåller stöd för kontraktsprogrammering i textredigeraren GNU Emacs. Tilläggsmodulen skall erbjuda följande stöd:

- Visning av kontrakt
- Infogning av kontrakt i kod
- Visning av eventuella kontraktsbrott

2.4 Semantik och kontrakt

En programmoduls syntax beskriver hur modulen ser ut medan dess semantik beskriver vad den gör. För att definiera syntax finns det formella system som underlättar möjligheten att kontrollera korrektheten i en moduls uppbyggnad. Att definiera en programmoduls semantik är mycket svårare då det inte existerar något enkelt tillämpbart formellt system för semantikbeskrivningar. För att beskriva semantiken hos en modul används därför ofta informella beskrivningar med hjälp av naturliga språk.[1]

För att ett system skall fungera korrekt är det viktigt att semantiken hos modulerna som utgör delsystemen upprätthålls vilket kallas för semantisk integritet.[3]

En metod för att beskriva modulers semantik och underlätta att den semantiska integriteten upprätthålls är kontraktsprogrammering. Programmering med kontrakt innebär att programmodulers användningsvillkor specificeras i kontraktsform. I kontraktet specificeras

vad en klient måste uppfylla för att använda en leverantörsmodul samt vad leverantörsmodulen lovar att utföra vid ett korrekt anrop. Genom att använda kontrakt ges moduler en semantisk beskrivning som är integrerad med källkoden och om kontrakten följs upprätthålls den semantiska integriteten.

Ett programspråk som har inbyggt stöd för kontraktsprogrammering är Eiffel [9] där kontrakt definieras med hjälp av nyckelord. Kontrakten i Eiffel kontrolleras av körtidsmiljön vilket leder till att kontraktsbrott enkelt upptäcks utan att explicit kontraktskontroll måste göras. Det finns, med undantag för Eiffel, inget inbyggt stöd för kontraktsprogrammering i dagens programspråk.

2.5 Befintliga verktyg

Det finns idag inget utbrett utbud av verktyg för att underlätta användning av kontrakt vid programmering. Detta medför att programmeraren själv måste söka i källkod efter de aktuella kontrakten.

Vi har funnit två typer av verktyg som angränsar till vårt arbete och innehåller idéer som kan vara intressanta att betrakta. Den ena typen rör användandet av kontrakt men ger inget stöd vid programmering. Några exempel på verktyg ur denna kategori är iContract [7], jContractor [20] och Jass [21]. Den andra typen ger ett mer generellt stöd genom syntaktisk och semantisk analys av programkod, men detta sker på programspråksnivå. Exempel på verktyg ur denna kategori är Harmonia [19] och Schütz Semantic Editor [25].

För att exemplifiera de två typerna har verktygen iContract och Harmonia valts ut och presenteras nedan.

2.5.1 iContract

iContract är ett verktyg som ger liknande stöd för kontraktsprogrammering i Java som det stöd som finns inbyggt i Eiffel.[7]

iContract är en preprocessor vilket innebär att verktyget analyserar javakod och genererar nya källkodsfiler med kod för kontraktshanteringen. Den genererade koden kompileras sedan med hjälp av en vanlig javakompilator vilket innebär att användandet av iContract resulterar i kod som uppfyller Java-standarden. När ett kontrakt bryts under körning av ett program kastas ett undantag med information om det aktuella kontraktsbrottet. Eftersom dessa undantag inte hanteras kommer ett kontraktsbrott att leda till att programmet avslutas.[7]

Vid användning av iContract använder man sig av Javas JavaDoc-system [22] för att specificera för- och eftervillkor samt invarianter. JavaDoc är ett system för att integrera

dokumentation direkt i källkoden vilket görs genom att använda en speciell kommentarsnotation som kan sammanställas till HTML-format med hjälp av verktyget JavaDoc. Kommentarsnotationen innebär att en kommentar som beskriver en klass, metod eller ett attribut inleds med dubbla asterisker för att den skall kunna identifieras som en JavaDoc-kommentar. I kommentaren används sedan text och taggar för att beskriva till exempel en metods parametrar och returvärde. Ett exempel på en standard JavaDoc-kommentar visas nedan.[22]

```
/**
* Adds object to top of stack.
*
* @param o The object to add on top of stack.
*/
public void push(Object o) {

    ...

};
```

För att definiera kontrakt används taggar som är specificerade av iContract. Dessa taggar är @pre, @post samt @invariant och används på liknande sätt som JavaDocs standardtaggar. Ett exempel på hur iContracts taggar används i JavaDoc-kommentarer visas nedan.

```
/**
* Adds object to top of stack.
*
* @pre !isFull()
* @post top() == o
* @post size() == size()@pre + 1
*/
public void push(Object o) {

    ...

};


/**
* The size of the stack is always greater than or equal to zero.
*
* @invariant size() >= 0
*
*/
```

Genom att använda taggsystemet inkluderas även kontrakten i dokumentationen då verktyget JavaDoc används.

2.5.2 Harmonia

Harmonia [19] är ett ramverk för att skapa språkmedvetna verktyg som utvecklas vid University of California at Berkeley. Harmonia är baserat på mer än 16 års forskning av språkbaserade programmeringsverktyg och bygger på två tidigare prototyper. Ramverket Harmonia har som mål att möjliggöra snabb utveckling av program som använder sig av analys och trädrepresentation av källkod. Exempel på sådana program är källkodsredigerare och dikteringssystem för programmering.[4]

En tillämpning av Harmonia-ramverket som utvecklats av Harmonia-projektet är en tilläggsmodul för textredigeraren Emacs. Tilläggsmodulen tillhandahåller bland annat stöd för automatisk indentering, syntax-highlighting och struktursökning. Den viktigaste funktionaliteten för tilläggsmodulen är kodanalys som är indelad i tre faser: lexikalanalys, syntaxanalys och semantisk analys. Den lexikala analysen grupperar tecken i enheter och används främst vid syntax-highlighting. Syntaxanalyseringen tolkar kodens struktur och bygger en trädrepresentation som används för att kontrollera kodens syntaktiska korrekthet. Vissa språkkonstruktioner, som till exempel funktionsnamn, kräver syntaxanalysering för att syntax-highlighting skall vara möjlig. Den tredje fasen är semantisk analys som har till uppgift att fastställa namn och typer på programmets konstruktioner samt att kontrollera programmets semantiska korrekthet. En kontroll av semantisk korrekthet som Harmonia utför är hur typer används i samband med uttryck. Ett exempel på ett felaktigt uttryck är multiplikation av strängar. [18]

2.5.3 Summering

iContract använder sig av en kommentarsnotation för att integrera kontrakt i källkoden. Detta är en bra metod för att mjukvarumässigt kunna identifiera kontrakt och kan därför bli aktuellt i vårt arbete.

Emacs-tillägget Harmonia utför syntaxanalys under programmering vilket eventuellt kan bli aktuellt för oss i samband med kontraktbrottsdetektering. Det var även intressant att se en tillämpning av ett språkverktyg utvecklat för Emacs.

2.6 GNU Emacs

GNU Emacs är en avancerad textredigerare som bland annat tillhandahåller stöd för automatisk indentering av källkod och syntax highlighting. Emacs har också funktioner för att

hantera text skrivet i naturliga språk där ord, meningar, stycken och sidor kan hanteras som enheter.

Emacs är skrivet i programspråket Emacs Lisp som utvecklats speciellt för Emacs. Större delen av Emacs är skrivet i detta språk men vissa delar är implementerade i programspråket C för att öka prestandan. Emacs Lisp är ett komplett språk som inte är bundet till Emacs utan kan användas tillsammans med en fristående interpretator. Emacs Lisp är dock anpassat med textredigeringsfunktionalitet vilket gör språket mycket användbart när det används tillsammans med Emacs. Emacs Lisp är ett funktionellt språk som har samma ursprung som det mer kända Common Lisp men de båda språken skiljer sig betydligt i vissa avseenden. Skillnaderna rör framför allt minneshanteringen där Emacs Lisp är designat för att minska minnesanvändningen i Emacs.[8]

Det som gör Emacs till ett kraftigt verktyg för texthantering är dess utbyggnads- och inställningsmöjligheter. Emacs har en inbyggd Emacs Lisp interpretator som gör det enkelt att skriva tilläggsprogram som kan användas i systemet direkt under körning. Det är också möjligt att modifiera befintlig funktionalitet genom att omdefiniera existerande funktioner eller ändra värden på systemvariabler.[13]

Emacs är självdokumenterande vilket innebär att dokumentationen för grundfunktioner och tilläggsprogram integreras med källkoden. Ett exempel på sådan dokumentation ses i Emacs Lisp koden nedan.

```
(defun rectangle-area (length width)
  "Calculates the area of a rectangle with
  sides LENGTH and WIDTH."
  ;; calculate the area to return
  (* length width))
```

I funktionsdefinitionen ovan ingår en dokumentationssträng som beskriver funktionen. Denna sträng visas sedan när information om funktionen slås upp i Emacs hjälpsystem.

Dokumentationen för de olika funktionerna och variablerna som för tillfället finns definierade i Emacs kan visas med ett enkelt kommando. Att dokumentationen är lättillgänglig underlättar utvecklingsarbete med utökningar och modifieringar av Emacs och förenklar inställningar i programmet.[13]

3 Kontraktsprogrammering

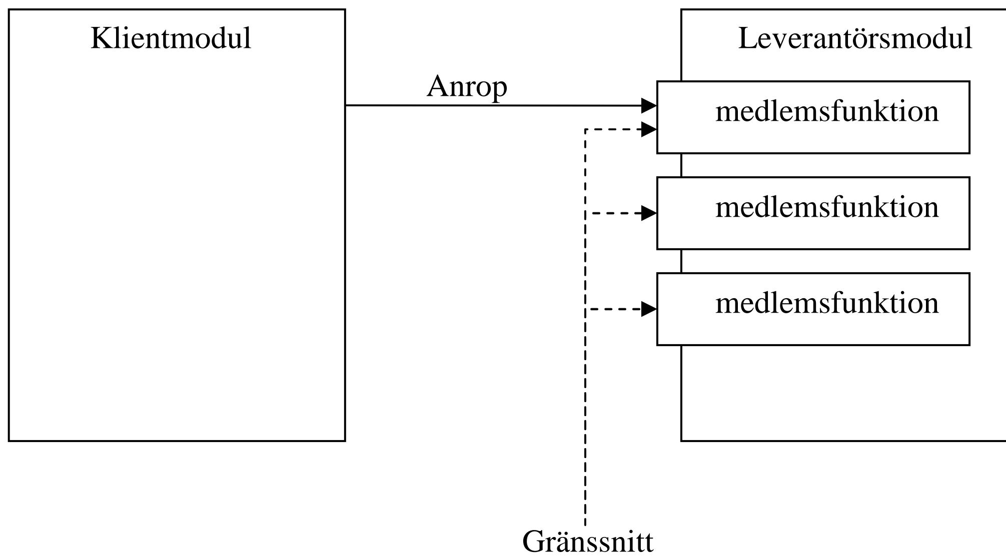
Detta kapitel ämnar ge en insikt i kontraktsprogrammering samt att behandla kontrakt med avseende på design av mjukvara. För att kunna följa uppsatsens efterkommande kapitel med behållning är det bra att ha kunskaper om kontraktsprogrammering då uppsatsen bygger på utveckling av en semantisk editor vilket innebär stöd för kontrakt. Kapitlet motiverar användandet av kontrakt samt beskriver grunderna i kontraktsprogrammering. I kapitlet behandlas kontrakt och design och även andra aspekter som är värda att beakta vid användandet av kontrakt tas upp.

3.1 Inledning

För att få bakgrund och med det förståelse för betydelsen av användning av kontrakt ges i detta kapitel både en beskrivning av de grundläggande delarna i kontraktsprogrammering men även djupare aspekter berörs. Anledningen till att kontrakt beskrivs är att detta är vad uppsatsen bygger på. Grunderna ges genom en beskrivning av semantik och kontrakt följd av beskrivningar av de olika kontraktsdelarna i form av förvillkor, eftervillkor samt invarianter.

Det är givetvis möjligt att använda kontrakt på olika sätt vilket resulterar i olika resultat. Hur kontrakt används beror till stor del på vad syftet med dess användande är. En annan aspekt är vilken kunskap utvecklaren av mjukvaran har vilket kan påverka användandet. En utvecklare som är kunnig och erfaren inom området kan använda kontrakt under hela utvecklingsprocessen medan en mindre kunnig endast använder kontrakt i syfte att dokumentera. Vilka konsekvenserna blir av att införa kontrakt vid olika tillfällen i utvecklingen av mjukvara behandlas därför i detta kapitel.

3.2 Terminologi



Figur 3.1: Illustration av använda termer.

3.2.1 Modul

En modul är en sammanhängande mängd av funktioner och variabler. Vissa moduler kan bestå av enbart funktioner, som till exempel funktionsbibliotek, medan andra består av bara variabler, till exempel records. Normalfallet är dock att en modul innehåller både funktioner och variabler som till exempel klasser. En moduls gränssnitt kan utgöras av funktioner eller variabler som är åtkomliga utifrån av till exempel ytterligare en modul.

3.2.2 Klient

Med en klient menas en användare av en modul. Klienten kan vara ytterligare en modul eller annan del av programmet. Klienten har ingenting med modulens utformning att göra utan använder den bara.

3.2.3 Leverantör

Med leverantör menas den modul som används av en klient. Leverantören specificerar kontrakten.

3.2.4 Funktion

Med en funktion menas en anropningsbar enhet i en modul. Detta skulle till exempel kunna vara en medlemsfunktion i en klass eller en funktion i ett bibliotek.

3.3 Semantik och kontrakt

Semantik sägs ofta ange vad något betyder eller har för innebörd vilket även gäller för en moduls semantik. Den semantiska beskrivningen av en modul anger alltså en moduls innebörd eller vad den utför, inte hur något utförs.

För att specificera en moduls semantik kan kontrakt användas för att beskriva vad som utförs utan att klienten skall behöva titta på hur modulen implementerats. Genom att beskåda denna semantiska beskrivning är det därför möjligt att snabbt få en uppfattning av vad modulen utför och vilka begränsningar som finns.

Semantiken i ett program är beroende av att semantiken för varje modul upprätthålls. Skulle den semantiska betydelsen brytas hos en medlemsfunktions i en modul kan detta påverka hela systemet eftersom en modul kan vara beroende av en annan modul eller att en invariant är bruten.

Om funktioners semantik dokumenteras med hjälp av endast löpande text är det stor risk att den semantiska beskrivningen blir otydlig och olika tolkningar kan göras. En beskrivning i löpande text har också en tendens att bli onödigt lång och innehålla redundant information vilket gör beskrivningens innebörd svårtillgänglig.

Även om kontrakt används är det viktigt att ge en kort textbeskrivning av funktionen som talar om vad den gör. Beskrivningen bör ge en överblick av vad funktionen gör för att underlätta för användaren medan för- och eftervillkoren tydligt beskriver hur den skall användas och dess resultat. Om detta uppfylls minimeras risken för att funktionen används på ett felaktigt sätt.

Nedan följer ett exempel på två funktionsbeskrivningar, en i ren text och en i kontraktsform:

```
/*
 * Calculates the trigonometric tangent of the angle object.
 * The angle object must have been assigned a value (which
 * can be checked with the hasValue( ) member function) and
 * its absolute value have to be less than PI/2. The value
 * of the passed argument must be either RADIAN or DEGREE
 * and specifies the unit of the calculated tangent value.
 *
 */
```

```
float angle::tan( int u )
{
  ...
}
```

```
/*
 * Calculates the trigonometric tangent of the angle object.
 *
 * @Pre    hasValue( )
 *         abs( ) < (PI / 2)
 *         u == RADIAN || u == DEGREE
 *
 * @Post   The trigonometric tangent of the angle object, in
 *         the unit determinded by argument 'u', is returned.
 */
float angle::tan( int u )
{
  ...
}
```

3.4 Ett kontrakts delar

3.4.1 Förvillkor

Förvillkor är den del av kontraktet som anger vad som måste vara uppfyllt för att funktionen skall garantera ett visst resultat. Inga kontroller sker i funktionen för att säkerställa att förvillkoret är uppfyllt utan detta förutsätts vara kontrollerat innan funktionen anropas. Om förvillkoret inte är uppfyllt tar funktionen inget ansvar för resultat utan kan utföra odefinierade operationer och till och med terminera. Ett förvillkor kan skrivas i klartext eller som ett exekverbart uttryck. Ett förvillkor utformas så att det till exempel anger vilka värden på inparametrar som är tillåtna eller vilket tillstånd objektet måste befinna sig i. För att underlätta utformningen av kontrakt är ofta predikatfunktioner, som returnerar sant eller falskt beroende på objekts tillstånd, ett bra sätt att förenkla utformningen av förvillkor samt underlätta vid kontroll av förvillkor hos klienter.

Genom att använda förvillkor på detta sätt kan partitiella funktioner skapas. Med partitiella funktioner menas funktioner som inte är definierade för alla typer av inargument. Det är alltså domänen av argument för vilka funktionen är definierad som anges med förvillkor. Då ett förvillkor tillåter alla inparametrar till funktionen kallas den total.[10]

3.4.2 Eftervillkor

Ett eftervillkor anger vad en funktion förbinder sig att utföra vid ett korrekt anrop. Med ett korrekt anrop menas att förvillkoren är uppfyllda vid anrop av funktionen. I detta fall ligger ansvaret helt hos funktionen och klienten kan lita på att eftervillkoret kommer att uppfyllas. Eftervillkor kan skrivas både som klartext och i kodformat. Då ett eftervillkor skrivs i kodformat är det inte för att det skall vara exekverbart utan för att det kan vara ett bra sätt att beskriva vad funktionen utför. Eftervillkoret skall inte vara bundet till funktionens implementation utan enbart ange vilket resultat den garanterar. Resultatet kan vara ett returvärde eller att ett tillstånd förändrats.

3.4.3 Invarianter

En invariant anger ett villkor som alltid måste vara uppfyllt av en modul. Invarianten bör initieras till ett tillåtet värde i konstruktorn och efter detta skall den alltid vara uppfylld. En invariant kan tillåtas att brytas inuti en funktion men den måste då alltid återställas innan funktionen nått sitt slut. Då en invariant är bruten är det mycket viktigt att detta inte resulterar i några sidoeffekter. Invarianten skall inte ange någonting om hur implementationen ser ut utan bara ange vilka villkor den måste uppfylla.

En typ av invariant är klassinvarianter. Med klassinvarianter menas värden för en klass som måste upprätthållas av alla metoder. Då förvillkoret uppfylls förpliktar sig funktionen att se till att klassinvarianten upprätthållas även vid dess slut. [10]

En annan typ av invariant är implementationsinvariant. Med detta menas den invariant som gäller för en specifik representation av en abstrakt datatyp. Även denna invariant måste upprätthållas av samtliga metoder. Ett exempel på en sådan invariant kan vara för en stack som är array-baserad och därmed har ett maxvärde som måste hållas. Denna invariant kommer inte från den abstrakta datatypen utan är implementationsspecifik.

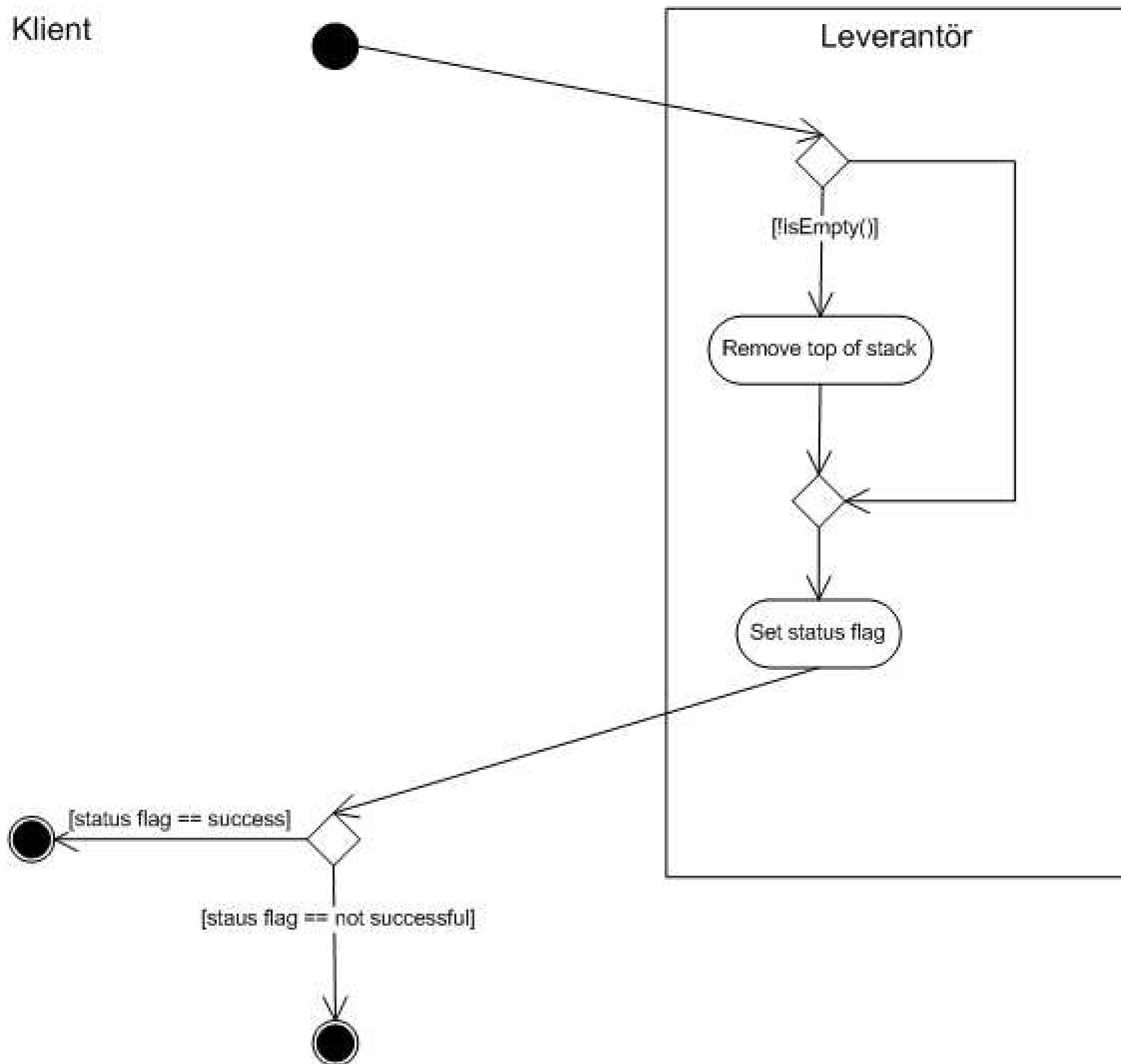
Alltid då en funktion anropas kan invarianten antas vara uppfylld och därmed kan den även antas vara uppfylld efter att funktionen avslutats om förvillkoret uppfyllts.

3.5 Mjukvarukvalitet

Förutom att ge en effektiv notation för att beskriva programmodulers semantik står kontraktsprogrammering för ett tankesätt som står i kontrast till användningen av defensiv programmering. Det gemensamma målet hos kontraktsprogrammering och defensiv programmering är hög mjukvarukvalitet men hur denna kvalitet skall uppnås skiljer sig mellan de båda sätten.

Defensiv programmering [5] är en programmeringsteknik som används för att detektera och hantera onormala och felaktiga tillstånd i mjukvarumoduler för att göra varje modul i ett mjukvarusystem robust. Genom att aldrig lita på klienters användning av en funktion undviks potentiella fel genom att varje funktion kontrollerar och säkerställer att felaktig användning inte resulterar i fel hos mjukvaran. Alla tänkbara källor till fel, även till synes osannolika källor, måste därför kontrolleras och hanteras på ett för leverantören felsäkert sätt.

Vid användning av defensiv programmering är det tänkt att målet, det vill säga hög kvalitet, skall uppnås genom att eliminera alla tänkbara källor till fel. Detta görs genom att till exempel kontrollera alla inparametrar till en funktion för att säkerställa att de tillhör den aktuella domänen eller att kontrollera det aktuella tillståndet hos en modul för att avgöra om anropet kan resultera i ett korrekt resultat. Eftersom utrymme för felaktiga anrop ges och det är den anropade funktionens ansvar att hantera sådana fel så utökas varje funktion till att inte bara kunna utföra sin tänkta uppgift utan också innehålla felkontroller och felhantering. Innebörden av detta är att modulers komplexitet kan öka betydligt. Funktioner som är robusta genom att alla möjliga felkällor kontrolleras måste också använda sig av en mekanism för att kunna tala om för dess klient om anropet lyckades, om något gick fel och i så fall vad som gick fel. Detta kan göras på ett antal olika sätt, till exempel med hjälp av undantag, flaggor, returvärde eller användning av pass-by-reference för att använda utargument. Gemensamt för dessa metoder är att de ökar funktioners semantiska komplexitet genom att dess resultat blir mer svårtolkat vilket resulterar i ökad komplexitet i klientens kod då resultatet måste tolkas. Ett exempel som illustrerar resultatet av defensiv programmering ses i Figur 3.2.

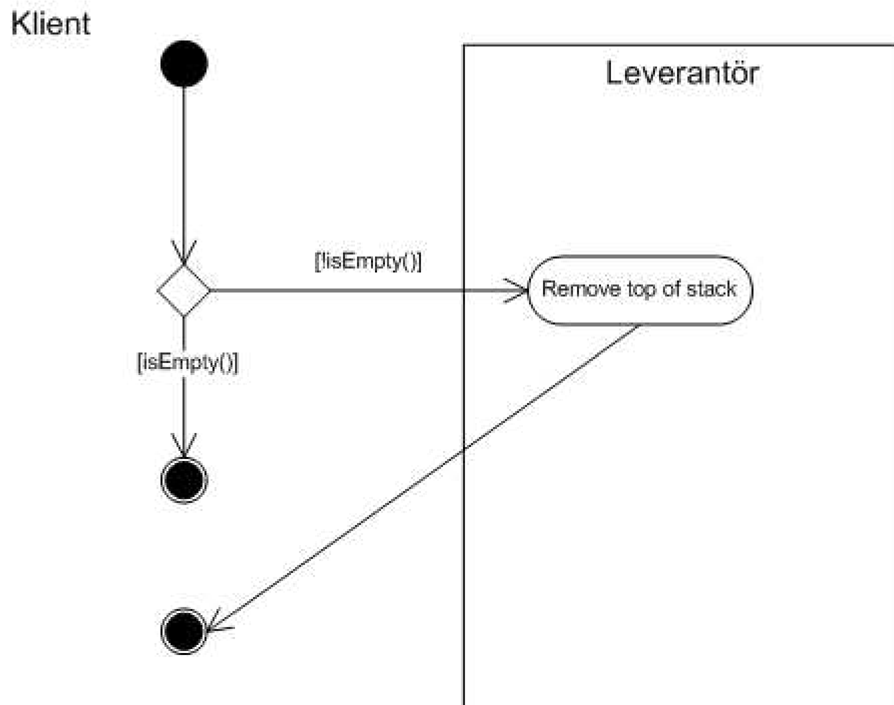


Figur 3.2: Användning av defensiv programmering för att säkerställa korrekthet.

Kontraktsprogrammering har ett annat sätt att nå målet om hög mjukvarukvalitet än det som defensiv programmering använder sig av. Då defensiv programmering används för att låta alla delar i varje leverantörsmodul själva ansvara för korrektheten använder sig kontraktsprogrammering av kontrakt som specificerar vilka åtaganden klienten och leverantören har vid anrop. Genom att dela upp åtagandena kan leverantörsmodulen koncentrera sig på sin primära uppgift vilket resulterar i mindre komplex kod då felkontroller och felhantering i stor utsträckning inte görs i leverantörskoden.

Även klientdelen blir enklare då resultatet av ett anrop vid uppfyllt förvillkor är beskrivet i eftervillkoret.

I Figur 3.3 visas motsvarande anrop som i Figur 3.2 men där komplexitet, i form av extra kontroll, som har införts på grund av defensiv programmering utelämnats.



Figur 3.3: Användning av kontrakt för att förenkla moduler.

Angripssättet som kontraktsprogrammering använder sig av för att uppnå en hög mjukvarukvalitet är baserat på teorin att komplexitet är den största orsaken till fel i mjukvara [10]. Genom att reducera antalet kontroller i leverantörskod minskas dess komplexitet. Även klientens komplexitet minskas genom användandet av kontrakt som klienten kan använda för att avgöra om ett anrop är tillåtet, detta görs ofta med hjälp av predikatfunktioner som tillhandahålls av leverantören och har som syfte att användas för att kontrollera om förvillkor uppfylls.

Genom att minska komplexiteten i ett mjukvarusystem med hjälp av kontrakts-programmering minskar också risken för fel. Med hjälp av defensiv programmering tillförs programkod för att förhindra att fel uppstår och ser man till enskilda moduler kan det tyckas att denna kod är hanterbar och inte belastar modulen nämnvärt. Ser man däremot till ett helt system med ett stort antal moduler ger den extra programkod i varje modul ett mera komplext system och risken för att fel uppstår ökar.

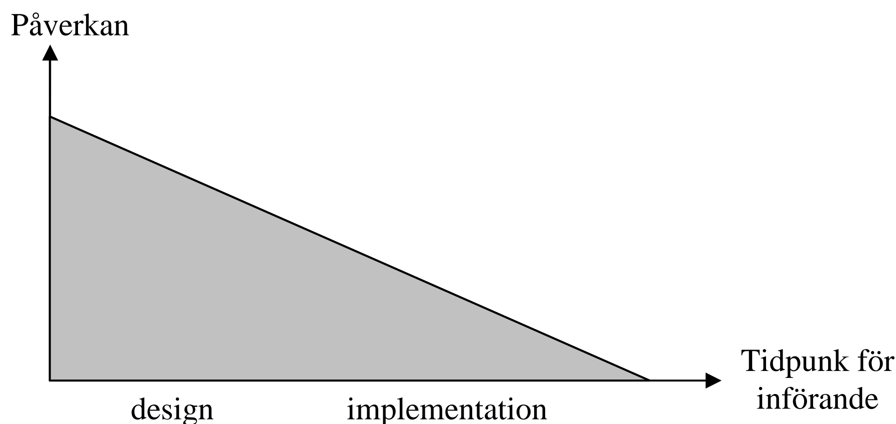
3.6 Användning av kontrakt vid mjukvaruutveckling

Kontrakt kan användas vid ett flertal tidpunkter under programutveckling. Ett sätt är att kontrakt används vid ett tidigt skede i utvecklingsarbetet, innan implementering påbörjas. Ett annat sätt att använda sig av kontrakt är att införa dessa under implementation. Det är också möjligt att använda kontrakt i dokumentationssyfte efter implementation.

Vid användningen av kontrakt efter implementationen av en modul blir kontraktens syfte att beskriva en befintlig semantik utifrån den färdiga implementationen. Att använda kontrakt på detta sätt resulterar inte i mer pålitlig kod då koden redan är skriven och det är risk för att defensiv programmering har använts. Även om kontrakten inte påverkar kodkvaliteten då de införs efter implementationen så fyller de ett syfte genom att ge en semantisk beskrivning av en modul som kan underlätta för modulens klienter samt vid underhåll.

När kontrakt införs i samband med implementation minskas risken för defensiv programmering påtagligt jämfört med när kontrakt införs efter implementationen. Att införa kontrakt i samband med implementation innebär att kontrakt identifieras och skrivs ner strax före, under samt strax efter kodning. Genom att använda för- och eftervillkor för att minska antalet kontroller i leverantörsmodulen minskas modulens komplexitet vilket minskar risken för fel. Problemet med att införa kontrakt först vid implementationsfasen är att parallell utveckling av moduler försvåras vilket beror på att modulens gränssnitt inte har en komplett beskrivning innan dess kontrakt har definierats och det blir därför svårare att utveckla klientmoduler parallellt. Då gränssnitten för alla klientmoduler på förhand är specificerade kan utveckling ske helt separat för att senare sättas ihop och i dessa fall är det även viktigt att alla moduler har en specificerad semantik. Om så inte är fallet finns risk att det sammansatta programmet inte fungerar som avsett på grund av att en modul i programmet används på ett felaktigt sätt.

I Figur 3.4 illustreras hur kontrakt har möjlighet att påverka kodkvalitet med avseende på vid vilken tidpunkt i utvecklingsarbetet de införs.



Figur 3.4: Illustration av kontrakts möjliga påverkan.

Vid genomförandet av en utvecklingsprocess finns det ingen anledning att vänta med att införa kontrakt till implementationsfasen beroende på att modulens semantik redan existerar i utvecklarnas tänkta lösning. Om semantiken i ett tidigt skede skrivs ner i form av kontrakt underlättas utveckling av klientmoduler och risken för missförstånd minskas.

För att kontraktsprogrammeringskonceptet skall vara så kraftfullt som möjligt bör kontrakt införas i ett tidigt skede i utvecklingsarbetet. Om kontrakt används redan från början, när de första modulerna tar form, så kommer kontrakten att bli en del av processen och mjukvarudelarna får en tydlig semantisk beskrivning genom hela utvecklingsarbetet. Det kan också tyckas vara det mest lämpliga sättet att använda kontrakt på då utvecklarna redan vid designfasen bör ha en klar bild av vilken mening de olika modulerna har.

Under designfasen av en utvecklingsprocess skapas den grundläggande arkitekturen hos ett system. Systemets olika delar kan under design beskrivas med en hög abstraktionsnivå för att resultera i ett elegant och väl fungerande slutsystem [10]. Detta kan göras med hjälp av abstrakta datatyper (ADT) men i praktiken används inte ADT:er i större utsträckning. Om de grundläggande modulerna i den tänkta implementationen är klasser så används abstrakta klasser, som är implementationer av abstrakta datatyper, för att beskriva olika delar i ett system. De abstrakta klasserna som används under design saknar i stor utsträckning implementering (av metoder etc.) och har därför en nära relation med den ADT den implementerar.

Det grundläggande kontraktet för en fullständigt implementerad klass eller en abstrakt klass härstammar från den ADT klassen implementerar. Förvillkor är en del av specifikationen av en ADT och används för att beskriva dess partitiella funktioner. Eftervillkor härstammar ur de axiom som beskriver resultaten av en ADTs funktioner och även klassinvarianten har sitt ursprung i axiomen. [10]

Även om inte abstrakta datatyper används som den första abstrakta beskrivningen av ett system så används abstrakta klasser som har en nära relation till en abstrakt datatyp. Detta gör att kontrakt kan användas redan från början för att beskriva de semantiska aspekterna och allt eftersom abstrakta klasser blir mer och mer konkreta under utvecklingsprocessen kan kontrakten också utvecklas för att bibehålla en god semantisk beskrivning. Då kontrakten ger en semantisk beskrivning genom hela utvecklingsprocessen och är en naturlig del i beskrivningen av moduler minskar risken för att fel uppstår och kvaliteten hos den utvecklade mjukvaran kan förbättras.

3.7 Sammanfattning

I kapitlet har metoden att använda kontrakt behandlats, dels genom en introduktion där olika begrepp har beskrivits och dels genom att resonemang om kontrakts inverkan vid eller efter kodning.

I avsnittet med begrepp har den terminologi som använts vidare i kapitlet beskrivits och med detta som underlag har sedan grundläggande begrepp inom kontraktsprogrammering beskrivits. Dessa begrepp är centrala inom kontraktsprogrammering och är förutom för att kunna följa uppsatsen även till nytta för den läsare som önskar att studera ämnet kontraktsprogrammering på en djupare nivå.

Då begreppen är beskrivna följer resonemang angående hur kontrakt inverkar på kodkvalitet samt hur det bör användas för att få bästa möjliga resultat. Eftersom kontrakt kan användas på flera olika sätt har användningen av kontrakt utretts samt fördelar och nackdelar presenterats. Sammanfattningsvis kan det konstateras att det är önskvärt att använda kontrakt så tidigt som möjligt i utvecklingsprocessen för att nå en så hög kodkvalitet som möjligt.

4 Framtagning av prototyp

Detta kapitel beskriver framtagningen av prototypen av den semantisk editorn. Prototypen utvärderas sedan i efterföljande kapitel.

Kapitlet inleds med en beskrivning av GNU Emacs som är den textredigerare som används vid framtagningen av prototypen. I samband med detta beskrivs Emacs Lisp som är det programspråk som används samt verktyget Collection of Emacs Development Environment Tools (CEDET) [15] som används vid utvecklingen. Efterföljande avsnitt beskriver funktionalitet med utgångspunkt från tidigare utförda arbeten med avseende på den prototyp som skall utvecklas.

Framtagningen av prototypen beskrivs sedan i efterföljande avsnitt genom dess funktionalitet och utformning. Kapitlet avslutas med en sammanfattning.

4.1 Inledning

Efter att i kapitel 3 ha läst om kontrakt och hur de påverkar utvecklingsarbetet är nästa steg att förverkliga dessa teorier. Genom att använda en editor som stöder användning av kontrakt kan det underlätta mycket för att på ett enkelt sätt förse användaren med kontrakt vid den tidpunkt de behövs. För att göra detta utvecklas prototypen av den semantiska editorn och i detta kapitel beskrivs framtagningen av prototypen.

För att utvecklingen av prototypen skall kunna beskrivas på ett bra sätt ges först en beskrivning av den editor som skall användas vid utvecklingen för att ge en större förståelse för hur utvecklingen har fortlöpt. Ytterligare en viktig aspekt att beakta är det programspråk som används. I detta fall används Emacs Lisp och av denna anledning ges en sammanfattning av programspråket. Vid utvecklingen används även ett verktyg som heter CEDET och därför beskrivs detta i ett eget avsnitt. För att ytterligare belysa de förutsättningar som fanns vid starten av utvecklingen finns en beskrivning av det tidigare utförda arbetet där de idéer som vi kan använda oss av plockas fram. Här beskrivs även den funktionalitet som prototypen i första hand skall tillhandahålla.

Efter att förutsättningarna beskrivits följer ett avsnitt som beskriver framtagningen av prototypen. Denna del är kapitlets största avsnitt och är viktig för uppsatsens efterföljande kapitel.

4.2 GNU Emacs

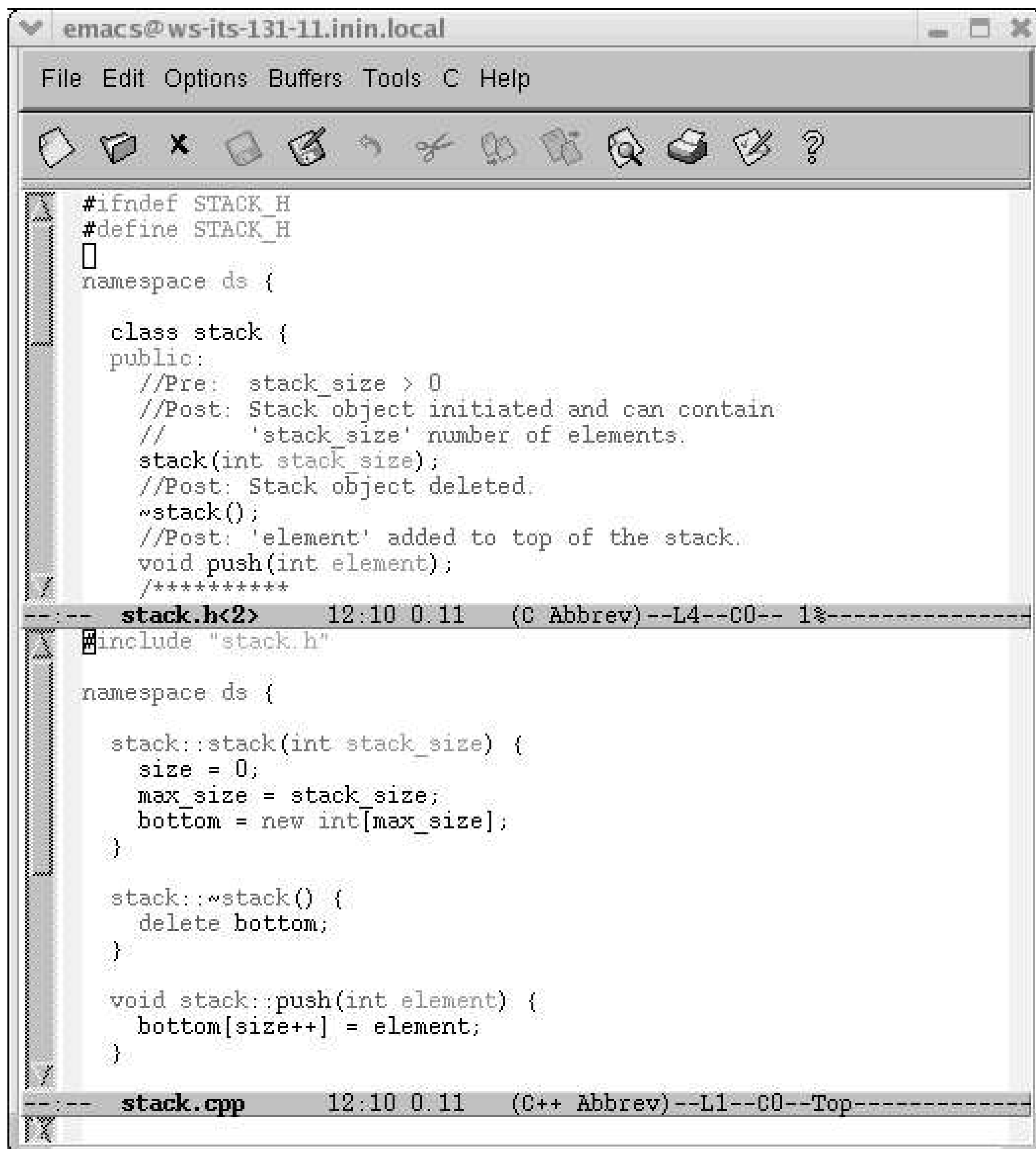
GNU Emacs är en textredigerare som kan anpassas med hjälp av Lisp-dialekten Emacs Lisp. Kärnan i Emacs består av en Emacs Lisp-interpretator med vars hjälp alla operationer i Emacs görs. I detta avsnitt ges en enkel beskrivning av hur Emacs fungerar samt vad som gör Emacs Lisp till ett kraftfullt språk för textredigering.

4.2.1 Textredigeraren Emacs

I Emacs utförs operationer genom så kallade kommandon. Kommandon kan vara grundläggande som till exempel insättning och borttagning av text, öppna och spara fil samt att klippa ur och klistra in text. Det finns även mer avancerade kommandon som till exempel sökning med hjälp av reguljära uttryck, utskrift till skrivare samt exekvering av skalkommando. Ett Emacs-kommando är en interaktiv Emacs Lisp funktion och det är med hjälp av kommandon som allt sker i Emacs. Mer om interaktiva funktioner tas upp i avsnitt 4.2.2.

Emacs använder sig av buffrar för att hantera text. En buffer är ett objekt som kan innehålla redigerbar text och all text som redigeras eller visas i Emacs finns i en buffer. Flera buffrar kan existera samtidigt men endast en buffer är aktiv. Det är i den aktiva buffern redigering och kommandon utförs. När en fil öppnas laddas den in i en buffer där ändringar kan utföras. När ändringar sparas skrivs den aktuella buffern till den fil som buffern är kopplad till. Buffrar är inte alltid bundna till filer utan kan innehålla temporär text som inte skall sparas.

När en buffer är synlig för användaren visas den i ett fönster. Varje fönster visar en buffer men en buffer kan visas i flera fönster. Emacs kan delas upp i godtyckligt antal fönster för att på så vis kunna visa flera olika buffrar samtidigt vilket visas i Figur 4.1.

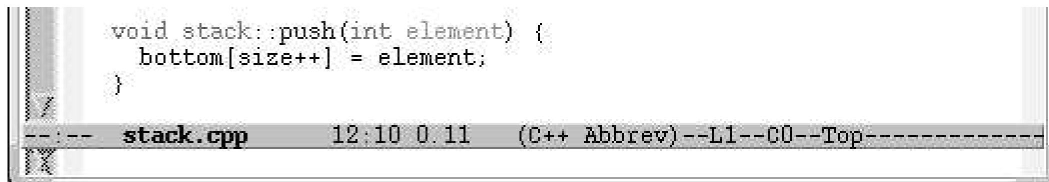


Figur 4.1: Emacs uppdelat i två fönster.

Om en buffer visas i flera fönster samtidigt visas ändringar gjorda i ett fönster i alla andra fönster som visar samma buffer. Varje fönster har en egen uppsättning av positions-information vilket medför att varje fönster kan visa olika delar av den aktuella buffern.

För att hantera hur texten i en buffer skall presenteras samt vilka kommandon som är möjliga används så kallade modes. Modes kan ses som tilläggsmoduler där regler för en buffer definieras. Exempel på regler som ett mode kan definiera är hur indenterad text skall se ut, vilka ord i en buffer som är nyckelord samt vilken tangentkombination som aktiverar ett specifikt kommando. Det finns två olika typer av mode: major och minor mode. Alla buffrar har ett och endast ett major mode. Ett major mode definierar de grundläggande reglerna för den text som finns i en buffer. Exempel på major modes är C++ mode, HTML mode, Emacs

Lisp mode och TeX mode. Minor modes tillför en eller flera egenskaper till en buffer utöver de som redan ingår i bufferns major mode. En buffer kan ha ett godtyckligt antal minor modes och kan aktiveras och avaktiveras efter behov. Exempel på egenskaper som kan styras genom minor modes är visning av syntax highlighting och visning av matchande parenteser. Vilka mode som är aktiva för tillfället kan visas i en så kallad *mode line*, vilket visas i Figur 4.2.

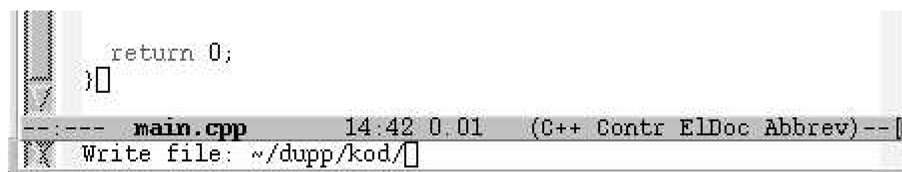


Figur 4.2: Här kan ses att C++ major mode och Abbrev minor mode är aktiva.

4.2.2 Emacs Lisp

Emacs Lisp är en Lisp-dialekt som är framtagen speciellt för utvecklingen av Emacs, som till största delen är skrivet i detta språk. Eftersom språket är framtaget för utvecklingen av en textredigerare finns det ett utbrett stöd för texthantering. Till exempel finns egenskaper för hur en textsträng skall visas i ett fönster inbyggt i varje strängobjekt. Det finns även en datatyp buffer som används i Emacs för att hantera redigerbar text.

I Emacs Lisp finns det en speciell typ av funktioner som heter kommandon. Dessa funktioner är interaktiva och kan anropas från Emacs. Kommandon kan interagera med användaren genom inmatning via en prompt. Det är de interaktiva funktionerna i Emacs som utgör gränssnittet gentemot användaren. Ett exempel på ett kommando ses i Figur 4.3.



Figur 4.3: Kommandot som sparar till fil interagerar med användaren.

4.2.3 Minor Modes

Anpassningen av Emacs till en semantisk editor som ger stöd vid användningen av kontrakt sker genom att utnyttja den befintliga Emacs-standarden för tilläggsprogram. Eftersom

funktionaliteten som skall tillhandahållas inte består av grundläggande beteende för programkodstext, utan är ett tillägg som kan användas vid kontraktsprogrammering, så kommer ett minor mode att implementeras.

Ett minor mode är i vissa fall svårare att implementera än ett major mode. Varje buffer har alltid ett major mode som tillhandahåller de grundläggande egenskaperna för den text buffern innehåller. Detta gör att major modes inte behöver ta hänsyn till andra major modes. De minor modes som eventuellt kan vara aktiva i en enskild buffer eller i alla existerande buffrar (globalt) har större krav på sig. De måste ta hänsyn till det major mode varje buffer har samt alla eventuella minor modes som kan vara aktiva. I praktiken innebär detta att ett minor mode skall kunna aktiveras oberoende av vilka andra major och minor modes som redan är aktiva hos en buffer. Ett minor mode skall också kunna utföra sin uppgift oberoende vilka andra modes som är aktiva om inte modet är konstruerat för att samverka med ett specifikt mode. Är så fallet måste det försäkras att modet är tillgängligt för att aktivering skall vara möjligt.

Grunden vid implementering av ett minor mode är följande:

- En mode-variabel med ett namn som slutar på ”-mode”. Variabeln kan anta värdena sant eller falskt och används för att avgöra om mode:et är på- eller avslaget.
- Ett kommando (interaktiv lisp-funktion) med samma namn som mode-variabeln. Detta kommando har till uppgift att slå av eller på mode:et och uppdatera värdet på mode-variabeln därefter.
- Skall det synas i Emacs *mode line* att mode:et är påslaget skall en teckensträng läggas till i en associativ lista som Emacs använder för att visa vilka modes som för tillfället är aktiva.

4.2.4 CEDET

CEDET [15] står för Collection of Emacs Development Environment Tools och är en samling verktyg som ger stöd för användning och utveckling av avancerade utvecklingsmiljöer i Emacs. Exempel på direkt användbara verktyg som ingår i CEDET är verktyg för code completion, grafisk klassnavigering och diagramsverktyg med stöd för Unified Modeling Language (UML) [26].

CEDET innehåller också verktyg för utveckling av utvecklingsmiljöer i Emacs. Ett sådant verktyg är Semantic, som bland annat innehåller en lexikalanalysator, parsrar för ett antal

programspråk samt en parser-generator. Det ingår också en databas som används för bestående lagring av parse-information.

Vid utvecklingen av prototypen kommer Semantic tillsammans med viss funktionalitet ur databassystemet att användas. Semantic innehåller som standard bland annat en parser för C++ som kommer att utnyttjas och databasen kommer främst att användas för att söka i den symboltabell som parsern skapar och upprätthåller. Den primära uppgiften hos databasen, att lagra information från parsning i en fil på hårddisken, kommer inte att utnyttjas men kan spela en viktig roll i en mer utarbetad version av prototypen.

4.3 Funktionalitet

Tidigare har två D-uppsatser som behandlar utveckling av semantiska editorer skrivits vid Karlstads universitet. I de båda uppsatserna har arbete genomförts för att ta fram en editor som ger stöd för kontraktsprogrammering.

Den första uppsatsen föregicks av en C-uppsats [11] där en befintlig editor modifierades för att kunna visa för- och eftervillkor vid programmering. Denna editor hade dock vissa begränsningar och därför valdes i stället utvecklingsmiljön NetBeans IDE för det fortsatta arbetet. Resultatet av den första D-uppsatsen blev ett stöd för visning och textbaserad infogning av kontrakt i NetBeans IDE. Den andra D-uppsatsen vidareutvecklade det arbete som utförts i den tidigare D-uppsatsen. De viktigaste tilläggen var parsning av exekverbara förvillkor samt kontroll av uppfyllda förvillkor.

Funktionaliteten hos prototypen som tas fram i denna uppsats bygger till största delen på den andra D-uppsatsen men indirekt även på den första. Eftersom denna uppsats inte är en vidareutveckling på det befintliga arbetet och använder sig av en annan plattform kommer funktionaliteten att behöva återimplementeras. Den nya implementationen kommer att vara annorlunda än den tidigare, främst på grund av den stora skillnaden mellan programspråken och de miljöer som skall anpassas. Den funktionalitet som i första hand kommer att ingå i prototypen är:

- Visning av kontrakt för användaren
- Parsning av förvillkor vid kontraktsvisning
- Parsning av förvillkor i leverantörskod
- Infogning av exekverbara förvillkor i kod
- Visning av eventuella kontraktsbrott

Språket som skall ges stöd för kontraktprogrammering i denna uppsats är C++. C++ är ett mycket omfattande språk som även inkluderar stora delar av programspråket C [14].

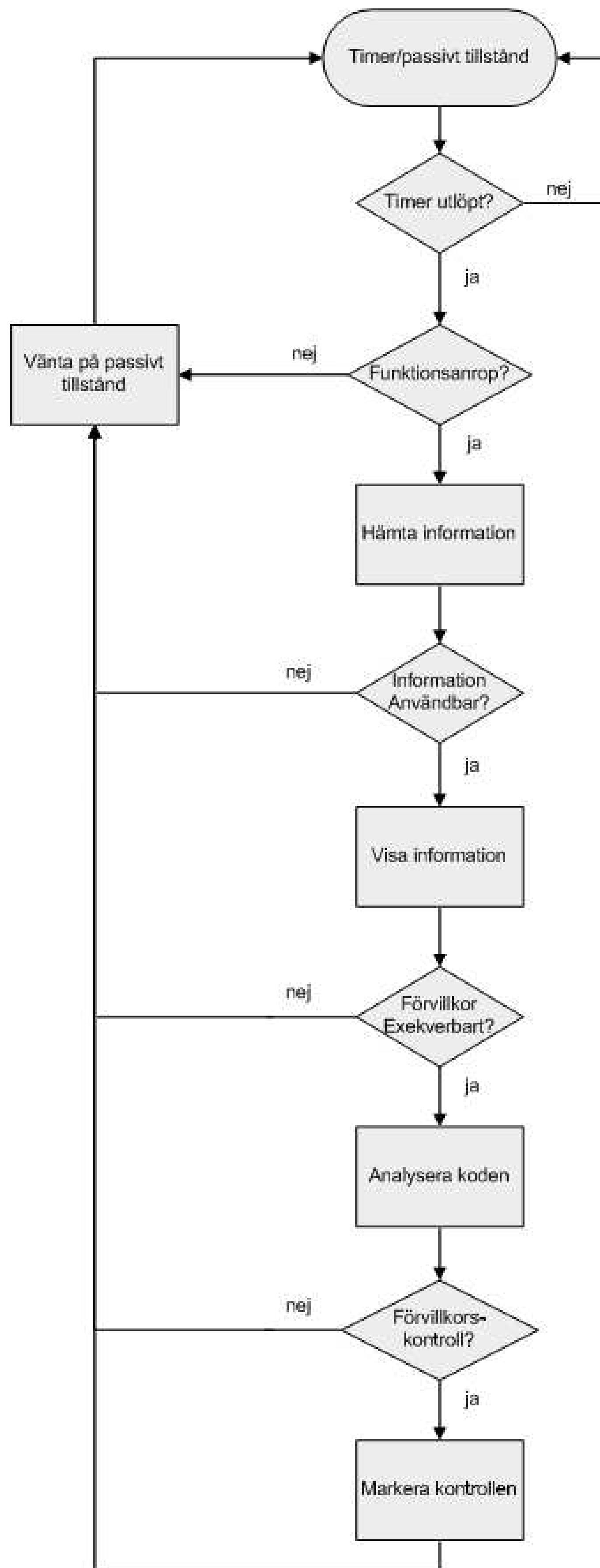
På grund av den tidsram som uppsatsen omfattas av måste vissa begränsningar göras. Exempel på begränsningar är:

- Inget stöd för C++ templates kommer att behandlas.
- Polymorfism och typomvandling av objekt.
- Arv

4.4 Prototyp

4.4.1 Överblick

Prototypen genomgår ett antal steg för att avgöra om det är möjligt att visa användbar information för användaren vid ett givet tillfälle. Användbar information är i första hand för- och eftervillkor för den funktion som anropas men även information som till exempel funktionssignatur kan vara användbart. När Emacs har varit inaktiv under en viss tid startar prototypens huvudfunktion. Denna funktion kontrollerar om det är möjligt att hämta information. Är det möjligt hämtas information och om relevant information lyckades sammanställas visas den för användaren. Ett översiktligt flödesdiagram över grundfunktionaliteten visas i Figur 4.4.



Figur 4.4: Grundfunktionalitet

4.4.2 Timer

Redigering av text i Emacs utförs med hjälp av kommandon. För att inte belasta Emacs i onödan används en timer som utlöses när programmet har befunnit sig i ett passivt tillstånd under en viss tid. Den passiva tiden är den tid som passerat sedan det senaste kommandot utfördes. Då en timer utnyttjas för att vänta tills en viss tid av inaktivitet har passerat körs inte programmet vid varje förändring av den redigerade texten och risken för störningar orsakade av för hög belastning minskas. Visning av kontraktsinformation, som är huvudmålet för prototypen, skall heller inte ske ögonblickligen eftersom informationsvisning inte alltid är önskvärd. Till exempel skulle en visning vid snabbt upprepade tangentnedtryckningar vara distraherande då låg passivitetstid ofta innebär att användaren redan vet vad som skall skrivas eller håller på att flytta inmatningsmarkören till en annan position i buffern. Vid användandet av en passivitetstimer undviks därför inte bara hög belastning utan gör också visningsfunktionen mer användarvänlig.

4.4.3 Kontroll av anrop

När Emacs har varit inaktiv under en inställningsbar tidsperiod startar timern huvudfunktionaliteten. Det första steget är att avgöra om det är aktuellt att visa kontraktsinformation vilket avgörs genom att analysera den text som markören i Emacs befinner sig vid. För att gå vidare med informationsinhämtning skall markören befinna sig vid ett funktionsanrop. I Figur 4.5 visas vilka markörpositioner som leder till att programmet går vidare till nästa steg.

```
s . push( element ) ;
```

= detektering av funktionsanrop

Figur 4.5: Positioner som leder till anropsdetektering.

4.4.4 Inhämtning av information

Om inmatningsmarkören befinner sig vid ett funktionsanrop kommer information om anropet att sammanställas. Denna information är funktionsnamn, returtyp, parametertyper och namn samt eventuellt objektnamn och typ om anropet är till en medlemsfunktion i en klass. Denna information avgör om och hur vidare information hämtas. Om funktionens namn och parametertyper stämmer överens med en funktionsdeklaration inhämtas den funktionens

returtyp vilket gör att överlagrade funktioner kan identifieras korrekt. Om anropet är till en medlemsfunktion måste denna funktion finnas deklarerad i den aktuella klassen.

Om tillräckligt med information om funktionsanropet kunde inhämtas kommer ett försök att hämta kontraktsinformation att göras. Kontraktsinformationen hämtas direkt ifrån kommentaren som finns ovanför funktioners deklaration. Denna deklaration använder sig av en speciell notation som identifierar kontraktets delar.

Tidigare D-uppsatser [2][6] har behandlat semantiska editorer med stöd för språket Java. I dessa har Javas JavaDoc system för dokumentering av kod använts för att ange kontrakt. JavaDoc använder sig av så kallade taggar inom speciella JavaDoc-kommentarer för att identifiera olika delar, som till exempel parametrar och undantag som kan kastas, i en kommentar. För att identifiera delar som hör till ett kontrakt har D-uppsatserna använt sig av taggarna @pre, @post och @invariant. I denna prototyp, som främst är inriktad mot C++, har denna notationen bibehållits då behovet av att kunna identifiera delar i ett kontrakt finns. @-tecknet används inte i någon språklig konstruktion i C++ och är därför lämplig för att tydligt markera kontrakt. En nackdel med att använda denna notation är att det i Object Constraint Language (OCL) [26], ett språk som används för att beskriva objektrestriktioner i Unified Modeling Language (UML) [26], används @pre för att hänvisa till ett tidigare tillstånd hos objekt. Eftersom @-notationen inte har någon betydelse i C++ samt att den kan ha andra användningsområden, som i UML, har det i prototypen inte antagits att det är denna notation som alltid kommer att används. Prototypen kan därför relativt enkelt modifieras för att stödja andra notationer.

4.4.5 Visning av information

Om kontraktsinformation lyckades sammanställas skall den visas för användaren. Visningen kan göras på två olika sätt: med hjälp av en *tooltip*-ruta eller i Emacs minibuffer längst ner i programmets huvudfönster. En *tooltip*-ruta, som är en vanligt förekommande grafisk komponent i fönstersystem, lägger sig ovanpå programmets huvudfönster och är ett enkelt sätt att visa information. Anledningen till att det behövs ett alternativ till *tooltip*-rutor är att dessa kräver stöd från ett fönstersystem. Då Emacs ursprungligen inte använde sig av ett grafiskt användargränssnitt och möjligheterna för att använda Emacs i ett textbaserat gränssnitt har bevarats är det viktigt att ge ett alternativt visningssätt. I den senaste stabila Emacs versionen för Windows har ännu inget stöd för *tooltip* implementerats.

4.4.6 Exekverbara förvillkor

Förvillkor kan uttryckas med hjälp av naturliga språk eller i form av kod. Användningen av kod för att beskriva förvillkor har fördelen att förvillkoren kan användas för att analysera klientkod, infoga förvillkorskontroller samt generering av kod för att detektera kontraktsbrott. Kodanalys och infogning av förvillkor, som är möjligt under redigering av kod, har implementerats i denna prototyp medans detektering av kontraktsbrott, som sker under körtid, möjliggörs av andra typer av verktyg. För att det skall vara möjligt att använda sig av förvillkor i form av exekverbar kod krävs ett sätt att kontrollera att denna kod är syntaktiskt korrekt samt ett sätt att anpassa förvillkoren till klientkod. Hur detta har lösts i denna prototyp beskrivs i resterande delen av detta avsnitt.

När predikatfunktioner används för att underlätta utformningen och kontroller av förvillkor, eller när inparametrars giltiga domäner skall uttryckas, är det ofta effektivt att utforma förvillkor som kod i det aktuella programspråket. En fördel med att uttrycka förvillkor med hjälp av kod är att det blir möjligt att analysera programkod. En programkodsanalys kan då kontrollera om det finns eventuella kontraktsbrott i klientkoden. Det blir också möjligt att infoga förvillkorskontroller i klientkod när förvillkoren är exekverbara.

För att avgöra om ett förvillkor är exekverbart eller inte parsas förvillkorstexten. Som utgångspunkt antas exekverbara förvillkor vara booleska-uttryck. Då C++ representerar booleska värden med hjälp av heltal har aritmetiska uttryck valts att tillåtas som förvillkor, vilket också är tillåtet som villkor i if-satser. Eftersom en stor mängd konstruktioner i C++ fungerar som uttryck, till exempel tilldelningar; olika varianter av funktionsanrop och pekararitmetik, har giltiga förvillkorsuttryck i prototypen begränsats till grammatiken som finns bifogad i bilaga A. Om förvillkors utformning stämmer överens med denna grammatik är förvillkoret exekverbart. Grammatiken som används vid parsning av förvillkor motsvarar endast en liten del av den fulla villkorsgrammatiken för C++ [14]. Även om begränsningar har gjorts så har uttryck som är vanligt förekommande i förvillkor inkluderats. Exempel på villkorselement som stöds är funktionsanrop, jämförelseoperatorer, aritmetiska operatorer samt variabelnamn. Grammatiken skulle kunna utökas för att stödja den fulla villkorsgrammatiken för C++ men detta innebär att även uttryck som, på grund av minskad läsbarhet, är mindre lämpliga i förvillkor skulle stödjas. Den grammatik som används i prototypen har en tillräcklig omfattning för att kunna utforma meningsfulla förvillkor.

I prototypen ges stöd för exekverbara förvillkor i både klient- och leverantörskod. I leverantörskod, där kontrakt definieras i en kommentar som direkt föregår själva

funktionsdefinitionen, parsas förvillkor när Emacs markör befinner sig inom ett visst område. Detta område visas i Figur 4.6 och har valts för att användaren själv skall kunna avgöra när förvillkorsuttrycket skall parsas. Användaren underrättas om förvillkoret är exekverbart eller inte genom att en *tooltip*-ruta visas. Misslyckas parsningen av förvillkoret informeras användaren om var i förvillkoret som fel påträffats. Denna felhantering är av en enklare typ och visar bara det första felet som påträffades.

```
/***********************************
 * Adds an element to the top
 * of the stack.
 *
 * @pre  !isFull()
 * @post top( ) == element
 **********************************/
void stack::push( int element );
```

Figur 4.6: Markörpositioner som leder till parsning av förvillkor.

Stödet för exekverbara förvillkor i klientkod är mer avancerat än stödet för leverantörskod. För att ett exekverbart förvillkor skall kunna användas av en klient för att avgöra om ett anrop är tillåtet eller inte måste det översättas, vilket görs i samband med parsningen. Även om förvillkor uttrycks med hjälp av kod har det en speciell form som inte är direkt användbar utanför dess sammanhang, som till exempel är en medlemsfunktion till en klass. Förvillkoret för en medlemsfunktion använder sig till exempel av de parameternamn som finns angivna i funktionens deklaration för att uttrycka begränsningar hos funktionen. Dessa parameternamn måste översättas till de namn eller värden som klienten använder som argument vid anrop till funktionen. Vid användandet av till exempel predikatfunktioner eller andra funktioner som är medlemfunktioner i samma klass används oftast inte något objektnamn eftersom räckvidden omfattar klassens medlemmar. När en klient anropar en medlemsfunktion måste denna använda sig av ett objektnamn tillsammans med en åtkommstoperator (förutsatt att medlemmen inte är statisk) för att kunna använda de medlemsfunktioner som förvillkoret använder sig av. Innebörden av detta är att anrop till medlemsfunktioner i förvillkor måste översättas till att använda sig av den aktuella objektnamnet för att vara användbara i klientkod. En annan översättning är nödvändig är användandet av nya rader. För att förvillkor

skall vara enkla att läsa är det vanlig att dela upp dem i flera rader. Dessa rader kan utläsas som delar i ett logiskt *och*-uttryck. För att inte behöva skriva ut C++-operatorn för logiskt *och* (&&) vid varje rad som utgör en del av ett förvillkor översätts istället nyradstecken till denna operator. Nedan följer ett exempel på resultatet av en översättning.

```
Leverantör:

/*
 * Adds a non-negative integer to the top of the stack.
 *
 * @Pre   !isFull()
 *        element >= 0
 * @Post  !isEmpty()
 */
void stack::push( int element )
{
  ...
}

Klientkod:

  stack* s = new stack( );
  int e;
  ...
  s->push( e );

Översättning av förvillkor:

!s->isFull( ) && e >= 0
```

Att tolka nyradstecken som implicita *och*-operatorer är inte att rekommendera eftersom det riskerar att göra kontrakt mindre tydliga. Anledningen till att nyradstecken kan användas som implicit *och*-operator är att vissa användare är vana att använda denna notation för att lista uttryck som samtliga måste vara sanna för att till exempel ett förvillkor skall vara uppfyllt. I prototypen finns möjlighet att ställa in om denna notation skall tillåtas eller ej. För att minska risken för feltolkningar används inte nyradstecken som *och*-operatorer i standardinställningen.

4.4.7 Kodanalys

För att kunna kontrollera om ett förvillkor är uppfyllt eller brutet genomsöks koden för att möjligen kunna fastställa om villkoret är uppfyllt. Ett förvillkor kan sägas vara uppfyllt om det med hjälp av en if-sats kontrolleras och finnes giltigt innan det aktuella anropet utförs. En metod för att kunna undersöka om de olika anropen i en funktion föregås av korrekta kontroller är att se på funktionen som en träd-representation av kontroller och funktionsanrop. Genom denna representation är det sedan möjligt att spåra kontroller av förvillkor utifrån det möjliga programflödet. Exempel på hur följande kodstycke kan representeras i form av ett träd kan ses i Figur 4.7.

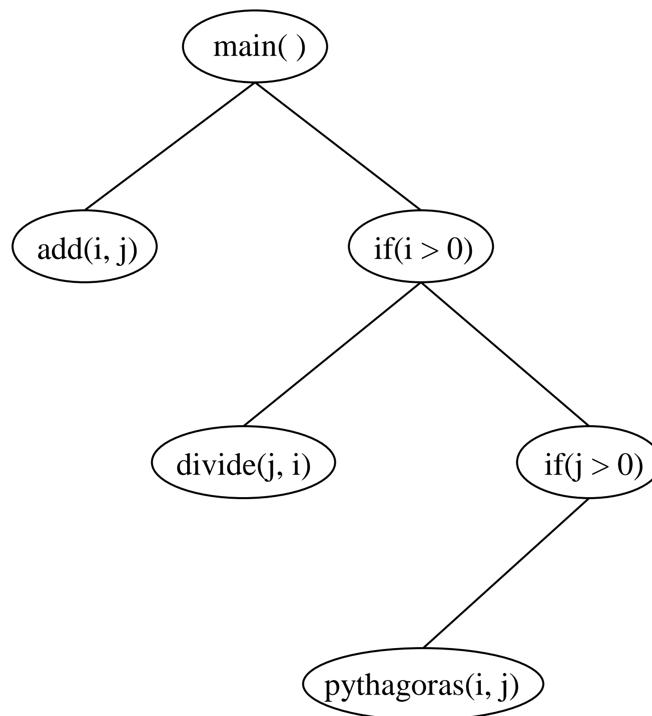
```
/*
 * Calculates the hypotenuse of a right triangle
 * using the legs 'i' and 'j'.
 *
 * @Pre   i > 0 && j > 0
 * @Post  result = length of the hypotenuse
 */
int pythagoras(i, j)
{
  . . .
}

int main()
{
  int i = 2, j = 4, value;

  add(i, j);

  if(i > 0)
  {
    divide(j, i);

    if(j > 0)
      value = pythagoras(i, j);
  }
}
```



Figur 4.7: Trädrepresentation av kod.

I koden som föregår Figur 4.7 går det att utläsa i förvillkoret för funktionen pythagoras att både X och Y skall vara större än 0. Genom att följa exekveringsvägen i koden och följa vilka kontroller som utförs går det att se att detta uppfylls. För att underlätta detta i större exempel finns trädrepresentationen som ses i Figur 4.7 där det är enkelt att följa exekveringsvägen för den aktuella funktionen och kontrollera vilka tester som utförs. Det går alltså att på detta sätt undersöka om förvillkoren för de olika funktionerna som anropas i main-funktionen kontrolleras.

Ett problem med denna lösning är att den inte tar hänsyn till möjligheten att andra funktioner kan anropas mellan en kontroll av förvillkor och anropet av funktionen vilket innebär att möjlighet finns att förvillkoret inte längre är uppfyllt vid funktionsanropet trots att kontroll utförts. Innebörden av detta är att den enda gången det går att säkerställa att villkoret verkligen är uppfyllt är då kontrollen av förvillkoret sker direkt före funktionsanropet. En lösning skulle kunna vara att varje funktions förvillkor även specificerar vilka effekter funktionen har på den aktuella modulen. Detta skulle dock kräva en helt ny notation för förvillkoren vilket ligger utanför detta arbetes gränser.

Ytterligare en svårighet vid kontroll av förvillkor är då villkor uppfylls implicit. Exempel på detta är om ett förvillkor anger att en storlek måste vara större än 0 och testet kontrollerar att storleken är större än 1. I de fall som kontrollen lyckas är det också tillåtet att anropa funktionen. Dessa fall kommer inte att klassas som korrekta i prototypen då enbart exakta förvillkor godkänns. Anledningen till att endast exakta förvillkor tillåts är att möjligheten till oönskade sidoeffekter då utesluts. Ett exempel på hur ett förvillkor kontrolleras på ett otillåtet sätt vilket resulteras i oönskade sidoeffekter ses i koden nedan.

```
/*
 * Gets the final price of the product.
 *
 * @Pre   subtractDiscount( )
 *        addValueAddedTax( )
 * @Post  The final price is returned
 */
int product::getFinalPrice( )
{
  ...
}

int main()
{
  int i;
  product p;
  ...
  if(p.addValueAddedTax() && p.subtractDiscount())
    i = p.getFinalPrice( );
  ...
}
```

I koden ovan kontrolleras förvillkoret men i en felaktig ordning. Förvillkoret anger att rabatten först skall dras av och därefter skall skatten läggas på. I kontrollen sker detta i omvänd ordning vilket ger ett felaktigt resultat.

Kontrollen av förvillkor sköts i den utvecklade prototypen på ett sätt som tar hänsyn till ovan angivna problem. Då möjligheten att klart definiera att ett kontrakt är uppfyllt eller brutet är begränsad konstateras inte heller detta. Vad som istället sker är att med start vid det

löv i träd-representationen där anropet sker startas en traversering av trädet. Denna traversering innebär att föräldern kontrolleras för att se om det är en korrekt kontroll av förvillkoret. Om så inte är fallet fortsätter traverseringen med en ny kontroll av nästa förälder. Traverseringen fortsätter till dess att en godkänd kontroll funnits eller funktionens början har nåtts. Genom att traversera trädet *bottom-up*, det vill säga att börja vid den nod som representerar det aktuella anropet och sedan fortsätta mot trädets rot, minimeras antalet noder som måste analyseras eftersom endast för anropet relevanta noder behandlas. Då en godkänd kontroll funnits markeras denna och användaren kan se närmaste godkända förvillkorskontroll för den aktuella funktionen utförs. På detta sätt får inte användaren något klart besked om förvillkoret är uppfyllt eller brutet men det är inte heller möjligt att ge. Ett annat sätt att angripa problemet på skulle ha kunnat vara att ange möjliga kontraktsbrott då kontrollen inte sker direkt före anropet. Att användaren istället får kontrollen markerad ger dock mer information till användaren.

4.4.8 Infogning av förvillkor

Då ett förvillkor skall infogas görs först en kontroll av förvillkoret där det undersöks om förvillkoret är exekverbart eller inte. Ett exekverbart förvillkor är ett villkor som är syntaktiskt korrekt vilket innebär att förvillkoret kan exekveras. Anledningen att detta kontrolleras är att endast tester av exekverbara förvillkor tillåts att infogas av anledningen att det endast är dessa förvillkor som kommer att resultera i godkänd kod. Kontroll sker genom att förvillkoret parsas och en närmare beskrivning av hur detta utförs ges i avsnitt 4.4.6.

Infogningen av test av förvillkor utförs av användaren genom tangentkombinationen Ctrl-c följt av ”_”. Denna kombination är vald så att den följer Emacs konvention för snabbkommandon och för att kommandot skall kännas enkelt och naturligt att använda i Emacs.

Användaren har möjlighet till infogning av test av förvillkor i samband med att kontraktet visas för användaren. Det test som skall infogas är om förvillkoret för den aktuella funktionen är uppfyllt och detta kontrolleras med en if-sats. För att få önskad effekt skall alltså kontrollen infogas före anropet och genom detta säkerställs att anropet inte utförs om förvillkoret inte är uppfyllt. För att göra detta infogas in if-sats innehållandes det parsade förvillkoret på raden som föregår anropet.

För att infogning av förvillkor skall fungera på ett korrekt sätt skall även testet ske med de argument som används i funktionsanropet. Skulle det parsade förvillkoret enbart infogas i

enlighet med funktionsdeklarationen är det möjligt att andra argumentnamn används vid anropet än vid funktionsdeklarationen. Ett exempel på detta kan ses nedan.

```
/*
 * Divides argument 'a' by argument 'b'.
 *
 * @Pre   b != 0 && inRange(a)
 * @Post  result of the division is returned.
 */
int integer::divide(int a, int b)
{
  …
}

void main()
{
  int x = 4, y =2, value;
  integer i;

  value = i.divide(x, y);
}
```

Skulle ett förvillkorstest för divide infogas utan att hänsyn tas till argumentnamn skulle det innebära att ett test enligt koden nedan skulle infogas.

```
if(b != 0 && inRange(a))
  value = i.divide(x, y);
```

I detta fall sker testet på variabler som inte existerar och fel uppstår. Argumentnamnen som är angivna i förvillkoret måste alltså översättas till motsvarande namn i anropet. Detta sker genom positionen på det argument som skall kontrolleras matchas gentemot motsvarande position i funktionsdeklarationen vid parsningen. Efter parsningen är det därför det faktiska argument som infogas.

Ytterligare en aspekt som det måste tas hänsyn till är då anropet är till en medlemsfunktion i en klass. För att infogningen skall bli korrekt måste klassnamnet läggas till i förvillkoret för att få det korrekta resultatet som kan ses i koden nedan.

```
if(y != 0 && i.inRange(x))
  value = i.divide(x, y);
```

Även denna korrigering sker i samband med parsningen där även fall då pekare används kontrolleras. Används pekare måste medlemsåtkomstsoperatorn ”->” användas istället för ”.” vid översättningen. Vid användandet av pekare skulle ett test, som kan ses i koden nedan, genereras.

```
if(y != 0 && i->inRange(x))
  value = i->divide(x, y);
```

4.5 Sammanfattning

Detta kapitel har givit en genomgång av de verktyg som använts vid framtagningen av prototypen. Dessa verktyg är utvecklingsmiljön GNU Emacs, programspråket Emacs Lisp, samt CEDET.

Efter beskrivningen av de verktyg som använts har en genomgång av den funktionalitet som prototypen skall tillhandahålla getts. Hur denna funktionalitet sedan har tagits fram beskrivs i efterföljande avsnitt.

Avsnittet som beskriver hur funktionaliteten tagits fram inleds med en överblick där flödet för programmets huvudfunktionalitet visas. Sedan följer en noggrannare beskrivningar av de olika stegen i programmets flöde. Efter denna beskrivning bör läsaren ha en uppfattning om hur funktionaliteten tagits fram, dock utan en detaljerad beskrivning på kodnivå. Anledningen till detta är att vi inte anser det vara relevant att veta på kodnivå utan det intressanta är vilken metodik som använts och vilket tankesätt som ligger bakom.

5 Utvärdering av prototyp

I detta kapitel utvärderas den framtagna prototypen som beskrevs i kapitel 4. Kapitlet inleds med ett avsnitt som ger en demonstration av den utvecklade prototypen. Avsnittet följs av en utvärdering av hur den utvecklade funktionaliteten fungerar i den faktiska prototypen. Prototypens funktionalitet utvärderas här stegvis i enlighet med den ordning som den presenterades i föregående kapitel.

Efter att funktionaliteten utvärderats ges en utvärdering av hur bra lämpat GNU Emacs är för utveckling av en semantisk editor. Sedan följer en sammanfattning av kapitlet.

5.1 Inledning

Kapitlet inleds med ett avsnitt med en demonstration av den framtagna prototypen. I detta avsnitt kan flertalet skärmbilder från användandet av prototypen ses samt en beskrivning av dess innebörder.

Efter detta avsnittet följer en utvärdering av den framtagna funktionaliteten. För att ansluta till den beskrivna funktionaliteten i kapitel 4 ges en mer ingående utvärdering av hur de olika funktionaliteterna i prototypen resulterat. Denna utvärdering är indelad på samma sätt som beskrivningen i kapitel 4 för att på ett enkelt sätt ge en återkoppling mot tidigare beskrivning.

Efter att den beskrivna funktionaliteten redovisats summeras användandet av GNU Emacs som bas för utveckling av den semantiska editorn. Detta görs dels genom att Emacs utvärderas som editor och dels genom att det vid utvecklingen använda programspråket Emacs Lisp utvärderas. I detta avsnitt ges både fördelar och nackdelar med användandet av Emacs och Emacs Lisp med avseende på utvecklingen av den semantiska editorn.

Kapitlet summeras sedan i det avslutande avsnittet.

5.2 Demonstration av prototyp

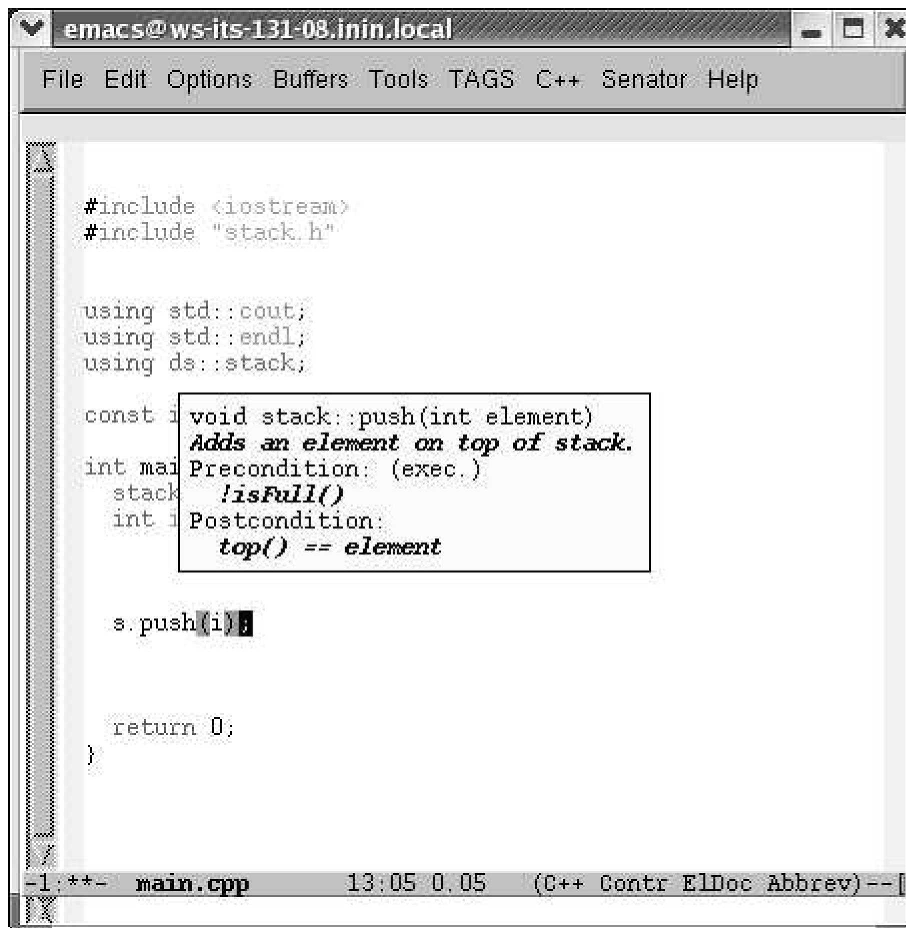
Innan funktionaliteten hos prototypen utvärderas i avsnitt 5.3 följer här demonstration som beskriver hur användandet av prototypens funktionalitet ser ut. För att demonstrationen skall vara så tydlig som möjligt används skärmbilder på Emacs när de olika funktionaliteterna har aktiverats.

5.2.1 Visning av kontrakt

Den automatiska visningen av kontrakt sker efter en inställningsbar tid när markören (svart ruta över ett semikolon i Figur 5.1) befinner sig på ett funktionsnamn eller efter slutparentesen för ett funktionsanrop. Informationen som visas är funktionens signatur, övergripande beskrivning från funktionens kontraktskommentar, förvillkor samt eftervillkor. Är förvillkoret exekverbart markeras detta med ”(exec.)” vid förvillkorsrubriken.

Eftersom visning av kontraktsinformation är implementerat som ett minor mode och måste vara påslaget för att kunna användas markeras detta i Emacs *mode line* med hjälp av texten ”Contr”. Emacs *mode line* återfinns i den nedre delen av huvudfönstret och anger även den aktuella filens namn och andra aktiva *modes*.

I Figur 5.1 ges ett exempel på hur visningen av kontraktsinformation kan se ut.

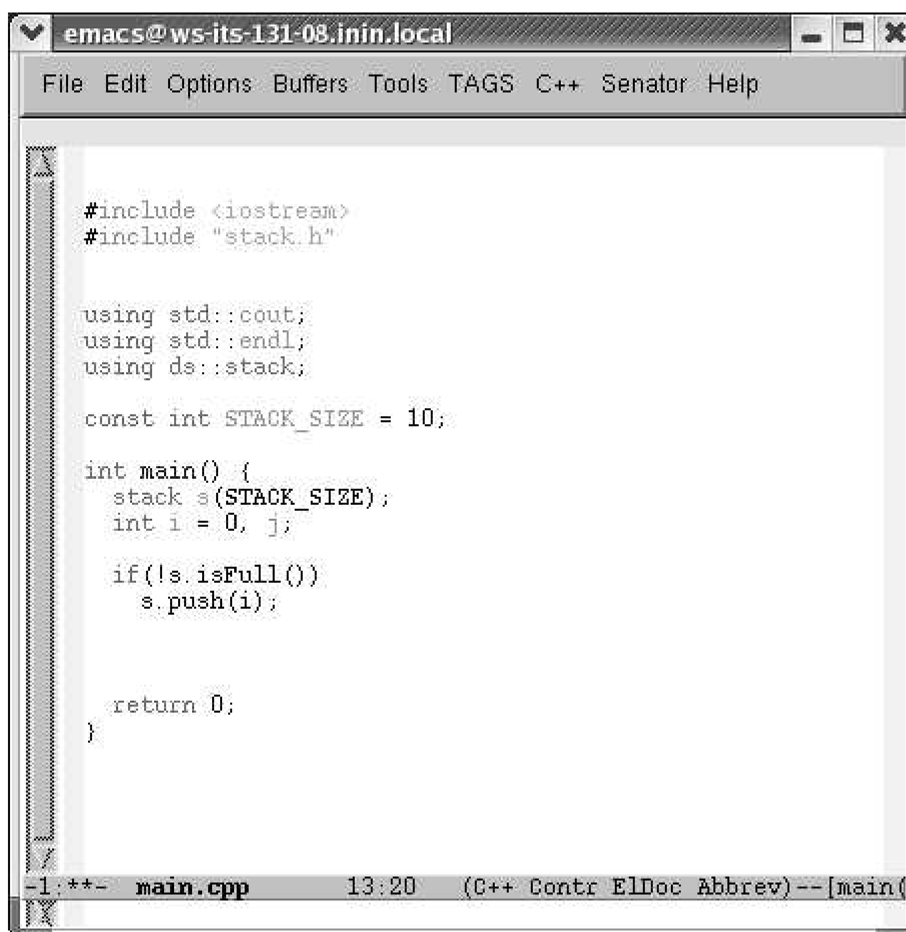


Figur 5.1: Visning av kontraktsinformation.

5.2.2 Infogning av förvillkor

Om ett förvillkor är exekverbart är det möjligt att infoga det direkt i koden så att det med hjälp av en if-sats bildar en förvillkorskontroll för anropet. Infogningen sker genom att förvillkoret översätts till den aktuella omgivningen för funktionsanropet vilket innebär att till exempel objektnamn och aktuella argument används istället för de signaturberoende namnen som används för att uttrycka förvillkoret. Infogning av förvillkor sker genom ett kommando som är bundet till en tangentkombination. Om förvillkoret inte är exekverbart påverkas inte koden av att detta kommando utförs utan ett felmeddelande visas i minibuffern.

I Figur 5.2 visas ett exempel på infogning av förvillkor.



Figur 5.2: Infogning av förvillkor.

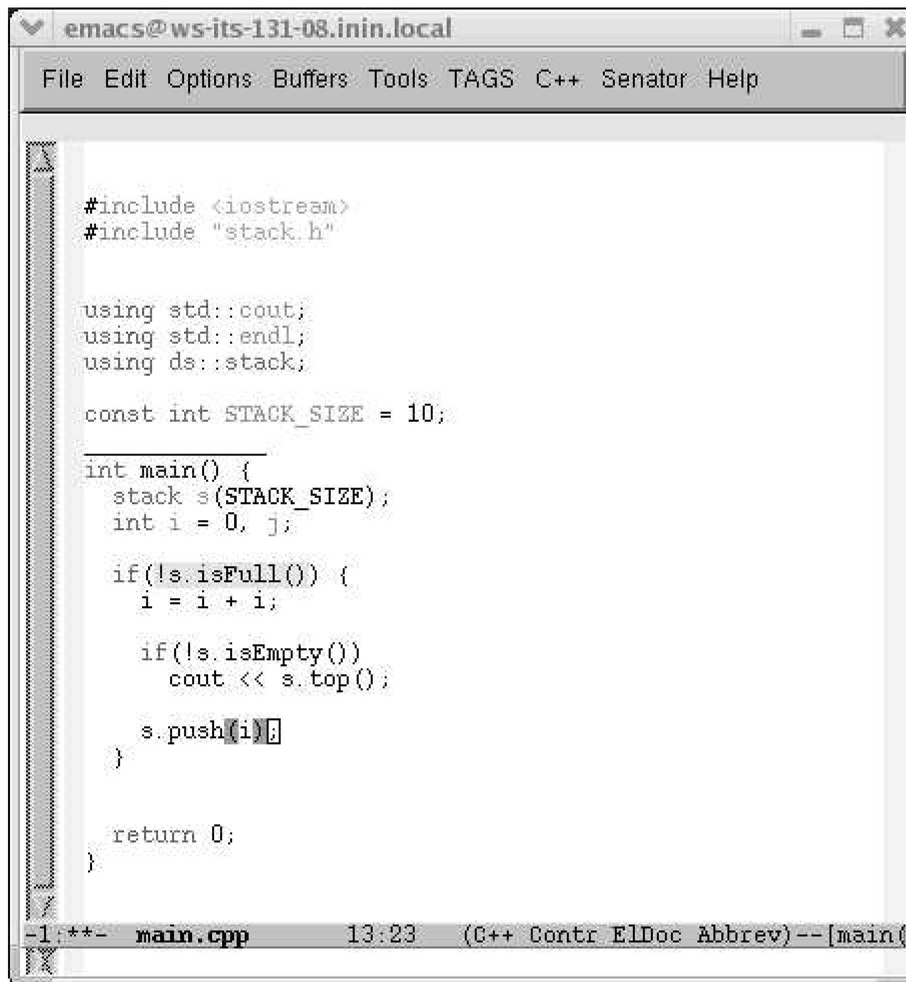
5.2.3 Markering av förvillkorskontroll

När markören befinner sig vid ett funktionsanrop och kontraktsinformation för anropet finns tillgängligt sker en sökning i klientkoden efter en befintlig kontroll av funktionens förvillkor.

Om en kontroll i form av en if-sats med funktionens förvillkor som villkor hittas markeras denna i koden. Att en förvillkorskontroll detekteras innebär inte att förvillkoret är uppfyllt då detta skulle kräva att kontrollen utförs direkt före funktionsanropet. Istället är markeringen ett sätt att uppmärksamma användaren på att det, någonstans i exekveringsvägen som leder till funktionsanropet, finns en kontroll av funktionens förvillkor. Användaren måste själv avgöra om kontrollen kan garantera att förvillkoret är uppfyllt vid själva anropet eller om till exempel andra funktionsanrop kan upphäva kontrollens giltighet innan anropet utförs. Helst skulle inte användaren behöva avgöra om förvillkoret är uppfyllt eller ej men eftersom det är svårt att garantera att förvillkor är uppfyllda vid ett anrop är metoden att markera en eventuell förvillkorskontroll till stor hjälp.

Markeringen av förvillkorskontroller görs temporärt och när något kommando (till exempel teckeninmatning) aktiveras avmarkeras kontrollen. Markeringen kan återskapas genom att markören återigen placeras vid det aktuella funktionsanropet.

I Figur 5.3 visas hur en förvillkorskontroll markeras i koden. Markören befinner sig efter slutparentesen i anropet till funktionen *stack::push*.



Figur 5.3: *Markering av förvillkorskontroll.*

5.2.4 Alternativ visning av kontrakt

Eftersom *tooltip*, det grafiska lättviktsfönstret som används för informationsvisning, inte stöds i alla Emacs-versioner kan information istället visas i minibuffern. Minibuffern är det lilla fönstret som finns längst ner i Emacs och används bland annat för inmatning av kommandonamn och parametrar. Finns det inget stöd för *tooltip*, som till exempel i Windowsversionen, används minibuffern automatiskt. Det är också möjligt att själv välja vilket visningssätt som skall användas i första hand.

I Figur 5.4 används minibuffern för att visa kontraktsinformation.

```
emacs@ws-its-131-08.inin.local
File Edit Options Buffers Tools TAGS C++ Senator Help

#include <iostream>
#include "stack.h"

using std::cout;
using std::endl;
using ds::stack;

const int STACK_SIZE = 10;

int main() {
  stack s(STACK_SIZE);
  int i = 0, j;

  if(!s.isFull()) {
    i = i + i;

    if(!s.isEmpty())
      cout << s.top();

    s.push(i);
  }


  return 0;
}
-1:**-  main.cpp       13:29 0.02    (C++ Contr ElDoc Abbrev)--[
void stack::push(int element)
Adds an element on top of stack.
Precondition: (exec.)
  !isFull()
Postcondition:
  top() == element
```

Figur 5.4: Alternativt visningssätt för kontraktsinformation.

5.2.5 Parsning av förvillkor i leverantörskod

För att kontrollera om ett förvillkor är syntaktiskt korrekt och därmed exekverbart har användaren möjlighet att parsa förvillkoret. Parsning utförs genom att markören placeras enligt Figur 4.6. Då markören är placerad på någon av dessa positioner utförs parsning av förvillkoret och resultatet visas med hjälp av en *tooltip*-ruta. Hur en lyckad parsning ser ut kan ses i Figur 5.5. Då parsningen misslyckas visas även detta i en *tooltip*-ruta vilket kan ses i Figur 5.6. I detta fall visas ett felmeddelande med det första påträffade felet.

```
stack.h
File  Edit  Options  Buffers  Tools  C++  Help

    /***********************************************
     * Removes the top element from the stack.
     * @Pre  !isEmpty()
     * @Post !isFull()
    Executable precondition:      ********************/
    !isFull() && element > 0

    /***********************************************
     * Adds a positive element on top of stack.
     * @Pre  !isFull() && element > 0
     * @Post top() == element
     ***********************************************/
    void push(int element);

    /***********************************************
     * Returns the top element of the stack.
     * @Pre  !isEmpty()
     * @Post result = element on top of the stack.
     ***********************************************/
    int top() const;
--(DOS)**  stack.h       14:08 0.02   (C++ Contr Abbrev)--L30--
```

Figur 5.5: Syntaktiskt korrekt exekverbart uttryck.

```
stack.h
File  Edit  Options  Buffers  Tools  C++  Help

    /***********************************************
     * Removes the top element from the stack.
     * @Pre  !isEmpty()
    Parse error near: and   ***********************/
    Remaining input:
    and element > 0
    /***********************************************
     * Adds a positive element on top of stack.
     * @Pre  !isFull() and element > 0
     * @Post top() == element
     ***********************************************/
    void push(int element);

    /***********************************************
     * Returns the top element of the stack.
     * @Pre  !isEmpty()
     * @Post result = element on top of the stack.
     ***********************************************/
    int top() const;
--(DOS)**  stack.h       14:16   (C++ Contr Abbrev)--L30--C8--2
```

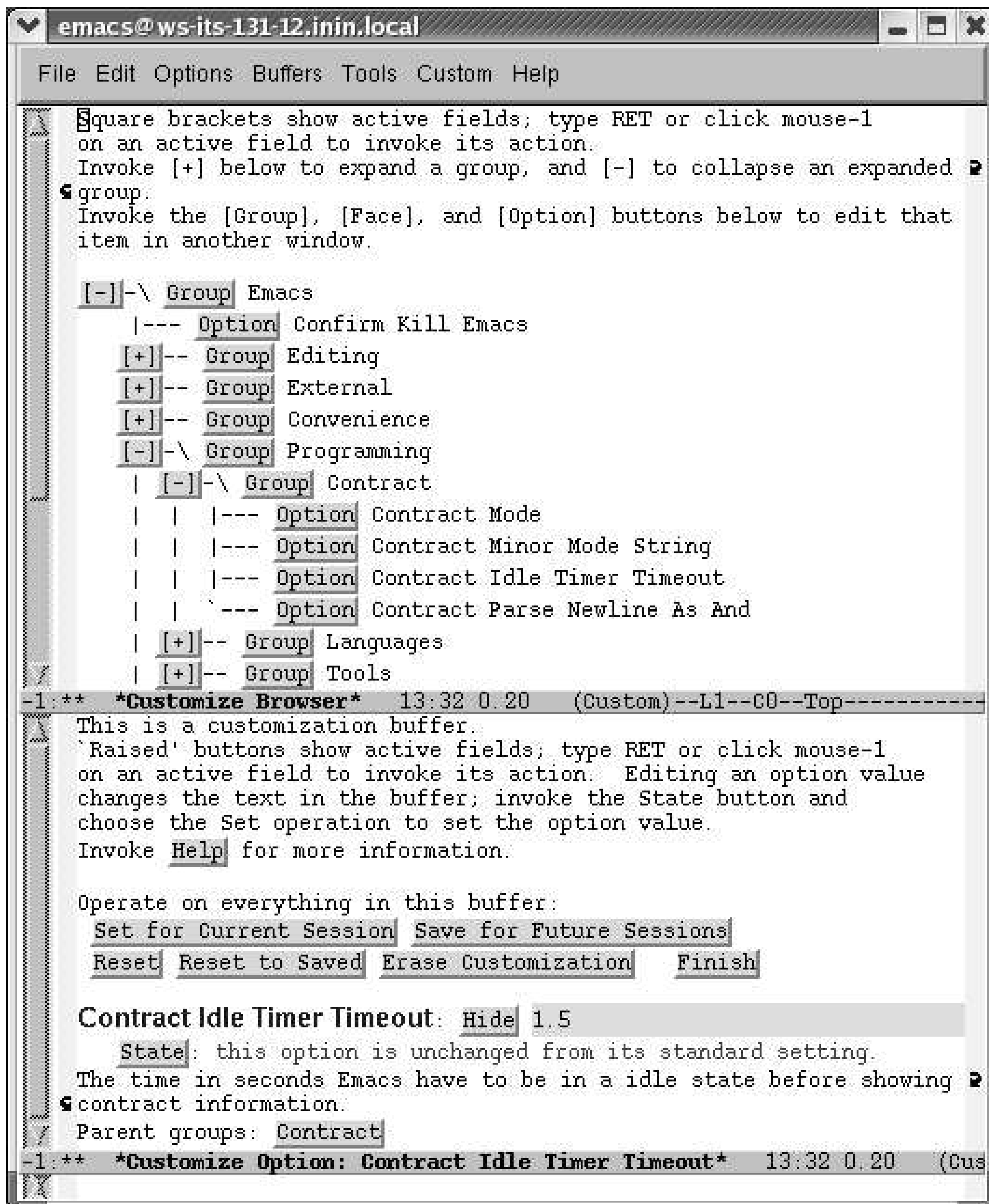
Figur 5.6: Ej exekverbart förvillkorsuttryck.

5.2.6 Inställningar via inställningspanelen

För att utföra personliga inställningar i den utvecklade prototypen finns en inställningspanel. Hur denna panel ser ut kan ses i det övre Emacs fönstret i Figur 5.5. För att nå inställningarna för contract-mode navigerar användaren genom trädstrukturen genom att öppna gruppen Emacs följt av grupperna programming och contract. Det är trädrepresentationen för detta läge som visas i det övre fönstret i Figur 5.7. Användaren har här möjlighet till fyra olika personliga inställningar. Dessa fyra inställningar är:

- Contract-mode. Här kan contract mode stängas av och sättas på.
- Contract Minor Mode String. Här anges den sträng som skall visas då contract mode är påslaget.
- Contract Idle Timer Timeout. Här anges värdet för timern.
- Contract Parse New Line As And. Här anges om ny rad skall tolkas som logiskt och.

För att exemplifiera en inställning har värdet för timern valts. För att ändra detta värde väljs ”Contract Idle Timer Timeout” från trädet och inställningar för timern blir då möjliga att utföra. Hur detta ser ut kan ses i det undre fönstret i Figur 5.7. Användaren kan här justera värdet för timern efter önskemål. I exemplet har värdet satts till 1,5 sekunder.



Figur 5.7: Inställning av Contract-Mode med hjälp av inställningspanel.

5.3 Utvärdering av funktionalitet

5.3.1 Timer

För att inte belasta systemet för hårt genom ständig exekvering har en timer använts för att avgöra när där är lämpligt att visa kontrakt. Timern startas då Emacs befinner sig i ett passivt tillstånd och efter att den inställda tiden har passerat startas exekveringen av

huvudfunktionaliteten. Eftersom Emacs befinner sig i ett passivt tillstånd störs inga andra aktiviteter som till exempel textredigering. Denna metod för att avgöra om visning av kontrakt är aktuell eller inte fungerar tillfredställande.

Då användaren skriver fort är detta ett tecken på att användaren redan vet vad som skall skrivas och ingen information behöver därför ges. Då användaren däremot skriver långsammare och gör uppehåll i skrivandet är detta ett tecken på att information kan vara önskvärt. På detta sätt är det även möjligt för användaren att då information önskas, ställa markören vid det anrop för vilket information önskas. Hur lång tid som då skall fortlöpa avgörs av ett inställningsbart värde som har ett utgångsvärde på en sekund vilket kan anses lämpligt [12]. Användaren kan sedan anpassa detta värde med hjälp av ett kommando för att få ett önskat timervärde. Lösningen att ge användaren möjlighet att själv justera timervärdet efter egna önskemål har ansetts vara bra då skrivhastighet är individuellt.

Sammantaget kan det sägas att metoden att använda en timer medför en god prestanda som inte medför någon belastning av Emacs som stör textredigering.

5.3.2 Kontroll av anrop

Kontroll av anrop sker då markören står på vissa bestämda positioner av ett anrop. Dessa positioner kan ses i Figur 4.5. De valda positionerna för detektering av anrop fungerar bra i prototypen. Då användaren vill ha information vid ett anrop känns det rimligt att placera markören över funktionsnamnet som anropas, över öppningsparentesen eller direkt efter slutparentesen. Dessa positioner har valts för att täcka in så stor del av anropet som möjligt men samtidigt möjliggöra att anrop som står som parametrar kan behandlas separat för kontraktsvisning. Då markören är placerad inom parenteser som omger parametrar visas alltså ingen information om anropsfunktionen. Anledningen till detta är att parametrarna i sig kan vara funktionsanrop vilket kan ses i Figur 5.8.

Figur 5.8: Visning av kontrakt för parameterfunktion.

I dessa fall kan användaren istället se information om den funktion som utgör parametern om så önskas. Avgränsningen för var markören skall placeras gäller alltså alltid då ett anrop utförs. Ett alternativ till denna lösning skulle kunna vara att alltid visa information om anropet även inom parenteserna. För användaren kan det dock vara värdefullt att även få information om funktionsanrop som står som argument. Ett exempel på detta kan ses i Figur 5.8 där stack::top( ) står som argument till stack::push(int).

5.3.3 Inhämtning av information

Vid framtagningen av prototypen har taggar använts för att specificera kontrakt vilket fungerar bra och är även den metod som använts i de två tidigare D-uppsatserna [2][6]. Ett val av hur taggarna skall se ut har gjorts men utformandet av taggarna kan ändras med enkelt ingrepp i koden. Nackdelen med den nuvarande utformningen är att notationen @pre används i OCL [26] men problemet med detta är inte akut då det inte är någon stor del i den nuvarande undervisningen av kontrakt vid Karlstads universitet.

Den information för ett anrop som inhämtas är funktionsnamn, returtyp, parametertyper och namn, eventuellt objektnamn och typ samt kontraktsinformation. Det som visas för användaren baseras sedan på denna inhämtade information som sätts ihop till det kontrakt som visas för användaren.

5.3.4 Visning av information

Kontrakt kan visas både genom en *tooltip*-ruta och genom Emacs minibuffer. Att visa kontraktet i minibuffern fungerar bra men i de versioner där *tooltip*-rutor stöds är detta oftast att föredra då det är fördelaktigt för användaren att få informationen i anslutning till det anrop som är aktuellt. Användaren får på detta sätt informationen presenterad i anslutning till anropet och behöver därför inte förflytta blicken till en annan del av skärmen. Denna typ av visning är vanlig i grafiska utvecklingsverktyg som till exempel Microsoft Visual Studio [23] och NetBeans IDE [24].

5.3.5 Exekverbara förvillkor

Att parsa förvillkor är nödvändigt för att avgöra om ett förvillkor är syntaktiskt korrekt eller inte. Denna kontroll fungerar bra och uttryck enligt grammatiken i bilaga A godkänns som exekverbara.

I den lösning som tagits fram vid utvecklingen av prototypen undersöks endast om ett förvillkor är korrekt eller inte. En vidareutveckling av denna parser skulle kunna vara en parser som anger fel i exekverbara förvillkor. Den utvecklade parsern hittar första felet i ett felaktigt skrivet exekverbart uttryck men har däremot ingen möjlighet till återhämtning för att vidare undersöka förvillkoret. Ett problem i detta fall är att parsern inte vet om användaren verkligen menar att skriva ett exekverbart förvillkor eller inte. En lösning på detta skulle kunna vara att parsern räknar antalet fel och en maxgräns anges för att avgöra om ett förvillkor skall behandlas som exekverbart eller inte. Alternativt skulle olika taggar kunna användas för att ange om ett förvillkor är tänkt som exekverbart eller inte. Dessa aspekter har dock inte behandlats vidare vid utvecklingen av prototypen då en parser med mer sofistikerad felhantering måste utvecklas vilket ansetts vara för tidskrävande.

Parsning av förvillkor i leverantörskod gör att användare enkelt kan säkerställa att ett förvillkorsuttryck som är menat som exekverbart har korrekt syntax och kan utnyttjas vid till exempel infogning och verifiering av förvillkorskontroller i klientkod. En ytterligare möjlighet som skulle kunna erbjudas i leverantörskod är permanent markering (*syntax highlighting*) av exekverbara förvillkor i leverantörskoden där förvillkor definieras.

Översättningen som sker i samband med parsning när klientkod editeras anses även vara lyckad. Att användaren kan infoga ett förvillkor som då det infogats är korrekt i sammanhanget är mycket viktigt för att prototypen skall vara så användbar och korrekt som möjligt.

5.3.6 Kodanalys

Metoden att märka ut närmaste utförda kontroll känns naturlig och ger användbar information. Alternativet till detta skulle vara att alla möjliga kontraktsbrott märks ut vilket i många fall skulle resultera i en stor mängd markeringar då det är svårt att säkerställa om ett kontrakt är uppfyllt eller inte. Valet att analysera eventuella kontraktsbrott i samband med att en funktions kontrakt visas känns också rätt. Ett alternativ hade varit att ständigt analysera källkoden och märka ut eventuella kontraktsbrott. Genom att istället analysera källkoden när kontraktsinformation visas så slipper användaren att markeringar visas konstant vid användning av prototypen.

5.3.7 Infogning av förvillkor

Infogning av förvillkor sker genom en tangentkombination i enlighet med Emacs standard för att det skall kännas igen av vana Emacs-användare, men även då det är en vanligt förekommande metod som är lätt att ta till sig för alla användare. Vid infogning av förvillkor kan endast exekverbara förvillkor infogas då det endast är dessa som är värdefulla att infoga.

Infogning sker i denna prototyp genom att en if-sats läggs till på raden innan anropet enligt koden nedan.

```
if(!s.isEmpty())
  s.pop();
```

Det kan tänkas att en användare önskar att använda en annan typ av indentering där till exempel måsvingar används för att innesluta anropet enligt koden nedan.

```
if(!s.isEmpty()) {
  s.pop();
}
if(!s.isEmpty()) {
  s.pop();
}
```

Ett sätt att hantera detta skulle kunna vara att ett antal olika varianter för indentering finns i prototypen och användaren får själv ange vilken typ som är önskvärd med hjälp av en inställning. Detta har dock inte utvecklats i prototypen då det inte ansets vara av stor vikt.

5.4 Utvärdering av GNU Emacs

GNU Emacs är en texteditor som ger stöd vid redigering av alla sorters text. Stödet kan vara av generell typ, som inte är beroende av textens format, eller speciellt inriktat på en viss typ av text, som till exempel kod skrivet i olika programspråk.

En av Emacs främsta egenskaper är dess utökningsmöjligheter och anpassningsbarhet vilket gör att speciellt stöd för en stor mängd texttyper är tillgänglig eller enkelt kan läggas till. Stödet för vanliga texttyper som till exempel kod i programspråken C++, Java, Ada, Perl och Fortran gör att grundläggande editorfunktionalitet som syntax highlighting och automatisk indentering finns som standard och kan samverka med ny tilläggsfunktionalitet. Detta medför att Emacs är en bra bas för utvecklingen av prototypen då stöd finns för programmspråket C++ och tillägsfunktionalitet kan implementeras på ett enkelt sätt.

En fördel med Emacs är dess tillgänglighet. Emacs är inte utvecklat för någon specifik plattform utan kan användas på de vanligaste operativsystemen som till exempel Linux, Windows och de flesta Unix-varianterna. Tack vare detta plattformsoberoende kan man använda Emacs som texteditor på de flesta plattformar vilket inte är så vanligt för andra editorer. Denna tillgänglighet gör att Emacs passar bra för utvecklingen av den semantiska editorn då den kan användas på olika plattformar. Emacs Lisp, som är det i särklass mest använda språket för att utöka och modifiera Emacs, använder sig av samma standardbibliotek oavsett vilken plattform man väljer att använda Emacs på vilket gör att tillägg också blir plattformsoberoende. Det enda som kan variera i Emacs på olika plattformar är att den senaste funktionaliteten ibland tar längre tid att införas på vissa plattformar vilket är fallet med *tooltip*-fönster som inte har införts i Windowsversionen ännu. I prototypen används istället Emacs minibuffer för att visa information när *tooltip* inte stöds, som till exempel i Windowsversionen eller i terminalläge. Att Emacs Lisp använder samma standardbibliotek oavsett plattform är en fördel då tilläggen därmed inte blir beroende av en viss plattform.

Eftersom Emacs är utgivet av Free Software Foundation [16] under GNU General Public License (GPL) [17] är det fri mjukvara, vilket innebär att källkoden är fritt tillgänglig och vem som helst får förändra eller göra tillägg i koden. Tack vare denna frihet finns det en stor

mängd tredjepartskod som också är fri att använda och mycket av koden i Emacs standardbibliotek härstammar från kod utvecklat av personer utanför GNU Emacs projektet. GNU Emacs är också gratis mjukvara och kan installeras på godtyckligt antal datorer då ingen licens eller annan begränsning för användning krävs. I och med att tredjepartskod finns tillgänglig finns möjligheten att vid programmutveckling använda kod som inte är skriven av personer i GNU Emacs projektet. Vid utveckling av prototypen har till exempel det beskrivna CEDET använts.

Emacs Lisp är en lisp-dialekt som är anpassad för att hantera text, främst i Emacs. Denna anpassning är gjord genom att tillhandahålla datatyper som gör texthantering enkelt samt ett standardbibliotek med en stor mängd funktioner för textbearbetning. Exempel på användbar funktionalitet som tillhandahålls av standardbiblioteket är sök- och matchningsfunktioner som använder sig av reguljära uttryck, funktioner för navigering över textblock samt funktioner som ändrar visningen av text. Möjligheterna att hantera text på ett bra sätt underlättar mycket vid framtagningen av prototypen då mycket handlar om att hantera strängar och att navigera i kodtext.

Den utökning som krävs för att kunna använda Emacs som en semantisk editor införs på ett bra sätt som ett minor mode. Stöd för visning av kontrakt, som behandlas i denna uppsats, är inte en grundläggande egenskap för den text de aktuella buffrarna innehåller. Det är istället en tilläggsfunktionalitet som kan slås av och på vilket är tanken med funktionalitet implementerat som ett minor mode. De grundläggande egenskaperna för texten, vilket i detta fallet är C++ kod, kan istället hanteras av redan befintlig funktionalitet i form av ett major mode vilket gör att stöd för kontrakt kan slås av och på utan att påverka övrig funktionalitet.

GNU Emacs användargränssnitt är väldigt annorlunda jämfört med de grafiska användargränssnitten dagens vanligaste texteditorer använder sig av. Även om Emacs användargränssnitt i senare versioner har utökats med grafiska komponenter för den vanligast använda funktionaliteten, är det kommandoanrop genom inskrivning av kommandots namn eller genom dess tangentbindning som är det effektivaste sättet att arbeta i Emacs. Grafiskt baserade användargränssnitt fungerar oftast tvärt om; all funktionalitet är åtkomlig via grafiska knappar eller menyer och endast de mest använda funktionerna är åtkomliga via tangentbindningar. Detta gör att inlärningskurvan för Emacs är ganska hög jämfört med andra texteditorer och det krävs viss inlärning för att kunna använda Emacs någorlunda effektivt.

Tillsammans med utökningsmöjligheterna är användargränssnittet det som gör Emacs till en kraftfull texteditor. Men det är också i stor utsträckning användargränssnittet som gör att en del personer inte kan tänka sig att använda Emacs. En anledning är att man bör vara

komfortabel när det gäller användningen av tangentbordet och vara relativt säker på tangenters placering för att kunna arbeta enkelt i Emacs. En annan anledning är att användargränssnittet är annorlunda och helt enkelt inte passar alla personer.

5.5 Sammanfattning

I detta kapitel har den utvecklade prototypen utvärderats. Först ges en genomgång av prototypen i form av en demonstration följs av en utvärdering av de metoder som använts för att ta fram viss funktionalitet. Detta har skett stegvis och matchats mot den tidigare beskrivningen av funktionaliteten i kapitel 4.

Efter att funktionaliteten utvärderats följer ett avsnitt där användandet av editorn GNU Emacs och programspråket Emacs Lisp utvärderas. Här har både fördelar och nackdelar med användandet beskrivits.

6 Slutsats

I detta avslutande kapitel behandlas resultatet av uppsatsen som har beskrivit och motiverat framtagningen av en prototyp som ger stöd för kontraktsprogrammering i editorn GNU Emacs. Det ges även förslag på framtida arbete som kan vara till nytta vid en eventuell vidareutveckling av prototypen.

6.1 Sammanfattning

Uppsatsen har behandlat prototyputvecklingen av en semantisk editor med den befintliga editorn GNU Emacs som bas. För att motivera framtagningen av den semantiska editorn har metoden att använda kontrakt vid programutveckling beskrivits samt olika aspekter som detta medför.

Det stöd som tagits fram och är tillgängligt för användaren rör området kontraktsprogrammering och därför är det viktigt att ha kunskap inom detta område. En hypotes som presenterats i uppsatsen är att kodkvalitet kan förbättras genom att kontrakt införs på ett tidigt stadium i utvecklingsarbetet vilket kan underlättas för användaren genom det stöd som ges i den framtagna prototypen. Stöd som att till exempel kunna se kontraktet för det aktuella anropet samt att kunna infoga korrekta förvillkor underlättar arbetet för användaren.

Framtagningen av prototypen har i uppsatsen behandlats i två steg. Först har framtagningsarbetet beskrivits och detta har följts upp med en utvärdering av resultatet. Vid framtagningen har beskrivningen skett stegvis där för prototypen viktiga delar har beskrivits. Den använda textredigeraren Emacs har också beskrivits samt det använda programspråket Emacs Lisp. Användandet av Emacs har medfört att nya områden har lärts in där speciella delar av Emacs såsom modes, minibuffer och *tooltip*-rutor har undersökts och slutligen använts. Programspråket Emacs Lisp var också tvunget att läras in innan arbetet kunde ta fart.

Utvärderingen presenterades likt framtagningen stegvis. Här har prototypens funktionalitet utvärderats och även användandet av textredigeraren Emacs och vad det medför. En genomgång av programmets funktionalitet med hjälp av skärmbilder har också getts.

Den funktionalitet som tagits fram fungerar tillfredsställande. Visningen av kontrakt fungerar bra och även infogning. Vid infogning sker även parsning av förvillkoren samt översättning till aktuella parameter- och objektnamn. Även kontroll av om kontrakt är

uppfyllda fungerar bra. I leverantörskod finns möjlighet att parsa förvillkor för att avgöra om förvillkorsuttryck är exekverbara och detta fungerar tillfredsställande.

Prototypens prestanda är bra och exekveringstiden för prototypens olika delar är tillräckligt kort för att inte vara märkbar.

Skillnader mot tidigare utförda arbeten är dels att stödet är ämnat för programspråket C++. Att stödet implementerats som ett så kallat minor mode innebär även att det kan slås av och på efter användarens önskemål vilket kan vara bra då en användare kanske inte alltid önskar att ha stödet påslaget vid alla tillfällen. Hänsyn har även tagits till att olika användare har olika önskemål om hur lång tid som skall fortlöpa innan kontrakt skall visas då markören positionerats för visning. En personlig inställning kan här göras för att anpassa stödet för att passa den aktuella användaren. Denna inställning har även betydelse i prototypen genom att kontrakt inte visas då en användare skriver tillräckligt fort för att denna tid inte skall löpa ut vilket oftast innebär att användaren vet vad som skall skrivas och inte behöver hjälp i form av kontrakt. Möjligheten till att parsa förvillkorsuttryck i leverantörskod är också nytt och det är bra att kunna bekräfta att ett förvillkor som är avsett som exekverbart är korrekt utformat.

Användandet av Emacs som bas vid utvecklingsarbetet har fungerat bra. Emacs har visat sig ha bra möjligheter för att lägga till funktionalitet och anpassas efter önskemål. Genom användandet av minor mode finns även möjligheten att slå av och på den tillagda funktionaliteten efter önskemål. Att Emacs är fritt och inte anpassat för en speciell plattform ger stor tillgänglighet vilket är en fördel gentemot många andra textredigerare.

Den slutliga prototypen ger ett fungerande stöd för kontraktsprogrammering i Emacs och har god prestanda.

6.2 Framtida arbete

Arbete som återstår och som skulle kunna utföras i en framtida vidareutveckling är en utveckling av den parser som används för att parsa förvillkor. Den förvillkorsparser som används i prototypen har en enkel felhantering där det första påträffade felet anges. Det skulle vara möjligt att förbättra denna så att den blir mer tolerant och när fel påträffas kunna fortsätta parsa resterande delen av förvillkoret. På detta sätt hanteras felaktiga förvillkor bättre.

Ytterligare arbete skulle även kunna läggas på att ge användaren möjlighet till att ange personliga inställningar i form av hur indentering vid infogning av förvillkor skall se ut.

Prototypen skulle även kunna utökas för att ge ett bredare stöd för C++ då detta har begränsats i denna uppsats. Ett förslag på möjlig utökning är till exempel hantering av arv där

medlemmar i basklassen som ingår i den aktuella klassens gränssnitt kan identifieras. Ytterligare ett förslag är utökning av grammatiken för exekverbara förvillkor. I samband med utökning av grammatiken kan det också vara lämpligt att resonera kring vilken omfattning av C++-uttryck som är lämpligt att använda i förvillkor med avseende på läsbarhet och flexibilitet.

En annan aspekt som skulle kunna utvecklas vidare är att hantera klientkod och leverantörskod olika. Leverantören har till exempel tillgång till sina privata medlemmar och skyddade (eng. *protected*) medlemmar i eventuella basklasser medan klienten inte har tillgång till dessa.

6.3 Slutord

De mål som sattes upp för uppsatsen har genomförts på ett lyckat sätt. Under arbetets gång har inga stora problem uppstått utan arbetet kunde hela tiden fortlöpa enligt den tidsplan som gjordes vid arbetets början.

Det som tog mest tid jämfört med planeringen var inlärningen av GNU Emacs och programspråket Emacs Lisp som båda var helt nya områden. Inlärningen tog större delen av tiden i arbetets början och pågick sedan i mindre omfattning under större delen av den resterande tiden.

Enligt den ursprungliga planeringen skulle 40 timmar i veckan läggas ner på arbetet vilket också har gjorts. Programmering och dokumentering har skett parallellt under hela utvecklingsarbetet men med olika tidsfördelning vid olika tidpunkter i arbetet. Totalt sett uppskattas de båda momenten ha upptagit ungefär lika mycket tid.

Referenser

[1] A. Aho, R. Sethi, J.D. Ullman. *Compilers*. Addison-Wesley, 1986.

[2] T. Bergh, H. Bergmark. *Utvecklingsmiljö för Java med stöd för kontraktsprogrammering*. D-uppsats Karlstads universitet, Karlstad 2003:06.

[3] M. Blom, E. J. Nordby, A. Brunström. *Method Description for Semla.* Karlstads universitet, Karlstad 2000.

[4] M. Boshernitsan. *Harmonia: A Flexible Framework for Constructing Interactive Language-Based Programming Tools.* University of California, Berkeley, 2001.

[5] B. Eckel, C. Allison, *Thinking in C++, Vol. 2: Practical Programming, 2nd Edition*. Prentice Hall, 2003.

[6] J. Ivarsson. *Utvecklingsmiljö för java med stöd för kontraktsprogrammering under redigering*. D-uppsats Karlstads universitet, Karlstad 2004:06.

[7] R. Kramer. i*Contract the Java Design by Contract*™ *Tool*. Från *Proceedings of the TOOLS'98 Conference*, Santa Barbara, USA, 1998.

[8] B. Lewis, D. Laliberte, Gnu Manual Group, R. M. Stallman. *Gnu Emacs Lisp Reference Manual.* Free Software Foundation, 2000.

[9] B. Meyer. *Eiffel: The Language.* Prentice Hall PTR, 1991.

[10] B. Meyer. *Object-Oriented Software Construction, 2nd Edition*. Prentice Hall PTR, 2000.

[11] L. Nguyen, E. Tångring. *Javaeditor som stöder för- och eftervillkor*. C-uppsats, Karlstads universitet, 2000.

[12] J. Nielsen, *Usability Engineering*. Morgan Kaufmann, 1994.

[13] R. M. Stallman. *GNU Emacs Manual, 15th Edition*. Free Software Foundation, 2002.

[14] B. Stroustrup. *The C++ Programming Language, 3rd Edition*. Addison-Wesley, 2000.

Webreferenser

[15] Collection of Emacs Development Environment Tools
http://cedet.sourceforge.net/
(2004-05-06)

[16] Free Software Foundation
http://www.fsf.org/
(2004-04-23)

[17] GNU General Public License
http://www.fsf.org/licenses/licenses.html
(2004-04-23)

[18] Harmonia Mode Manual
http://www.cs.berkeley.edu/~harmonia/harmonia/projects/harmonia-mode/doc/
(2004-03-15)

[19] Harmonia Research Project
http://www.cs.berkeley.edu/~harmonia/harmonia/
(2004-03-15)

[20] jContractor
http://jcontractor.sourceforge.net/
(2004-06-19)

[21] Jass
http://csd.informatik.uni-oldenburg.de/~jass/
(2004-06-19)

[22] Javadoc Tool
http://java.sun.com/j2se/javadoc/
(2004-03-16)

[23] Microsoft Visual Studio
http://msdn.microsoft.com/vstudio/
(2004-05-10)

[24] NetBeans IDE
http://www.netbeans.org/
(2004-03-15)

[25] Schütz Semantic Editor
http://www.cs.wichita.edu/~rodney/scheutz/
(2004-06-21)

[26] Unified Modeling Language
http://www.uml.org/
(2004-04-14)

A Grammatik för exekverbara förvillkor

```
Subset of C++ expressions grammar
 Start symbol: expr
 Nonterminals: expr expr_rest fun_call expr_list
               expr_list_rest id number
 Terminals    : addop mulop relop letter digit

 expr           = number expr-rest
                | id fun-call expr-rest
                | '(' expr ')' expr-rest
                | addop expr
                | negop expr
                ;

 expr_rest      = addop expr
                | mulop expr
                | relop expr
                | logop expr
                | (* empty *)
                ;

 fun_call       = '('expr_list')'
                | '.'  id fun_call
                | '->' id fun_call
                | (* empty *)
                ;

 expr_list      = expr expr_list_rest
                | (* empty *)
                ;

 expr_list_rest = ',' expr_list
                | (* empty *)
                ;
```

```
id              = (letter|'_')(letter|digit|'_')*;
number          = digit+['.'digit+];
addop           = '+'|'-';
mulop           = '*'|'/' | '%';
relop           = '<'|'<='|'>'|'>='|'=='|'!=';
letter          = 'a'|'b'|'c'| ... |'z'|'A'|'B'|'C'| ... |'Z';
digit           = '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9';
```

B Användarmanual

Vid användandet av det semantiska stödet antas användaren ha grundläggande kunskap om editorn GNU Emacs. Det semantiska stödet för GNU Emacs är implementerat i form av ett så kallat minor mode vilket innebär att stödet kan slås till och från av användaren under körning efter önskemål. Det stöd som ges är i form av:

- Visning av kontrakt för funktioner
- Möjlighet till infogning av kontroller för kontrakt
- Möjlighet att spåra om en funktion har kontrollerat sitt förvillkor före anrop utförs

För att anpassa stödet kan även personliga inställningar göras via en inställningspanel. Hur dessa inställningar utförs beskrivs vidare i avsnitt B.8.

B.1 Att utföra kommandon i GNU Emacs

För att utföra vissa saker i Emacs används så kallade kommandon. I fallet med kontraktstöd för GNU Emacs används kommandon för att slå på eller av stödet samt för att nå inställningspanelen. För att kunna utföra dessa kommandon är det därför viktigt att veta hur ett kommando utförs. Användaren trycker ner tangentkombinationen Alt-X varpå det aktuella kommandot matas in. Vid inmatning kan användaren se inmatning av kommando i Emacs minibuffer.

B.2 Att ladda stödet

För att stödet skall vara tillgängligt i GNU Emacs måste det först laddas vilket enklast görs med hjälp av Emacs initieringsfil ”.emacs”. Filen ”.emacs” återfinns vanligast i den aktuella hemkatalogen. Innan Emacs startas upp skall följande rad läggas till sist i ”.emacs” för att stödet skall laddas vid uppstart.

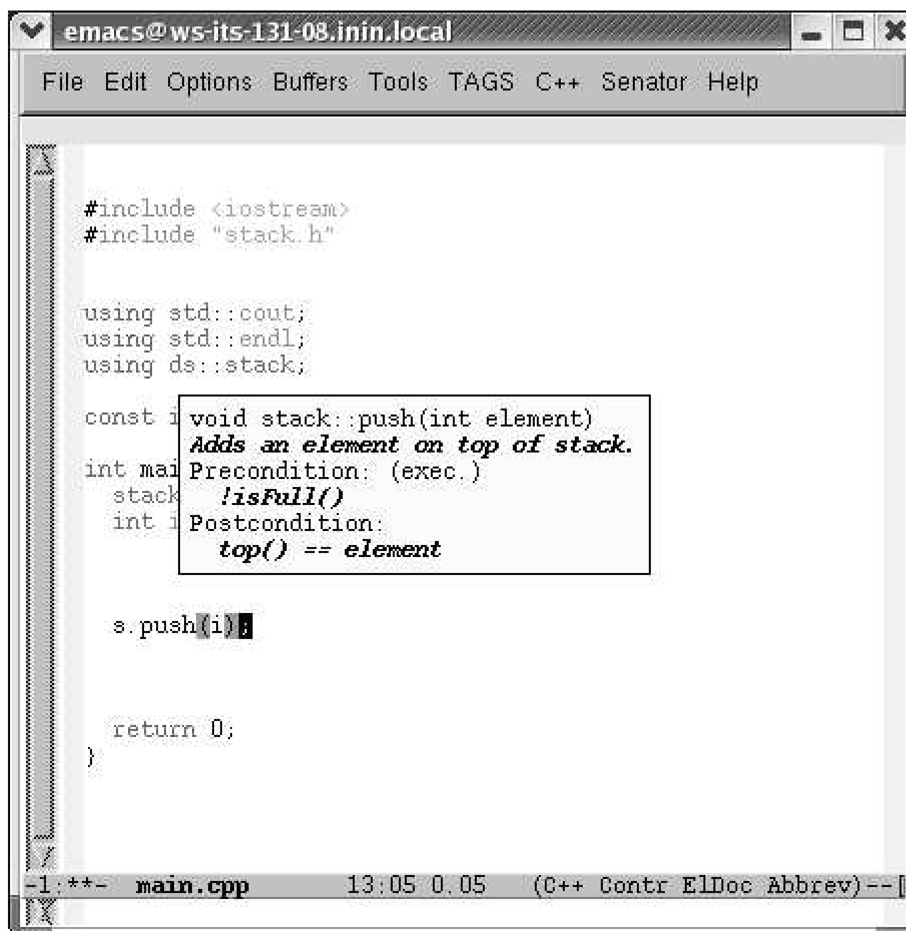
```
(require 'contract-load (expand-file-name "~/code/contract-load.el"))
```

Sökvägen till filen”contract-load.el” måste ändras för att stämma överens med den aktuella sökvägen. Sökvägar till övriga filer i prototypen (se bilaga C) samt till CEDET som finns i

filen ”contract-load.el” måste även de ändras till aktuella sökvägar. För att stödet skall fungera måste CEDET:s filer finnas tillgängligt på den aktuella datorn.

B.3 Att slå på och av stödet

Stödet slås på och av genom kommandot contract-mode. Då stödet antingen kan vara på eller av innebär det att då kommandot används byter stödet tillstånd från antingen av till på, eller på till av. Då stödet är påslaget kan detta ses i Emacs *mode line* enligt Figur B.1.



Figur B.1: Emacs mode line anger att Contract-mode är påslaget (Contr).

B.4 Utformning av kontrakt

För att ett kontrakt skall kunna tolkas på ett korrekt sätt är det viktigt att det utformas enligt vissa direktiv. Hur kontrakten skall utformas kan ses nedan.

```
/*
 *  Här skrivs en kort beskrivningen av funktionen
 *
 *  @pre  Här skrivs förvillkor
 *  @post Här skrivs eftervillkor
*/
Här följer funktionen
```

Ett exempel på hur detta kan se ut följer nedan.

```
/*
 * Adds a non-negative integer to the top of the stack.
 *
 * @Pre   !isFull()
 *        element > 0
 * @Post  !isEmpty()
 */
void stack::push( int element )
{
  ...
}
```

I exemplet ovan skrivs förvillkoret på två rader. Detta för att det kan öka läsbarheten vid långa förvillkor samt att många användare är vana vid att lista upp förvillkoret på detta sätt. Vid infogning av förvillkor tolkas detta som ett logiskt och (&&) mellan raderna vilket ger:

```
!isFull() && element > 0
```

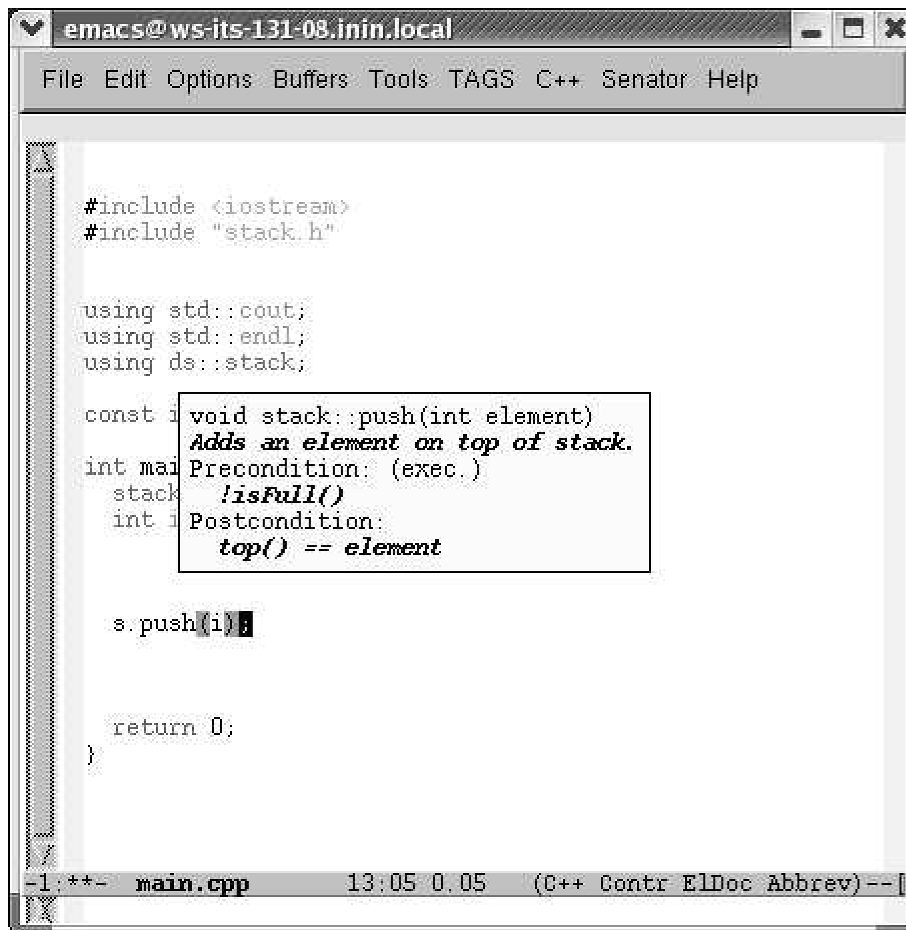
Då detta kan anses som att förvillkoret inte är skrivet som ett exekverbart uttryck är det även möjligt att skriva ut och-tecken (&&) i förvillkoret. Vilken variant som användaren vill använda sig av går att ställa in via inställningspanelen som beskrivs senare i manualen.

Exempel på utformningen av förvillkor då varianten där användaren skriver ut och-tecken kan ses nedan:

```
/*
 * Adds a non-negative integer to the top of the stack.
 *
 * @Pre   !isFull() && element > 0
 * @Post  !isEmpty()
 */
void stack::push( int element )
{
  ...
}
```

B.5 Visning av kontrakt för funktioner

Det semantiska stödet ger användaren möjlighet till att få information om funktioner i form av kontrakt visade. Kontraktet kan visas dels i form av en *tooltip*-ruta och dels i Emacs minibuffer. Att dessa olika alternativ finns beror på att vissa versioner av Emacs inte stöder *tooltip*-rutor. Då stöd finns för *tooltip*-rutor används detta automatiskt medan minibuffern används då inget stöd finns. Hur alternativet med *tooltip*-ruta ser ut visas i Figur B.2 medan alternativet med minibuffern visas i Figur B.3.



Figur B.2: Exempel där kontrakt visas med en tooltip ruta.

```
emacs@ws-its-131-08.inin.local
File  Edit  Options  Buffers  Tools  TAGS  C++  Senator  Help

#include <iostream>
#include "stack.h"


using std::cout;
using std::endl;
using ds::stack;

const int STACK_SIZE = 10;

int main() {
  stack s(STACK_SIZE);
  int i = 0, j;

  if(!s.isFull()) {
    i = i + i;

    if(!s.isEmpty())
      cout << s.top();

    s.push(i);
  }


  return 0;
}
-1:**-  main.cpp       13:29 0.02    (C++ Contr ElDoc Abbrev)--[
void stack::push(int element)
Adds an element on top of stack.
Precondition: (exec.)
  !isFull()
Postcondition:
  top() == element
```

Figur B.3: Exempel där kontrakt visas i minibuffern.

För att ett kontrakt för en funktion skall visas krävs, förutom att ett kontrakt är skrivet, att markören placeras på bestämda positioner av det anrop där information önskas. Dessa positioner kan ses i Figur B.4.

s . push(element) ;

= detektering av funktionsanrop

Figur B.4: Positioner som resulterar i anropsdetektering.

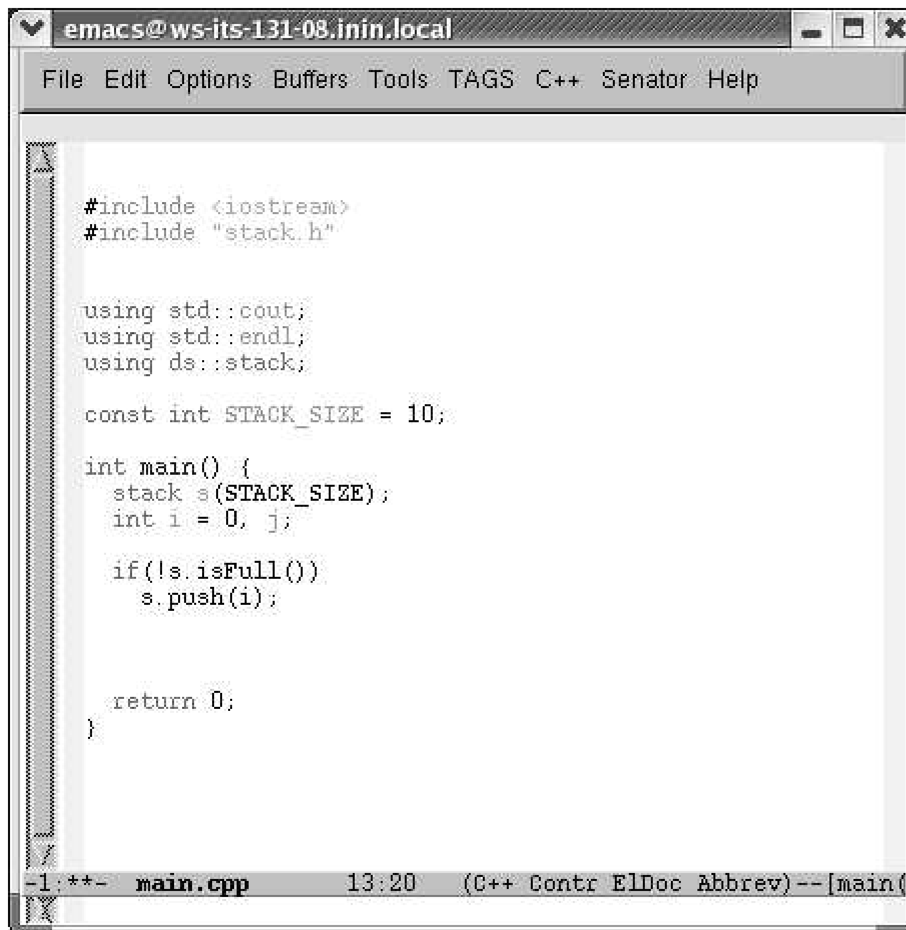
Dessa positioner gäller alltid för att få information om en speciell funktion, det vill säga även för en funktion som används som argument vid ett anrop.

För att kontraktet skall visas krävs att markören står placerad på en korrekt position under en viss tid. Denna tid har ett förinställt standardvärde på en sekund men är möjlig att ändra om mer eller mindre tid skulle önskas.

B.6 Infogning av kontroll av förvillkor

För att en kontroll av ett förvillkor skall kunna infogas krävs det att det skrivna förvillkoret är exekverbart vilket innebär att förvillkoret är syntaktiskt korrekt enligt grammatiken i bilaga A. Detta kontrolleras innan infogning sker och det är därför endast möjligt att infoga kontrakt som är exekverbara. Då ett kontrakt visas indikeras det om förvillkoret är exekverbart genom att (exec.) skrivs ut före förvillkoret vilket kan ses i Figur B.2.

Infogning sker genom tangentkombinationen Ctrl-c följt av ”_”. Då denna kombination utförs kommer en kontroll av förvillkoret att infogas före det aktuella anropet i de fall då förvillkoret är exekverbart. Ett exempel på hur detta kan se ut kan ses i Figur B.5.

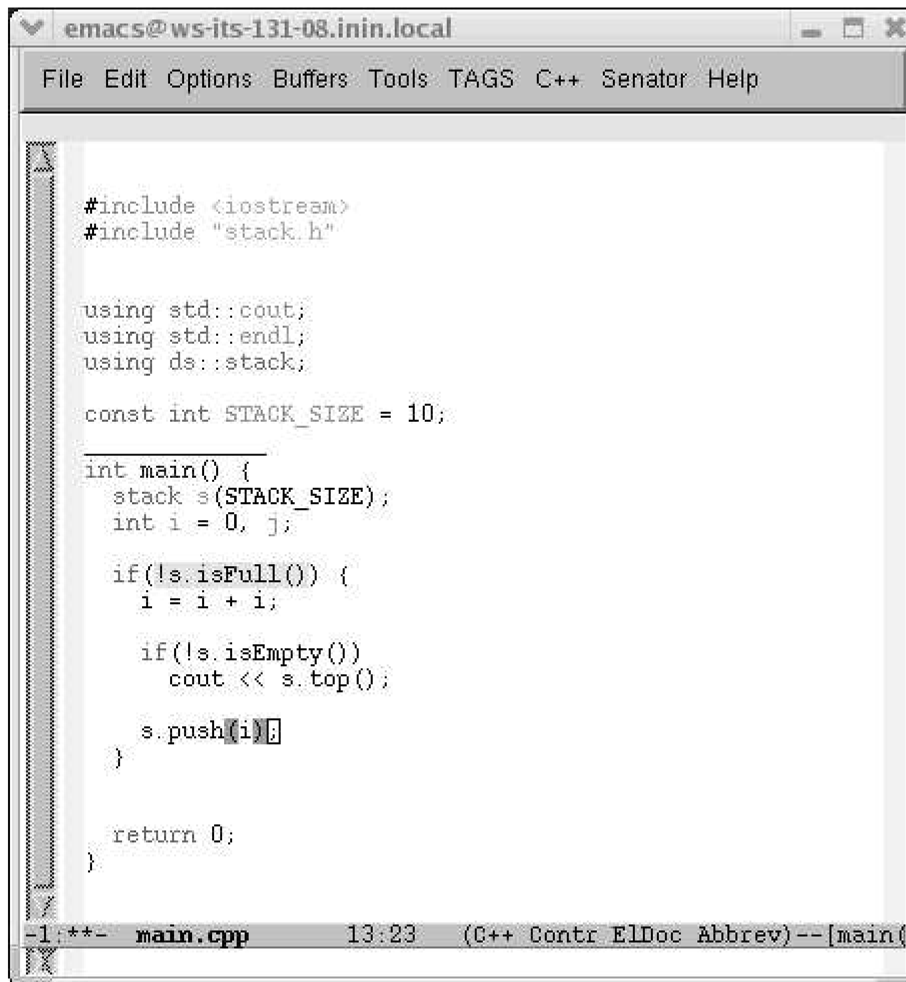


Figur B.5: Infogning av test om förvillkor är uppfyllt.

Vid infogning kommer även de angivna argumentnamnen i de skrivna förvillkoren att automatiskt översättas till de faktiska argumentnamn som används i koden.

B.7 Att spåra om en funktion uppfyller sitt förvillkor

För att undersöka om en funktion uppfyller sitt förvillkor eller inte ges möjligheten för användaren att få eventuellt utförd kontroll markerad. För att starta denna kontroll placeras markören på någon av de positioner som anges i Figur B.4. Då markören placeras på någon av dessa positioner söks den närmast föregående kontrollen som uppfyller förvillkoret upp och markeras. Hur denna markering ser ut kan ses i Figur B.6.



Figur B.6: Närmast föregående kontroll av förvillkor markeras.

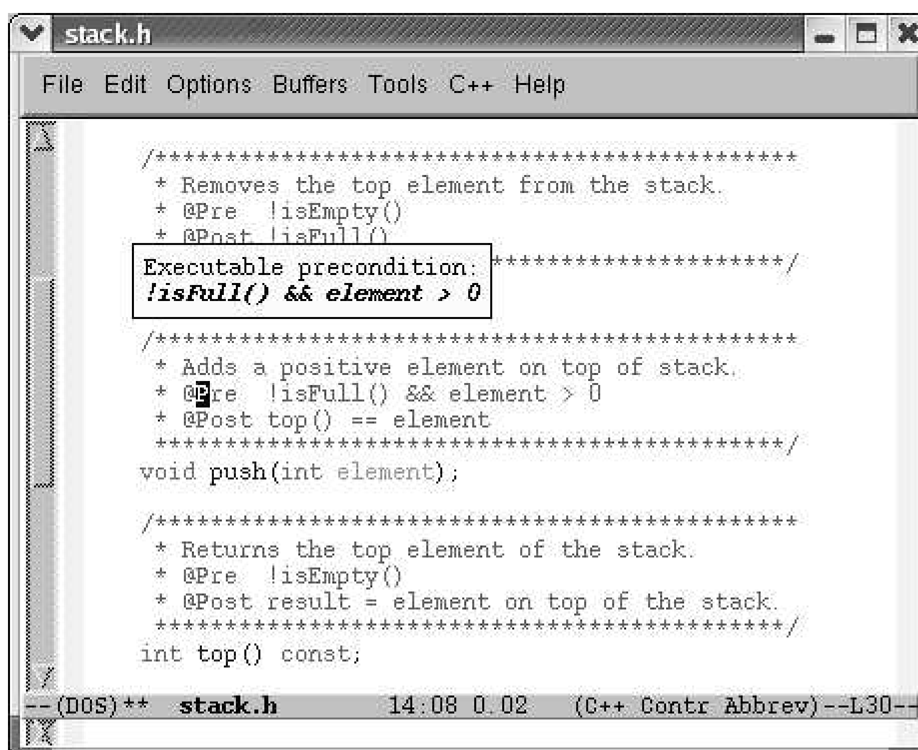
B.8 Parsning av förvillkor

För att undersöka om ett förvillkor som är menat att vara exekverbart är syntaktiskt korrekt är det möjligt att parsa förvilkoret. Genom att placera markören på någon av de positioner som anges i Figur B.7 startar parsningen av förvillkoret.

```
/**********************************
 * Adds an element to the top
 * of the stack.
 *
 * @pre  !isFull()
 * @post top( ) == element
 *********************************/
void stack::push( int element );
```

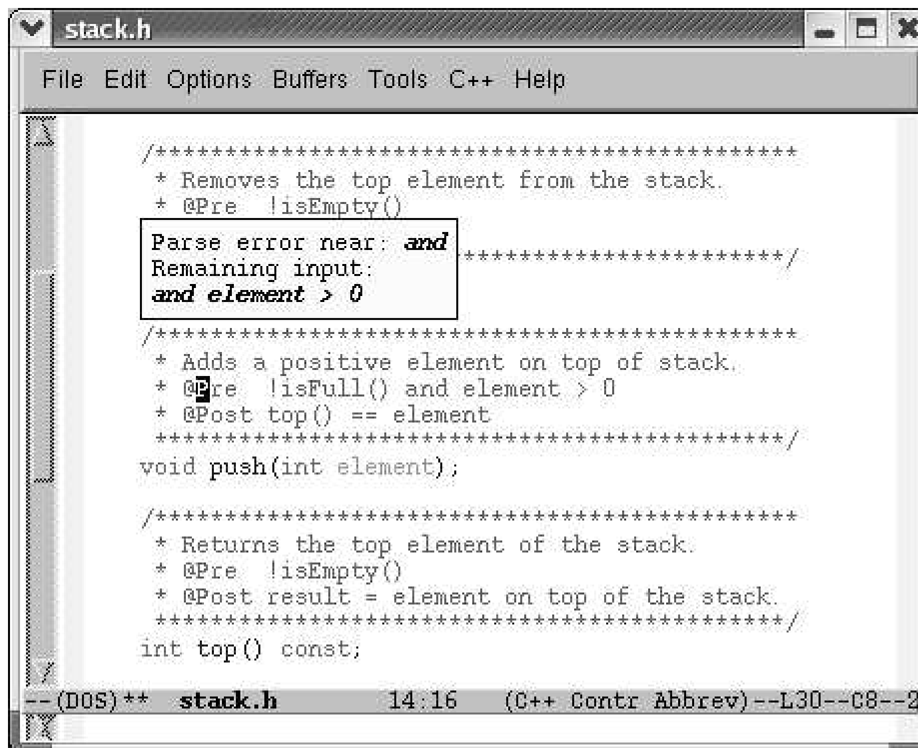
Figur B.7: Markörpositioner som leder till parsning av förvillkor.

Resultatet visas med hjälp av en *tooltip*-ruta i det aktiva Emacs-fönstret. Ett exempel på en lyckad parsning kan ses i Figur B.8.



Figur B.8: Syntaktiskt korrekt exekverbart uttryck.

Då en parsning misslyckas visas ett felmeddelande i *tooltip*-rutan. Felmeddelandet anger det första påträffade felet i förvillkoret. Ett exempel på detta kan ses i Figur B.9.



Figur B.9: Ej exekverbart förvillkorsuttryck.

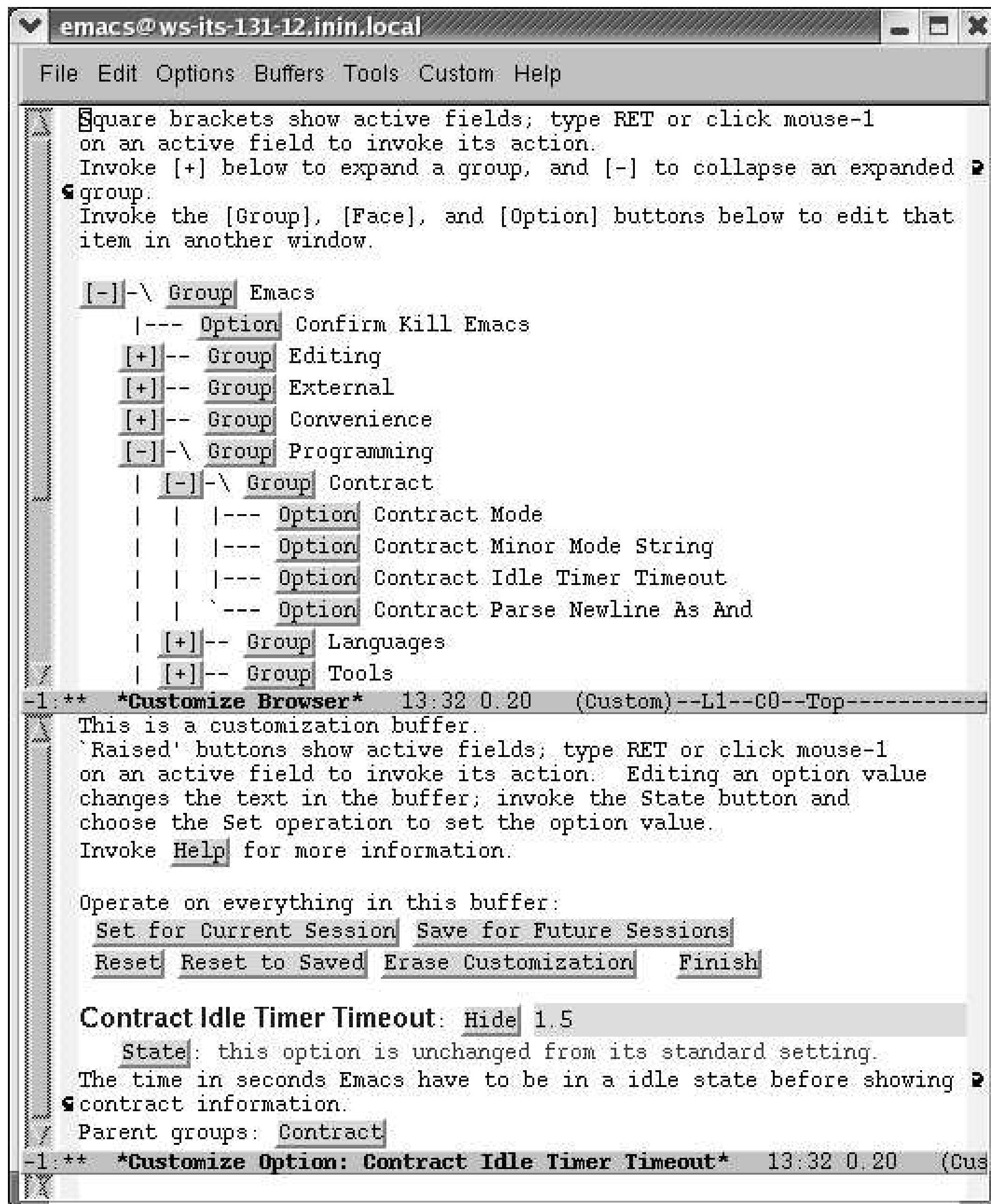
B.9 Inställningspanel

För att utföra personliga inställningar i Emacs finns en inställningspanel som nås genom kommandot customize-browse. Då detta kommando angetts dyker inställningspanelen upp i det aktiva fönstret i Emacs. Den panel som dyker upp vid detta tillfälle kan ses i det övre fönstret i Figur B.10. För att nå inställningarna för contract-mode öppnas först gruppen Emacs följt av gruppen programming och gruppen contract. Detta läge representeras av en trädrepresentation som kan ses i det övre fönstret i Figur B.10. Användaren har här möjlighet till fyra olika personliga inställningar. Dessa fyra inställningar är:

- Contract-mode. Här kan contract mode stängas av och sättas på.
- Contract Minor Mode String. Här anges den sträng som skall visas i Emacs *mode line* då contract mode är påslaget.
- Contract Idle Timer Timeout. Här anges värdet för timern.
- Contract Parse New Line As And. Här anges om ny rad skall tolkas som logiskt och.

För att exemplifiera ett speciellt fall har inställning av timers värde valts ut. I detta fall väljs ”Contract Idle Timer Timeout” ur trädrepresentationen. Då detta är gjort är inställning av

timerns värde möjligt och hur detta ser ut kan ses i det undre fönstret i Figur B.10. För att anpassa värdet ändras den angivna tiden som i det exemplifierade fallet är satt 1,5 sekunder.



Figur B.10: Inställningspanel för Contract-Mode.

B.10 Standardvärden för inställningar

Inställning	Standardvärde
• Contract Mode	False
• Contract Minor Mode String	Contr
• Contract Idle Timer Timeout	1 sek
• Contract Parse NewLine As And	Off

C Systembeskrivning

Prototypens kod har dokumenterats genom funktions- och variabelbeskrivningar vid definitionerna. Denna dokumentation beskriver hur respektive funktion och variabel skall användas och det är denna dokumentation som syns i GNU Emacs hjälpsystem. Koden i varje funktion har också dokumenterats för att göra det enklare att förstå vad som uträttas i olika delar av koden. Denna systembeskrivning beskriver prototypens olika delar översiktligt samt vilken roll de har i systemet. Verktyg som används i prototypen tas också upp.

C.1 GNU Emacs

Prototypen har utvecklats i GNU Emacs version 21.2.1 för Linux och version 21.1.1 för Windows. Framtida versioner av Emacs kommer med stor sannolikhet att kunna användas tillsammans med prototypen eftersom bakåtkompatibilitet är något eftersträvas vid utvecklingen av Emacs. Äldre versioner än 21 har inte testats men eftersom inget krav på visning med grafiska komponenter finns i prototypen är det inte omöjligt att även de versionerna fungerar.

C.2 CEDET

Prototypen använder sig av Collection of Emacs Development Environment Tools (CEDET) 1.0 beta 1c. I denna CEDET-version ingår Semantic 2.0 beta 1 som innehåller funktionalitet för inkrementell (eng. *incremental*) parsning av källkod samt sökning i den symboltabell som parsningen resulterar i. En stor skillnad mellan tidigare versioner av Semantic och den som används i denna uppsats är namngivningen av funktioner och variabler som har ändrats. Då namnen i den äldre versionen gav en dålig beskrivning av användningen har en stor del av namngivningsstandarden gjorts om. De gamla namnen finns dock fortfarande kvar vilket kan vara förvirrande. Vid utvecklingen av prototypen har de nya funktions- och variabelnamnen använts i så stor utsträckning som möjligt eftersom prototypen på så vis skall kunna användas tillsammans med nyare versioner av Semantic.

C.2.1 Parser

I prototypen används en inkrementell parser för C/C++ som ingår i Semantic. Med inkrementell parsning menas att källkoden inte antas vara syntaktiskt korrekt eller fullständig

utan information till symboltabellen sammanställs så gott det går. Den inkrementella parsningen arbetar i bakgrunden och är inte speciellt krävande då endast modifierad eller ny kod behandlas mellan parsningarna. Symboltabellen byggs på så vis upp och uppdateras utan att det stör användaren.

Semantic har också funktioner som används för att söka i symboltabellen som skapas och uppdateras vid parsning.

C.2.2 Symboltabell

Den symboltabell som skapas och uppdateras av den inkrementella parsern är mycket detaljerad och byggs upp med hjälp av nästlade listor. Exempel på information som lagras i symboltabellen är namn på klasser, variabler, funktioner och namnutrymmen. Vidare lagras attribut till namn som till exempel returtyper, argumentlistor, variabeltyper och basklasser. Även räckvidd (eng. scope) lagras i symboltabellen genom nästling av listor och bufferpositioner för symboler som finns i filer som är öppna i Emacs lagras. Då informationen som lagras i symboltabellen är mycket grundlig blir symboltabeller mycket stora. Nedan ges ett exempel på hur en symboltabell ser ut för en klass som deklarerats i ett namnutrymme och som innehåller två medlemsfunkioner samt en medlemsvariabel.

```
namespace myspace {

  class myclass {

  public:
    void f(int i);
    int  g();

  private:
    int value;

  };
}
```

Symboltablell:

```
(
("myspace" type
(:members
 (
 ("myclass" type
 (:members
  (
  ("public" label nil (reparse-symbol classsubparts)
    #<overlay from 44 to 51 in test.h>)
  ("f" function (prototype t :arguments
    (
      ("i" variable (:type "int") (reparse-symbol arg-sub-list)
        #<overlay from 61 to 67 in test.h>)
    ) :type "void")
    (reparse-symbol classsubparts)
    #<overlay from 54 to 68 in test.h>
  )
  ("g" function (prototype t :type "int")
    (reparse-symbol classsubparts)
    #<overlay from 70 to 78 in test.h>)
  ("private" label nil (reparse-symbol classsubparts)
    #<overlay from 82 to 90 in test.h>)
  ("value" variable (:type "int") (reparse-symbol classsubparts)
    #<overlay from 92 to 102 in test.h>)
  )
  :type "class"
 )
 (reparse-symbol namespacesubparts)
 #<overlay from 25 to 108 in test.h>
 )
 )
:type "namespace") nil #<overlay from 2 to 110 in test.h>)
)
```

Även om det till och med är svårt att överblicka liststrukturen i det korta exemplet ovan är det viktigt att förstå den grundläggande principen hos symboltabellen samt dess uppbyggnad. När symboltabellen används i prototypen används funktioner i Semantic för att söka efter namn i symboltabeller samt funktioner som plockar ut specifik information som till exemel en funktions returvärde eller parameterlista.

C.3 Översikt

Prototypen är implementerad som ett minor mode som kallas Contract-Mode. Källkoden för Contract-Mode är uppdelad på följande filer:

- **contract-load.el**
  Innehåller initieringskod som laddar contract-mode och koden som contract-mode är beroende av.
- **contract-mode.el**
  Denna fil innehåller grundfunktionaliteten för contract-mode samt de funktioner och variabler som utgör själva modet.
- **contract-parse.el**
  Denna fil innehåller funktioner och variabler som används vid parsning av förvillkor.
- **contract-utils.el**
  Innehåller generella funktioner som inte har någon direkt anknytning till funktionaliteten som tillhandahålls av Contract-Mode men används av funktioner i de övriga filerna.

C.4 Beskrivning

Här följer en beskrivning över viktiga funktioner och aspekter som finns i Contract-Modes huvudfiler contract-mode.el och contract-parse.el.

C.4.1 contract-mode.el

Denna fil innehåller kod som gör att prototypen fungera som ett *minor mode* samt kod för den grundläggande funktionaliteten hos prototypen. Mode delen består av följande delar:

- Mode-variabeln *contract-mode*, som anger om modet är på- eller avslaget.
- Mode-strängen *contract-minor-mode-string*, som är den sträng som visas i Emacs *mode line* när Contract Mode är påslaget.
- Kommandot *contract-mode*, som slår av och på Contract-Mode.
- Funktionen *turn-on-contract-mode* som är ett enklare sätt att slå på Contract-Mode i kod.

Prototypens grundfunktionalitet sköts av funktionen *contract-show-info* som är den funktion som anropas när Emacs inaktivitetstimer har löpt ut. Denna funktion anropar direkt eller indirekt större delen av de övriga funktionerna och samlar in tillgänglig information som sedan visas för användaren. En liknande funktion som anropas via ett snabbkommando är *contract-paste-precondition* som infogar en förvillkorskontroll om möjligt. För att få en överblick över koden som utgör prototypen är det dessa två funktioner som är utgångspunkter för prototypens funktionalitet.

C.4.2 contract-parse.el

Denna fil innehåller kod som hanterar parsning och översättning av förvillkor. Parsern är en *recursive descent predictive* parser som analyserar förvillkors syntaktiska korrekthet enligt grammatiken i bilaga A. Översättning av förvillkor som till exempel skall infogas i klientkod görs av parsern genom att en associativ lista med formella och faktiska parametrar tillsammans med ett eventuellt objektnamn skickas som parameter till funktionen som utgör parserns gränssnitt. Nedan visas ett exempel på en parsning med översättning.

```
(parse "a < b && isOK()" '(("a" . "c") ("b" . "d")) "obj")
Resultat: ”c < d && obj.isOK()”
```

De två sista argumenten till parse-funktionen, argumentlistan och objektnamnet, är valfria. Utnyttjas inte de valfria argumenten sker ingen översättning utan endast syntaktisk analys av förvillkoret. Det finns också möjlighet att ange ett fjärde argument, som skall vara icke-falskt, för att parsern skall returnera ett felmeddelande om ett förvillkor inte har korrekt syntax. Nedan visas ett exempel på ett sådant felmeddelande.

```
(parse "a < b isOK()" '(("a" . "c") ("b" . "d")) "obj" t)
Resultat:
```

```
"Parse error near: isOK
Remaining input:
isOK ( )"
```

Parsern använder sig av en lexikalanalysator som med hjälp av reguljära uttryck genererar en lista av tokens som representerar det aktuella förvillkorets uppbyggnad. En token-lista innehåller par av token och dess lexeme. Nedan visas hur en token-lista är uppbyggd samt ett exempel.

```
((TOK1 . LEX1) (TOK2 . LEX2) ... (TOKn . LEXn))
“a + b” ger:
((:id . ”a”) (:addop ”+”) (:id ”b”))
```

D Källkod

Här följer den källkod som utgör prototypen som har utvecklats i denna uppsats.

D.1 contract-load.el

```
;;; contract-load.el
;;; Peter Labraaten & Stefan Larsson
;;; Loads CEDET and Contract Mode support
;;; Initiation code
(add-to-list 'load-path (expand-file-name "~/dupp/cedet/semantic"))
(add-to-list 'load-path (expand-file-name "~/dupp/cedet/common"))
(add-to-list 'load-path (expand-file-name "~/dupp/cedet/eieio"))
(add-to-list 'load-path (expand-file-name "~/dupp/cedet/speedbar"))
(setq semantic-load-turn-everything-on t)
(require 'semantic-load)

(require 'contract-mode  (expand-file-name "~/dupp/contract-mode.el"))
(require 'contract-parse (expand-file-name "~/dupp/contract-parse.el"))
(require 'contract-utils (expand-file-name "~/dupp/contract-utils.el"))

(provide 'contract-load)
```

D.2 contract-mode.el

```
;;; contract-mode.el
;;; Peter Labraaten & Stefan Larsson
;;; Main functionality of Contract Mode

;;; This section defines variables that is part of Contract Mode.

;; Contract mode group.
(defgroup contract nil
  "Handles programming by contract."
  :group 'programming)

;; The buffer local mode variable for contract mode.
(defcustom contract-mode nil
```

```
  "*If non-nil, Contract Mode is enabled."
  :type 'boolean
  :group 'contract)
(make-variable-buffer-local 'contract-mode)

;; String to show in the mode line when contract mode is enabled.
(defcustom contract-minor-mode-string " Contr"
  "*String to display in mode line when Contract Mode is enabled."
  :type 'string
  :group 'contract)
(add-to-list 'minor-mode-alist '(contract-mode contract-minor-mode-string))

(defcustom contract-idle-timer-timeout 1.0
  "*The time in seconds Emacs have to be idle before showing contract
info."
  :type 'number
  :group 'contract)

(defvar contract-keymap nil "Keymap used by Contract-Mode.")

;; Key map used by contract mode.
(if contract-keymap
  nil
  (setq contract-keymap (make-sparse-keymap))
  (define-key contract-keymap "\C-c_" 'contract-paste-precondition))
(add-to-list 'minor-mode-map-alist (cons 'contract-mode contract-keymap))

;; Variable that holds the ID of the used timer.
(defvar contract-idle-timer nil "")

;; Contract Mode enable/disable command
(defun contract-mode (&optional prefix)
  "*Enable or disable contract mode.

If called interactively with no prefix argument, toggle current condition
of the mode.
If called with a positive or negative prefix argument, enable or disable
the mode, respectively."
  (interactive "P")
  (setq contract-mode
    (if prefix
```

```
      (> (prefix-numeric-value prefix) 0)
      (not contract-mode)))
  (if contract-mode
      (progn
    ;; Setup idle timer if not already active.
    (or (memq contract-idle-timer timer-idle-list)
      (setq contract-idle-timer
          (run-with-idle-timer contract-idle-timer-timeout
                     t
                     'contract-show-info))))
  (if contract-idle-timer
    ;; Remove timer
    (progn
      (cancel-timer contract-idle-timer)
      (setq contract-idle-timer nil))))
  (and (interactive-p)
     (message "contract-mode %s" (if contract-mode "enabled" "disabled")))
  contract-mode)

;; Convenient function for turning contract mode on.
(defun turn-on-contract-mode ()
  "Unequivocally turn on contract-mode."
  (interactive)
  (contract-mode 1))

(defun contract-show-info () (interactive)
  "The entry function of the contract mode core functionality.
This function will display contract information for the function call at
point, if such information is available. If a precondition check is found
in the execution path of the function call, the check will be highlighted."

  ;;(if contract-mode ...

  (let (function-info-list ; used for temporary storage of gathered info
    object
    function
    arg-list
    arg-list-string-list
    doc              ; list of function documentation:
                   ; (DESCRIPTION PRECOND POSTCOND)
    exec-pre
```

```
  tag
  display-string          ; string to be displayed
  (old-point (point)))

(save-excursion

 ;; Get function call info if point is located at a function call.
    (if (contract-function-call-p)
    (progn
    ;; Get the available function info.
    (setq function-info-list (contract-function-call-info))
    (setq object   (nth 0 (car function-info-list)))
    (setq function (nth 1 (car function-info-list)))
    (setq arg-list (nth 2 (car function-info-list)))
    (setq tag (cdr function-info-list))

    ;; Get documentation info like description, precondition
    ;; and postcondition
    (if (contract-get-comment tag)
        (setq doc (contract-filter-comment
             (contract-get-comment tag))))

    ;; If there is a precondition, parse it.
    (if (nth 1 doc)
        (setq exec-pre
          (parse (nth 1 doc)
             ;; construct alist used in translation
             (contract-merge-lists
              (mapcar (lambda (x) (semantic-tag-name x))
                  (semantic-tag-function-arguments tag))
              (mapcar 'cdr arg-list))
             (cdr object))))

    ;; Construct string to display.
    (and
     (car function)
     (setq display-string
         (concat display-string (car function) " "))
     (car object)
     (setq display-string
         (concat display-string (car object) "::")))
```

```
    ;;(contract-call-by-pointer-p)

    (setq display-string
        (concat display-string
            (cdr function)
            "("
            (contract-arg-list-string tag)
            ")"
            "\n" (nth 0 doc)
            "\n" "Precondition: "
            (if exec-pre
                "(exec.)\n  "
              "\n  ")
            (if (nth 1 doc) (nth 1 doc) "True")
            "\n" "Postcondition:\n  " (nth 2 doc)
            "\n"))

    ;; Display gathered contract information.
    (contract-display display-string)

    ;; Highlight precondition check if there is one.
    (let (region)
      (and exec-pre
         (setq region (contract-check-precondition exec-pre))
         (and region)
         (contract-highlight-region (nth 0 region)
                      (nth 1 region))
         (add-hook 'pre-command-hook
             'contract-unhighlight-region nil t))))

;; Point isn't located at a function call. Check if point is located
 ;; at a precondition definition and display the precondition  parse
  ;; information.
  (let (comment-pos output-string)
    (goto-char old-point)
    (if (setq comment-pos (contract-point-at-precondition))
      (progn
      ;; Parse the precondition if found.
      (if (parse (nth 1 (contract-filter-comment
                 (buffer-substring (car comment-pos)
```

```
                          (cdr comment-pos))))
             nil nil)
          (setq output-string "Executable precondition:\n"))

        ;; Build string to be displayed.
        (setq output-string
            (concat output-string
                (parse
                 (nth 1 (contract-filter-comment
                     (buffer-substring
                    (car comment-pos)
                    (cdr comment-pos))))
                 nil nil t)))))

      ;; Display parse info.
      (if output-string
        (contract-display output-string)))))))

(defun contract-point-at-precondition ()
  "Returns a cons cell containing the begin and end locations of the
comment header if it contains a '@pre' tag. Otherwise non-nil is returned."
  (save-excursion
  (let (comment-beg-pos comment-end-pos)

    (if (or
       (looking-at "@[Pp]re")
       (save-excursion
       (goto-char (1- (point)))
       (looking-at "@[Pp]re"))
       (and (condition-case nil (progn (backward-sexp) t) (error nil))
        (goto-char (1- (point)))
        (looking-at "@[Pp]re")))
      (and
       (re-search-backward "/\\*" nil t nil)
       (setq comment-beg-pos (point))
       (re-search-forward "\\*/" nil t nil)
       (setq comment-end-pos (point))))

    (if (and comment-beg-pos comment-end-pos)
      (cons comment-beg-pos comment-end-pos)))))
```

```
(defun contract-check-precondition (precond)
  "Checks weather there is a check for the precondition PRECOND in the
current path of execution. Return a list containing the  begin and end
location of the precondition check if one is found. Else return 'nil'"
  (interactive "s")
  (save-excursion
    (let (beg end found)
      (backward-sexp 2)
      (or
       (save-excursion
     ;; First check for an if statement immediately before point.
     (if (and
        (condition-case nil (progn (backward-sexp) t) (error nil))
        (if (looking-at "[a-zA-Z_][a-zA-Z0-9_]*[ \t\n]*\\(\\.\\|->\\)")
          (condition-case nil (progn (backward-sexp) t) (error nil))
        t)
        (condition-case nil (progn (backward-sexp) t) (error nil))
        (looking-at "if[ \t\n]*("))
       (contract-preconditions-equal
        precond
        (buffer-substring
         (progn (forward-sexp) (setq beg (1+ (point))))
         (progn (forward-sexp) (setq end (1- (point))))))))

     ;; Move up one scope. Check for if statement. Repeat until no scope
     ;; left or a  matching precondition check found.
       (while (and (not found) (not (semantic-up-context)))
     (setq found
         (and
        (condition-case nil (progn (backward-sexp 2) t) (error nil))
        (looking-at "if[ \t\n]*(")
        (contract-preconditions-equal
         precond
         (buffer-substring
          (progn (forward-sexp) (setq beg (1+ (point))))
          (progn (forward-sexp) (setq end (1- (point))))))))))

       (if (not found) (setq beg nil end nil)))
    (if (and beg end) (list beg end) nil))))

(defun contract-preconditions-equal (precond1 precond2)
```

```
  "Compare if the two preconditions PRECOND1 and PRECOND2 are equal.
Current implementation returns non-nil if the preconditions are
string equal."
  (string-equal precond1 precond2))

(defun contract-paste-precondition ()
  "This function pastes a precondition test sourrounding the current
function call. If point is located at a function call, an if-statement
using a translated version of the functions precondition as the if
condition."
  (interactive)
  (let (function-info-list
    object function
    arg-list
    arg-list-string-list
    doc
    tag
    precond)

    (if (not (contract-function-call-p))
    nil
    (setq function-info-list (contract-function-call-info))
    (setq object   (nth 0 (car function-info-list)))
    (setq function (nth 1 (car function-info-list)))
    (setq arg-list (nth 2 (car function-info-list)))
    (setq tag (cdr function-info-list))

    (if (contract-get-comment tag)
      (setq doc (contract-filter-comment (contract-get-comment tag))))

    ;; Get translated and successfully parsed precondition.
    (setq precond (parse (nth 1 doc)
               (contract-merge-lists
              (mapcar (lambda (x) (semantic-tag-name x))
                  (semantic-tag-function-arguments tag))
              (mapcar 'cdr arg-list))
               (cdr object) nil))

    ;; Construct if-statement and paste around function call.
    (and precond
       (save-excursion
```

```
        (let ((length (point)))
          (backward-sexp)
          (backward-sexp)
          (goto-char (1- (point)))

          (if (looking-at ">")
            (goto-char (1- (point))))
          (if (looking-at "\\.\\|->")
            (backward-sexp)
          (goto-char (1+ (point))))

          (setq length (- length (point)))
          (insert "if(")
          (insert precond)
          (insert ")\n")
          (indent-region (point) (+ (point) length) nil)))))))

(defun contract-call-by-pointer-p ()
  "Returns non-nil if a memeber function is called using the pointer
memeber access operator."
  (and (condition-case nil (backward-sexp 2) (error nil))
      (goto-char (1- (point)))
      (looking-at ">")
      (goto-char (1- (point)))
      (looking-at "-")))

(defun contract-function-call-p ()
  "Returns non-nil if point is located at a function call.
If there is a function call, move point to right after the closing
parenthesis. Note: No control of name correctness, e.g. '6foo' could
pass as a name."
  (let ((old-point (point)))
    (or
      ;; Check if point is looking at a function call's opening
    ;; parenthesis or on a whitespace character sequence preceding one.
      (and
       (if (looking-at "[ \t\n]")
       (re-search-forward "[ \t\n]+" nil t)
    t)
       (looking-at "(")
       (prog2 (re-search-backward "[^ \t\n][ \t\n]*" nil t)
```

```
      (looking-at "[a-zA-Z0-9_]")
    (goto-char (1+ (point))))
      (condition-case nil
      (not (forward-sexp))
    (error nil)))

     (not (goto-char old-point)) ;restore point's position for next check.

     ;; Check if point is located on a function name.
     (and
      (looking-at "[a-zA-Z0-9_]")
      (re-search-forward "[a-zA-Z0-9_]+" nil t)
      (looking-at "[ \t\n]*(")
      (condition-case nil
      (not (forward-sexp))
    (error nil))
      )

     (not (goto-char old-point)) ;restore point's position for next check.

;; Check if point is located immediately after the closing parenthesis
     ;; of a function call.
     (and
      (prog2
      (goto-char (1- (point)))
      (looking-at ")")
    (goto-char (1+ (point))))
      (save-excursion
    (and
     (condition-case nil
       (not (backward-sexp))
       (error nil))
     (looking-at "(")
     (re-search-backward "[^ \t\n][ \t\n]*" nil t)
     (looking-at "[a-zA-Z0-9_]"))))

     ;; No function call detected - restore point.
     (not (goto-char old-point)))))

(defun contract-resolve-arguments (beg end)
  "Returns a list of the types of the argument list found between
```

```
BEG and END in the current buffer."
  (let (arg-list fun-info text args count (commas 0))
    (save-excursion
      (save-restriction
    (narrow-to-region beg end)
    (goto-char (point-min))

    ;; Count number of commas in the argument list.
    (save-excursion
      (while (re-search-forward "[(,]" nil t)
      (goto-char (1- (point)))
      (if (looking-at "(")
        (condition-case nil (forward-sexp) (error nil))
        (goto-char (1+ (point)))
        (setq commas (1+ commas)))))

    ;; Calculate number of arguments in the argument list.
    (cond ((= commas 0)
         (if (looking-at "[ \t\n]*.+[ \t\n]*")
           (setq args 1) (setq args 0)))
        (t (setq args (1+ commas))))

    (goto-char (point-min)))

    ;; Get the types of the arguemnts in the argument list.
    (dotimes (count args (nreverse arg-list))
    (re-search-forward "[ \t\n]*" nil t)

    (cond
    ;; check for numeric constants and decide type (int or float).
     ((looking-at "[\\+-]?[0-9]+\\(\\.[0-9]+\\)?")
      (setq arg-list
        (cons
         (cons
          (cond
           ((integerp (string-to-number (match-string 0))) "int")
           ((floatp   (string-to-number (match-string 0))) "float")
           (t nil))
          (match-string 0))
         arg-list)))
  ;; check for function call and lookup return type in symbol table.
```

```
     ((looking-at
       "[a-zA-Z_][a-zA-Z0-9_]*[  \t\n]*\\(\\.\\|->\\)?[  \t\n]*[a-zA-Z_][a-
zA-Z0-9_]*[ \t\n]*(")
      (setq text
        (buffer-substring
         (point)
         (progn
           (re-search-forward
          "[a-zA-Z_][a-zA-Z0-9_]*[    \t\n]*\\(\\.\\|->\\)?[    \t\n]*[a-zA-
Z_][a-zA-Z0-9_]*[ \t\n]*"
          nil t)
           (condition-case nil (forward-sexp) (error nil))
           (point))))
      (setq arg-list (cons
              (cons
               ;; Get return type of function
               (car (nth 1 (car (get-function-call-info))))
               text)
              arg-list)))

     ;; check for variable name and look up type in symbol table.
     ((looking-at "[a-zA-Z_][a-zA-Z0-9_]*")
      (setq text (match-string 0))
      (setq arg-list (cons
              (cons (contract-lookup-type-in-buffer text) text)
              arg-list)))

     ;; Skip ',' and spaces.
     ((or (looking-at "[.\n]*,?") (looking-at "[.\n]*"))
      (setq arg-list (cons (cons nil (match-string 0)) arg-list)))
     ;; Default condition: Can't decide argument type  - add 'nil'
     ;; to list of types.
     (t (setq arg-list (cons nil arg-list))))

    (re-search-forward "," nil t)))))

(defun contract-function-call-info ()
  "Extract function call information like name, arguments, object name if
dealing with a member function.
Format of the returned list:
(((OBJ_TYPE . OBJ_NAME)
```

```
  (RET_TYPE . FUN_NAME)
  ((ARG1_TYPE . ARG1_TEXT) (ARG2_TYPE . ARG2_TEXT)))
 TAG)"
  (save-excursion
    (let ((call-end (point)) old-point object function arg-list tag)
      (backward-sexp)          ;go back to opening parenthesis
      (setq arg-list
      (contract-resolve-arguments (1+ (point)) (1- call-end)))
    (setq old-point (point))
      (backward-sexp)
      (setq function
      (contract-extract-string
       (buffer-substring (point) old-point) "[^ \t\n]*"))
    (setq old-point (point))
      ;; Check if call to member function
      (and (condition-case nil
         (not (backward-sexp))
       (error nil))
     (looking-at "[a-zA-Z_][a-zA-Z0-9_]*\\(\\.\\|->\\)")
     (looking-at "[a-zA-Z_][a-zA-Z0-9_]*")
     (setq object (match-string 0))
     ;; Lookup object type.
     (setq object (cons (contract-lookup-type-in-buffer object) object)))
      (setq function
      (cons
     (progn
       (setq tag (contract-find-tag (car object) function arg-list))
       (if (stringp (semantic-tag-type tag))
           (semantic-tag-type tag)
         (car (semantic-tag-type tag))))
     function))

      (cons (list object function arg-list) tag))))

(defun contract-arg-list-string (tag)
  "Return a string representation of the arguments in TAG.
Example return string: \"int, float, int\""
  (let ((arg-string-list
     (mapcar
      (lambda (x) (concat
             (semantic-tag-type x)
```

```
              " "
              (semantic-tag-name x)))
       (semantic-tag-function-arguments tag)))
     (count 0)
     (arg-string "")
     item)

     (dolist (item arg-string-list arg-string)
       (if (= count 0)
       (progn (setq count (1+ count))
          (setq arg-string (concat item ", ")))
     (setq arg-string (concat arg-string item ", "))
     (setq count (1+ count))))
     (if (> (length arg-string-list) 0)
     (substring arg-string 0 (- (length arg-string) 2)))))

(defun contract-filter-comment (string)
  "Removes comment characters from a comment block and return a list of the
following format: (DESCR PRE POST)"
  ;; Remove comment block start.
  (string-match "^/\\*[\\* \t\n]*" string)
  (setq string (substring string (match-end 0)))
  ;; Remove comment block end.
  (setq string (substring string
              0
              (string-match "[ \t\n\\*]*\\*/$" string)))
  ;; Remove all '*' in front of comment line
  (while (string-match "^[ \t]*\\*+ ?" string)
    (setq string (concat
          (substring string 0 (match-beginning 0))
          (substring string (match-end 0) (length string)))))

  (let (beg end descr-part pre-part post-part)
  (with-temp-buffer
      (insert string)
      (goto-char (point-min))
    (if (looking-at "[ \t\n]*") (goto-char (match-end 0)))

    (if (looking-at "@[Pp]")
      nil
    (setq beg (point))
```

```
    (setq end (if (re-search-forward "@[Pp][ \t\n]*" nil t)
            (match-beginning 0)
          (point-max)))
    (while (= ?\n (char-before end)) (setq end (1- end)))
    (setq descr-part (buffer-substring beg end)))

    (goto-char (point-min))

    (if (re-search-forward "@[Pp]re[ \t\n]*" nil t)
      (progn (setq beg (point))
         (setq end
             (if (re-search-forward "@[Pp]" nil t)
               (match-beginning 0)
             (point-max)))
         (while (= ?\n (char-before end)) (setq end (1- end)))
         (setq pre-part (buffer-substring beg end))))

    (goto-char (point-min))

    (if (re-search-forward "@[Pp]ost[ \t\n]*" nil t)
      (progn (setq beg (point))
         (setq end
             (if (re-search-forward "@[Pp]" nil t)
               (match-beginning 0)
             (point-max)))
         (while (= ?\n (char-before end)) (setq end (1- end)))
         (setq post-part (buffer-substring beg end)))))

  (list descr-part pre-part post-part)))

(defun contract-get-comment (tag)
  "Returns the comment header if found at TAG's declaration.
If comment header can't be found, nil is returned."
  (save-excursion
    (let (buffer comment-beg)
      (and (semantic-tag-p tag)
       (setq buffer (get-buffer (semantic-tag-buffer tag)))
       (set-buffer buffer)
       (goto-char (semantic-tag-start tag))
       (re-search-backward "[^ \t\n][ \t\n]*")
       (goto-char (1- (point)))
```

```
        (looking-at "\\*/")
        (setq comment-beg (+ (point) 2))
        (re-search-backward "/\\*"))
       (if comment-beg
       (buffer-substring comment-beg (point))
     nil))))

(defun contract-arg-list-equal (tag arg-list)
  "Returns non-nil if the types of the arguments in TAG is equal to the
C argument list in ARG-LIST."
  (let ((tag-arg-list (mapcar 'semantic-tag-type
                (semantic-tag-function-arguments tag)))
    (result t)
    item)
    (if (and (not tag-arg-list) arg-list) (setq result nil))
    (dolist (item tag-arg-list result)
      (if (string-equal item (car arg-list))
      (setq arg-list (cdr arg-list))
    (setq result nil)))))

(defun contract-find-tag (parent name arg-list)
  "Returns the tag (see CEDET/Semantic documentation for description of
tag)
for NAME in class PARENT. If ARG-LIST is a list of arguments, the arguments
in the tag must match the arguments in ARG-LIST."
  (if (and parent name)
    ;; Get PARENT tag
      (let* ((find-result (semanticdb-deep-find-tags-by-name parent nil
nil))
        (tags (cdr (car find-result))) ;not the best way
        (db (car (car find-result))) ;...
        (parent-tag nil)
        (member-tags nil)
        (target-tag nil)
        item)

    (while (and (not parent-tag) tags)
      (if (semantic-tag-of-class-p (car tags) 'type)
        (setq parent-tag (car tags))
      (setq tags (cdr tags))))
```

```
    (setq member-tags (semantic-tag-type-members parent-tag))

    (while (and (not target-tag) member-tags)
      (if (and (string-equal name (semantic-tag-name (car member-tags)))
           (contract-arg-list-equal (car member-tags)
                      (mapcar 'car arg-list)))
        (setq target-tag (car member-tags))
      (setq member-tags (cdr member-tags))))
    target-tag)

  ;; Get NAME from the parent tag
    (let* ((find-result (semanticdb-deep-find-tags-by-name name nil nil))
       (tags (cdr (car find-result))) ;not the best way
       (db (car (car find-result))) ;...
       (target-tag nil))
      (dolist (item tags target-tag)
    (if (and (string-equal name (semantic-tag-name (car tags)))
         (contract-arg-list-equal
          (car tags)
          (mapcar 'car arg-list)))
      (setq target-tag (car tags))
      (setq tags (cdr tags))))
      target-tag)))

(defun contract-lookup-type-in-buffer (name)
  "Return the string representation of the type of NAME."
  (save-excursion
    (let ((scopes 2)          ;two scopes: local and global
      ;; Ignore 'no-context'-errors
      (var-tags (condition-case nil
            (semantic-get-local-variables)
            (error nil)))
      (type nil))

      (while (> scopes 0)
    (while (and (not type) var-tags)
      (if (string-equal name (semantic-tag-name (car var-tags)))
        (progn (setq type (semantic-tag-type (car var-tags)))
           (if (semantic-tag-p type)
             (setq type (semantic-tag-name type))))
      (setq var-tags (cdr var-tags))))
```

```
    (if type
      (setq scopes 0)        ;type found - skip global scope
      (setq scopes (1- scopes))
      (setq var-tags (semantic-bovinate-toplevel)))) ;get global tags
      type)))

(provide 'contract-mode)
```

D.3 contract-parse.el

```
;;; contract-parse.el
;;; Peter Labraaten & Stefan Larsson
;;; Precondition parser used by Contract Mode

(defcustom contract-parse-newline-as-and nil
  "Specifies if newline characters are used as implicit AND operators.
If new line chars are used as logical AND operators, '&&' will be inserted
by the lexer when a new line character is found. Otherwise new line
characters will be ignored like white space characters."
  :type 'boolean
  :group 'contract)

(defun parse (precondition-string arg-alist object &optional error-msg)
  "Used to parse and optionally translate a precondition string.
Arguments: PRECONDITION-STRING - The precondition to parse.
           ARG-ALIST - Associative list containing the formal and actual
           arguments used in translation.
           ((FRM1 . ACL1) (FRM1 . ACL1) ... (FRMn . ACLn))
           OBJECT - The name of the object used to access a variable
           or function. Examples: myobj, myobj*
           ERROR-MSG - If true, return error message instead of nil."
  (let* ((token-stream (lex precondition-string))
     (lookahead (car token-stream))
     (token-stream (cdr token-stream))
     output)

    (cond ((and (expr) (not lookahead)) output)
      (error-msg (parse-error-msg)))))

;;; Functions used by 'parse'.
```

```
(defun lex (c-expr)
  "Lexical analyzer used to tokenize the input precondition C-EXPR.
Uses the varible contract-parse-newline-as-and to decide how to treat
new line characters.
Format of tokenized stream:
  ((TOK1 . LEX1) (TOK2 . LEX2) ... (TOKn . LEXn))"
  (let (token-list (token t))
    (with-temp-buffer
      (insert c-expr)
      (goto-char (point-min))

      (looking-at "[ \t]*")
      (goto-char (match-end 0))
      (while (and (/= (point) (point-max)) token)
    (setq token
        (cond ((looking-at "[0-9]+\\(\\.[0-9]+\\)?")
           (cons :number (match-string 0)))
          ((looking-at "[a-zA-Z_][a-zA-Z_0-9]*")
           (cons :id (match-string 0)))
          ((looking-at "\\+\\+")
           (cons :incop (match-string 0)))
          ((looking-at "--")
           (cons :decop (match-string 0)))
          ((looking-at "\\(\\.\\|->\\)")
           (cons :memacc (match-string 0)))
          ((looking-at "[\\+-]")
           (cons :addop (match-string 0)))
          ((looking-at "[\\*/%]")
           (cons :mulop (match-string 0)))
          ((looking-at "\\&\\&\\|||")
           (cons :logop (match-string 0)))
          ((looking-at "==\\|!=\\|<=\\|>=\\|[<>]")
           (cons :relop (match-string 0)))
          ((looking-at "!")
           (cons :negop (match-string 0)))
          ((looking-at "(")
           (cons :lparen (match-string 0)))
          ((looking-at ")")
           (cons :rparen (match-string 0)))
          ((looking-at ",")
           (cons :comma (match-string 0)))
```

```
          ((looking-at "\n")
           (cons :newline (match-string 0)))
          (t nil)))
    ;; Check for newline. If newline found, check if translation
        ;; to '&&' is wanted.
    (if (eq (car token) :newline)
      (if contract-parse-newline-as-and
        ;; Translate to '&&'.
        (setq token-list (cons (cons :logop "&&") token-list))
        ;; No translation wanted - skip the newline.
        )
      ;; No newline - add token to list.
      (setq token-list (cons token token-list)))
    (goto-char (match-end 0))
    (looking-at "[ \t]*")
    (goto-char (match-end 0)))
      (if token (nreverse token-list) nil)
    )))

(defun lookahead-token ()
  "Used by 'parse' to handle token pairs like: (TOKEN . LEXEME).
Returns the token from a token-lexeme pair."
  (car lookahead))

(defun lookahead-text ()
  "Used by 'parse' to handle token pairs like: (TOKEN . LEXEME).
Returns the lexeme from a token-lexeme pair."
  (cdr lookahead))

(defun match (token)
  "Used by 'parse' to check if TOKEN matches the current token.
If there is a match, non-nil is returned and the next token is set as
the current token. If there is a mismatch, 'nil' is returned."
  (if (eq token (lookahead-token))
      (progn (setq lookahead (car token-stream))
       (setq token-stream (cdr token-stream))
       t)
    nil))

(defun emit (string)
  "Used by 'parse' to build up an output string by appending STRING to the
```

```
output already processed."
  (setq output (concat output string)))

(defun parse-error-msg ()
  "Used by 'parse' to generate an error message."
  (message "Parse error near: %s\nRemaining input:\n%s"
       (lookahead-text)
       (concat (lookahead-text) " " (mapconcat 'cdr token-stream " "))))

;;;

(defun expr () ""
  (cond
   ((eq :number (lookahead-token)) (and (emit (lookahead-text))
                    (match :number)
                    (expr-rest)))
   ((eq :id     (lookahead-token)) (and
                  (let ((id-text (lookahead-text)))
                    (match :id)
                    (fun-call))
                  (expr-rest)))
   ((eq :lparen (lookahead-token)) (and (emit (lookahead-text))
                    (match :lparen)
                    (expr)
                    (eq :rparen (lookahead-token))
                    (emit (lookahead-text))
                    (match :rparen)
                    (expr-rest)))
   ((eq :addop  (lookahead-token)) (and (emit (lookahead-text))
                    (match :addop)
                    (expr)))
   ((eq :negop  (lookahead-token)) (and (emit (lookahead-text))
                    (match :negop)
                    (expr)))))

(defun expr-rest () ""
  (cond
   ((eq :addop (lookahead-token))
  (and (emit (concat " " (lookahead-text) " "))
     (match :addop) (expr)))
   ((eq :mulop (lookahead-token))
```

```
  (and (emit (concat " " (lookahead-text) " "))
     (match :mulop) (expr)))
   ((eq :relop (lookahead-token))
  (and (emit (concat " " (lookahead-text) " "))
     (match :relop) (expr)))
   ((eq :logop (lookahead-token))
  (and (emit (concat " " (lookahead-text) " "))
     (match :logop) (expr)))
   (t)))

(defun fun-call () ""
  (cond
   ((eq :lparen (lookahead-token))
    (and
   (if object
     ;; Translation
     (progn
       (string-match "[a-zA-Z_][a-zA-Z0-9_]*" object)
       (emit (match-string 0 object))
       (if (string-match "\\*" object)
         (emit "->")
       (emit "."))
       t)
       t)

     (emit id-text)
     (emit (lookahead-text))
     (match :lparen)
     (expr-list)
     (eq :rparen (lookahead-token))
     (emit (lookahead-text))
     (match :rparen)))
   (t
    (let (arg)
      (setq arg (cdr (assoc id-text arg-alist)))
      (emit (if arg arg id-text)))

  (if (not (eq :memacc (lookahead-token)) )
    t               ; empty production
    ;; Member access production
    (and (emit (lookahead-text))
```

```
       (match :memacc)
       (eq :id (lookahead-token))
       (setq id-text (lookahead-text))
       (match :id)
       (fun-call))))))

(defun expr-list () ""
  (cond
   ((or (eq :number (lookahead-token))
    (eq :id     (lookahead-token))
    (eq :lparen (lookahead-token))
    (eq :addop  (lookahead-token))
    (eq :negop  (lookahead-token)))
    (and (expr)
     (expr-list-rest)))
   (t)))

(defun expr-list-rest () ""
  (cond
   ((eq :comma (lookahead-token)) (and (emit (concat (lookahead-text) " "))
                     (match :comma)
                     (expr-list)))
   (t)))

(provide 'contract-parse)
```

D.4 contract-utils.el

```
;;; contract-utils.el
;;; Peter Labraaten & Stefan Larsson
;;; Utility functions used by Contract Mode

(require 'avoid)          ; needed for pixel pos calculation

(defvar contract-overlay nil "")
(defvar contract-overlay-properties '(face (:background "yellow")) "")

(defun contract-highlight-region (beg end)
  "Highlights the region between BEG and END by using
the 'contract-overlay' overlay."
  (let (face (count 0))
```

```
  (setq contract-overlay (make-overlay beg end))

  (while (< count (length contract-overlay-properties))
    (overlay-put contract-overlay
           (nth count      contract-overlay-properties)
           (nth (1+ count) contract-overlay-properties))
    (setq count (+ count 2))))
  t)

(defun contract-unhighlight-region ()
  ""
  (delete-overlay contract-overlay)
  (setq contract-overlay nil)
  (remove-hook 'pre-command-hook 'disable-temp-highlight-region t))

(defun contract-display (string) ""
  (let* ((params '((name . "tooltip")
           (internal-border-width . 5)
           (border-width . 1)
           (border-color . "black")))
     (pos (contract-point-approx-pixel-pos 0 -62))
     (left
      (assq 'left
        (car (cdr (car (cdr (current-frame-configuration)))))))
     (top
      (assq 'top
        (car (cdr (car (cdr (current-frame-configuration))))))))

    (setq left (cons 'left (+ (cdr left) (car (cdr pos)))))
    (setq top  (cons 'top  (+ (cdr top)  (cdr (cdr pos)))))

    (add-to-list 'params left)
    (add-to-list 'params top)

    (if (fboundp 'x-show-tip)
    (x-show-tip string (car pos) params nil nil nil)
      (message "%s" string))))

(defun contract-point-approx-pixel-pos (&optional adjust-x adjust-y)
  "Returns the approximate position of point as \(FRAME X . Y\).
X and Y are pixel values of a position within FRAME that
```

```
is located near point.
Optional arguments ADJUST-X and ADJUST-Y provides an easy way to
modify the returned X and Y values by passing positive or negative
pixel offset values."
  (let* ((point  (mouse-avoidance-point-position)) ; (FRAME X . Y)
      (frame  (car point))
      (x (car (cdr point)))
      (y (cdr (cdr point))))
     (setq adjust-x (if adjust-x (+ adjust-x) 0))
     (setq adjust-y (if adjust-y (+ adjust-y) 0))
     (cons frame
       (cons
        (+ (round (* (/ (float x) (frame-width))  (frame-pixel-width)))
         adjust-x)
        (+ (round (* (/ (float y) (frame-height)) (frame-pixel-height)))
         adjust-y)))))

(defun contract-merge-lists (list1 list2)
  "Merges LIST1 and LIST2 by returning a list of cells where the car of
the nth cell is the nth element of LIST1 and the cdr is the nth element
of LIST2.
((L1E0 . L2E0) (L1E1 . L2E1) ... (L1En . L2En))"
  (let (merged-list)
  (if (/= (length list1) (length list2))
    nil
    (dotimes (count (length list1) merged-list)
    (add-to-list
     'merged-list
     (cons (nth count list1) (nth count list2)) t)))))

(defun contract-first-string-from-vector (vector)
  "Returns the first string object found in VECTOR.
If no string object is found or VECTOR isn't a vector, nil is returned."
  (if (vectorp vector)
    v      (let ((file-name nil) (index 0))
         (while (not (stringp file-name))
         (setq file-name (elt vector index))
         (setq index (1+ index)))
         file-name)
    nil))
```

```
(defun contract-extract-string (string regexp)
  "Return the first match of REGEXP in STRING.
If no match is found, nil is returned.
Example call:
(extract-string \" foo  \" \"[^ ]*\")
  ->\"foo\""
  (string-match regexp string 0)
  (match-string 0 string))

(provide 'contract-utils)
```