Eindrapport bij voorstudie Rosetta-reparatie

GridLine BV, mei 2007

Inhoudsopgave

Samenvatting

GridLine heeft een vooronderzoek gedaan naar de repareerbaarheid van de Rosetta-vertaalsoftware. Dit rapport doet verslag van de de bevindingen in dit vooronderzoek. Gebleken is dat het repareren van Rosetta moeilijk, maar niet ondoenlijk is. Concreet verwacht GridLine dat de reparatie van Rosetta, gegeven de randvoorwaarden van dit vooronderzoek, grofweg 18 maanden werk zal opleveren. Voor een preciezere prognose is een tweede vooronderzoek nodig.

Het Rosetta-vertaalsysteem werd eind jaren 80 ontwikkeld in het research-lab van Philips in samenwerking met een groot team van taalkundig onderzoekers. Het systeem heeft indertijd daadwerkelijk gefunctioneerd voor de talen Nederlands, Engels en Spaans, maar voordat het systeem voltooid kon worden werd de financiering stopgezet en het ontwikkelplatform ontmanteld. In het kader van het Stevin-project IRME kwam er belangstelling voor de reanimatie van Rosetta. Dit leidde ertoe dat GridLine de opdracht kreeg om in vier weken tijd een inschatting te maken van de benodigde reparatietijd door een pilot-onderzoek te doen naar de repareerbaarheid van één complete Pascal-module.

Toen GridLine met deze opdracht aan de slag ging, bleek dat het project complexer was dan voorzien. Zo was de module-structuur van Rosetta minder transparant dan de functionele beschrijving deed vermoeden. De functionele modules waren namelijk niet zichtbaar in de directorystructuur, maar moesten gereconstrueerd worden door de make-files te analyseren. Dit leidde tot het inzicht dat de functionele modules in feite clusters zijn van een lager type modules, de operationele modules. Ook bleek dat de functionele modules elkaar flink overlappen en dat een deel van de operationele modules buiten het bereik viel van de functionele modules. De functie van deze operationele modules was bovendien slecht gedocumenteerd.

Ondanks deze complicaties is GridLine er toch in geslaagd om een functionele component te reconstrueren (namelijk “Amorph”) en gedeeltelijk te hercompileren. Eerst is een systeemanalyse uitgevoerd, hetgeen gedetailleerde informatie opleverde over de aanwezige files en hun modulaire samenhang. Op basis van deze informatie kon een indruk worden verkregen van de omvang en de complexiteit van de reparatieklus. Ook werd een alternatief gezocht voor de oorspronkelijke Pascal-compiler; na een korte experimenteerperiode bleek de GNU Pascal-compiler het geschikst te zijn. Hierna is ca. 46 uur besteed aan een poging tot het hercompileren van Amorph. Hierbij zijn in totaal 9 grote aanpassingen doorgevoerd, waardoor 18 files volledig konden worden gecompileerd (behoudens niet-traceerbare file- en functie-verwijzingen). Veel andere files ondergingen een of meer verbeteringen.

Uitgaande van deze gegevens schat GridLine dat het hercompileren van het Pascal-gedeelte van het Rosetta-systeem ca. 6 maanden gaat kosten. Hiernaast zal het vervangen van een aantal cruciale componenten zo’n 3 maanden vergen. Voor het oplossen van runtime-errors en functionele problemen zal tot slot zo’n 9 maanden nodig zijn. Bij elkaar opgeteld zal een inspanning van ca. 18 maanden (met een marge van 4 maanden) nodig zijn om het complete Rosetta-systeem te repareren. Omdat deze prognose tal van onzekerheden kent, valt het aan te raden om een tweede vooronderzoek op te zetten van ca. 8 weken.

1 Inleiding

Het Rosetta-vertaalsysteem werd eind jaren 80 ontwikkeld in het research-lab van Philips. Er was een groot team van taalkundigen bij betrokken, die veel inventiviteit toonden in het uitwerken van een taalkundig regelsysteem op basis van de principes van compositionaliteit en omkeerbaarheid. Het systeem heeft indertijd daadwerkelijk gefunctioneerd voor de talen Nederlands, Engels en Spaans, maar voordat het systeem voltooid kon worden, werd de financiering stopgezet. Hierna werd het oorspronkelijke ontwikkelplatform ontmanteld.

In het kader van het Stevin-project IRME is het voornemen ontstaan om Rosetta opnieuw in gebruik te nemen. De reanimatie van Rosetta bleek echter lastiger dan gehoopt, aangezien het oude platform niet meer bestaat. Hetzelfde geldt voor de compiler, het filebeheersysteem, de grafische interface en de lexicale database. Daarom werd besloten om eerst een voorstudie uit te voeren naar de repareerbaarheid van het systeem. De opdracht voor deze voorstudie ging naar GridLine. Het doel van deze opdracht was om een prognose te geven van de benodigde reparatietijd door een pilot-onderzoek naar de repareerbaarheid van één complete Pascal-module. Voor deze opdracht kreeg GridLine een budget van vier werkweken.

Dit rapport doet verslag van de werkzaamheden en bevindingen van GridLine en biedt de gevraagde prognose. Het rapport heeft de volgende opbouw. Hoofdstuk 2 geeft een korte typering van GridLine, de uitvoerder van deze opdracht. Hoofdstuk 3 behandelt de opdracht en de door GridLine gekozen uitwerking. Hoofdstuk 4 bespreekt de kennis die is opgedaan door het verkennen van de documentatie. Hoofdstuk 5 presenteert de resultaten van de systeemanalyse. Hoofdstuk 6 biedt een overzicht van de activiteiten en observaties in de reparatie-fase. In hoofdstuk 7 wordt op basis van de resultaten in hoofdstuk 5 en 6 een prognose uitgewerkt voor de tijdsinspanning die nodig is voor de reparatie van het Rosetta-systeem als geheel (conform de bepalingen in de opdracht); hierbij worden ook enkele technische problemen belicht. Hoofdstuk 8 zet de conclusies op een rijtje en biedt een aantal aanbevelingen voor het vervolgtraject.

2 Over GridLine

GridLine is een dynamisch, research-gericht ICT-bedrijf dat zich specialiseert in advies en productontwikkeling op het terrein van content-management en taaltchnologie. Hierbij benut GridLine steeds de nieuwste technische mogelijkheden, van Ajax tot SharePoint 2007 en van RDF tot SOAP. Bij de uitwerking van haar projecten stelt GridLine altijd de wensen en behoeften van de klant voorop, zowel in de functionaliteit als in de technische uitwerking.

Wat de taaltechnologie betreft specialiseert GridLine zich in ontwikkeling en beheer van conceptueel geordende vaktermsystemen (zoals thesauri en ontologieën) en hierop gebaseerde toepassingen (zoals collectie-ontsluiting). Op dit terrein beschikt GridLine over veel contacten en ruime ervaring met productontwikkeling, implementatie en educatie. Hierbij zijn drie projectdomeinen te onderscheiden, te weten:

- projecten m.b.t. beheer en toepassing van *bestaande* vaktermsystemen (zoals thesauri)
- projecten m.b.t. de (semi-)automatische opbouw van vaktermsystemen (zoals thesauri) door terminologische analyse van vakspecifieke documentcorpora en automatische integratie van bestaande datacollecties met terminologische en lexicale gegevens;
- adhoc-projecten met een taaltechnologische component

GridLine is aangesloten bij branche-organisatie NOTaS en werkt veel samen met Nederlandse en Vlaamse kennisinstellingen, waaronder de universiteiten van Groningen, Twente, Tilburg, Utrecht, Amsterdam, Antwerpen en Leuven, de Lessius-hogeschool, het CWI, het Meertens-instituut, de TaalUnie, het INL en uitgever Van Dale Lexicografie. GridLine organiseert haar eigen symposia (zoals de tweejaarlijkse Thesauruslunches) en seminars (bijv. de GridWalker Seminars), en verzorgt op aanvraag workshops-op-maat voor bedrijven en instellingen.

GridLine heeft uitsluitend hoogopgeleid personeel in dienst. Het bedrijf telt inmiddels zeven senior-medewerkers (waarvan drie gepromoveerd) en heeft daarnaast een wisselend aantal junior-medewerkers, veelal studenten informatica, CKI en taalkunde. De eindverantwoordelijkheid ligt bij de eigenaar van Gridline, Tigran Spaan (die is afgestudeerd in de logica). Het Rosetta-project is uitgevoerd door dr. Oele Koornwinder (R&D-manager voor taaltechnologie), dr. Erik Aarts (software developer) en Maarten Leeuwrik (student taalkunde).

3 Opdracht en uitwerking

3.1 Oorspronkelijk projectplan

Het oorspronkelijke projectplan van J. Odijk luidde als volgt:

Ik zal beschikbaar stellen:

- *copie van de Rosetta CD*
- *copie van de RBS software (staat niet op de CD)*
- *RUNROS script (script om Rosetta te draaien) (staat waarschijnlijk ook op de CD)*
- *PDF versies van de belangrijkste in LaTeX geschreven documenten. Tekstuele versie van documenten die in een daarvoor gebruikt formateringsnotatie zijn geschreven*
- *Index van de Rosetta documentatie (2 Excel sheets)*

Ik ga ervan uit dat jullie de initiele onderzoekingen op een computer bij jullie zullen doen (onder Linux of Unix), maar het uiteindelijke resultaat moet natuurlijk op een computer van de UU komen te staan.

Ik heb de indruk dat de sourcefiles van het lexicon ontbreken op de CD. Ik zal proberen die alsnog te pakken te krijgen. Indien dat niet lukt, moeten we die wellicht reconstrueren met een script vanuit de afgeleide files (die wel aanwezig zijn)

Taakbeschrijving

Assessment van de te verwachten effort nodig om van Rosetta weer een draaiende versie te maken waarop weer development gedaan kan worden. Daartoe:

- *Bestuderen algemene documentatie. met name*
  - *Chapter 4 van het Rosetta boek*
  - *doc r0073 Global design*
  - *doc R0303; The Rosetta Module index*
  - *doc r0064 programming standards (opgesteld toen het systeem nog onder VAX/VMS werkte)*
  - *RBS documentatie*
- *bestuderen gebruikte Pascal syntax en selectie alternatieve compiler (bij voorkeur freeware). Ik weet niet meer welke compiler wij gebruikten maar ik meen me te herinneren dat het van een software bedrijf uit Utah was (en dat meen ik ter ziele is). Inschatten wat de effort is om de Pascal source code aan te passen aan de nieuwe compiler, en wat de effort is om compilers aan te passen die vanuit lingware (linguistische informatie neergeschreven in speciaal ontwikkelde taaltjes) notatie pascal files opleveren (bij voorkeur door gebruik te maken van de compiler generator)*
- *Speciaal kijken naar gebruik C-ISAM (zijn daar freeware versies voor en zijn die compatibel met hetgeen in Rosetta gebruikt wordt.*
- *Extraheren van Rosetta software om eventueel corrupte files te detecteren. de directory strucuur daarbij behouden.*
- *Delen van Rosetta zijn in C geschreven. Het betreft hier met name de grafische user interface, die gebruik maakt van X-Windows OSF Motif. De grafische user interface heeft lagere prioriteit, draaien in batch mode is voorlopig voldoende (die mogelijk*

wordt door Rosetta geboden), Checken of de in C geschreven modules met standaard C-compilers behandeld kunnen worden

In tweede fase:

- *Installeren RBS systeem en opzetten RBS archief compatibel met hetgeen in gebruik is voor Rosetta. SCCS wordt gebruik vor versie controle, dus dat is ook nodig (maar zit meen ik standaard op Unix systemen)*
- *Installeren Pascal compiler, evt. C-compiler, evt C-ISAM, evt. X-Windows OSF MOTIF.*
- *Proberen een nieuwe versie van Rosetta te bouwen met behulp van RBS. Ik vrees dat dit grotendeels trial en error zal zijn: uitproberen, kijken of en waar het vastloopt, aanpassingen doen, en weer op nieuw proberen te bouwen. Dit kan --indien de afhankelijkheden tussen de modules goed gekend zijn incrementeel gebeuren. De grafische interface heeft hierbij lagere prioriteit.*

Voor alle vragen en problemen kunnen jullie bij mij terecht, en ik heb ook contact met een informaticus, een voormalige medewerker in het Rosetta project die grote delen van de software ontwikkeld heeft (o.a. RBS), en die bereid is vragen te beantwoorden (wel in zijn vrije tijd, en naast zijn huidige baan, en over een systeem dat 15-20 jaar geleden ontwikkeld is, dus we kunnen niet alijd snelle respons garanderen)

3.2 Overeengekomen opdracht

In de door GridLine aangeboden offerte is het oorspronkelijke plan i.v.m. het beperkte budget op een aantal punten ingeperkt. De door GridLine aangenomen opdracht luidt als volgt:

GridLine stelt voor in eerste instantie uitsluitend een voorstudie te doen op basis van het door de opdrachtgever aangeleverde materiaal. De beoogde voorstudie heeft als doel om in een periode van 3 weken een gefundeerde begroting te maken van de inspanning (met een vertaling naar tijd en kosten) die nodig is voor de reparatie van de Rosetta-vertaalsoftware. In dit kader hebben we een aantal randvoorwaarden geformuleerd:

a) we gaan ervan uit dat de Rosetta-software op een Linux-systeem kan draaien; zo niet, dan zullen we de opdracht alleen uitvoeren als we in staat worden gesteld in de oorspronkelijke server-omgeving te werken (te weten een Sun machine of een Silicon Graphics machine);
b) we laten de grafische interface buiten beschouwing;
c) we steken geen tijd in het herstellen van het oorspronkelijke RBS-systeem; in plaats daarvan zullen we geen of een moderner beheersysteem gebruiken;

Wat betreft de uitvoering van de voorstudie onderscheiden we twee fasen, Een verkennende fase, en een proef waarbij geprobeerd wordt één geselecteerde component van de Rosetta software weer werkend te krijgen.
GridLine zal op basis daarvan een degelijk onderzoeksrapport produceren dat een goede basis biedt om de markt op te gaan, bijvoorbeeld voor het werven van aanvullende fondsen voor het repareren van de overige componenten.

3.2.1 Fase 1

Een verkenning van de opdracht, door het lezen van de documentatie, de analyse van het aangeleverde materiaal en het opstellen van een componentenlijst; vervolgens zullen we op basis van de componentenlijst een Pascal-genererende component selecteren die we in de

volgende fase weer werkend proberen te maken; hier valt te denken aan een concrete taalkundige deeltaak, zoals de syntactische analyse van de input; verder zullen we nagaan wat er bekend is over het gebruik van C-ISAM.
De eerste fase wordt afgesloten met de oplevering van een plan van aanpak voor fase 2. Dit plan omvat ook een sectie met evaluatiecriteria voor fase 2. Op basis van deze vooraf opgestelde evaluatiecriteria zal later, aan het eind van fase 2, een gefundeerde begroting worden gemaakt voor de reparatie van het gehele Rosetta-pakket.

Nadat het plan van aanpak voor fase 2 is goedgekeurd (al dan niet na aanpassingen), gaan we fase 2 uitvoeren.

3.2.2 Fase 2

Een concrete inspanning om de in fase 1 geselecteerde component werkend te maken; hiertoe zullen we een of meer Pascal-compilers selecteren en kijken hoe goed ze werken voor de geselecteerde component; vervolgens selecteren we de beste compiler en gaan we de te repareren component zo proberen aan te passen dat hij goed werkt met deze compiler; tot slot evalueren we het resultaat, en schrijven we aan de hand van de vooraf in fase 1 geformuleerde evaluatiecriteria een eindrapport met een testverslag en een gemotiveerde begroting voor de reparatie van het hele Rosetta-pakket (onder de gegeven randvoorwaarden).

3.3 Stappenplan

Bij de uitvoering van deze opdracht is GridLine uitgegaan van het onderstaande stappenplan.

1. Systeemanalyse
a) inventarisatie van filetypes c.q. extensies
b) identificatie van startscripts (zoals make-files)
c) inventarisatie van benodigde files (globaal en per module) via script-interne file-referenties
d) inventarisatie van ontbrekende files
e) koppeling systeem aan documentatie
f) Keuze van Pascal-compiler op basis van documentatie en script-kenmerken

2. Activering van make-files en executables
a) script-aanpassingen ivm overgang naar nieuwe Unix-omgeving
b) automatische aanmaak van benodigde .o-files door aanmaak van meta-compiler (= simulatie van RBS-systeem m.b.t. de compilatie-volgorde)
c) activering van executables

3. verkenning en activering van C-ISAM
a) compilatie van C-files
b) heractivering van C-ISAM

4. Reparatie van action-scripts
a) script-aanpassingen ivm overgang naar nieuwe Unix-omgeving
b) oplossen van resterende problemen

5. Activering van Pascal-scripts
a) Testen en werkend maken van scripts zonder inputfiles
b) Aanpassen van de formele module-structuur

c) Compilatie van interfaces (.pf-files); resulteert in .gpi-files
d) Compilatie van Pascal-scripts (.p-files); resulteert in .o-files
e) linken van .o-files door toepassing van make-files en .exe-files
f) aanmaak van taalspecifieke versies van scripts
g) compilatie van .o-files met inverse-werking (ANA > GEN)

6) Activering van een complete functionele module
a) compilatie van de module-interne .p-scripts (testen en aanpassen)
b) compilatie van complete module (via make-file)
c) generaliseerbaarheid naar overige modules

De beoogde voorstudie verkent alleen de problematiek die samenhangt met de eerste zes werkstappen. Om het Rosetta-systeem weer werkend te krijgen zijn echter nog meer werkstappen nodig, blijkens het onderstaande overzicht. De verkenning van de problemen die met deze werkstappen samenhangen is bij voorbaat uitgesteld naar een eventuele vervolgstudie.

Werkstappen voor een mogelijk vervolgproject

7. Herstel datastromen tussen functionele modules (inclusief het lexicon)

8. Verkenning van problemen in overige file-types dan .p, .pf en make/action

9. Activering van interactief controle-systeem voor de gebruiker

10. Vervanging van het RBS-systeem (een filebeheersysteem dat als doel had om ontwikkelaars in staat te stellen het systeem stap voor stap te testen en aan te passen) door een modern filebeheersysteem

4 Verkenning documentatie

4.1 Samenstelling documentatie

De aangeleverde Rosetta-cd bevat uitgebreide informatie over de functionele opzet van het Rosetta-systeem (te weten hoofdstuk 4 van de betreffende boekpublicatie) en over de technische implementatie van dit systeem. Deze documentatie biedt echter weinig houvast voor het vaststellen van de functies van de files die deel uitmaken van het Rosetta-systeem, laat staan voor de reconstructie van de modulaire en operationele samenhang van deze files. Zo specificeert de Rosetta Module Index tal van filetype-specificaties die niet zijn terug te vinden in de file-extensies. Omgekeerd zijn veel file-extensies niet terug te vinden in de index. Voor onze voorstudie had deze documentatie dan ook een beperkte waarde.

4.2 Functionele beschrijving Rosetta

Rosetta is een onvoltooid software-pakket waarmee automatisch vertalingen kunnen worden aangemaakt van en naar zinnen in het Nederlands, Engels en Spaans, voorzover ondersteund door het interne woordenboek. Deze vertalingen komen tot stand door toepassing van een algoritme dat uitgaat van het compositionaliteitsprincipe: dit vertaalprincipe stelt dat zin B een correcte vertaling is van zin A als alle basiseenheden (met name woorden) van zin A een equivalent hebben in zin B en als de combinatieregels voor het opbouwen van zin A een equivalent hebben in zin B, terwijl ook het omgekeerde van toepassing moet zijn. Een vertaling die aan dit principe voldoet is isomorf met de originele zin.
De compositionele benadering van deze vertaalmethode leent zich uitstekend voor implementatie in een Montague-grammatica. Voor de uitwerking van Rosetta is een speciale variant van Montague-grammatica ontwikkeld, namelijk M-GRAMMAR. In deze variant kunnen de syntactische bouwstenen interne structuur bezitten, zodat er een aparte submodule nodig is om deze bouwstenen naar een morfologische (c.q. orthografische) vorm om te zetten. Dit is met name handig voor het oplossen van afwijkingen tussen de structuur van de brontaal en die van de doeltaal. Voor elke bouwsteen geldt dat vorm, betekenis, vertaling en syntactische combinatiemogelijkheden (per taal) in een centraal lexicon worden vastgelegd.
Het Rosetta-systeem verbindt de beschikbare talen door gebruik te maken van een intermediaire betekenisrepresentatie. Het vertaalproces is over drie verschillende modules verdeeld, namelijk een controle-module (voor interactie tussen de gebruiker en het vertaalsysteem), een analyse-module (die zinnen uit de brontaal naar een intermediare betekenisrepresentatie omzet) en een generatie-module (die de intermediare betekenisrepresentaties omzet naar zinnen in de doeltaal). Elke analyse- en generatiestap is reversibel, zodat er geen fundamenteel verschil is tussen het vertalen van taal A naar taal B en het vertalen van taal B naar taal A. Dit heeft als groot voordeel dat de analyse- en generatiemodules per taal op dezelfde scripts kunnen worden gebaseerd (al kan wel een aparte compilatie nodig zijn van deze modules). In de praktijk komen er ook wel vertaalstappen voor die niet reversibel zijn, maar deze vormen een kleine minderheid.
Bij het omzetten van een zin uit de brontaal naar een intermediaire betekenisrepresentatie worden de volgende stappen doorlopen:

A-MORPH$_{SL}$:	omzetting van morfologische (c.q. orthografische) representatie naar syntactische bouwstenen (S-trees)
S-PARSER$_{SL}$:	categoriale identificatie van syntactische bouwstenen (S-trees) en opbouw van een reeks surface-trees (set of S-trees)

M-PARSER$_{SL}$: opbouw van syntactische derivatie-tree (D-TREE) door identificatie van derivatiestappen en omzetting van S-trees naar expressienamen
A-TRANSFER$_{SL}$: omzetting van derivatie-tree naar betekenis-tree (IL-representation)

Bij het omzetten van de intermediaire representatie naar de vertaling worden feitelijk dezelfde modules doorlopen, maar dan in omgekeerde richting. Wegens die omkering dragen de modules ook andere namen, te weten:

G-TRANSFER$_{TL}$ omzetting van betekenis-tree (IL-representation) naar derivatie-tree
M-GENERATOR$_{TL}$ ontleding van syntactische derivatie-tree (D-TREE) in set of S-trees
LEAVES$_{TL}$ ontleding van set of S-trees in reeks syntactische bouwstenen (S-trees)
G-MORPH$_{TL}$ omzetting van S-trees naar morfologische representaties

Bij elke data-overgang wordt een gespecialiseerde interface-module geactiveerd, blijkens het onderstaande interface-model.

```
*
 *  Type name      : INTERFACES_leveltype
 *  Description    : Definition of the interface levels ( level -1 has been
 *                   added for reasons of initialization )
 *                     -1                        9
 *                   ALAYOUT                 G-LAYOUT
 *                      0                        8
 *                   AMORPH                  G-MORPH
 *                      1                        7
 *                   SURFPARSER              LINEARIZER
 *                      2                        6
 *                   M-PARSER                M-GENERATOR
 *                      3                        5
 *                   A-TRANSFER              G-TRANSFER
 *                                  4
```

Tot slot zijn er overkoepelende modules voor geheugenbeheer en gebruikerscontrole.

4.3 Selectie Pascal-component

De door ons uitgevoerde voorstudie had als doel om voor een vantevoren geselecteerde Pascal-component na te gaan hoeveel moeite het kost om de onderliggende modules zo aan te passen dat ze operationeel kunnen worden gemaakt met een nieuw te implementeren Pascal-compiler. Met het oog op deze doelstelling moest eerst een Pascal-component worden geselecteerd. Uit de functionele beschrijving in sectie 4.2 blijkt dat de operationele ordening van de onderscheiden componenten een sterk lineaire opbouw vertoont, waarbij Amorph de eerst te doorlopen component is. Dit impliceert dat deze component (waarschijnlijk) geen andere componenten verist om gecompileerd te kunnen worden, zodat het compilatietraject beperkt kan blijven tot het compileren van de component-interne modules. Om die reden lijkt Amorph de meest geschikte component voor het uitvoeren van de voorstudie.
Voor de zekerheid hebben we deze conclusie eerst met opdrachtgever J. Odijk besproken, onder meer om te informeren of er complexiteitsverschillen waren te verwachten tussen Amorph en de andere componenten. In dit overleg bleek dat de component Amorph inderdaad een geschikt testdomein zou zijn voor onze voorstudie. In dit verband kregen we het advies om bij het testen van Amorph (en andere componenten) in eerste instantie met Nederlandse data te werken. Het Nederlandse deelsysteem van Rosetta is namelijk verder ontwikkeld dan het Engelse en het Spaanse deelsysteem.

5 Systeemanalyse

5.1 Inleiding

De systeemanalyse had als doel om inzicht te krijgen in de samenstelling en de structuur van het filesysteem op de aangeleverde Rosetta-cd. In dit kader hebben we enkele programma's ontwikkeld voor het automatisch analysen van de file-directories en de Pascal-scripts.

Als eerste stap is voor alle Rosetta-directories een inventarisatie gemaakt van de aanwezige files en hun extensies (sectie 5.2). Hiermee is een subinventarisatie van te compileren Pascal-files geïdentificeerd, te weten de module-types .p (programma) en .pf (interface); voor de meeste modules bleek ook een gecompileerde versie (.o) beschikbaar te zijn. Uitgaande van deze inventarisatie is vervolgens (sectie 5.3) een lijst van file-relaties aangemaakt door een script te ontwikkelen waarmee automatisch declaraties van input- en output-files kunnen worden opgespoord.
Hierna (sectie 5.4) is onderzocht in hoeverre de benodigde input-files terug waren te vinden op de aangeleverde Rosetta-cd. Verder (sectie 5.5) hebben we een poging gedaan om de oorspronkelijke module-hierarchie te reconstrueren (die deels is terug te vinden in de make-files) door de file-relaties hiërarchisch te clusteren. Het hieruit voortgekomen rapport kan worden gebruikt om functionele componenten te identificeren en hun module-opbouw te vergelijken. Met het oog op deze voorstudie (sectie 5.6) is voor de al bekende functionele componenten, waaronder de in H4 geselecteerde component Amorph, een nadere analyse uitgevoerd. Hierbij is per component nagegaan welke modules ertoe behoren, welke overlap er tussen de componenten bestaat en welke modules werkelijk beschikbaar zijn.
Het hoofdstuk wordt afgesloten met een overzicht van de belangrijkste kencijfers van het Rosetta-systeem (sectie 5.7) en een conclusie (sectie 5.8).

5.2 Inventarisatie van Rosetta-files

Voorafgaande aan de analyse van het Rosetta-filesysteem is de aangeleverde directory-structuur enigszins vereenvoudigd: hierbij is de inhoud van de subdirectory ARCHIVE overgeheveld naar de ROSETTA3-folder. De oorspronkelijke onderverdeling had te maken met de toepassing van het filebeheersysteem RBS; dit niet meer operatieve beheersysteem maakte het mogelijk om een deelsysteem te testen door een selectie te maken uit de folders en files in het Rosetta-archief; deze werden dan gekopieerd naar de ROSETTA3-folder, alwaar ze vervolgens gemodificeerd en gecompileerd konden worden. Voor onze doeleinden was het handiger om onnodige structuurlagen te verwijderen.

De inventarisatie van Rosetta-files heeft een aantal tabellen opgeleverd die zijn opgenomen in een Excel-file genaamd **rapport_Rosetta_directory_analyse.xls**. Het betreft de volgende tabellen:

a) directory-inhoud
b) statistieken bij files
c) statistieken bij extensies
d) files op alfabet
e) extensies op alfabet
f) extensies op aantal
g) extensies op data-omvang
h) files zonder extensie

Tabel a) geeft een complete inventarisatie van folders en files in het filesysteem van Rosetta. Hierbij is voor elke folder nagegaan hoeveel subfolders en files er zijn, welke extensies deze files hebben, hoeveel opslagruimte elke folder en file in beslag neemt, wat de laatste bewerkingsdatum is van elke file en of de files gecomprimeerd zijn opgeslagen. Bij deze analyse bleek dat veel folders een subfolder SCCS kennen, waarin oude script-versies werden bewaard. De inhoud van deze subfolders is bij de overige analyses genegeerd.
Tabel b) en c) bieden een modificeerbare samenvatting van de informatie in tabel a), in de vorm van een op folder geordende opsomming van resp. de beschikbare files en de beschikbare extensies, tezamen met frequentie- en omvanggegevens.
Tabel d) en e) bieden een alfabetisch geordende lijst van resp. kale file-namen (dus zonder padspecificatie) en hun extensies, waarbij per filenaam de vindplaatsen en de bijbehorende file-gegevens worden gespecificeerd; bij de extensies worden bovendien alle beschikbare files gegeven. Tabel d) maakt het mogelijk om snel na te gaan of een filenaam bestaat, in welke folders een filenaam voorkomt. Tabel e) maakt het mogelijk om snel na te gaan welke extensies er allemaal bestaan en op welke files deze extensies van toepassing zijn, hetgeen handig is voor het onderzoek naar de functie van deze files.
Tabel f) en g) bieden resp. een op gebruiksfrequentie en een op totale data-omvang geordend overzicht van de beschikbare extensies. Tabel h) ten slotte laat zien welke files helemaal geen extensie bezitten. Van deze 164 files kan worden aangenomen dat ze overbodig voor het Rosetta-systeem.
In sectie 5.7 staat een samenvatting van deze analyses in de vorm van een tabel met kencijfers per file-categorie.

5.3 Inventarisatie van module-relaties

De inventarisatie van module-relaties had als doel om voor alle Pascal-modules (files met extensie .p en .pf) na te gaan welke modules als input-file worden gedeclareerd. Hiertoe zijn de betreffende modules met een programma geanalyseerd op het voorkomen van specifieke file-declaraties. Deze inventarisatie heeft een aantal tabellen opgeleverd die zijn opgenomen in een Excel-file genaamd **rapport_Rosetta_module_relaties.xls**. Het betreft de volgende tabellen:

a) inputfiles bij modules
b) modules bij inputfiles
c) aantal modules bij inputfiles

Tabel a) biedt een alfabetisch gesorteerde lijst van Pascal-modules en de bijbehorende input-files (aangeduid als “reffiles”). Tabel b) biedt een alfabetisch gesorteerde lijst van inputfiles met de Pascal-modules waarvoor ze benodigd zijn. Tabel c ten slotte biedt een overzicht van het aantal modules per inputfile.

5.4 Inventarisatie van module-omissies

De inventarisatie van module-omissies had als doel om na te gaan welke input-files uit de inventarisatie in sectie 5.3 daadwerkelijk beschikbaar zijn in de Rosetta-directory. Dit heeft een aantal tabellen opgeleverd die zijn opgenomen in een Excel-file genaamd **rapport_Rosetta_reffile_omissies.xls**. Het betreft de volgende tabellen:

a) reffile-existence
b) non-existent, op reffreq
c) existent, op reffreq
d) directory-files

Tabel a) specificeert voor alle file-verwijzingen wat hun directory-status is: de markering [+path,+ext] geeft aan dat de verwijzing exact overeenkomt met een bestaande file, dat wil zeggen, met een file, waarvan naam locatie (pad) en extensie (ext) identiek zijn aan de opgegeven specificaties; de markering [+path,-ext] geeft aan dat de opgegeven file alleen kan worden teruggevonden na aanpassing van de extensie; de markering [-path,-ext] geeft aan dat de file niet in de opgegeven directory kan worden teruggevonden, ook niet na aanpassing van de extensie.
Tabel b) biedt een overzicht van alle file-referenties die (binnen de opgegeven directory) met een niet-bestaande file corresponderen (ongeacht de extensie). Tabel c) biedt een overzicht van alle file-referenties die juist wel met een bestaande file corresponderen. Tabel d) ten slotte specificeert nogmaals (zie ook sectie 5.2) naam, extensie en locatie van alle bestaande files.

5.5 Inventarisatie van module-clusters

De inventarisatie van module-clusters had als doel om na te gaan welke modules operationeel gezien tot dezelfde component behoren. Hiertoe is voor alle Pascal-modules een file-cluster geconstrueerd door de file-relaties recursief te volgen; de resulterende module-hiërarchie staat in een Excel-file genaamd **rapport_Rosetta_module_hierarchie.xls**. Op basis van deze hiërarchie is voor alle samenhangende componenten (c.q. module-clusters) een modulelijst afgeleid. Dit heeft een aantal tabellen opgeleverd die zijn opgenomen in een Excel-file genaamd **rapport_Rosetta_componenten.xls**. Het betreft de volgende tabellen:

a) modules met reffiles
b) module-lijst op clusteromvang
c) modulelijst op alfabet
d) reffiles op reffreq
e) reffiles op alfabet

Tabel a) specificeert voor alle Pascal-modules de recursief benodigde input-files (c.q. reffiles). In tabel b) en c) wordt deze informatie samengevat door voor elke component de module-omvang te bepalen, waarbij tabel b) de componenten van groot naar klein ordent, terwijl tabel c) een alfabetische volgorde aanhoudt. Tabel d) en e) bieden een overzicht van alle benodigde inputfiles (reffiles) en hun cumulatieve verwijsfrequentie (uitgaande van de startmodules); hierbij staan de items in tabel d) op volgorde van afnemende verwijsfrequentie, terwijl tabel e) gewoon een alfabetische ordening aanhoudt.

5.6 Inventarisatie van hoofdcomponenten

De inventarisatie van hoofdcomponenten had als doel om voor de al bekende functionele componenten van het Rosetta-systeem (zie hoofdstuk 4) na te gaan wat de samenstelling is van de bijbehorende module-clusters (met recursief benodigde inputmodules) en wat voor overlap deze clusters vertonen. De resulterende tabellen zijn opgenomen in een Excel-file genaamd **rapport_Rosetta_hoofdcomponenten.xls**. Het betreft de volgende tabellen:

a) samenstelling componenten
b) omvang hoofdcomponenten
c) overlap reffiles
d) componenten per reffile
e) existentie reffiles
f) overlap en existentie
g) GENERAL-modules
h) samenstelling van Amorph

i) p-modules in Amorph
j) pf-modules in Amorph

Tabel a) specificeert voor elke hoofdcomponent in de GENERAL-directory (te weten, elke GENERAL-module die qua naam overeenkomt met een functionele component) welke modules tot de bijbehorende module-cluster behoren. Tabel b) vat tabel a samen door voor elke hoofdcomponent de omvang van de module-cluster te specificeren.Tabel c) en d) laten zien hoe deze componenten overlappen door per inputmodule (c.q. reffile) bij te houden hoeveel componenten deze module benutten (tabel c) en een opsomming te geven van deze componenten (tabel d). Tabel e) specificeert voor elke reffile (c.q. benodigde inputmodule) of hij ook werkelijk bestaat. Tabel f) combineert de gegevens in tabel c) en e), waarbij de reffiles (hier beperkt tot de .p-modules) worden geordend op gebruiksfrequentie (aantal verwijzingen per reffile). Met deze tabel kan snel worden nagegaan welke van de veel gevraagde input-modules wel en welke niet beschikbaar zijn. Tabel g) een complete inventarisatie van de beschikbare GENERAL-files.
Tabel h) toont de module-samenstelling van de in deze voorstudie behandelde component Amorph; tabel i) en j) laten zien welke van de door Amorph gebruikte p-modules resp. pf-modules echt zijn terug te vinden in de GENERAL-directory.

De analyse van de module-samenstelling van de 10 hoofdcomponenten wijst uit dat ze gemiddeld 91 modules tellen (minimaal 56, maximaal 111) en dat er slechts 76 p-modules en 85 pf-modules nodig zijn om deze hoofdcomponenten te construeren. Hiervan worden 117 modules door 2 of meer componenten gedeeld, te weten 51 p-modules en 64 pf-modules. Van de 76 p-modules zijn er slechts 50 als file beschikbaar; de overige 26 ontbreken, waaronder 8 modules die door alle hoofdcomponenten gebruikt worden. Dit is zo'n hoog aantal, dat de vraag rijst of die modules ooit bestaan hebben en echt vereist zijn.
Wat betreft de component Amorph laat deze inventarisatie zien dat de bijbehorende file-cluster in totaal 108 modules telt, waaronder 48 p-modules en 58 pf-modules. Van de p-modules zijn er 13 niet terugvindbaar, en van de pf-modules blijken er 9 te ontbreken.

5.7 Kencijfers bij het Rosetta-systeem

De onderstaande tabel biedt een overzicht van de belangrijkste kencijfers van het Rosetta-systeem. Hierbij wordt onderscheid gemaakt tussen gegevens over het complete filesysteem (inclusief de SCCS-files) en het actieve filesysteem (exclusief SCCS-files).

kenmerk	**aantal files**	**data-omvang**
Rosetta-files	**6323**	**412.619 kb**
* actieve files (niet in sccs-folders)	5133	367.181 kb
* files in sccs-folders	1190	45.438 kb
- Pascal-files in sccs-folders	404	4.585 kb
- make-files in sccs-folders	8	336 kb
actieve Rosetta-files met script-status	**1694**	**33.301 kb**
* Pascal-files (.p/.pf)	1645	31.441 kb
- Pascal-programma-files (.p)	427	26.556 kb
- Pascal-interface-files (.pf)	1218	4.885 kb
* make-files (.make)	25	1.619 kb
* c-files (.c)	24	241 kb

actieve Rosetta-files met compiler-status	**1301**	**119.656 kb**
* Pascal-programma's (.o)	1262	81.593 kb
* executables (.exe)	81	118.063 kb
Rosetta-files met taaldata	**1541**	**128.298 kb**
* algemeen (mrule, dat, dict, lex, kdf)	559	61.992 kb
* alleen DUTCH (mrule, dat, dict, lex,)	436	21.768 kb
* alleen ENGLISH (mrule, dat, dict, lex)	274	31.578 kb
* alleen SPANISH (mrule, dat, dict, lex)	272	12.960 kb

Het onderzoek heeft zich toegespitst op Pascal-modules in de GENERAL-directory. De onderstaande tabel geeft een paar kencijfers bij deze directory.

files in de GENERAL-directory	**aantal files**
p-modules	101
pf-modules	132
o-modules	85
overige files	16
totaal	334

De onderstaande tabel toont de hoofdcomponenten en hun verwachte module-omvang.

component	**module-omvang**
general/alayout.p	56
general/amorph.p	108
general/atransfer.p	74
general/glayout.p	59
general/gmorph.p	109
general/gtransfer.p	97
general/linearizer.p	93
general/mgenerator.p	104
general/mparser.p	99
general/surfparser.p	111
Totaal	**910**

De onderstaande tabel vergelijkt de verwachte en de werkelijke module-omvang van de hoofdcomponenten als groep en van de component Amorph.

Pascal-modules in de 10 hoofdcomponenten	**benodigd aantal**	**beschikbaar aantal**
benodigde en beschikbare p-modules	76	50
benodigde en beschikbare pf-modules	85	56
totaal	161	106

Pascal-modules in de component Amorph	**benodigd aantal**	**beschikbaar aantal**
benodigde en beschikbare p-modules	48	35
benodigde en beschikbare pf-modules	58	49
totaal	106	84

6 Reparatie van het Rosetta-systeem

6.1 Inleiding

In dit hoofdstuk wordt verslag gedaan van de werkzaamheden en resultaten in het kader van het reparatiegedeelte van onze voorstudie naar de repareerbaarheid van het Rosetta-systeem. Hierbij wordt achtereenvolgens aandacht besteed aan de selectie van de Pascal-compiler (sectie 6.2), de reparatie van C-ISAM (sectie 6.3), de reparatie van de MAKE-files (sectie 6.4) en de reparatie van Pascal-modules, in het bijzonder de Pascal-modules in de component Amorph (sectie 6.5). Tot slot volgt een overzicht van de resultaten (sectie 6.6).

6.2 Selectie Pascal-compiler

Keuze van Pascal-compiler op basis van documentatie en script-kenmerken

- Rosetta is oorspronkelijk op een Sun Sparcs-computer ontwikkeld; dit type is verouderd.
- de oorspronkelijke Pascal-compiler blijkt 'Metaware Pascal' te heten; deze is niet meer leverbaar en bestaat ook niet voor hedendaagse Unix-omgevingen; het betreffende bedrijf reageerde niet op ons informatie-verzoek
- het was dus noodzakelijk om een nieuwe compiler te selecteren; de voorkeur ging uit naar een moderne, gratis verkrijgbare Pascal-compiler met als basiseisen dat hij uitgaat van standaard-Pascal (ISO-norm) met de mogelijkheid van escape-commando's en dat hij geschikt is voor de toepassing van module-structuur.
- Er is gekozen voor het gebruik van de GNU Pascal compiler (http://www.gnu-pascal.de); een mogelijk alternatief is Free Pascal, maar deze compiler lijkt minder geschikt voor Unix-systemen dan GNU Pascal.
- De gekozen Pascal-compiler kent een structurele scheiding van interfaces (.pf) en implementaties (.p); deze moeten apart gecompileerd worden.

6.3 Reparatie C-ISAM

a) compilatie van C-files

- De C-files waren gecompileerd voor sun solaris, zie -DSUN4 vlag in actions/c
- De C-files in de directory Rosetta3/UNIX (nodig voor C-ISAM) zijn gehercompileerd
- Problemen die hierbij optraden:
  - FUNCDEF staat in gendef.h
    #ifdef sparc weggehaald in gendef.h, dan kan FUNCDEF gevonden worden.
  - Het type boolean is onbekend, daarom compileren met c99 vlag.
    gcc -std=c99 unix/source/globbuf.c

b) heractivering van C-ISAM

- Er wordt verwezen naar files in de folder Informix, maar deze bestaat niet
  - Het lukt om unix/cisam.o te bouwen maar de library /home/informix/lib/libisam.a is er niet. Het kan zijn dat bij het linken (maken van executable) deze library wel nodig is.

6.4 Reparatie make-files

script-aanpassingen i.v.m. verschillen in file-systeem en Unix-omgeving

a) Reparatie van action-scripts

De pascal compiler wordt aangeroepen in actions/pas. In actions/pas is ‘pp’ (Metaware Pascal) vervangen door ‘gpc’ (Gnu Pascal Compiler).

b) Reparatie van make-files

- aanpassingen in file-referenties: padstructuur en foldernamen
- include-statements die naar het RBS-systeem verwijzen zijn verwijderd

c) automatische aanmaak van benodigde .o-files door aanmaak van meta-compiler (= simulatie van RBS-systeem m.b.t. de compilatie-volgorde)

Van een make-file verwacht men dat de afhankelijkheden zo beschreven zijn dat elk target ge-"maakt” kan worden. Indien eerst andere targets gemaakt moeten worden, gebeurt dit automatisch. De huidige make-files werken echter niet zo. Het is onduidelijk of dit vroeger wel op die manier werkte.

Nu is het zo dat de sources (de targets die eerst gemaakt moeten worden voordat de huidige target gemaakt kan worden) vaak .pf en .p bestanden zijn. Dit zijn echter bronbestanden. De makefile checkt dus of deze bronbestanden bestaan maar dwingt geen volgorde van compileren af.
Om dit goed te krijgen moeten de makefiles anders opgezet worden. Het kan ook zijn dat ergens anders (in build) staat in welke volgorde alles gecompileerd moet worden maar daar ziet het niet naar uit. Een alternatief zou zijn om de door gpc gegenereerde interface definitie files (.gpi files) in de makefile op te nemen en zo af te dwingen dat de interfaces in de correcte volgorde gecompileerd worden).

Wegens de hier beschreven problemen is besloten om af te zien van de reparatie van de make-files. In plaats daarvan zijn we verder gegaan met het direct compileren van Pascal-modules.

6.5 Reparatie Pascal-modules

a) Testen en werkend maken van scripts zonder inputfiles

Er is een eerste test uitgevoerd met het script 'pp/inc/actions/strings.pf'. Dit had succes. Het is ook gelukt om de files string.o en tstring.o te genereren met de Pascal compiler. Hieronder volgt een rapport van de bevindingen.

Om te compileren dient men naar de directory ROSETTA3 te gaan.

Hier kan dan het make-commando gegeven worden:

```
make -f general/large.make tools/tstring.o
make -f  general/large.make general/string.o
```

Om zonder de makefile de compiler direct aan te spreken gebruikt men:

```
gpc -c —interface-only -x Pascal general/string.pf
```

gpc -c –-implementation-only -x Pascal general/string.p

Metaware specifieke code:
Er wordt in de software gebruikt gemaakt van functies die specifiek zijn voor de Metaware compiler. Sommige zijn niet-standaard Pascal functies:

- ✓ Maxlength
- ✓ set_length
- ✓ string()

Deze functies kunnen eenvoudig vervangen worden door eigen code.
Ook worden een aantal libraries van de compiler gebruikt. De interfaces van deze libraries staan in pp/inc. Het gaat om de volgende files:

- ✓ strings.pf
- ✓ arg.pf
- ✓ converts.pf
- ✓ filep.pf
- ✓ heap.pf

Strings.pf wordt vaak aangeroepen. De code in deze library is verangen door eigen code (waarbij deletestring is vervangen door substr), en de include van strings.pf is verwijderd in alle source files. De andere libraries moeten nog gedaan worden. Deze worden hooguit 4 keer aangeroepen.

Andere wijzigingen die in de code zijn doorgevoerd

- ✓ || vervangen door +
- ✓ "ord applied to integers has no effect": ord() weggehaald.

b) Aanpassen van de formele module-structuur

GPC beschikt over een module systeem met gescheiden files voor de interfaces en de implementaties van die interfaces.

Hiervoor moeten de .p en .pf files geconverteerd worden. Hiervoor zijn twee Perl scripts gemaakt die all files in een directory in 1 keer omzetten.

Het converteren van het import-statement leidde tot diverse complicaties; deze zijn voorlopig nog handmatig opgelost.

De aanpak nu is om alle interfaces eerst te compileren met –interface.
Daarvan zijn er nu 60 gedaan.

c) Aanpassingen bij de reparatie van Amorph

'GPC' heeft een andere syntax om bestanden te includen. De verandering is:

```
Metaware:   pragma c_include('$foo');
GPC:        #include "$foo");
```

'GPC' heeft een andere syntax om aan te geven welke modulen er gebruikt worden:

```
Metaware:   with $foo, $bar;
GPC:        import $foo; $bar;
```

(Belangrijk: let op de vervanging van komma's door puntkomma's!)

Binnen "GPC" bestaat het keyword "export" niet meer als zodanig. Deze regel kan verwijderd worden in *.p bestanden. In *.pf bestanden blijft export gehandhaafd in de volgende vorm:

```
GPC:        module $foo interface;
            export $foo=all;
```

Binnen "GPC" bestaan er nu modulen, in plaats van losse packages en programs. Hiervoor moet

het volgende veranderen:

Metaware:	*.p	program $foo;
	*.pf	package $foo;
GPC	*.p	module $foo implementation;
	*.pf	module $foo interface;

Binnen de codebase zijn er *.pf bestanden, die echter geen interface naar modulen zijn. Dit zijn bestanden die enkele losse typen en constanten definieren. GPC beschouwt echter elk bestand dat met #include wordt aangegeven als een interface; dit moet nog opgelost worden.

Enkele standaard-modulen uit Metaware bestaan niet meer binnen GPC. De twee belangrijkste modulen zijn Heap en Loopholes. Loopholes is gedeeltelijk te vervangen door gebruik te maken van nieuwe GPC syntax:

Metaware:	loopholes.sizeof($foo);
	loopholes.retype($foo, $type);
GPC:	sizeof($foo);
	$type ($foo);

Enkele andere referenties gaan problemen opleveren, bijv. “LOOPHOLES.ADRESS”.

De module 'heap' was onderdeel van de MetaWare compiler. GPC lijkt genoeg interne functies te hebben om 'heap' overbodig te maken.

“GPC” geeft vaak foutmeldingen over “checksum mismatch”. Vaak komt dit door de volgorde die gehanteerd wordt bij het compileren. Als module A de lijn “import B;” heeft, samen met “#include B.pf”, dan moet module B eerst gecompileerd worden (hoewel linken nog overbodig is). Soms lost dit echter de checksum mismatch niet op; de reden is nog onbekend.

6.6 Resultaten

In dit hoofdstuk is verslag gedaan van de werkzaamheden en resultaten in het kader van de reparatie van een aantal deelcomponenten van Rosetta, te weten de reparatie van C-ISAM, de MAKE-files en Amorph.

Wat betreft de component C-ISAM is gconstateerd dat de onderliggende C-modules wel compileerbaar zijn, maar dat er een essentiele library ontbreekt, te weten *libisam.a*. Hierdoor was het niet mogelijk om C-ISAM werkend te krijgen.

Wat betreft de MAKE-files is geconstateerd dat het zonder een vervanging van het oude filebeheersysteem RBS ondoenlijk is om deze hulpprogramma's en de bijbehorende action-scripts aan de praat te krijgen. Dit betekent dat er vervangende scripts moeten komen voor de automatische compilatie van samenhangende clusters van Pascal-modules. Dit was uiteraard een belangrijke hindernis bij het hercompileren van de Pascal-modules.

Wat betreft Amorph is de nodige vooruitgang geboekt. Ondanks de problemen met de MAKE-files is het gelukt om de voor deze component benodigde module-verzameling te reconstrueren en 18 van de 108 modules te compileren, terwijl de overige modules (voor zover beschikbaar) hier dichterbij zijn gekomen. Hiervoor is in totaal 48 uur gebruikt.

De reconstructie van Amorph werd mogelijk door het natrekken van de file-verwijzingen in de input-statements van de startmodule (*Amorph.p*). Dit leverde een verzameling van 108 modules op, waaronder 48 p-modules, 58 pf-modules en 2 h-modules. Een deel van de benodigde modules bleek overigens te ontbreken: zo konden slechts 50 van de 76 p-modules worden teruggevonden; de overige 26 ontbraken, waaronder 8 modules die door alle hoofd-componenten gebruikt worden. Dit is zo'n hoog aantal, dat de vraag rijst of deze modules ooit bestaan hebben en echt vereist zijn. Dit zal nog moeten worden uitgezocht.

Voor het compileren van de Amorph-modules is een programma ontwikkeld waarmee automatisch script-aanpassingen konden worden doorgevoerd. Bij het opsporen en verhelpen van compilatie-problemen viel op dat sommige problemen van algemene aard waren (zodat alle modules van de aanpassingen profiteren) terwijl andere problemen nogal locaal waren. Dit betekent dat het compileren van alle Pascal-modules een langdurige aangelegenheid kan worden. Bovendien blijkt een deel van de benodigde modules (waaronder libraries) te ontbreken, zodat veel runtime-errors kunnen worden verwacht.

7 Prognoses

7.1 Inleiding

Het centrale doel van deze voorstudie was om een prognose te maken van de tijdsinspanning die nodig is voor de complete reparatie van het Rosetta-systeem. Een dergelijk project omvat in elk geval de volgende onderdelen:

1) inventarisatie van files en file-modules (c.q. file-hierarchie) in het gearchiveerde Rosetta-systeem en opsporing/koppeling van bijbehorende documentatie
2) compilatie van Pascal-files met nieuwe Pascal-compiler en nieuw operating system; oplossen van compileerproblemen (o.a. aanpassing script en vervanging van ontbrekende files en functie-libraries)
3) compilatie van samenhangende Pascal-modules (door verbeteren of vervangen van make-files)
4) herstel van overige files en modules, waaronder C-ISAM en het lexicon
5) heractivering vertaalsysteem door opsporen en wegwerken van runtime-errors bij het verwerken van concrete data
6) vervanging van het oude file-beheersysteem (RBS) t.b.v. de facilitering van nieuw ontwikkelwerk (waarbij het systeem modulegewijs moet kunnen worden getest)
7) ontwikkeling van een Windows-interface (of een andere grafische user-interface)

De door ons uitgevoerde voorstudie beperkte zich tot de onderdelen 1, 2 en 3. Op basis van de hieruit voortgekomen informatie kunnen we een grove prognose geven van de tijdsinspanning die nodig is voor het compleet herstellen van het Rosetta-systeem. Voor de overige onderdelen kunnen we alleen algemene inschatting geven van de benodigde werkzaamheden en de hierbij op te lossen problemen.

7.2 Prognose voor reparatie Pascal-modules

Zoals in hoofdstuk 6 werd uiteengezet, hebben we in het kader van deze voorstudie geprobeerd om de Pascal-component Amorph weer werkend te maken. Hiertoe zijn (wegens problemen met de make-files) eerst de onderliggende Pascal-modules opgespoord (voor zover beschikbaar) en is vervolgens geprobeerd om deze geschikt te maken voor compilatie met de GNU-compiler. Hiervoor bleken veel aanpassingen nodig te zijn, onder meer door verschillen in de vereiste syntax, een andere opzet van de module-structuur en het gebruik van afwijkende functie-libraries. Binnen de beschikbare tijd kon slechts een klein deel van deze problemen worden opgelost. Het is dan ook niet gelukt om de component Amorph volledig te compileren, laat staan functioneel werkend te maken. Toch is voldoende voortgang geboekt om een gemotiveerde prognose te kunnen afgeven voor de tijdsinspanning die nodig is voor het hercompileren van alle 1645 Pascal-modules in het actieve deel van het Rosetta-systeem.

Methode voor het voorspellen van de benodigde werktijd
De voorstudie heeft als doel om een schatting te maken van de totale werktijd die nodig is voor het herstellen van het Rosetta-systeem. Deze onderzoeksvraag kan worden onderverdeeld in een aantal deelvragen:

a) hoeveel tijd is er nodig om het hele Rosetta-systeem te compileren;
   - hoeveel tijd is er nodig voor het compileren van alle Pascal-files?

b) hoeveel tijd is er nodig om alle runtime-errors op te lossen?
   - hoeveel tijd is er nodig om de geheugenmodule te vervangen en werkend te maken?
   - hoeveel tijd is er nodig om C-ISAM te vervangen en werkend te maken?
   - hoeveel tijd is er nodig om de makefiles te vervangen en werkend te maken?
   - hoeveel tijd is er nodig om resterende runtime-errors op te lossen?

c) hoeveel tijd is er nodig om Rosetta (weer) functioneel werkend te krijgen?

Door de beperkte vorderingen bij het vooronderzoek kunnen we alleen een extrapolatie maken voor vraag a; voor de overige deelvragen kunnen we alleen een globale schatting afgeven op basis van onze algemene ervaring.

In beginsel zijn er twee mogelijkheden voor de extrapolatie van de resultaten uit het vooronderzoek:

1) door de genormaliseerde werktijd voor de compilatie van één component C (c.q. module-cluster) lineair te projecteren door dezete vermenigvuldigen met het aantal te repareren componenten;

2) door de werktijd per gecompileerde file lineair te projecteren (door vermenigvuldiging met een tijdsinterval) en eventueel een exponentiële versnellingsfactor toe te passen;

Toen we aan deze voorstudie begonnen, waren we van plan om methode 1 te gebruiken. Deze methode werkt echter alleen goed wanneer er een complete component is gecompileerd en als duidelijk is hoeveel componenten er zijn. Aan geen van beide condities wordt echter voldaan: we hebben slechts 10% van de geselecteerde component kunnen compileren, en het is verre van duidelijk hoeveel componenten er zijn. De functionele componenten die van tevoren bekend waren, blijken slechts een klein deel van de beschikbare modules te dekken, terwijl ze

onderling veel overlap vertonen. Hierdoor is het vrijwel onmogelijk om vanuit de resultaten van Amorph te extrapoleren naar de rest van de componenten.

Onze voorkeur gaat daarom uit naar een algoritme dat gebaseerd is op methode 2. Dit algoritme kan als volgt worden uitgewerkt:

r(F,t) = [W(t1) * t] ^ V(t)

verklaring van de symbolen:
F = fileset met alle Pascal-files in Rosetta (.p/.pf): ca. 1650 files
r(F,t) = aantal gerepareerde files in fileset F na t uur
W(t) = r(F,t) / t = werksnelheid = aantal verbeterde files per uur na t uur
V(t) = exponentiële versnellingsfactor op tijdstip t
A^E = A tot de macht E

t1 = tijdsduur van pilot-studie naar het compileren van Pascal files
r(t1) = aantal gecompileerde Pascal-files aan einde pilot-studie
f(t1) = verbetersnelheid aan eind pilot-studie (= aantal gerealiseerde verbeteringen t.o.v. het totaal aantal benodigde reparaties per t uur)
V(t) = 1 + f * t = de versnellingsfactor na t uur
W(t1) = r(F,t1) / t1 = werksnelheid = aantal verbeterde files per uur tijdens de pilot-studie

Men kan het bovenstaande algoritme als volgt beschrijven: gegeven een nulmeting van de gemiddelde werksnelheid W (gedefinieerd als het aantal gerepareerde files per tijdseenheid na tijdsinterval t1) kan men voor elk gewenst tijdstip na t1 (te weten tijdsinterval t) het aantal gerepareerde files voorspellen door de werksnelheid W met het tijdsinterval t te vermenigvuldigen en hier een exponentiële versnellingsfactor V op toe te passen.

Bij de beapling van de werksnelheid W blijft het voorwerk (zoals de systeemanalyse, de installatie van een geschikte compiler en het lezen van de compiler-manual) buiten beschouwing.

Men kan de versnellingsfactor V schatten door te kijken hoeveel procent van alle benodigde reparaties in tidjsinterval t1 is bereikt, of, equivalent, door te meten hoeveel de nog niet gecompileerde files dichterbij het compilatiepunt zijn gekomen.

Toepassing van het algoritme

Basisgegevens uit de nulmeting (tijdsinterval t1):
* tijdsinterval t1 = 46 uur
* er zijn 18 files gecompileerd
* er zijn 9 aanpassingen uitgevoerd
* per aanpassing worden dus twee files bereikt
* in totaal moeten er ruim1600 files worden aangepast
* het aantal benodigde aanpassingen wordt geschat op 800
* versnellingsfactor V = 9/800 = 0.01 aanpassingen per tijdseenheid
* werksnelheid W(t1) = r(F,t1) / t1 = 18 files / 46 uur = .39 file per uur

p(F,t) = [W(t1) * t]

r(F,t) = [p(F,t)] ^ V

t = 46 uur =>
V(t) = 1 + (.01 * 1) = 1.01
p(F,t) = [.39 * 1 * 46] = 18 files
r(F,t) = [.39 * 1 * 46] ^ 1.01 = 19 files

t = 2 * 46 uur =>
V(t) = 1 + (.01 * 2) = 1.02
p(F,t) = [.39 * 2 * 46] = 36 files
r(F,t) = [.39 * 2 * 46] ^ 1.02 = 39 files

t = 3 * 46 uur =>
V(t) = 1 + (.01 * 3) = 1.03
p(F,t) = [.39 * 3 * 46] = 54 files
r(F,t) = [.39 * 3 * 46] ^ 1.03 = 61 files

....

t = 23 * 46 uur =>
V(t) = 1 + (.01 * 24) = 1.23
p(F,t) = [.39 * 23 * 46] = 413 files
r(F,t) = [.39 * 23 * 46] ^ 1.23 = 1649 files (netto 413 files)

Uit deze berekening blijkt dat na een periode van 23 keer de duur van de pilotstudie (ofwel na 26 weken) alsnog het moment is bereikt waarop alle 1650 Pascal-files gecompileerd zullen zijn (voor zover dat mogelijk is), ofwel dat dit de helft van een werkjaar vergt. Bij deze prognose moet een flinke onzekerheidsfactor worden gehanteerd: het kan makkelijk een maand meer worden.

7.3 Kernproblemen

7.3.1 Geheugenmodule

Een groot probleem binnen het Rosetta-project is de reparatie van de geheugenmodule 'MEM'. Twee belangrijke modules die door deze module gebruikt worden, namelijk 'HEAP' en 'LOOPHOLES', zijn niet langer beschikbaar onder de nieuwe GPC compiler. Om de MEM module wederom werkend te krijgen, moet de functionaliteit van deze modules vervangen worden door bestaande GPC-mogelijkheden, of moeten ze worden herschreven.
De geheugenroutines die MEM aanbiedt, en die gebaseerd zijn op de missende modules, worden door de hele code van Rosetta heen gebruikt. De grootste problemen zijn te verwachten bij de modules voor de speciale boomstructuren. Deze vormen de basis van het Rosetta project, maar het is niet te verwachten dat een nieuwe geheugenmodule zich exact hetzelfde gedraagt als de oude module. Dit betekent dat er vele onvoorspelbare run-time errors op kunnen treden wanneer het programma gebruikt wordt, welke moeizaam te traceren zullen zijn. Het aanpassen van de geheugenmodule, en vervolgens het oplossen van de te verwachten run-time errors, zal minstens een hele maand in beslag nemen.

7.3.2 Reparatie C-ISAM

C-ISAM is een in C geprogrammeerde database-module. De bestaande ISAM-module is wel te compileren, maar een van de door deze module gebruikte libraries is niet meer beschikbaar. Een vervanging vinden van deze library is niet gelukt.
Er zijn ongeveer 45 Pascal-modules die de ISAM-module gebruiken. Een mogelijke oplossing is om de ISAM-module te vervangen door een nieuwe database-module, bijvoorbeeld via MySQL. Hiervoor moet dan een interface ontwikkeld worden die de oude ISAM-routines kan vertalen naar MySQL commando's. Onze inschatting is dat het vervangen van de complete database-module van Rosetta zeker een maand werk zal kosten.

7.3.3 Reparatie MAKE-files

De MAKE-files moeten grotendeels herschreven worden wegens het wegvallen van het oude filebeheersysteem (RBS) en de overgang naar een nieuwe module / interface-systematiek (als gevolg van de vervanging van de Pascal-compiler). De hiervoor benodigde action-scripts zijn waarschijnlijk niet langer bruikbaar en moeten dus ook worden vervangen.
Om de make-files te kunnen repareren, moeten eerst alle interfaces compileerbaar worden gemaakt. Hierna dienen de modules in een van tevoren vastgelegde volgorde worden gelinkt. Als de gewenste linking-volgorde van de modules afwijkt van het oude systeem, zullen de MAKE-files helemaal opnieuw moeten worden opgezet. Dit vereist per pascal-bestand (*.p) een overzicht van alle afhankelijkheden.
Voor een component als Amorph is het mogelijk om een schatting te maken. Voor deze component dienen ongeveer 50 compileerbare Pascal-modules in de juiste volgorde geïntegreerd te worden. Het analyseren van deze bestanden en het opzetten van de basis-structuur voor een make-file heeft ons een dag gekost. Het ontwikkelen en testen van de bijbehorende MAKE-routine zal waarschijnlijk nog een dag in beslag nemen.

Gegeven het feit dat er 408 p-modules moeten worden gecompileerd (die met 357 pf-modules corresponderen), zouden dus 800 (400 keer 2) dagen nodig zijn om de bijbehorende make-files te ontwikkelen en testen. Inmiddels is het analyseren van de file-afhankelijkheden inmiddels voor het grootste deel geautomatiseerd, zodat de werktijd voor volgende modules beperkt kan worden tot ca. 1 dag. Door nog meer aspecten te automatiseren (door hier een week voor uit te trekken), zal de werktijd per make-routine misschien teruggebracht kunnen worden tot ca. een kwartier (waarbij vooral problemen zijn te verwachten als gevolg van niet-bestaande modules of modules met afwijkende extensies). Zelfs dan zal nog steeds zo'n 100 uur nodig (c.q. 2,5 week) nodig zijn om alle make-routines aan te maken, ervan uitgaande dat de onderliggende p- en pf-files al compileerbaar zijn gemaakt.
Nadat voor alle modules een MAKE-routine is aangemaakt, moeten alle MAKE-files aan elkaar worden gekoppeld om het Rosetta-programma te bouwen. Deze laatste stap vergt waarschijnlijk enkele dagen. In totaal kost het dus zeker een maand om alle MAKE-files te repareren.

7.4 Eindprognose

Het in dit rapport beschreven onderzoek heeft uitgewezen dat de reparatie van het Rosetta-systeem wat betreft de in Pascal geschreven onderdelen een tijdrovende opgave is. Het onderzoek had als doel om een van tevoren geselecteerde component, te weten Amorph, geschikt te maken voor een nieuwe Pascal-compiler en te meten hoeveel tijd het kost om deze component volledig werkend te maken. Dit gegeven zou vervolgens als basis dienen voor een prognose van de totaal benodigde werktijd.

In dit hoofdstuk is de gevraagde prognose opgedeeld in een aantal deelvragen. Hierbij is de meeste aandacht uitgegaan naar de vraag hoeveel tijd nodig is voor het compileren van alle Pascal-modules uit het Rosetta-systeem. Voor de overige vragen is alleen een grove indicatie gegeven van de te verwachten problemen en de benodigde oplossingstijd. Hieronder wordt voor elke deelvraag aangegeven wat de bijbehorende tijdschatting is, waarna optelling van de deelprognoses een eindprognose oplevert. Die eindprognose luidt dat de reparatie van het Rosetta-systeem, met een onzekerheidsmarge van 4 maanden, zo'n 18 maanden zal duren, behoudens problemen die het gevolg zijn van ontbrekende modules en libraries; het oplossen van deze omissies valt buiten onze prognose.

vraag	**prognose**
hoeveel tijd is er nodig voor het compileren van alle Pascal-files?	6 (±1) maanden
hoeveel tijd is er nodig om de geheugenmodule te vervangen en werkend te maken?	1 maand
hoeveel tijd is er nodig om C-ISAM weer werkend te maken?	1 maand
hoeveel tijd is er nodig om de MAKE-files te vervangen en werkend te maken?	1 maand
hoeveel tijd is er nodig om resterende runtime-errors op te lossen?	6 (±2) maanden
hoeveel tijd is er nodig om Rosetta (weer) functioneel werkend te krijgen?	3 (±1) maanden
hoeveel tijd is er nodig voor een complete reparatie van Rosetta?	**18 (±4) maanden**

8 Conclusie

In het kader van het Stevin-project IRME kreeg GridLine de opdracht om een voorstudie uit te voeren naar de repareerbaarheid van het Rosetta-systeem. Hierbij diende GridLine in vier weken een prognose te maken van de benodigde reparatietijd door een pilot-onderzoek uit te voeren naar de repareerbaarheid van één complete Pascal-module.

Toen GridLine met deze opdracht aan de slag ging, bleek dat het project complexer was dan voorzien. Zo was de module-structuur van Rosetta minder transparant dan de functionele beschrijving deed vermoeden. De functionele modules waren namelijk niet zichtbaar in de directorystructuur, maar moesten gereconstrueerd worden door de make-files te analyseren. Dit leidde tot het inzicht dat de functionele modules in feite clusters zijn van een lager type modules, de operationele modules. Ook bleek dat de functionele modules elkaar flink overlappen en dat een deel van de operationele modules met andere functies corresponderen dan die van de functionele modules. De eigenschappen van deze operationele modules waren bovendien matig gedocumenteerd.

Ondanks deze complicaties is GridLine er toch in geslaagd om een functionele component te reconstrueren (namelijk Amorph) en deze gedeeltelijk te compileren. Hiertoe is eerst een systeemanalyse uitgevoerd, hetgeen inventarisatierapporten opleverde van de aanwezige files en hun extensies, hun onderlinge relaties en hun beschikbaarheid. Op basis van deze informatie kon een indruk worden verkregen van de omvang en de complexiteit van de reparatieklus. Vervolgens is uitgezocht welke Pascal-compiler een geschikte vervanging zou zijn van de oorspronkelijke Metaware-compiler. Hierna is ca. 46 uur besteed aan een poging om de Amorph-modules geschikt te maken voor compilatie met de nieuwe compiler, de GNU Pascal-compiler. Hiertoe is stap voor stap een programma ontwikkeld waarmee de benodigde aanpassingen automatisch konden worden doorgevoerd.

In deze procedure zijn in totaal 9 grote aanpassingen gerealiseerd, waardoor 18 van de 108 benodigde modules (waarvan er 22 bleken te ontbreken) volledig konden worden gecompileerd, terwijl de compilatie van de overige files dichterbij kwam. Hierbij werden alle niet-traceerbare file-verwijzingen genegeerd. Er zullen dan ook zeker runtime-errors ontstaan bij het verwerken van concrete data, die alleen kunnen worden opgelost door de betreffende modules te vervangen.

Als van deze complicaties wordt geabstraheerd, zal het hercompileren van het Pascal-gedeelte van het Rosetta-systeem ca. 6 maanden kosten. Voor de reparatie van de geheugenmodule, de database-module C-ISAM en de make-files verwachten we drie maanden nodig te hebben. Het oplossen van resterende runtime-errors kost waarschijnlijk zo'n 6 maanden. Om Rosetta (en het bijbehorende lexicon) weer functioneel werkend te maken kost nog eens 3 maanden. De eindprognose luidt dat de reparatie van het Rosetta-systeem, met een onzekerheidsmarge van 4 maanden, zo'n 18 maanden zal duren, behoudens problemen die het gevolg zijn van ontbrekende modules en libraries; het oplossen van deze omissies valt buiten onze prognose.

Het verdient aanbeveling om een tweede vooronderzoek op te zetten van ca. 8 weken. Dit vervolgonderzoek is nodig om de prognose voor het compilatietraject nader te testen, om uit te zoeken waarom er zoveel Pascal-modules lijken te ontbreken en om te experimenteren met de vervanging van een aantal cruciale componenten (zoals C-ISAM en de geheugenmodule). Dit tweede onderzoek zal een preciezere prognose moeten opleveren voor de inspanning die gemoeid is met de reparatie van het Rosettasysteem.