Eindhoven University of Technology
Department of Mathematics and Computing Science

Language for Describing Financial Products

by

B.R.T. Arnold, A. van Deursen and M. Res

95/22

ISSN 0926-4515

All rights reserved
editors: prof.dr. J.C.M. Baeten
prof.dr. M. Rem

Computing Science Report 95/22
Eindhoven, June 1995

An Algebraic Specification of a Language for Describing Financial Products

B.R.T. Arnold

Bank MeesPierson, Rotterdam; and CAP Volmac, Utrecht

P.O. Box 2575, 3500 GN Utrecht, The Netherlands

A. van Deursen*

Department of Mathematics and Computing Science

TU Eindhoven, 5600 MB P.O. Box 513, The Netherlands

arie@win.tue.nl, http://www.win.tue.nl/win/cs/fm/arie/

M. Res†

Royal Library, Prins Willem Alexanderhof 5, 2595 BE The Hague

The Netherlands. Email: martijn@konbib.nl

Abstract

We report on the use of formal methods and supporting tools during the development of a language applied in a banking environment. This language, called RISLA, is used to define the nature of the interest products offered by a bank. A RISLA description fixes the cash flows (amounts of money coming in or going out on particular dates) resulting from a product, and is used to generate COBOL code. The language has been developed with the use of algebraic specifications, the role of which is discussed.

Note: An extended abstract appeared under the same title in M. Wirsing (editor), *Proceedings of the ICSE-17 Workshop on Formal Methods Application in Software Engineering Practice*, Seattle, April 1995.

Keywords & Phrases: algebraic specifications, financial engineering, COBOL, formal methods experience, tool support;

*Work done while at CWI, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands.

†Work done while at the Programming Research Group of the University of Amsterdam.

1 Introduction

Among the most exciting banking activities is the inter-bank trade of *interest rate products* [Tuc91, Cog95]. Large amounts of money are transferred, in order to fulfill the bank's current or future financial needs. Interest rates can change at any moment, and trade in financial products is not without risk. These risks are further enlarged by the liberalization of the international financial market, making borrowing or lending in foreign currencies easier than ever.

The simplest interest rate product is the loan: a fixed amount in a certain currency is borrowed for a fixed period at a given interest rate. The risks involved here include, e.g., changes in the exchange rate of the currency or in the current market interest rate. More complicated products, such as the *financial future*, *option*, or the more recent *swap*, all aim at *risk reallocation*. Futures, e.g., fix a loan for a future period, and help the borrower to reduce the risk involved in rising interest rates. Every bank can invent new types of products, giving clients new risk reallocation opportunities. Whether the introduction of such a new instrument is a (financial) success largely depends on the bank's abilities to offer new products faster than the competition.

Offering a new product to a large number of clients, however, is not without consequences for the bank's automated systems. The *financial administration* needs to be able to process the new products (buyers, actual rates and amounts, interest payment periods, ...), and *management information systems* should provide bankers with up to date information concerning additional risks the bank is taking.

How can the bank's automated systems be made so flexible that they can easily process new kinds of products? CAP Volmac[1], commissioned by Bank MeesPierson[2], started working out the following scenario. For each interest rate product, a high-level description is given in a new language called RISLA[3]. The basic assumption is that a product can be characterized by describing its *cash flows*. A cash flow is a series of ⟨amount, date⟩ pairs, indicating an amount of money leaving or entering the bank at a certain date. The kernel of a product description consists of a set of rules defining how to compute cash flows from initial contract values. These product descriptions suffice to *generate* the necessary COBOL procedure code. The core of this code consists of call to COBOL routines taken from a MeesPierson library developed over the past years. The code can be invoked to enter new contracts (instantiated products) and to retrieve information from entered contracts. (As required by the financial administration of management information systems.)

The key to success in this scenario is RISLA. Therefore, CAP Volmac and Bank MeesPierson decided to use formal methods during the design of this language. A cooperation was started with CWI, the Dutch national center for mathematics and computer science. An algebraic specification describing the language was to be written [AD92, Res94, Deu94], using the formalism ASF+SDF and its supporting system [BHK89, Kli93].

[1]The work on RISLA was in particular initiated by the Orfis group — the Organization for Financial Information Systems; now part of CAP Volmac.

[2]The cooperation originally started with Bank Mees & Hope.

[3]A Dutch/English acronym for Rente (interest) Informatie Systeem LAnguage. Earlier documents use "RPM" as language name, an acronym for Rente Produkt Modelleringstaal.

In this document, we sketch an overview of the RISLA project, give the main ingredients of the specifications written, and discuss reasons of success, suggestions for improvement, and areas for further research.

2 Related Work

A completely independent, but remarkably similar project is described by Eggenschweiger and Gamma from the Union Bank of Switzerland [EG92]. They describe how ET++, a class library for C++, has been used to implement a swap valuation system.

The idea of using domain-specific languages to increase software productivity is studied at various sites. A recent Internet discussion on `comp.software-eng`, initiated by Philips Research Laboratories, included reactions from AT&T, Motorola, NASA, and various universities. The approach investigated by Philips is referred to as "end-user programming". For a certain mature application area, domain experts and software engineers together develop a domain-specific formalism (DSF). As part of that, they establish a mapping between language constructs and graphical user-interface (GUI) components (dialogues, pull-down menus, data-entry). The software experts implement this mapping and use compiler generation technology to build tools for the DSF. The domain specialists can use the language to build and maintain their future (GUI-based) applications.

The SDRR (Software Design for Reliability and Reuse) group from Oregon advocates the development of small, domain-specific design languages, and the use of these languages as front-ends to program generation systems [WH95]. Related to that is the Amphion approach in use at Nasa. Amphion is based on a formally documented library of geometrical (FORTRAN) routines. Domain experts state their problem in a high-level formalism, and theorem proving is used to find the right call solving that problem correctly [LPPU94].

3 Project Overview

Before the RISLA formalization started, a folder [AG92] concerning the language RISLA was available, containing a number of product definitions. The notation used looked relatively stable, although small variations were visible between different products, and certain descriptions resorted to ad-hoc extensions of the notation.

Various built-in functions were assumed. Some of these dealt with date manipulation, for instance when computing the number of working days during which interest has to be paid; depending on the currency certain days may or may not count (e.g., the fourth of July or 14me Juillet), depending on the date computation convention applicable for the particular product. Other functions reflected operations that were frequently needed when manipulating interest computations. These functions were either based on existing COBOL routines, or informally described in natural language.

No context-free (BNF) grammar for RISLA existed at that moment. Type checking requirements were implicit: it was assumed that certain operations were only allowed on operands of particular types, but this was not explicitly stated.

3.1 First Formalization

Spring 1992, Bank MeesPierson, CAP Volmac, and CWI agreed to start a two person-month project aiming at the further development the RISLA application language. A formal definition, using algebraic specifications, of RISLA was to be given, and some prototype tools, such as a syntax-directed editor and type checker were to be generated. Moreover, ten representative interest rate products had to be described using the newly developed RISLA, in order to illustrate its expressive power. The results of this project included a specification of the fundamental data types, a proposal for a context free grammar, and the definition of 10 example products [AD92].

In addition to developing the new language RISLA, CAP Volmac, CWI, and the University of Amsterdam (UvA), studied whether an *existing* language could be used. As it was conjectured that interest products involved certain "processes", like "pay interest", or "redeem a loan", a language tailored towards process description was chosen, namely the Process Specification Formalism PSF [MV90]. In a pilot study, four interest products were defined using PSF. Surprisingly, the process specification facilities were *not* used to define these products; the definitions only involved functions and data structures. Finally, it was decided to stop experimenting with existing languages. The main reasons for this were that RISLA was by that time sufficiently developed and (i) it seemed that all products could easily be defined in RISLA; (ii) these product descriptions looked the way CAP Volmac and Bank MeesPierson wanted them to look; and (iii) RISLA was so small that the extra time needed to build a specialized compiler was considered worth the effort.

3.2 Building Tools in C, Lex, and Yacc

The bank's reaction to the two-month formalization effort was positive. The increased understanding of the data types of RISLA, as well as the fixed context-free syntax strengthened the management's faith in the RISLA approach. Work continued by defining the remaining (30) interest products of Bank MeesPierson, which triggered several extensions of the RISLA syntax. At that moment it was decided to base all processing of Bank MeesPierson's interest rate products on RISLA descriptions.

The bank could have decided to use ASF+SDF to construct prototypes of the RISLA to COBOL compiler or a RISLA type checker. There were, however, several reasons not to do this. First of all, one should realize that the idea of using the specific RISLA language already was a small revolution within the bank. Introducing ASF+SDF in addition to RISLA would involve too many new things at the same time, and therefore endanger the acceptance of the RISLA project. Secondly, so far no experience existed with the use of ASF+SDF or algebraic specifications to generate COBOL programs. Lastly, the design phase of RISLA was considered to be completed, so a prototyping phase could safely be skipped.

For these reasons, the implementation of a RISLA to COBOL compiler was started immediately, using traditional tools like Lex, Yacc, and C. Type checking of RISLA specifications was not considered an urgent facility, so the analysis performed by the RISLA to COBOL compiler was at a minimal level.

In October 1994, after extensive testing of the generated COBOL programs (the bank's

auditors have to check that only correct money transfers can occur), the first RISLA-based product descriptions were fully operational.

3.3 Reverse Engineering

CWI could understand the decision to use traditional C and Lex/Yacc development tools, but naturally did not share the hesitations concerning the feasibility and benefits of using algebraic specifications for the full specification of RISLA, including type checking and compilation to COBOL. To support this, they decided to formalize type checking and to come up with a specified version of the RISLA to COBOL translation. This project was started in 1993, and took 6 person months to complete [Res94].

The bank's reaction to this project was favorable. While developing their C-based compiler, it turned out that the design of RISLA was not yet fully mature: consequently, it had to be changed several times. It was clear that (i) such changes in the design were more easily handled when using ASF+SDF, and that (ii) having started with a full specification in ASF+SDF, followed by a C-implementation once the design phase was fully completed, would have saved time and money. Moreover, the bank realized that a type checker is necessary, and that reasoning about the correctness of the compilation was easier using a formal specification.

3.4 Current Work

At the moment work transferring all interest rate product software to the RISLA approach is in full progress. A continuation of the cooperation between CWI and CAP Volmac is moreover planned: Starting June 1995, a re-design phase of will start, aiming at building a modular layer on top of RISLA, and at facilitating extensions to a graphical user interface easily. Six months are planned, during which time the bank's people will intensively use the ASF+SDF Meta-environment.

4 Specification Results

In order to appreciate of the specification activities involved in the RISLA project, we briefly discuss the specifications of some fundamental data types, the syntax and the translation to COBOL (see [AD92, Res94, Deu94] for the full specifications).

4.1 Data Types

The *cash flow* is the most important data type of RISLA. It can be modeled as a list of ⟨amount, date⟩ pairs ordered by date, An algebra of cash flows has been specified, introducing several elementary operations on flows, such as addition, netting, or merging (See Appendix A). Related to cash flows are *balances*, which are modeled as lists of ⟨amount, interval⟩ pairs, indicating that a certain amount of money will be available for the bank during the given date interval. Specifying the data types of RISLA [AD92, Appendix A.1] took

```
product LOAN

declaration
  contract data
    PAMOUNT : amount                          %% Principal Amount
    STARTDATE : date                          %% Starting date
    MATURDATE : date                          %% Maturity data
    INTRATE : int-rate                        %% Interest rate
    RDMLIST := [] : cashflow-list             %% List of redemptions.
  information
    PAF : cashflow-list                       %% Resulting Principal Amount Flow
    IAF : cashflow-list                       %% Resulting Interest Amount Flow
  registration
    RDM(AMOUNT : amount, DATUM : date)        %% Register one redemption.
  local
    FPA(CHFLLIST : cashflow-list) : amount    %% Final Principal Amount
    FRDM : cashflow                           %% Final redemption
  error checks
    "Wrong term dates" in case of STARTDATE >= MATURDATE
    "Negative amount"  in case of PAMOUNT < 0.0

implementation
  local
    define FPA  as IBD(CHFLLIST, -/-PAMOUNT, MATURDATE)
    define FRDM as <-/-FPA(RDMLIST), MATURDATE>
  information
    define PAF as [<-/-PAMOUNT, STARTDATE>] >> RDMLIST >> [FRDM]
    define IAF as [< -/-CIA( BL(RDMLIST, <-/-PAMOUNT, <STARTDATE, MATURDATE>>),
                             INTRATE ),  MATURDATE >]
  registration
    define RDM as
      error checks
         "Date not in interval" in case of (DATUM < STARTDATE) or (DATUM >= MATURDATE)
         "Negative amount"      in case of AMOUNT <= 0.0
         "Amount too big"       in case of FPA(RDMLIST >> [<AMOUNT, DATUM>]) > 0.0
      RDMLIST := RDMLIST >> [<AMOUNT, DATUM>]
```

Figure 1: A description of a loan in Risla.

about 15 modules, covering 25 pages, using about 100 equations were needed to specify these data types (excluding the modules defining standard data types like Booleans and Integers).

Although constructing these data type specifications was straightforward, it had a clarifying effect. Before, a question like "what is a cash flow" resulted in an answer explaining that it was something which could be used to deal with flows of money resulting from interest products. Later on, the answer became that a cash flow is an element of the sort CASH-FLOW. Secondly, the specifications fixed the argument and result types of many of the built-in functions. In most cases this was again straightforward, but in others various people involved in the project had different opinions about, e.g., the number of arguments given to functions used in RISLA definitions. In those cases, the formal specification served as a fruitful design aid.

module Compile-Error-Checks
imports COBOL Risla Envs Comp-Exprs
exports
context-free syntax
$cp[\![$ CHECK $]\!]_{\text{CP-ENV}}$ → CALC
error-code(ERR-MESSAGE) → NAT
variables
$\mathcal{E}$ → CP-ENV
DeclSet → DECL-SET
Mess → RIS-MESSAGE
ϵ → RIS-EXPR
$\mathcal{S}^+$ → COBOL-STATS

equations
[1] $cp[\![\ \epsilon\ ]\!]_{\mathcal{E}}^{bool-exp} = \langle\ DeclSet,\ \mathcal{S}^+\ \rangle$

$cp[\![$ *Mess* IN CASE OF $\epsilon\ ]\!]_{\mathcal{E}} =$
⟨ *DeclSet*,
IF RET-CODE EQUAL *val(TRUE)*
$\mathcal{S}^+$
IF RET-CODE EQUAL *val(FALSE)*
MOVE *err-code(Mess)* TO RET-CODE
ELSE MOVE 0 TO RET-CODE
END-IF
END-IF ⟩

Figure 2: An example compilation equation

4.2 Risla Syntax

The RISLA language can be used to manipulate these data types. Each RISLA description defines a *product*, for which a name is given and, most importantly, functions computing the resulting cash-flows are defined. A simple definition for a *loan* is shown in Figure 1 (taken from [Res94]). The *contract data* of a loan are the principal amount, start and maturity date, and interest rate, which are given a value whenever an instance of a loan is created (i.e., when two parties have agreed on a contract). The initially empty RDMLIST indicates at what dates the loan is redeemed, and with which amounts. The *information* that can be extracted from a loan are the principal amount flow PAF, and the interest amount flow IAF. The *implementation* section defines how these cash flow lists can be computed, taking, e.g., the redemption list into account. Several built-in functions on cash-flows and balances are used, such as IBD, "Initial Balance on Date", or CIA, "Calculate Interest Amount".

The original folder on RISLA [AG92] did not contain a context-free grammar for RISLA. Together with the author of most of the initial example product definitions, we constructed an initial version of a grammar, tested whether a simple product description fitted well in this syntax, updated the syntax where necessary, and tried the next product. After several iterations, we had defined ten products ranging from the simple Loan to the "Fixed Rate Agreement", and a stable syntax. The interactive setting of the ASF+SDF Meta-environment proved ideal for these purposes.

4.3 COBOL Generation

Although it is quite standard to use algebraic specifications to describe translations between languages, it was intriguing to specify a translation of RISLA to COBOL. For mapping RISLA abstract syntax trees (ASTs) to COBOL ASTs the specification has to build up an environment containing the user-defined variables and functions, compute intermediate values, infer types, and so on. Figure 2 shows one equation handling error checks. The condition is used to produce the COBOL code for evaluating the case expression. The COBOL code (in concrete syntax, written in teletype) in the right-hand side of the conclusion

inspects the result value and sets the RET-CODE variable accordingly.

Using term rewriting this specification can be executed, thus yielding a (not very efficient, but useful) RISLA to COBOL compiler. Note that this term rewriting systems produces an abstract syntax tree, and that proper pretty printing of this COBOL tree is essential (indenting at correct column levels). We were able to use the pretty print generator implemented by Van den Brand for these purposes [Bra93].

5 Discussion

- Financial engineering is extremely suitable as application area for formal methods. There are, of course, no safety critical issues, as, e.g., in nuclear or railway applications, but the financial damage due to incorrect data provided by management information systems can be considerable. Moreover, the arithmetical nature of interest computations makes formalizations over financial products relatively easy.

- The formal method used, algebraic specification, is easy to learn, and is moreover supported by tools. In order to get a first grip on algebraic specifications, all one needs to understand is the simple notion of term rewriting. For this reason, we were able to involve employees of CAP Volmac and Bank MeesPierson actively in the project. The incremental syntax-directed editing facilities of the ASF+SDF Meta-environment gave immediate feedback whenever the user made syntactic errors.

- The crucial parts of the specification must be executable; The possibility to test the specification and to validate that the functions defined indeed have the expected behavior is essential for building up confidence in the formalization choices.

- Algebraic specifications typically offer little (or no) built-in data types. The importance of a well-developed library of modules defining (variations of) standard data types cannot be overestimated: it is impossible to explain to experienced software developers that now they are using sophisticated new technology they have to start defining an integer module themselves. For RISLA we used library versions for the Booleans and Integers. We also needed Real numbers, but had to write our own specification for these.

- Some support for record types with named fields is desirable. Record types are not specifically supported by algebraic specifications, and neither are they by ASF+SDF. Such types can be introduced as abbreviations for clean specifications, with an underlying efficient implementation.

- The problem when applying formal methods in real-life is not to write the specification, but to relate the clean specification to the dirty running systems. We addressed this by describing the translation of RISLA to sequences of COBOL routine calls, but have the impression that much more research in this area is necessary.

- The formalism used should support the notation with which the application specialists are familiar. Naturally, this is of utmost importance if the specification involves the definition of languages. The use of concrete syntax in ASF+SDF equations makes the translation equations look as natural as is possible.

- The project helps to compare non-formal and formal language definitions. Initially, a small formalization was performed; then the language was implemented in Lex/Yacc/C, and then re-engineered using formal methods.

  Every one involved in the project agrees that it would have been better had the full formalization been undertaken immediately. The design of RISLA was not yet finished, while implementation was started already, requiring several time consuming re-implementations. During a formalization, such changes in the design are processed more easily.

- Theorem proving played no role. Conform Hall, *the fact is that formal methods are all about specifications* [Hal90], we only *described* the RISLA language formally.

- Crucial for the success of a formal methods project is the attitude taken by the partners. We were greatly helped by a clear separation of expertise (financial specialist versus specification guru), respect for each other's decisions, and an eagerness to learn about the other's specialism.

References

[AD92] B.R.T. Arnold and A. van Deursen. Algebraic specification of a language defining interest rate products. CWI, Amsterdam; ORFIS International, Huis ter Heide, 1992.

[AG92] B.R.T Arnold and H. Gouw. Internal draft papers on RPM. ORFIS International, Huis ter Heide, The Netherlands, 1992.

[BHK89] J.A. Bergstra, J. Heering, and P. Klint, editors. *Algebraic Specification*. ACM Press Frontier Series. The ACM Press in co-operation with Addison-Wesley, 1989.

[Bra93] M.G.J. van den Brand. Prettyprinting without losing comments. Report P9315, University of Amsterdam, 1993. Available by *ftp* from ftp.cwi.nl:/pub/gipe/reports as Bra93.ps.Z.

[Cog95] Ph. Coggan. *The Money Machine: How the City Works*. Pinguin, 1995. Third edition.

[Deu94] A. van Deursen. *Executable Language Definitions: Case Studies and Origin Tracking Techniques*. PhD thesis, University of Amsterdam, 1994.

[EG92] Th. Eggenschwiler and E. Gamma. ET++ SwapsManager: Using object technology in the financial engineering domain. In *OOPSLA'92 Seventh Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 166–177. ACM, 1992. SIGPLAN Notices 27(10).

[Hal90] A. Hall. Seven myths of formal methods. *IEEE Software*, 7(5):11–19, 1990.

[Kli93] P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2(2):176–201, 1993.

[LPPU94] M. Lowry, A. Philpot, T. Pressburger, and I. Underwood. A formal approach to domain-oriented software design environments. In *KBSE'94*, 1994.

[MV90] S. Mauw and G.J. Veltink. A process specification formalism. *Fundamenta Informaticae*, 12:85–139, 1990.

[Res94] M. Res. A generated programming environment for RISLA, a specification language defining financial products. Master's thesis, Progr. Res. Group, Univ. of Amsterdam, 1994.

[Tuc91] A.L. Tucker. *Financial Futures, Options, and Swaps*. West Publishing Company, 1991.

[WH95] L. Walton and J. Hook. Design automation: Making formal methods relevant. In M. Wirsing, editor, *ICSE-17 Workshop on Formal Methods Application*, pages 93–100, April 1995.

A Flows

In this module, we list the ASF+SDF code of a slightly simplified definition of the cash flows, based on [AD92].

imports Dates

exports

sorts CASH-FLOW FLOW-LIST

context-free syntax

"⟨" "*amount*:" INT "," "*date*:" DATE "⟩" → CASH-FLOW
CASH-FLOW "." *date* → DATE
CASH-FLOW "." *amount* → INT
CASH-FLOW "." *amount* ":=" INT → CASH-FLOW

"[" {CASH-FLOW ","}* "]" → FLOW-LIST
ins(CASH-FLOW, FLOW-LIST) → FLOW-LIST
merge(FLOW-LIST, FLOW-LIST) → FLOW-LIST
nett(FLOW-LIST) → FLOW-LIST
split-before(FLOW-LIST, DATE) → FLOW-LIST
split-after(FLOW-LIST, DATE) → FLOW-LIST
inv(FLOW-LIST) → FLOW-LIST
FLOW-LIST "−" FLOW-LIST → FLOW-LIST {**left**}
FLOW-LIST "+" FLOW-LIST → FLOW-LIST {**left**}

hiddens

context-free syntax

sorted-insert(CASH-FLOW, FLOW-LIST) → FLOW-LIST

exports

variables

C [0-9]* → CASH-FLOW
C [0-9]*"*" → {CASH-FLOW ","}*
F [0-9]* → FLOW-LIST

equations

Querying and updating a single cash flow

$$\langle \textit{amount}\text{: } N, \textit{date}\text{: } D\rangle \text{ . } \textit{date} = D \quad [1]$$

$$\langle \textit{amount}\text{: } N, \textit{date}\text{: } D\rangle \text{ . } \textit{amount} = N \quad [2]$$

$$\langle amount\colon N_1,\ date\colon D\rangle\ .\ amount := N_2\ =\ \langle amount\colon N_2,\ date\colon D\rangle \quad [3]$$

Inserting elements in a sorted list

$$sorted\text{-}insert(C, []) = [C] \quad [4]$$

$$\frac{C_1\ .\ date \leq C_2\ .\ date = true}{sorted\text{-}insert(C_1, [C_2, C^*]) = [C_1, C_2, C^*]} \quad [5]$$

$$\frac{C_1\ .\ date \leq C_2\ .\ date = false}{sorted\text{-}insert(C_1, [C_2, C^*]) = sorted\text{-}insert(C_2, sorted\text{-}insert(C_1, [C^*]))} \quad [6]$$

Insertion of non-empty elements.

$$ins(C, F) = F \quad \textbf{when}\ \ C\ .\ amount = 0 \quad [7]$$

$$ins(C, F) = sorted\text{-}insert(C, F) \quad \textbf{when}\ \ C\ .\ amount \neq 0 \quad [8]$$

Merging two flow lists.

$$merge(F, []) = F \quad [9]$$

$$merge(F, [C, C^*]) = ins(C, merge(F, [C^*])) \quad [10]$$

Making dates unique, netting the amounts.

$$nett([]) = [] \quad [11]$$

$$nett([C]) = [C] \quad [12]$$

$$\frac{C_1\ .\ date = C_2\ .\ date}{nett([C_1, C_2, C^*]) = nett(ins(C_1\ .\ amount := C_1\ .\ amount + C_2\ .\ amount, [C^*]))} \quad [13]$$

$$\frac{C_1\ .\ date \neq C_2\ .\ date}{nett([C_1, C_2, C^*]) = ins(C_1, nett([C_2, C^*]))} \quad [14]$$

"Arithmetic" operations

$$inv([]) = [] \quad [15]$$

$$inv([C, C^*]) = ins(C\ .\ amount := -\ C\ .\ amount, inv([C^*])) \quad [16]$$

$$F_1 + F_2 = nett(merge(F_1, F_2)) \quad [17]$$

$$F_1 - F_2 = F_1 + inv(F_2) \quad [18]$$

Taking first and second parts of a flow list

$$split\text{-}before([], D) = [] \quad [19]$$

$$split\text{-}before([C^*, C], D) = [C^*, C] \quad \textbf{when}\ \ C\ .\ date < D = true \quad [20]$$

$$split\text{-}before([C^*, C], D) = split\text{-}before([C^*], D) \quad \textbf{when}\ \ C\ .\ date \geq D = true \quad [21]$$

$$split\text{-}after(F, D) = F - split\text{-}before(F, D) \quad [22]$$

Computing Science Reports

In this series appeared:

Department of Mathematics and Computing Science
Eindhoven University of Technology

93/01	R. van Geldrop	Deriving the Aho-Corasick algorithms: a case study into the synergy of programming methods, p. 36.
93/02	T. Verhoeff	A continuous version of the Prisoner's Dilemma, p. 17
93/03	T. Verhoeff	Quicksort for linked lists, p. 8.
93/04	E.H.L. Aarts J.H.M. Korst P.J. Zwietering	Deterministic and randomized local search, p. 78.
93/05	J.C.M. Baeten C. Verhoef	A congruence theorem for structured operational semantics with predicates, p. 18.
93/06	J.P. Veltkamp	On the unavoidability of metastable behaviour, p. 29
93/07	P.D. Moerland	Exercises in Multiprogramming, p. 97
93/08	J. Verhoosel	A Formal Deterministic Scheduling Model for Hard Real-Time Executions in DEDOS, p. 32.
93/09	K.M. van Hee	Systems Engineering: a Formal Approach Part I: System Concepts, p. 72.
93/10	K.M. van Hee	Systems Engineering: a Formal Approach Part II: Frameworks, p. 44.
93/11	K.M. van Hee	Systems Engineering: a Formal Approach Part III: Modeling Methods, p. 101.
93/12	K.M. van Hee	Systems Engineering: a Formal Approach Part IV: Analysis Methods, p. 63.
93/13	K.M. van Hee	Systems Engineering: a Formal Approach Part V: Specification Language, p. 89.
93/14	J.C.M. Baeten J.A. Bergstra	On Sequential Composition, Action Prefixes and Process Prefix, p. 21.
93/15	J.C.M. Baeten J.A. Bergstra R.N. Bol	A Real-Time Process Logic, p. 31.
93/16	H. Schepers J. Hooman	A Trace-Based Compositional Proof Theory for Fault Tolerant Distributed Systems, p. 27
93/17	D. Alstein P. van der Stok	Hard Real-Time Reliable Multicast in the DEDOS system, p. 19.
93/18	C. Verhoef	A congruence theorem for structured operational semantics with predicates and negative premises, p. 22.
93/19	G-J. Houben	The Design of an Online Help Facility for ExSpect, p.21.
93/20	F.S. de Boer	A Process Algebra of Concurrent Constraint Programming, p. 15.
93/21	M. Codish D. Dams G. Filé M. Bruynooghe	Freeness Analysis for Logic Programs - And Correctness, p. 24
93/22	E. Poll	A Typechecker for Bijective Pure Type Systems, p. 28.
93/23	E. de Kogel	Relational Algebra and Equational Proofs, p. 23.
93/24	E. Poll and Paula Severi	Pure Type Systems with Definitions, p. 38.
93/25	H. Schepers and R. Gerth	A Compositional Proof Theory for Fault Tolerant Real-Time Distributed Systems, p. 31.
93/26	W.M.P. van der Aalst	Multi-dimensional Petri nets, p. 25.
93/27	T. Kloks and D. Kratsch	Finding all minimal separators of a graph, p. 11.
93/28	F. Kamareddine and R. Nederpelt	A Semantics for a fine λ-calculus with de Bruijn indices, p. 49.
93/29	R. Post and P. De Bra	GOLD, a Graph Oriented Language for Databases, p. 42.
93/30	J. Deogun T. Kloks D. Kratsch H. Müller	On Vertex Ranking for Permutation and Other Graphs, p. 11.

93/31	W. Körver	Derivation of delay insensitive and speed independent CMOS circuits, using directed commands and production rule sets, p. 40.
93/32	H. ten Eikelder and H. van Geldrop	On the Correctness of some Algorithms to generate Finite Automata for Regular Expressions, p. 17.
93/33	L. Loyens and J. Moonen	ILIAS, a sequential language for parallel matrix computations, p. 20.
93/34	J.C.M. Baeten and J.A. Bergstra	Real Time Process Algebra with Infinitesimals, p.39.
93/35	W. Ferrer and P. Severi	Abstract Reduction and Topology, p. 28.
93/36	J.C.M. Baeten and J.A. Bergstra	Non Interleaving Process Algebra, p. 17.
93/37	J. Brunekreef J-P. Katoen R. Koymans S. Mauw	Design and Analysis of Dynamic Leader Election Protocols in Broadcast Networks, p. 73.
93/38	C. Verhoef	A general conservative extension theorem in process algebra, p. 17.
93/39	W.P.M. Nuijten E.H.L. Aarts D.A.A. van Erp Taalman Kip K.M. van Hee	Job Shop Scheduling by Constraint Satisfaction, p. 22.
93/40	P.D.V. van der Stok M.M.M.P.J. Claessen D. Alstein	A Hierarchical Membership Protocol for Synchronous Distributed Systems, p. 43.
93/41	A. Bijlsma	Temporal operators viewed as predicate transformers, p. 11.
93/42	P.M.P. Rambags	Automatic Verification of Regular Protocols in P/T Nets, p. 23.
93/43	B.W. Watson	A taxonomy of finite automata construction algorithms, p. 87.
93/44	B.W. Watson	A taxonomy of finite automata minimization algorithms, p. 23.
93/45	E.J. Luit J.M.M. Martin	A precise clock synchronization protocol,p.
93/46	T. Kloks D. Kratsch J. Spinrad	Treewidth and Patwidth of Cocomparability graphs of Bounded Dimension, p. 14.
93/47	W. v.d. Aalst P. De Bra G.J. Houben Y. Kornatzky	Browsing Semantics in the "Tower" Model, p. 19.
93/48	R. Gerth	Verifying Sequentially Consistent Memory using Interface Refinement, p. 20.
94/01	P. America M. van der Kammen R.P. Nederpelt O.S. van Roosmalen H.C.M. de Swart	The object-oriented paradigm, p. 28.
94/02	F. Kamareddine R.P. Nederpelt	Canonical typing and Π-conversion, p. 51.
94/03	L.B. Hartman K.M. van Hee	Application of Marcov Decision Processe to Search Problems, p. 21.
94/04	J.C.M. Baeten J.A. Bergstra	Graph Isomorphism Models for Non Interleaving Process Algebra, p. 18.
94/05	P. Zhou J. Hooman	Formal Specification and Compositional Verification of an Atomic Broadcast Protocol, p. 22.
94/06	T. Basten T. Kunz J. Black M. Coffin D. Taylor	Time and the Order of Abstract Events in Distributed Computations, p. 29.
94/07	K.R. Apt R. Bol	Logic Programming and Negation: A Survey, p. 62.
94/08	O.S. van Roosmalen	A Hierarchical Diagrammatic Representation of Class Structure, p. 22.
94/09	J.C.M. Baeten J.A. Bergstra	Process Algebra with Partial Choice, p. 16.

94/10	T. verhoeff	The testing Paradigm Applied to Network Structure. p. 31.
94/11	J. Peleska C. Huizing C. Petersohn	A Comparison of Ward & Mellor's Transformation Schema with State- & Activitycharts, p. 30.
94/12	T. Kloks D. Kratsch H. Müller	Dominoes, p. 14.
94/13	R. Seljée	A New Method for Integrity Constraint checking in Deductive Databases, p. 34.
94/14	W. Peremans	Ups and Downs of Type Theory, p. 9.
94/15	R.J.M. Vaessens E.H.L. Aarts J.K. Lenstra	Job Shop Scheduling by Local Search, p. 21.
94/16	R.C. Backhouse H. Doombos	Mathematical Induction Made Calculational, p. 36.
94/17	S. Mauw M.A. Reniers	An Algebraic Semantics of Basic Message Sequence Charts, p. 9.
94/18	F. Kamareddine R. Nederpelt	Refining Reduction in the Lambda Calculus, p. 15.
94/19	B.W. Watson	The performance of single-keyword and multiple-keyword pattern matching algorithms, p. 46.
94/20	R. Bloo F. Kamareddine R. Nederpelt	Beyond β-Reduction in Church's $\lambda\rightarrow$, p. 22.
94/21	B.W. Watson	An introduction to the Fire engine: A C++ toolkit for Finite automata and Regular Expressions.
94/22	B.W. Watson	The design and implementation of the FIRE engine: A C++ toolkit for Finite automata and regular Expressions.
94/23	S. Mauw and M.A. Reniers	An algebraic semantics of Message Sequence Charts, p. 43.
94/24	D. Dams O. Grumberg R. Gerth	Abstract Interpretation of Reactive Systems: Abstractions Preserving $\forall$CTL*, $\exists$CTL* and CTL*, p. 28.
94/25	T. Kloks	$K_{1,3}$-free and W_4-free graphs, p. 10.
94/26	R.R. Hoogerwoord	On the foundations of functional programming: a programmer's point of view, p. 54.
94/27	S. Mauw and H. Mulder	Regularity of BPA-Systems is Decidable, p. 14.
94/28	C.W.A.M. van Overveld M. Verhoeven	Stars or Stripes: a comparative study of finite and transfinite techniques for surface modelling, p. 20.
94/29	J. Hooman	Correctness of Real Time Systems by Construction, p. 22.
94/30	J.C.M. Baeten J.A. Bergstra Gh. Ştefanescu	Process Algebra with Feedback, p. 22.
94/31	B.W. Watson R.E. Watson	A Boyer-Moore type algorithm for regular expression pattern matching, p. 22.
94/32	J.J. Vereijken	Fischer's Protocol in Timed Process Algebra, p. 38.
94/33	T. Laan	A formalization of the Ramified Type Theory, p.40.
94/34	R. Bloo F. Kamareddine R. Nederpelt	The Barendregt Cube with Definitions and Generalised Reduction, p. 37.
94/35	J.C.M. Baeten S. Mauw	Delayed choice: an operator for joining Message Sequence Charts, p. 15.
94/36	F. Kamareddine R. Nederpelt	Canonical typing and Π-conversion in the Barendregt Cube, p. 19.
94/37	T. Basten R. Bol M. Voorhoeve	Simulating and Analyzing Railway Interlockings in ExSpect, p. 30.
94/38	A. Bijlsma C.S. Scholten	Point-free substitution, p. 10.

94/39	A. Blokhuis T. Kloks		On the equivalence covering number of splitgraphs, p. 4.
94/40	D. Alstein		Distributed Consensus and Hard Real-Time Systems, p. 34.
94/41	T. Kloks D. Kratsch		Computing a perfect edge without vertex elimination ordering of a chordal bipartite graph, p. 6.
94/42	J. Engelfriet J.J. Vereijken		Concatenation of Graphs, p. 7.
94/43	R.C. Backhouse M. Bijsterveld		Category Theory as Coherently Constructive Lattice Theory: An Illustration, p. 35.
94/44	E. Brinksma R. Gerth W. Janssen S. Katz M. Poel C. Rump	J. Davies S. Graf B. Jonsson G. Lowe A. Pnueli J. Zwiers	Verifying Sequentially Consistent Memory, p. 160
94/45	G.J. Houben		Tutorial voor de ExSpect-bibliotheek voor "Administratieve Logistiek", p. 43.
94/46	R. Bloo F. Kamareddine R. Nederpelt		The λ-cube with classes of terms modulo conversion, p. 16.
94/47	R. Bloo F. Kamareddine R. Nederpelt		On Π-conversion in Type Theory, p. 12.
94/48	Mathematics of Program Construction Group		Fixed-Point Calculus, p. 11.
94/49	J.C.M. Baeten J.A. Bergstra		Process Algebra with Propositional Signals, p. 25.
94/50	H. Geuvers		A short and flexible proof of Strong Normalazation for the Calculus of Constructions, p. 27.
94/51	T. Kloks D. Kratsch H. Müller		Listing simplicial vertices and recognizing diamond-free graphs, p. 4.
94/52	W. Penczek R. Kuiper		Traces and Logic, p. 81
94/53	R. Gerth R. Kuiper D. Peled W. Penczek		A Partial Order Approach to Branching Time Logic Model Checking, p. 20.
95/01	J.J. Lukkien		The Construction of a small CommunicationLibrary, p.16.
95/02	M. Bezem R. Bol J.F. Groote		Formalizing Process Algebraic Verifications in the Calculus of Constructions, p.49.
95/03	J.C.M. Baeten C. Verhoef		Concrete process algebra, p. 134.
95/04	J. Hidders		An Isotopic Invariant for Planar Drawings of Connected Planar Graphs, p. 9.
95/05	P. Severi		A Type Inference Algorithm for Pure Type Systems, p.20.
95/06	T.W.M. Vossen M.G.A. Verhoeven H.M.M. ten Eikelder E.H.L. Aarts		A Quantitative Analysis of Iterated Local Search, p.23.
95/07	G.A.M. de Bruyn O.S. van Roosmalen		Drawing Execution Graphs by Parsing, p. 10.
95/08	R. Bloo		Preservation of Strong Normalisation for Explicit Substitution, p. 12.
95/09	J.C.M. Baeten J.A. Bergstra		Discrete Time Process Algebra, p. 20
95/10	R.C. Backhouse R. Verhoeven O. Weber		Math∫pad: A System for On-Line Prepararation of Mathematical Documents, p. 15

95/11	R. Seljée	Deductive Database Systems and integrity constraint checking, p. 36.
95/12	S. Mauw and M. Reniers	Empty Interworkings and Refinement Semantics of Interworkings Revised, p. 19.
95/13	B.W. Watson and G. Zwaan	A taxonomy of sublinear multiple keyword pattern matching algorithms, p. 26.
95/14	A. Ponse, C. Verhoef, S.F.M. Vlijmen (eds.)	De proceedings: ACP'95, p.
95/15	P. Niebert and W. Penczek	On the Connection of Partial Order Logics and Partial Order Reduction Methods, p. 12.
95/16	D. Dams, O. Grumberg, R. Gerth	Abstract Interpretation of Reactive Systems: Preservation of CTL*, p. 27.
95/17	S. Mauw and E.A. van der Meulen	Specification of tools for Message Sequence Charts, p. 36.
95/18	F. Kamareddine and T. Laan	A Reflection on Russell's Ramified Types and Kripke's Hierarchy of Truths, p. 14.
95/19	J.C.M. Baeten and J.A. Bergstra	Discrete Time Process Algebra with Abstraction, p. 15.
95/20	F. van Raamsdonk and P. Severi	On Normalisation, p. 33.
95/21	A. van Deursen	Axiomatizing Early and Late Input by Variable Elimination, p. 44.