The United Nations University

UNU/IIST

International Institute for
Software Technology

A Financial Information System

Đỗ Tiến Dũng, Chris George, Hoàng Xuân Huấn, Phùng Phương Nam

July 1997

UNU/IIST Report No. 115

T

UNU/IIST

UNU/IIST enables developing countries to attain self-reliance in software technology by: (i) their own development of high integrity computing systems, (ii) highest level post-graduate university teaching, (iii) international level research, and, through the above, (iv) use of as sophisticated software as reasonable.

UNU/IIST contributes through: (a) advanced, joint industry–university advanced development projects in which rigorous techniques supported by semantics-based tools are applied in case studies to software systems development, (b) own and joint university and academy institute research in which new techniques for *(1) application domain* and computing platform modelling, *(2) requirements capture*, and *(3) software design & programming* are being investigated, (c) advanced, post-graduate and post-doctoral level courses which typically teach Design Calculi oriented software development techniques, (d) events [panels, task forces, workshops and symposia], and (e) dissemination.

Application-wise, the advanced development projects presently focus on software to support large-scale infrastructure systems such as transport systems (railways, airlines, air traffic, etc.), manufacturing industries, public administration, telecommunications, etc., and are thus aligned with UN and International Aid System concerns. UNU/IIST is a leading software technology centre in the area of infrastructure software development.

UNU/IIST is also a leading research centre in the area of Duration Calculi, i.e. techniques applicable to *real-time, reactive, hybrid & safety critical systems*. The research projects parallel and support the advanced development projects.

At present, the technical focus of UNU/IIST in all of the above is on applying, teaching, researching, and disseminating Design Calculi oriented techniques and tools for trustworthy software development. UNU/IIST currently emphasises techniques that permit proper development steps and interfaces. UNU/IIST also endeavours to promulgate sound project and product management principles.

UNU/IIST's primary dissemination strategy is to act as a clearing house for reports from research and technology centres in industrial countries to industries and academic institutions in developing countries. At present more than 200 institutions worldwide contribute to UNU/IIST's report collection while UNU/IIST at the same time subscribes to more than 125 international scientific and technical journals. Information on reports received (and produced) and on journal articles is to be disseminated regularly to developing country centres — which are then free to order a reasonable number of report and article copies from UNU/IIST.

Dines Bjørner, Director — 02.7.1992 – 01.7.1997
Zhou Chaochen, Director — 01.8.1997 – 31.7.2001

UNU/IIST Reports are either $\mathcal{R}$esearch, $\mathcal{T}$echnical, $\mathcal{C}$ompendia or $\mathcal{A}$dministrative reports:

$\boxed{\mathcal{R}}$ Research Report • $\boxed{\mathcal{T}}$ Technical Report • $\boxed{\mathcal{C}}$ Compendium • $\boxed{\mathcal{A}}$ Administrative Report

The United Nations
University

UNU/IIST

International Institute for
Software Technology

P.O. Box 3058
Macau

A Financial Information System

Đỗ Tiến Dũng, Chris George, Hoàng Xuân Huấn, Phùng Phương Nam

Abstract

In this document we continue the work started in "Developing a Financial Information System", UNU/IIST Research Report 81. That document concentrated on the taxation system and on its detailed development. Here we take a broader view, sketching the taxation system and also the budget, treasury and external aid and loan systems. Then we show how these may all be combined, allowing not only "vertical" communication within each system but also "horizontal" communication between components of different systems. We thus provide a top-level specification of a national financial information system.

Đỗ Tiến Dũng, and Phùng Phương Nam are fellows at UNU/IIST, 28 March - 14 April 1997.

Đỗ Tiến Dũng and Phùng Phương Nam are staff at the Vietnamese Ministry of Finance, under the Information Technology Applications and Management Committee.

Hoàng Xuân Huấn is fellow at UNU/IIST, 15 January 1997 - 14 July 1997. He is on leave from the Faculty of Information Technology, Vietnam National University Hanoi, Vietnam. He obtained Ph.D from there in mathematics and has been a Lecturer there since 1980. His current research interests are economic mathematics and information technology in economics.

Chris George is a Senior Research Fellow at UNU/IIST, 1 September 1994 — 31 August 1999. He is one of the main contributors to RAISE, particularly the RAISE method, and that remains his main research interest. Before coming to UNU/IIST he worked for companies in the UK and Denmark.

Contents

1 Introduction **1**

2 Financial Information System - an Overview **1**

2.1 Ministry of Finance . . . 1
2.2 Relationship of MoF and Social Entities . . . 3
2.3 FIS' component systems . . . 5
2.3.1 Tax System . . . 6
2.3.2 Treasury System . . . 6
2.3.3 Budget System . . . 7
2.3.4 External System . . . 8
2.3.5 Combining and Distributing the separate systems . . . 8

3 Tax System **10**

3.1 Narrative . . . 10
3.1.1 Taxation functions . . . 10
3.1.2 The Tax System Structure . . . 11
3.1.3 A Taxpayer . . . 12
3.1.4 Specification Structure Design . . . 14
3.2 Models . . . 15
3.2.1 Basic types . . . 15
3.2.2 A Registration . . . 16
3.2.3 Regime . . . 19
3.2.4 Roll . . . 20
3.2.5 Base . . . 20
3.2.6 Bases . . . 22
3.2.7 Tax_Account . . . 23
3.2.8 Taxpayer . . . 24
3.2.9 Group . . . 27
3.2.10 Tax Offices . . . 30
3.2.11 The Tax Policy . . . 33

4 Budget System **35**

4.1 Narrative . . . 35
4.1.1 Structure of the Budget System . . . 35
4.1.2 The budget objects . . . 36
4.1.3 Functions of the Budget System . . . 37
4.2 Formal model . . . 41
4.2.1 Specification of the basic concepts . . . 41
4.2.2 The Budget Plan . . . 43
4.2.3 Budget Objects . . . 46
4.2.4 Budget Departments . . . 49
4.2.5 Hierarchical development . . . 53

5 Treasury System **55**

5.1 Narrative 55
5.1.1 Role and Structure of the Treasury System 55
5.1.2 Relationship between the Treasury system and other parts of the MOF . 57
5.1.3 Management of Treasury information 58
5.2 Formal model 59
5.2.1 Budgets 59
5.2.2 Accounts 63
5.2.3 Treasury 65
5.3 Further work 68

6 External System 69
6.1 Survey for External Financial Information system 69
6.1.1 Social entities of EFIS 69
6.1.2 Component Systems 70
6.2 External aid reception 73
6.2.1 Narrative 74
6.2.2 Formal Model 75
6.3 External debt 79
6.3.1 Narrative 79
6.3.2 Model 81

7 Combining and distributing separate subsystems 87
7.1 Narrative 87
7.2 Formal Model 90
7.2.1 The message system 90
7.2.2 Additional modules 91
7.2.3 Adapting existing components 95
7.2.4 The complete system 101
7.3 Conclusion 102

8 Conclusion 104
8.1 Implementation 104
8.2 Future work 104

A Other schemes 105

1 Introduction

This document is a production of the MoFIT (Ministry of Finance Information Technology) project, which started with a visit by Chris George to the Ministry of Finance (MoF) in Hanoi in September 1995, where the basic requirements of a substantial project supported by the World Bank were discussed. This was continued in January 1996 by Chris George presenting a two-week course on RAISE [1, 2] to software engineers from the MoF and Institute of Information Technology (IoIT). The project began in earnest in April 1996, when five Fellows, four from the MoF and one from IoIT, came to UNU/IIST for six months. In the first step (see [3]), they have specified the tax system and considered the problems of Security and Taxation Policy.

In this document we present a domain analysis for a national financial information system as input to the major project in Vietnam to produce such a system. The formal components are written in the RAISE specification language [1], using the RAISE development method [2].

Section 2 presents an overview for Financial Information System and decomposes it into subsystems.

Section 3 presents the Tax System

Section 4 presents the Treasury System

Section 5 presents the Budget System

Section 6 presents the External System

Section 7 presents the combination and distribution of the separate systems

Section 8 draws some conclusions

2 Financial Information System - an Overview

The Financial Information System (FIS) involves financial informations relating to activities of collecting, storing, distributing and processing the state budget. The FIS is familiar to everybody but is very difficult to describe. This is a complex system containing a huge information volume.

In order to describe FIS, first of all we consider the Ministry of Finance and its relationship to other social entities.

2.1 Ministry of Finance

In Vietnam the ministry of Finance (MoF) is the Government agency performing the function of state management in the financial field nation-wide. The following tasks are identified:

1. Guiding ministries, government offices and provincial peoples committees subsidiary to the central management to build the yearly state budget plan. Includes: state budget planning, implementing and making statement reports.
2. Coordinating with the Ministry of Planning and Investment in building long-term, medium-term and short-term financial plans, economic-society development plans, basis construction plans, etc. And consulting to the ministries, government offices and provincial peoples committees on the direction of development, finance investment policy, wages, prices and other social-economic issues involving the financial and state budget.
3. Drafting laws and other legal documents on tax collection. Managing and guiding tax, fee and other state budget collections.
4. Managing the State budget, reserve and foreign currency funds. Distributing state budget expenditure amounts for each state budget holder following the plan.
5. Performing financial support measures in order to implement social-cultural purposes as authorised by the Government.
6. Managing capital, assets and resources of the State.
7. Managing the State's capital and assets in enterprises. Being the State's representative for capital and assets in enterprises authorised by the Government.
8. Making decisions to revoke money which is used for the wrong purposes, does not follow the plan or violates the financial management regulations of the State. And reporting all such decisions to the Prime Minister.
9. Managing the loan and debt payments (including internal and external debt); managing aid. Being involved in appraisal on the financial side of all projects which use foreign-borrowed capital following assignments of the Government.
10. Managing insurance services; bond, bill issue; social funds; lottery; auditing and accounting services and other financial services; managing the capital market.
11. Examining and inspecting the administrative organisations (non-business organisations), State enterprises and other organisations which have State financial relationships.
12. Being the representative of the Government in international financial organisations.

Temporarily, the organisation structure of MoF is divided into three groups. The fourteen *central departments* carry out the state management function; the five *professional line organisations* carry out specialised financial activities such as managing capital for investment and development, managing the State's capital and assets in enterprises, managing revenue collection, managing state budget expenditure and financial inspection from the central to the local. There are three levels in such an organisation: the central level, the provincial level, and the district level. These correspond to the centre, the province and the district in administration management of the State; the *other institutes* are involved in research and training.

The main duty of MoF is *State budget Management*:state budget collection, expenditure and storage functions; Budget planning for short and long time: Making necessary reports of financial situations to the Government.

Collection sources are taxes, fees, charges; State economic activities; contributions from organisations and individuals; aid; State's borrowing.

Expenditures include all amounts for defence, education, health; social and economic development; maintaining state activities; debt payments.

Essentially, the Treasury system manages state budget expenditure and the Taxation system manages revenue collection. The State Budget department serves to synthesise information from the Treasury and Taxation systems.

Budget planning is an annual event. As part of this a national revenue plan for tax collection is established, and allocated among the provinces and districts.

Taxpayers are allocated to either province or district tax offices. They report their income, profits, turnover etc. to that office, receive demands for taxes and pay the Treasury department at the same (province or district) level. Treasury departments report on amounts collected to tax departments. Tax departments make daily and annual reports up the tax system hierarchy. They also forecast and report on expected revenues.

Based on the approved expenditure plan which the State budget produces, budget holders make their detailed expenditure plans and get them approved (perhaps modified). Proposed actual expenditures are then checked against the detailed expenditure plans and, if approved, made by the Treasury department at the appropriate level. Treasury departments make daily and annual reports up the Treasury system hierarchy.

A national summary reports on budget performance are made at the end of the fixed periods.

In order to describe FIS more in detail, we sketch out the relationship of MoF and other Social Entities (FIS Social Entities)

2.2 Relationship of MoF and Social Entities

The FIS Social Entities involve the entities which participate in activities of MoF. They include:

- Government Organisations (Government ORGs)
- Ministries
- External Organisations (External ORGs)[1]

[1] Include external Lenders and external Aiders via Governments

- Taxpayers
- Collecting fee ORGs[2]
- State Bank
- Receivers
- Lenders
- Other revenue resources [3]
- Investment Holders [4]

In this system, the MoF performs mainly the following duties:

1. Budget collection from: Taxpayers; Collecting fee ORGs; Other revenue resources, Lenders
2. Issues state bonds and stocks to Lenders
3. Monitors and manages the external aid reception, borrows and settles external debts[5]
4. Restores and manages budget funds
5. Based on Government's commands, issues money or receives money from the state bank in order to balance the state budget.
6. Estimates the plans for state budget collection and makes expenditure plans for the budget objects
7. Performs the budget distribution
8. Synthesises and sends the necessary reports to Government ORGs

In order to illustrate the FIS, we consider figure 1.

[2]The Government offices authorised to collect common fees such as:Transport fees, insurance fees ...
[3]For example from exploiting mines, oil...
[4]The domestic projects or cooperative projects with foreign organisations via Government
[5]in this section we use the name 'currency' for both goods and foreign money

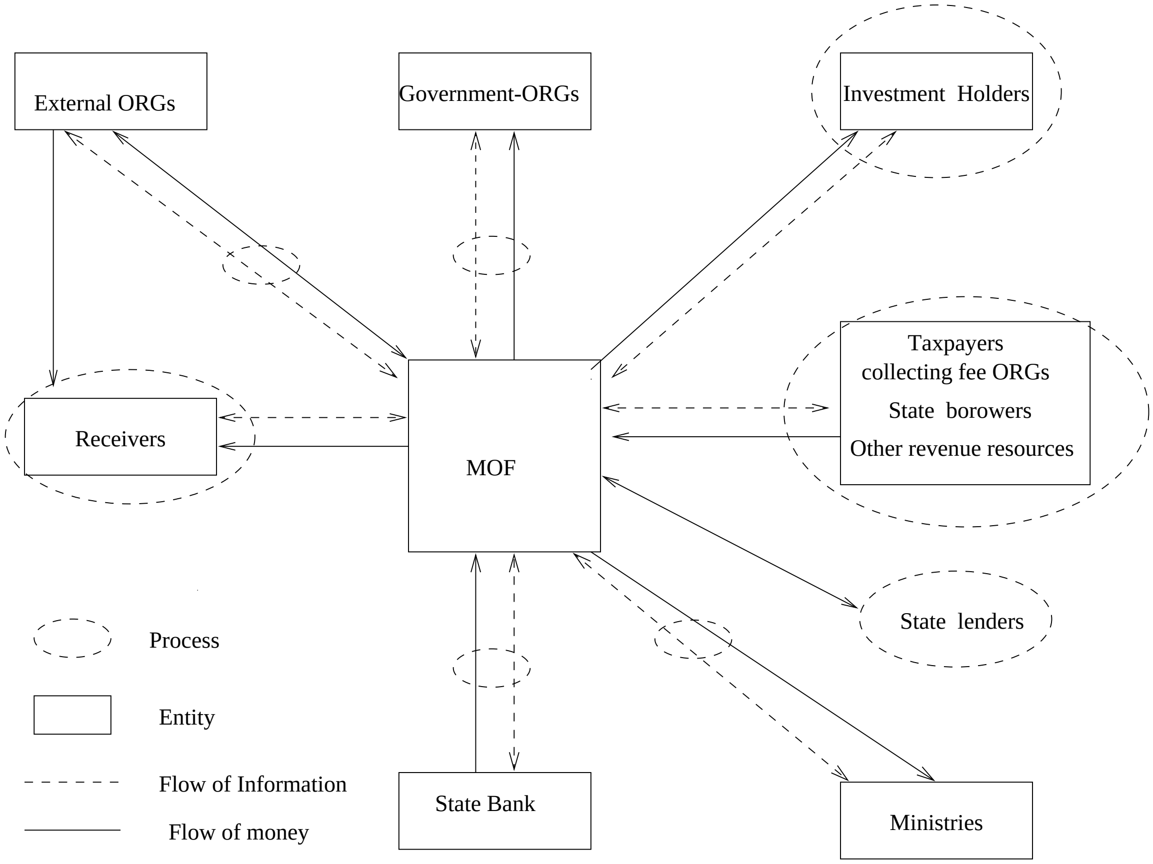

Figure 1: Data flow Diagram of MoF and Social Entities

2.3 FIS' component systems

Based on the analyse above and organisation structure of the MoF, we decompose the FIS into the following subsystems

- Tax System
- Treasury System
- Budget System
- External System

2.3.1 Tax System

The tax system is hierarchical. The general tax department is the highest level, down to the net of provincial tax departments and district tax departments (the lowest level). It has the main duties of identifying tax amounts for each taxpayer and administering tax collection activities so that tax leakages can be avoided; sending the synthesised report to related organisations.

The tax payers can settle their tax amount directly to treasury system or via tax office or state bank. We consider the figure 2. This system will be described in detail in section 3.

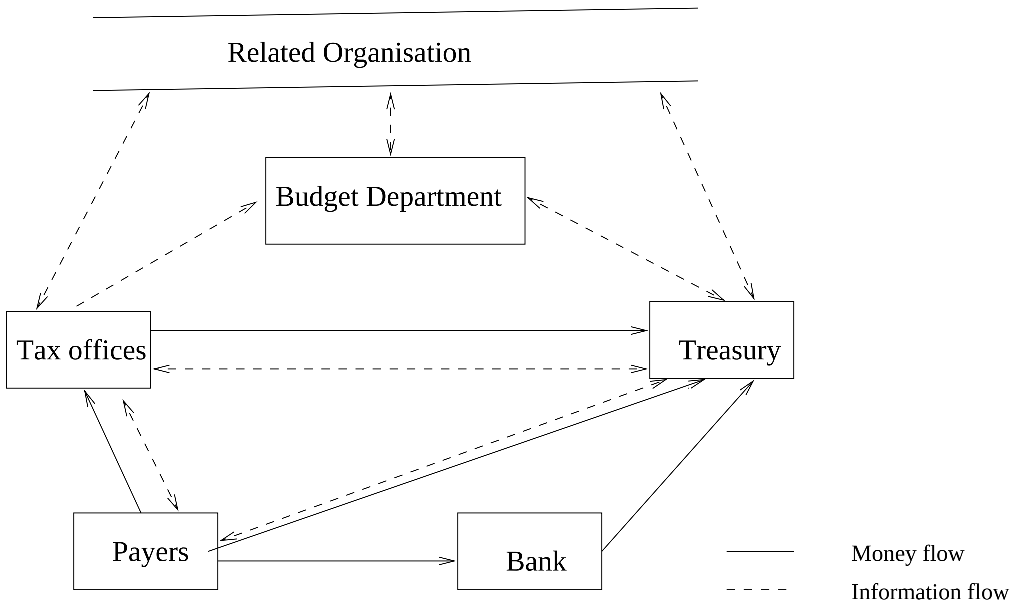

Figure 2: Data flow Diagram of Tax Collection

2.3.2 Treasury System

The treasury system is an important subsystem of the Ministry of Finance. It manages the State budget funds, mobilises capital for the State budget by issuing bills and bonds, performs payment for the budget holders following the plan which was approved, exchanges information with the tax system and other entities in order to realise the state activities of budget collection and expenditure, makes necessary periodic reports to budget department and other related offices. Likes the tax system, the treasury system is hierarchical, downs from the state central treasury (highest level) to to the net of provincial treasuries (middle level) and district treasuries (the lowest level).

The information flow can be illustrated by figure 3 .

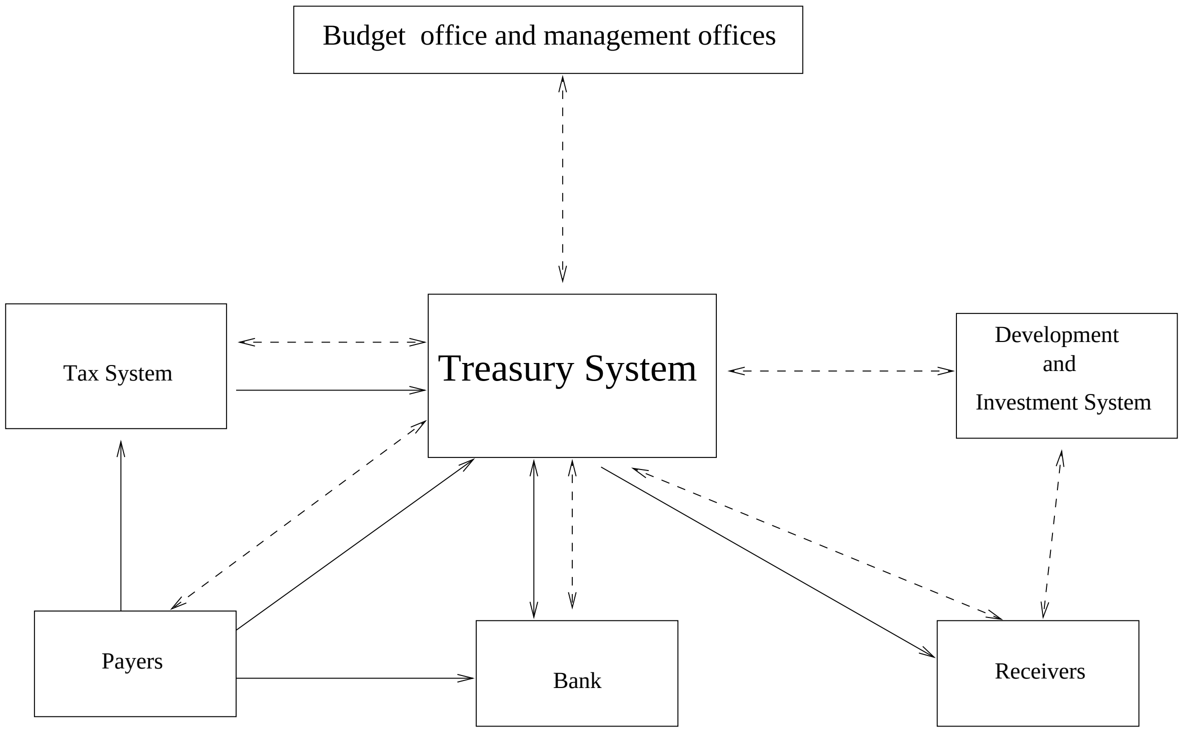

Figure 3: Data flow Diagram of Treasury System

This system will be described in detail in section 5.

2.3.3 Budget System

The central task of the Vietnam financial system is budget processing, which is performed by the budget system. Similar to tax and treasury systems, the budget system is composed of a hierarchical budget department system corresponding to the administration offices.

The budget department manages and processes information of financial collection and expenditure. Based on executive information, the budget department cooperates with other organisations to estimate collection plans and build spending plans, issues spending commands, synthesises and sends reports of the budget situation to other organisations. We consider figure 4. In order to proply estimate and optimally use the state budget we have to design a group of decision support making systems and expert systems for forecasting and optimising expenditure. In this document we only describe how it could be. The description in detail of the budget system will be presented in section 4.

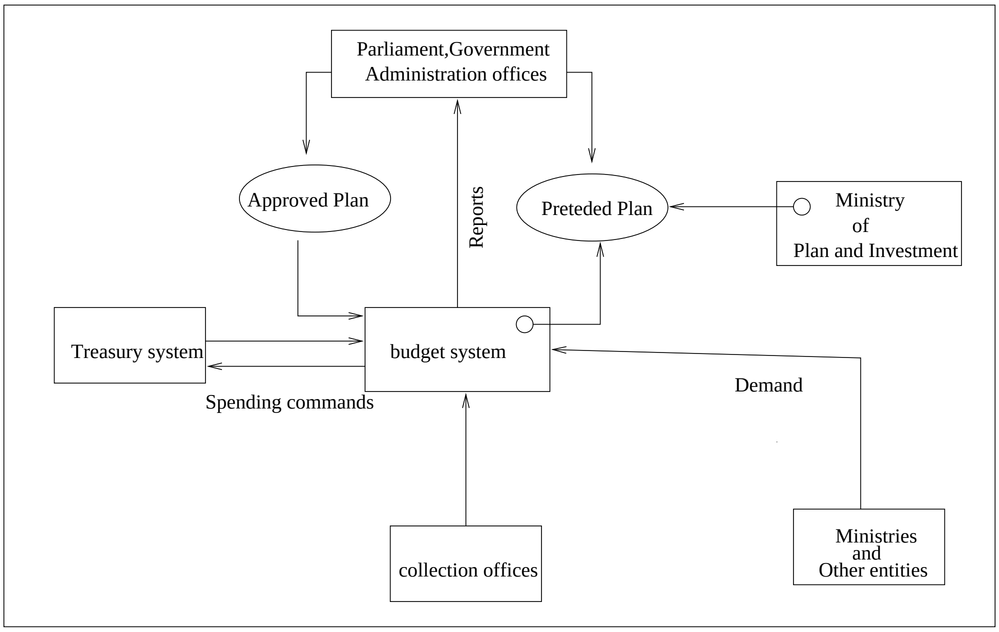

Figure 4: Information Diagram of Budget System

2.3.4 External System

The external information system includes the information of foreign currency collection and expenditure; aid reception and external debts. Vietnam is a developing country, the process of industrialising demands a huge foreign capital. Therefore external financial information is managed especially. This system will be described in detail in section 6 .

2.3.5 Combining and Distributing the separate systems

As has been described above, the financial system is a complex system which includes subsystems . Each of these component system has its own functions and is managed and implemented relatively independently. However, in order to realize common task of financial management they need to communicate with each other. For example, the treasury system needs expenditure plans of its customers from the budget system and, conversely, the budget system needs "feedback" information from the treasury system to make any further plan for these customers, or the office of the ministry may want to collect reports from all district financial departments. Omitting the connection in every subsystem, we can sketch this system by figure 5.

This system requires us to design a good user interface system and study FIS more in depth with

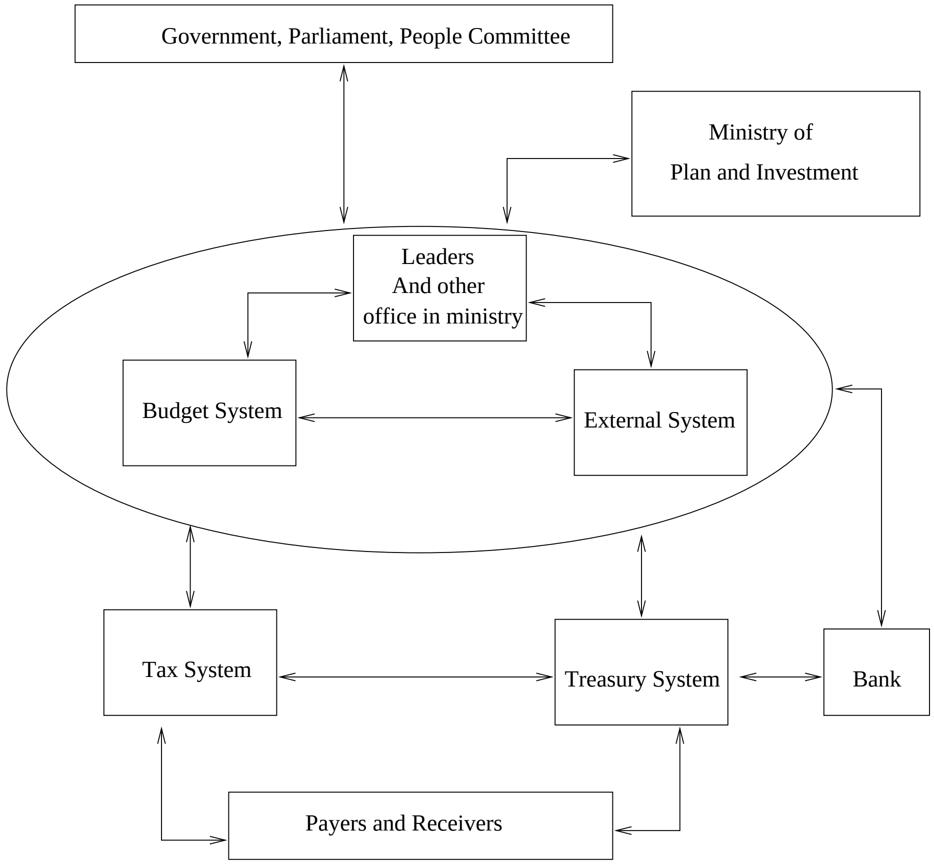

Figure 5: FIS Diagram

reference to the capacities of hardware. Temporarily, in section 7 we describe how the systems could be combined and distributed.

3 Tax System

This section is a summary of [4] based on [3]. The specification is only applicative.

3.1 Narrative

The tax system is considered as a system which manages all taxation activities. It based on the tax law and tax policy to manage tax professional activities. The input information of the system is tax transaction (normally between taxpayers and tax offices). The output information is tax reports; we consider the figure 6. The tax system, in analysis, can be seen from two points

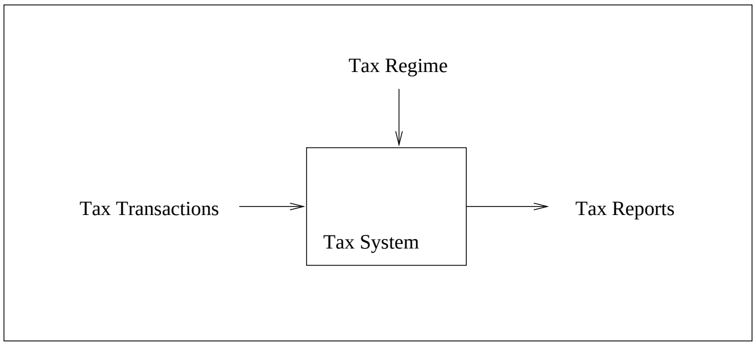

Figure 6: Tax System

of view: structure and functions. The tax system is organised into different levels and it has different functions. At each level, it may have some functions and a tax function can be included in several levels. We will discuss these points of view in more detailed later.

3.1.1 Taxation functions

The tax system consists of four main functions: tax management, planning, accounting and statistics. Tax management and planning functions carry on management and planning processes and record them in the system. Thus, in the view point of the tax system, these functions are considered as "generators". On the contrary, tax accounting and statistic functions are seen as "observers", they only get data from the system and summarise it to get reports. We consider figure 7

We will discuss a bit more elaborately about generator functions. The tax planning function is a process in which leaders impose tax collection planning for tax units. In fact, it doesn't effect the tax activities of a particular taxpayer. All the activities of the taxpayer are managed by

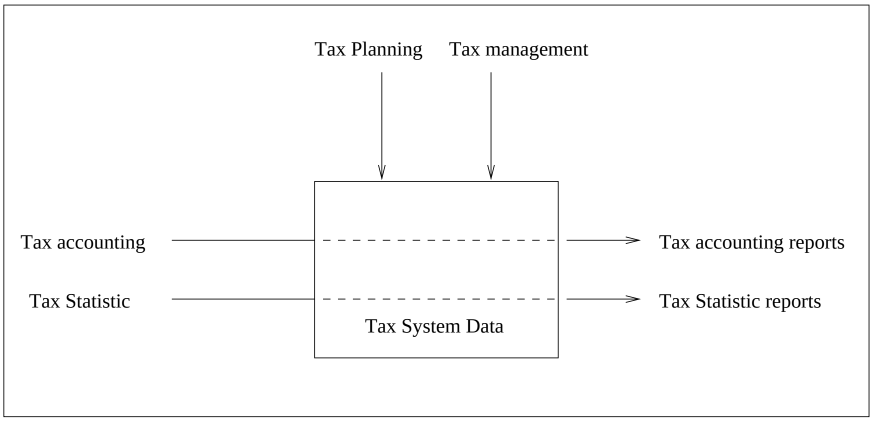

Figure 7: Tax Functions

management processes; see figure 8.

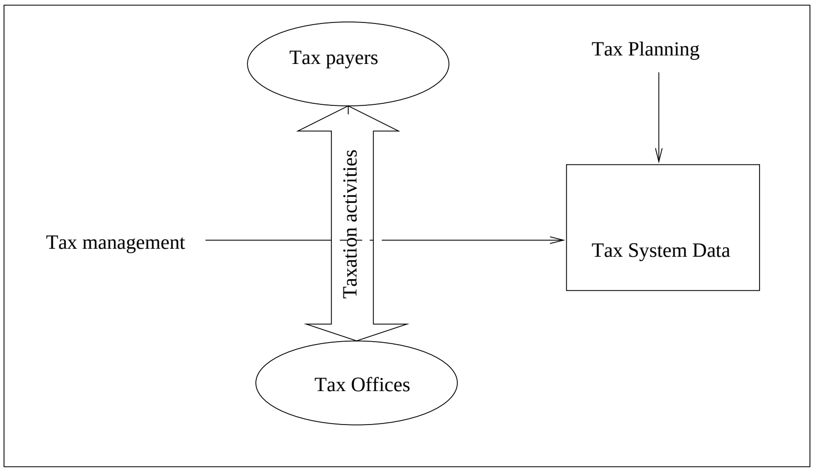

Figure 8: Tax management and planning

3.1.2 The Tax System Structure

The Tax System is organised into a hierarchy with three levels. The general tax department (GTD) is the highest level which manages province tax departments (PTD). A PTD manages several district tax departments (DTD) and a number of taxpayers. Each DTD only manages taxpayers. The taxpayers under different tax offices are separated, it means there is no taxpayer which is managed by two tax offices; see figure 9.

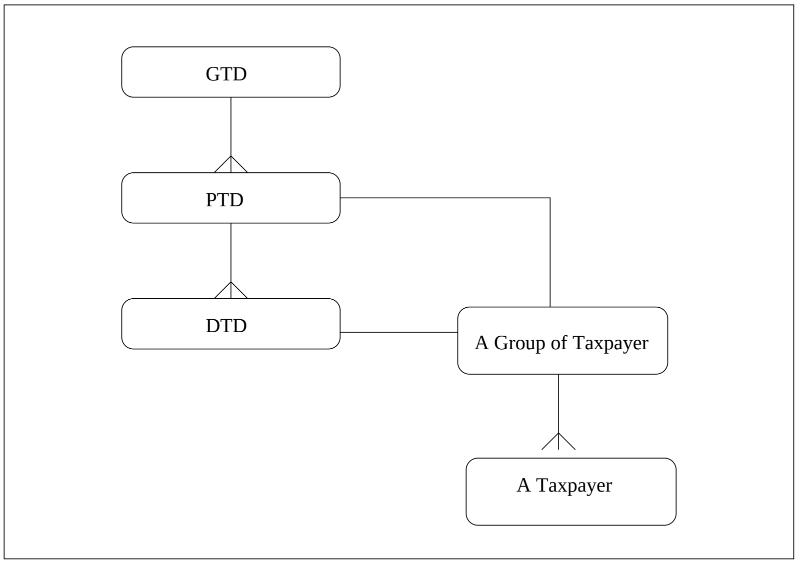

Figure 9: The Tax System Structure

3.1.3 A Taxpayer

The tax activities of a taxpayer can be divided into three kinds:

registration the taxpayer registers when he starts running his business.

declaration the taxpayer declares his tax base (profit,turnover,etc.) for a period.

transaction the taxpayer settles his tax amounts.

This can be illustrated by figure 10

Let's see how the above processes are implemented in reality. The register information is recorded as the roll of the taxpayer. The roll consists of needed information about taxpayer's business. Based on the tax law, it defines what kinds of taxation in liable for the taxpayer. A declaration of a period is called a base for this period. It is recorded in the bases of the taxpayer. The tax office uses the base to compute the tax amount for the taxpayer. In order to settle the tax amount, the taxpayer needs to do some tax transactions which are tax payments or exemptions. This process can be sketched by figure 11. Let's look a bit more carefully at the information of a taxpayer. The tax roll is considered as static information in the tax processes of the taxpayer. It means, the roll is not affected by declaration or payment activities, it reflects information of the taxpayer's specialisation, what kinds of business the taxpayer is involved in and so on. In fact the tax roll determines the pairs of categories and specialisations, which are included in the tax declaration of what the taxpayer needs to pay tax on.

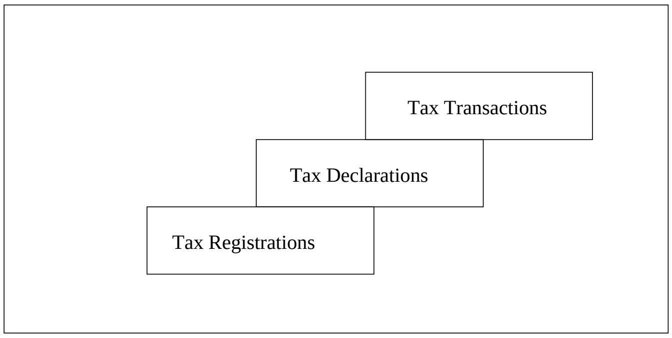

Figure 10: Tax payer

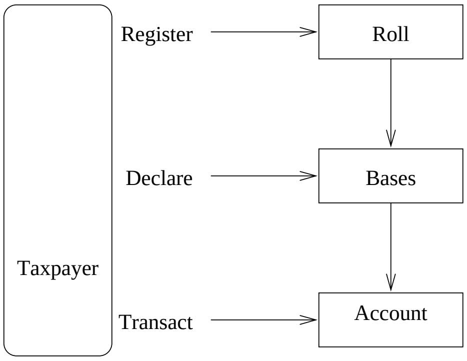

Figure 11: Tax management of a Taxpayer

The tax bases record all the declarations of the taxpayer. It doesn't have any meaning in the tax management directly. It is needed for supervising tax declarations of the taxpayer and for the tax accounting, statistical processes.

After having declarations, the tax office calculates and sends to the taxpayer tax demands. The taxpayer then does some payment (or exemption) to settle them. These activities are recorded in the tax account by transaction records. The are some rule of managing tax transactions. They are showed below:

1. When the taxpayer settles for a period, he needs to settle the debt of the previous periods first.
2. When the tax office calculates demand, it needs first subtracts any amount the taxpayer has overpaid.
3. If a taxpayer has a tax debt, each day he has to bear an interest defined in the tax regime.

4. Tax transactions in different categories are independent.

3.1.4 Specification Structure Design

The specification is written with the principle developing from abstract to concrete, applicative to concurrent, from basic components to the whole system. Firstly, we will write the applicative specification trying to match all the requirements. Then we will develop it to the imperative and concurrent versions. With the idea that developing the system from its components, we try to specify the agents of the tax system separately in order to get a clear sight of the system and make it easy to develop.

In particular:

1. The Regime module defines the tax law. It will be use for the whole system.
2. T object defines the basic types of the system.
3. RT object defines the register information of a taxpayer when he starts running his business.

The Taxpayer scheme defines a taxpayer. It has mainly three components which are specified in three scheme:

1. The ROLL scheme defines the Roll type which includes all the register information (and its changes) of a taxpayer.
2. The BASES scheme manages the declarations of a taxpayer.
3. The ACCOUNT scheme performs the tax transactions between tax offices and a taxpayer: tax demands and tax settlements.

It also declares a Base object to calculate tax demands for the taxpayers from tax declarations.

Based on the Taxpayer object, the Group scheme, which determines tax activities of a group of taxpayers, is defined. The group object will be used as the basis component to specify the tax structure with three levels: DTD, PTD and GTD.

The tax offices have following characteristics:

1. Besides the management activities, each tax office needs functions to summarise its information, to make reports.

2. A tax office, a separate component, also has its plan.

To follow the principle developing the specification separately, we will try to specify the report and planning systems independently. So there are modules specifying them. The report policy is modelled as the RP object to be used for other modules. The planning part of the different tax levels are specified in different schemes. The above modules are used to specify the tax offices.

3.2 Models

3.2.1 Basic types

The object T defines basic types of the tax system. It also determines the linear property of date types. Dates are totally ordered.

```
object
  T :
    class
      type
        Date,
        Specialization,
        Category,
        Period = Nat,
        Amount = Nat,
        Base = (T.Specialization × T.Category) →m T.Amount,
        Tid,
        DTDid,
        PTDid,
        GTDid

      value
        start_date : Date • ∀ d : Date • start_date ≤ d,

        ≤ : Date × Date → Bool,

        next : Date → Date
        next(d) as d′
          post d ≤ d′ ∧ d ≠ d′ ∧ (∀ d0 : Date • d0 ≤ d ∨ d′ ≤ d0),

        previous : Date →~ Date
        previous(d) as d′ post d = next(d′) pre d ≠ start_date,
```

```
      start_period : Period = 0,

      next : Period → Period
      next(p) ≡ p + 1

    axiom
      /∗ dates are totally ordered ∗/
      [date_reflexive] ∀ d : Date • d ≤ d,

      [date_total] ∀ d1, d2 : Date • d1 ≤ d2 ∨ d2 ≤ d1,

      [date_antisymmetric]
        ∀ d1, d2 : Date • d1 ≤ d2 ∧ d2 ≤ d1 ⇒ d1 = d2,

      [date_transitive]
        ∀ d1, d2, d3 : Date • d1 ≤ d2 ∧ d2 ≤ d3 ⇒ d1 ≤ d3
  end
```

3.2.2 A Registration

The object RT defines the register of a taxpayer when he starts his business. This information is used to define which pair of tax categories and specialisations the taxpayer needs to pay for.

```
object
  RT :
    class
      type
        Register,
        /∗ components of a registration ∗/
        Management_level,
        Economic_component,
        Chapter,
        Type_article_class,
        Locality,
        Collector_team,
        /∗ a component of a registration ∗/
        Reg_item =
          Management_level | Economic_component | Chapter |
            Type_article_class | Locality | Collector_team,
        /∗  kinds of components of a registration ∗/
        Reg_item_kind ==
          management_level |
```

```
        economic_component |
        chapter |
        type_article_class |
        locality |
        collector_team

    value
      /* generator */
      change_item : Reg_item_kind × Reg_item × Register ~→ Register
      change_item(ik, i, reg) post true pre is_fix(ik, i),

      /* observer */
      get_item : Reg_item_kind × Register → Reg_item
      get_item(ik, reg) as i post is_fix(ik, i),

      /* derived */
      is_fix : Reg_item_kind × Reg_item → Bool
      is_fix(ik, i) ≡
        case ik of
          management_level → ∃ i′ : Management_level • i′ = i,
          economic_component → ∃ i′ : Economic_component • i′ = i,
          chapter → ∃ i′ : Chapter • i′ = i,
          type_article_class → ∃ i′ : Type_article_class • i′ = i,
          locality → ∃ i′ : Locality • i′ = i,
          collector_team → ∃ i′ : Collector_team • i′ = i
        end

    axiom
      [get_change]
        ∀ ik1, ik2 : Reg_item_kind, i : Reg_item, reg : Register •
          get_item(ik2, change_item(ik1, i, reg)) ≡
            if ik1 = ik2 then i else get_item(ik2, reg) end
            pre is_fix(ik1, i)
  end
```

The object RP and the scheme SUM_DATA; SUM is used for other modules

```
object
  RP :
    class
      type
        Report,
        Report_form,
        Line,
```

```
    Report_level == taxpayer | group | DTD | PTD | GTD

value
    /* report generator */
    empty : Report_form → Report,

    summary : Report × Report ~→ Report
    summary(rp1, rp2) post true pre form(rp1) = form(rp2),

    /* report observers */
    form : Report → Report_form,

    amount : Line × Report ~→ T.Amount
    amount(l, rp) post true pre is_in(l, form(rp)),

    /* report_form observers */
    is_in : Line × Report_form → Bool,

    level : Report_form → Report_level,

    /* report_form derived */
    lower_level : Report_form ~→ Report_form
    lower_level(rf) as rf′
      post
        level(rf′) =
        case level(rf) of
          GTD → PTD, PTD → DTD, DTD → group, group → taxpayer
        end
      pre level(rf) ≠ taxpayer

axiom
    [ form_empty ] ∀ rf : Report_form • form(empty(rf)) ≡ rf,

    [ amount_empty ]
      ∀ rf : Report_form, l : Line •
        is_in(l, rf) ⇒ amount(l, empty(rf)) = 0,

    [ form_summary ]
      ∀ rp1, rp2 : Report •
        form(summary(rp1, rp2)) ≡ form(rp1) pre form(rp1) = form(rp2),

    [ amount_summary ]
      ∀ l : Line, rp1, rp2 : Report •
        amount(l, summary(rp1, rp2)) ≡ amount(l, rp1) + amount(l, rp2)
          pre form(rp1) = form(rp2) ∧ is_in(l, form(rp1))
```

```
    end

scheme
  SUM_DATA =
    class type Domain,  Range   value sum : Range × Range → Range
  end

scheme
  SUM(SD : SUM_DATA) =
    class
      type Map = SD.Domain ⇸ SD.Range

      value
        sum : SD.Range × Map → SD.Range
        sum(null, m) ≡
          if m = [ ] then
            null
          else
            let d : SD.Domain • d ∈ dom m in
              SD.sum(m(d), sum(null, m \ {d}))
            end
          end
    end
```

3.2.3 Regime

The scheme AP_REGIME defines the tax regime. In particular, for a particular regime, it defines the tax amount for a tax calculation amount in a given category and specialisation on a certain date, the tax penalty rate for any date, and checks if a register includes a given pair of category and specialisation

```
scheme
  AP_REGIME =
    class
      type Regime,  Rate = {| r : Real • r ≥ 0.0 |}

      value
        is_included :
          T.Specialization × T.Category × RT.Register × Regime → Bool,
        tax_amount :
```

```
        T.Date × T.Specialization × T.Category × T.Amount × Regime →
          T.Amount,
      penalty_rate : T.Date × Regime → Rate
    end
```

3.2.4 Roll

The AP_ROLL scheme define the Roll type to store all the registration of the taxpayer. It means the taxpayer may change his registration information. We can define the registration from the roll for any date.

```
scheme
  AP_ROLL =
    class
      type
        Reg_list = (T.Date × RT.Register)*,
        Roll = {| rl : Reg_list • len rl > 0 |}

      value
        mk_roll : T.Date × RT.Register → Roll
        mk_roll(d, rg) ≡ ⟨(d, rg)⟩,

        change_roll : T.Date × RT.Register × Roll → Roll
        change_roll(d, rg, rl) ≡ ⟨(d, rg)⟩ ^ rl,

        get_reg : T.Date × Roll ~→ RT.Register
        get_reg(d, rl) ≡
          let (d′, rg) = hd (rl) in
            if T.(≤)(d′, d) then rg else get_reg(d, tl rl) end
          end
          pre T.(≤)(start_date(rl), d),

        start_date : Roll → T.Date
        start_date(rl) ≡ let (d, rg) = rl(len rl) in d end
    end
```

3.2.5 Base

Defines functions to calculate tax amounts from a tax base. Because of the requirement of management tax activities in categories, the tax demands are computed for each category separately.

```
scheme
 AP_BASE(RG : AP_REGIME) =
  hide SM in
   class
    value
     is_legal : RT.Register × T.Base × RG.Regime → Bool
     is_legal(rt, b, rg) ≡
      (
       ∀ s : T.Specialization, c : T.Category •
        (s, c) ∈ dom b ⇒ RG.is_included(s, c, rt, rg)
      ),

     demand_amounts :
      T.Date × T.Base × RG.Regime → (T.Category →m T.Amount)
     demand_amounts(d, b, rg) ≡
      [ c ↦ a |
       c : T.Category, a : T.Amount •
        a = demand_amount(d, c, b, rg) ∧ a ≠ 0
      ],

     demand_amount :
      T.Date × T.Category × T.Base × RG.Regime → T.Amount
     demand_amount(d, c, b, rg) ≡
      let
       sa =
        [ s ↦ a |
         s : T.Specialization, a : T.Amount •
          (s, c) ∈ dom b ∧
          a = RG.tax_amount(d, s, c, b(s, c), rg)
        ]
      in
       SM.sum(0, sa)
      end

    object
     SM : SUM(SD),
     SD :
      class
       type Domain = T.Specialization,  Range = Int

       value
        sum : Int × Int → Int
        sum(x, y) ≡ x + y
      end
   end
```

3.2.6 Bases

The Bases scheme defines the bases type to store all declarations of the taxpayer.

```
scheme
  AP_BASES =
    class
      type
        Bases = Declaration*,
        Declaration ::
          date : T.Date
          period : T.Period
          specialization : T.Specialization
          category : T.Category
          amount : T.Amount

      value
        declare : T.Date × T.Period × T.Base × Bases → Bases
        declare(d, p, b, bs) ≡
          if b = [ ] then
            bs
          else
            let
              s : T.Specialization,
              c : T.Category,
              a : T.Amount • (s, c) ∈ dom b ∧ a = b(s, c)
            in
              ⟨mk_Declaration(d, p, s, c, a)⟩ ^
              declare(d, p, b \ {(s, c)}, bs)
            end
          end,

        calculation_amount :
          T.Date × T.Period × T.Specialization × T.Category × Bases →
            T.Amount
        calculation_amount(d, p, s, c, bs) ≡
          case bs of
            ⟨⟩ → 0,
            ⟨dc⟩ ^ bs′ →
              if
                d = date(dc) ∧
                p = period(dc) ∧ s = specialization(dc) ∧ c = category(dc)
              then
                amount(dc)
```

```
            else
              0
            end +
            calculation_amount(d, p, s, c, bs′)
        end
    end
```

3.2.7 Tax_Account

The AP_ACCOUNT scheme defines the Account type to do tax transactions (see appendix A). There's a note about using numbers when calculating:

1. Every amount got from or sent to a customer is a Nat.
2. The debt amount is specified as real numbers to get precise calculations.

```
scheme
  AP_ACCOUNTS(RG : AP_REGIME) =
    hide AC, get_account in
      class
        object AC : AP_ACCOUNT(RG)

        type
          Accounts = T.Category →m AC.Account,
          Bill = AC.Bill,
          Receipt = AC.Receipt

        value
          demand :
            T.Date × T.Date × T.Period × T.Category × T.Amount ×
              Accounts × RG.Regime
            ~→
              Accounts × Bill
          demand(d1, d2, p, c, a, acs, r) ≡
            let
              ac = get_account(c, acs),
              (ac′, b) = AC.demand(d1, d2, p, a, ac, r)
            in
              (acs † [c ↦ ac′], b)
            end
            pre can_demand(d1, d2, c, acs),
```

```
    pay :
      T.Date × T.Period × T.Category × T.Amount × Accounts ×
        RG.Regime
      ∼→
        Accounts × Receipt
    pay(d, p, c, a, acs, r) ≡
      let
        ac = get_account(c, acs), (ac′, rc) = AC.pay(d, p, a, ac, r)
      in
        (acs † [c ↦ ac′], rc)
      end
      pre can_settle(d, c, acs),

    exempt :
      T.Date × T.Period × T.Category × T.Amount × Accounts ×
        RG.Regime
      ∼→
        Accounts × Receipt
    exempt(d, p, c, a, acs, r) ≡
      let
        ac = get_account(c, acs), (ac′, rc) = AC.exempt(d, p, a, ac, r)
      in
        (acs † [c ↦ ac′], rc)
      end
      pre can_settle(d, c, acs),

    can_demand : T.Date × T.Date × T.Category × Accounts → Bool
    can_demand(d1, d2, c, acs) ≡
      AC.can_demand(d1, d2, get_account(c, acs)),

    can_settle : T.Date × T.Category × Accounts → Bool
    can_settle(d, c, acs) ≡ AC.can_settle(d, get_account(c, acs)),

    get_account : T.Category × Accounts → AC.Account
    get_account(c, acs) ≡ if c ∈ dom acs then acs(c) else ⟨⟩ end
end
```

3.2.8 Taxpayer

The scheme AP_TAXPAYER_MANAGEMENT defines a taxpayer and its management functions.

The AP_TAXPAYER scheme is a extension of taxpayer_management with extra functions to make reports for the taxpayer. The tax_management scheme includes four objects to do particular tasks of tax management. They will be talked about in the parts below.

```
scheme
  AP_TAXPAYER_MANAGEMENT(RG : AP_REGIME) =
    hide RL, B, BS, AC in
      class
        object
          RL : AP_ROLL,
          B : AP_BASE(RG),
          BS : AP_BASES,
          AC : AP_ACCOUNTS(RG)

        type
          Taxpayer ::
            roll : RL.Roll ↔ re_roll
            bases : BS.Bases ↔ re_bases
            account : AC.Accounts ↔ re_account,
          Bills = T.Category →m AC.Bill,
          Receipt = AC.Receipt

        value
          register : T.Date × RT.Register → Taxpayer
          register(d, rt) ≡ mk_Taxpayer(RL.mk_roll(d, rt), ⟨⟩, [ ]),

          change_roll : T.Date × RT.Register × Taxpayer → Taxpayer
          change_roll(d, rg, t) ≡
            re_roll(RL.change_roll(d, rg, roll(t)), t),

          declare :
            T.Date × T.Date × T.Period × T.Base × Taxpayer × RG.Regime
            ~→
              Taxpayer × Bills
          declare(d1, d2, p, b, t, rg) ≡
            let
              ca = B.demand_amounts(d1, b, rg),
              bs = BS.declare(d1, p, b, bases(t)),
              (ac, bills) = declare(d1, d2, p, ca, account(t), rg)
            in
              (re_bases(bs, re_account(ac, t)), bills)
            end
            pre can_declare(d1, d2, b, t),

          declare :
            T.Date × T.Date × T.Period × (T.Category →m T.Amount) ×
```

```
      AC.Accounts × RG.Regime
    ~→
      AC.Accounts × Bills
  declare(d1, d2, p, ca, ac, rg) ≡
    if ca = [ ] then
      (ac, [ ])
    else
      let
        c : T.Category • c ∈ dom ca,
        (ac1, bills) = declare(d1, d2, p, ca \ {c}, ac, rg),
        (ac2, bill) = AC.demand(d1, d2, p, c, ca(c), ac1, rg)
      in
        (ac2, bills ∪ [ c ↦ bill ])
      end
    end
    pre
      T.(≤)(d1, d2) ∧
      (
        ∀ c : T.Category •
          c ∈ dom ca ⇒ AC.can_demand(d1, d2, c, ac)
      ),

  pay :
    T.Date × T.Period × T.Category × T.Amount × Taxpayer ×
      RG.Regime
    ~→
      Taxpayer × Receipt
  pay(d, p, c, a, t, rg) ≡
    let (ac, rc) = AC.pay(d, p, c, a, account(t), rg) in
      (re_account(ac, t), rc)
    end
    pre can_settle(d, c, t),

  exempt :
    T.Date × T.Period × T.Category × T.Amount × Taxpayer ×
      RG.Regime
    ~→
      Taxpayer × Receipt
  exempt(d, p, c, a, t, rg) ≡
    let (ac, rc) = AC.exempt(d, p, c, a, account(t), rg) in
      (re_account(ac, t), rc)
    end
    pre can_settle(d, c, t),

  can_declare : T.Date × T.Date × T.Base × Taxpayer → Bool
```

```
    can_declare(d1, d2, c, t) ≡
      T.(≤)(d1, d2) ∧
      T.(≤)(RL.start_date(roll(t)), d1) ∧
      let ac = account(t) in
        ∀ c : T.Category •
          c ∈ dom ac ⇒ AC.can_demand(d1, d2, c, ac)
      end,

    can_settle : T.Date × T.Category × Taxpayer → Bool
    can_settle(d, c, t) ≡ AC.can_settle(d, c, account(t))
  end

scheme
  AP_TAXPAYER(RG : AP_REGIME) =
    extend AP_TAXPAYER_MANAGEMENT(RG) with
      class
        value
          mk_report : RP.Report_form × Taxpayer ~→ RP.Report
          mk_report(rf, t) post true pre RP.level(rf) = RP.taxpayer
      end
```

The schemes DATA and AP_DATABASE (see appendix A) are used for other modules

3.2.9 Group

The scheme AP_GROUP_MANAGEMENT defines a group of taxpayers and its management functions.

The AP_GROUP scheme is an extension of AP_TAXPAYER_MANAGEMENT with adding functions to make reports for groups (see figure 12).

```
scheme
  AP_GROUP_MANAGEMENT(RG : AP_REGIME) =
    hide D in
      class
        object
          TP : AP_TAXPAYER(RG),
          D : class type Key = T.Tid,  Data = TP.Taxpayer end,
          DB : AP_DATABASE(D)

        type Group = DB.Database
```

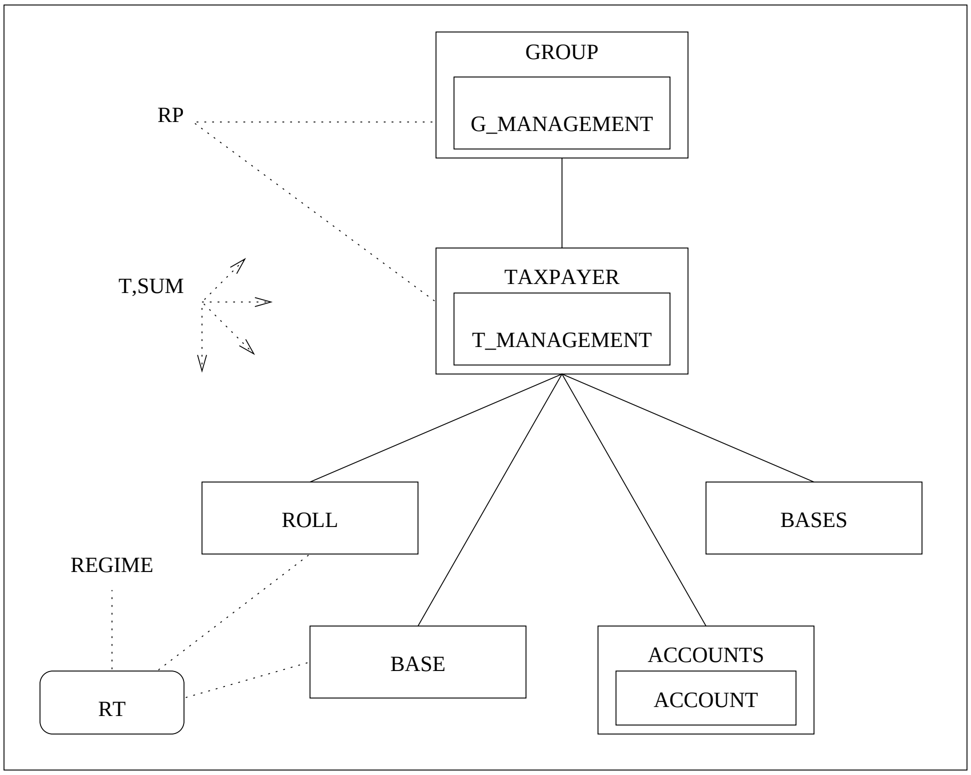

Figure 12: Group Specification Structure

value
  register : T.Tid × T.Date × RT.Register × Group $\stackrel{\sim}{\to}$ Group
  register(tid, d, rg, g) ≡
    DB.insert(tid, TP.register(d, rg), g)
    pre ∼ DB.defined(tid, g),

  change_roll : T.Tid × T.Date × RT.Register × Group $\stackrel{\sim}{\to}$ Group
  change_roll(tid, d, rg, g) ≡
    DB.insert(tid, TP.change_roll(d, rg, DB.lookup(tid, g)), g)
    pre DB.defined(tid, g),

  declare :
    T.Tid × T.Date × T.Date × T.Period × T.Base × Group ×
      RG.Regime
    $\stackrel{\sim}{\to}$
      Group × TP.Bills
  declare(tid, d1, d2, p, b, g, r) ≡
    let (t, bills) = TP.declare(d1, d2, p, b, DB.lookup(tid, g), r) **in**

```
        (DB.insert(tid, t, g), bills)
      end
      pre
        DB.defined(tid, g) ∧
        TP.can_declare(d1, d2, b, DB.lookup(tid, g)),

    pay :
      T.Tid × T.Date × T.Period × T.Category × T.Amount × Group ×
        RG.Regime
      ~→
        Group × TP.Receipt
    pay(tid, d, p, c, a, g, r) ≡
      let (t, rc) = TP.pay(d, p, c, a, DB.lookup(tid, g), r) in
        (DB.insert(tid, t, g), rc)
      end
      pre DB.defined(tid, g) ∧ TP.can_settle(d, c, DB.lookup(tid, g)),

    exempt :
      T.Tid × T.Date × T.Period × T.Category × T.Amount × Group ×
        RG.Regime
      ~→
        Group × TP.Receipt
    exempt(tid, d, p, c, a, g, r) ≡
      let (t, rc) = TP.exempt(d, p, c, a, DB.lookup(tid, g), r) in
        (DB.insert(tid, t, g), rc)
      end
      pre DB.defined(tid, g) ∧ TP.can_settle(d, c, DB.lookup(tid, g))
  end

scheme
  AP_GROUP(RG : AP_REGIME) =
    hide DB, TP, SD, SM in
      extend AP_GROUP_MANAGEMENT(RG) with
        class
          value
            mk_report : RP.Report_form × Group → RP.Report
            mk_report(rf, g) ≡
              let
                rps =
                  [ tid ↦ rp |
                    tid : T.Tid, rp : RP.Report •
                      DB.defined(tid, g) ∧
                      rp = TP.mk_report(RP.lower_level(rf), DB.lookup(tid, g))
                  ]
              in
```

```
          SM.sum(RP.empty(rf), rps)
        end
        pre RP.level(rf) = RP.group

    object
      SD :
        class
          type Domain = T.Tid,  Range = RP.Report

          value
            sum : Range × Range → Range
            sum(rp1, rp2) ≡ RP.summary(rp1, rp2)
        end,
      SM : SUM(SD)
  end
```

3.2.10 Tax Offices

The three levels structure of the tax system is specified in three modules: AP_GTD, AP_PTD and AP_DTD. Each office of higher levels has a tax planning object PL and a number of offices of the lower level which are specified as an object array. The office in the lowest level is only a group of taxpayers and it own planning. AP_PTD also has a number of taxpayers itself. We can sketch this system as figure 13.

```
scheme AP_DTD_PLAN = class type Plan end

scheme
  AP_DTD(RG : AP_REGIME) =
    hide GR, PL, add_plan in
      class
        object GR : AP_GROUP(RG),  PL : AP_DTD_PLAN

        type Office :: plan : PL.Plan  taxpayers : GR.Group

        value
          mk_report : RP.Report_form × Office → RP.Report
          mk_report(rf, o) ≡
            add_plan(GR.mk_report(RP.lower_level(rf), taxpayers(o)), plan(o))
            pre RP.level(rf) = RP.group,

          add_plan : RP.Report × PL.Plan → RP.Report
          add_plan(rp, pl) as rp′
```

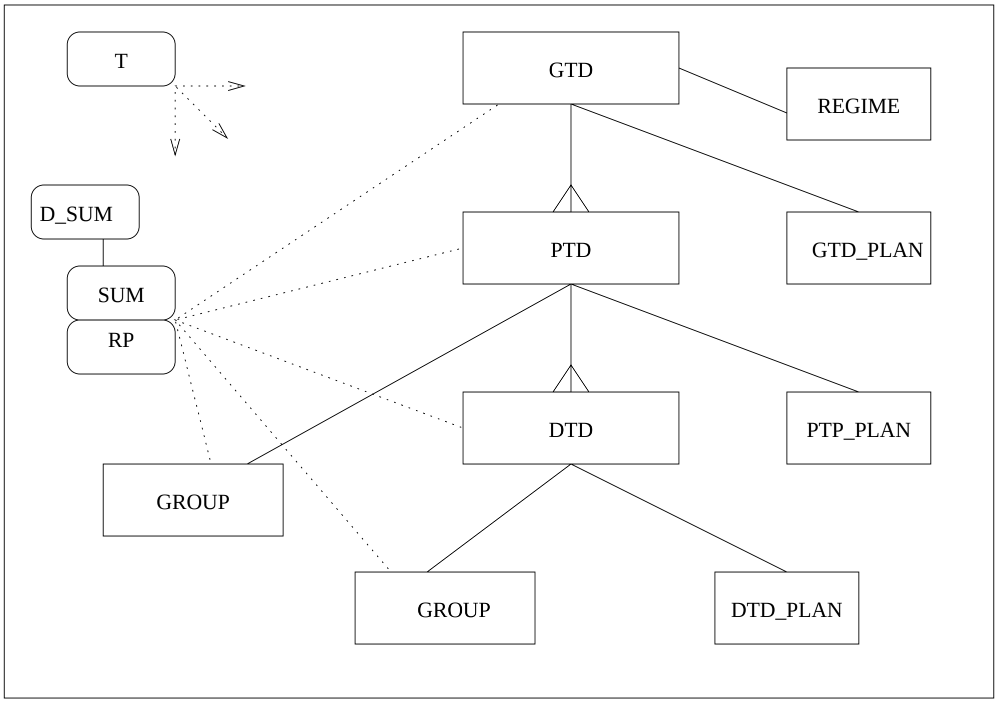

Figure 13: The Specification Structure

```
        post RP.level(RP.form(rp′)) = RP.DTD
        pre RP.level(RP.form(rp)) = RP.group
    end


scheme
  AP_PTD(RG : AP_REGIME) =
    hide SD, SM, GR, DTD, add_plan, change_form in
      class
        object
          SD :
            class
              type Domain = T.DTDid,  Range = RP.Report

              value
                sum : Range × Range → Range
                sum(rp1, rp2) ≡ RP.summary(rp1, rp2)
            end,
          SM : SUM(SD),
```

```
      GR : AP_GROUP(RG),
      PL : AP_PTD_PLAN,
      DTD : AP_DTD(RG)

    type
      Office ::
        plan : PL.Plan
        taxpayers : GR.Group
        offices : T.DTDid →m DTD.Office

    value
      mk_report : RP.Report_form × Office → RP.Report
      mk_report(rf, o) ≡
        let
          rf′ = RP.lower_level(rf),
          dtds_reports =
            [ id ↦ rp |
              id : T.DTDid, rp : RP.Report •
                id ∈ dom offices(o) ∧
                rp = DTD.mk_report(rf′, offices(o)(id))
            ],
          dtds_report = SM.sum(RP.empty(rf), dtds_reports),
          gr_report = GR.mk_report(RP.lower_level(rf′), taxpayers(o)),
          gr_report = change_form(gr_report)
        in
          add_plan(RP.summary(dtds_report, gr_report), plan(o))
        end
        pre RP.level(rf) = RP.PTD,

      add_plan : RP.Report × PL.Plan → RP.Report
      add_plan(rp, pl) as rp′
        post RP.level(RP.form(rp′)) = RP.PTD
        pre RP.level(RP.form(rp)) = RP.DTD,

      change_form : RP.Report → RP.Report
      change_form(rp) as rp′
        post RP.level(RP.form(rp′)) = RP.DTD
        pre RP.level(RP.form(rp)) = RP.group
  end
scheme AP_GTD_PLAN = class type Plan end
scheme
  AP_GTD =
    hide SD, SM, RG, PL, PTD, add_plan in
    class
      object
```

```
    SD :
      class
        type Domain = T.PTDid,  Range = RP.Report

        value
          sum : Range × Range → Range
          sum(rp1, rp2) ≡ RP.summary(rp1, rp2)
      end,
    SM : SUM(SD),
    RG : AP_REGIME,
    PL : AP_GTD_PLAN,
    PTD : AP_PTD(RG)

  type Office :: plan : PL.Plan  offices : T.PTDid →m PTD.Office

  value
    mk_report : RP.Report_form × Office → RP.Report
    mk_report(rf, o) ≡
      let
        rps =
          [ id ↦ rp |
            id : T.PTDid, rp : RP.Report •
              let ptds = offices(o) in
                id ∈ dom ptds ∧
                rp = PTD.mk_report(RP.lower_level(rf), ptds(id))
              end
          ]
      in
        add_plan(SM.sum(RP.empty(RP.lower_level(rf)), rps), plan(o))
      end
      pre RP.level(rf) = RP.GTD,

    add_plan : RP.Report × PL.Plan → RP.Report
    add_plan(rp, pl) as rp′
      post RP.level(RP.form(rp′)) = RP.GTD
      pre RP.level(RP.form(rp)) = RP.PTD
end
```

3.2.11 The Tax Policy

In the future, the tax management will progress to managing taxpayers more closely. When this has been done, with the supervision of the taxpayers' business more detailed, the tax offices can calculate the tax base for them. We define the Finance type in the scheme Finance and the

REGIME scheme is now inserted a new rule: for a specialisation, a category and a period, each "finance" is defined an amount.

The modules to be changed are: AP_ROLL, AP_TAXPAYER, AP_GROUP.

4 Budget System

The budget system performs the processing of collecting and supporting expenses for the state activities, which is the central task of the Vietnam financial system. Be low, we present a description for this system [6].

4.1 Narrative

The budget system consists of budget departments and budget objects. There's a budget department in each locality to carry on all the budget work. Each budget department manages a number of budget objects. Budget objects are objects of the budget system, they may pay or spend money to/from the state budget.

Yearly, before the new year, the budget departments estimates the amounts that will be collected and makes expenditure plans for the budget objects for the next year. These estimated collections and spending plans are called budget estimates.

The state budget consists of state revenue resources and expenditures in the budget estimates which are approved and implemented in one year.

The state budget fund is all the amounts in the budget accounts of all the state budget departments in the country. The budget fund is managed by the state treasury.

4.1.1 Structure of the Budget System

The State budget is organised into a hierarchy corresponding to the administration offices. It consists of the central budget and local budget departments at three levels: province, district and commune. Each department manages a number of the departments of the locality at the lower level, see figure 14.

There are two different kinds of commune budget called village and quarter budget which are located in villages and quarters respectively.

Beside the departments at the lower level, a budget department also has a number of budget objects. One budget object is managed by one and only one budget department.

In a locality, the budget department manages its budget objects and itself is steered by the administrative office, see figure 15

[6]This section is quoted from [5]

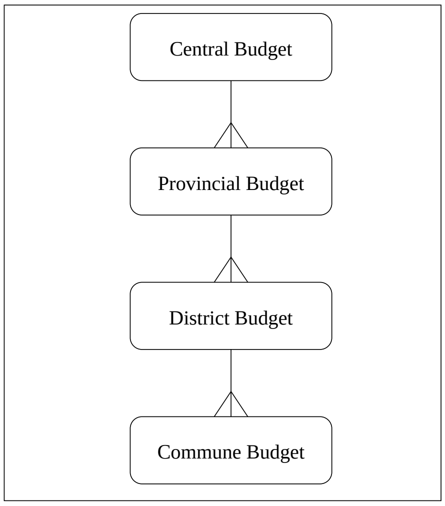

Figure 14: The Budget System

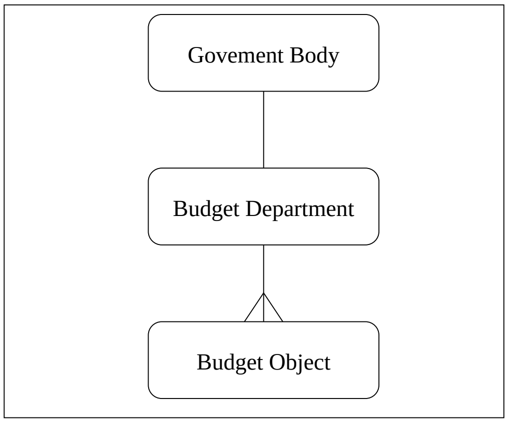

Figure 15: Budget System in a Locality

4.1.2 The budget objects

We just defined budget objects as objects of the state budget. A budget object may pay or spend the state budget directly or itself is divided into some other objects. We call budget objects which pay or spend amounts to the state budget, budget holders. So, a budget object may be a budget holder or itself consists of of a number of other budget objects.

Each budget object is managed by a particular budget department. All the sub-objects of a certain budget object have to belong to the same budget department. The budget departments perhaps manage a budget object directly or via intermediate objects. The procedures depend on the budget objects with their business or sometimes on the particular management processes.

In the budgetary field, besides the budget office, a budget object could be also managed by its upper object. The upper object might be at the same level with the object (so it's a sub-object of the upper one as said above) or at the upper budget level. Anyway, state budget is always done in the budget levels separately, see figure 16.

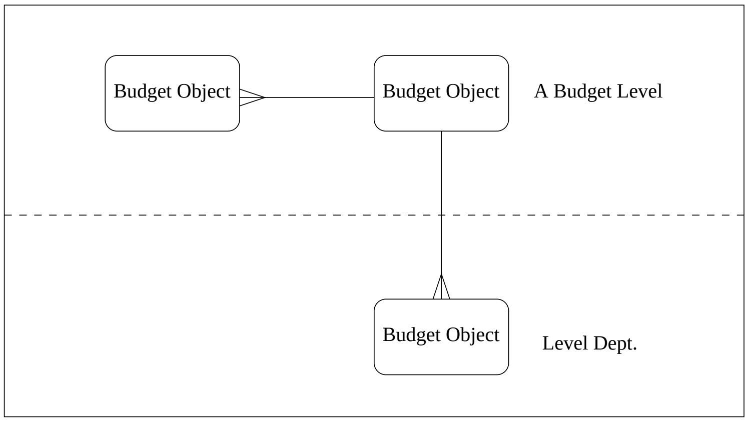

Figure 16: A Budget Object

4.1.3 Functions of the Budget System

There are three essential processes of one budget year: Budget Planning, Budget Controlling and Budget Accounting.

Budget planning is the process determining the plan for a budget year. It mainly consists of two processes: budget estimating and budget distributing.

Budget Controlling consists of two parallel activities: budget collection and budget expenditure. The budget system doesn't perform these activities by itself, it is assisted by some other professional system. These are the tax system, customer office, aid reception management department, treasury system, investment development system, etc.

Budget accounting includes all the accounting activities. In fact, besides assisting budget management, it is supposed to serve two other activities: making budget statement reports and inspection processes.

Budget Estimating The budget estimating is a process estimating the amounts of state budget that will be collected and spent in the next year. The process is implemented from the lowest level to the highest. The amounts are shown corresponding to the collecting and spending categories.

The process is performed on the budget system with the units at high levels of management. It means, there are some budget objects at the lower side of the "budgetary-tree" that don't take part in the process. Depending on the role in the system, a budget object may be a budget estimated object or not. Because the process is implemented from lower to the upper objects, the upper object of a budget estimated object is always an estimated one.

The main procedures of the process in a budget level are:

1. The budget estimated objects make the budget estimate for themselves and report to the budget department or to the immediate-upper budget object.
2. The budget department inspects, summarises budget estimates, plans the distribution to report to the administration office.
3. The administration office decides the budget estimate and distribution and reports to the financial office at the higher level.

We consider figure 17.

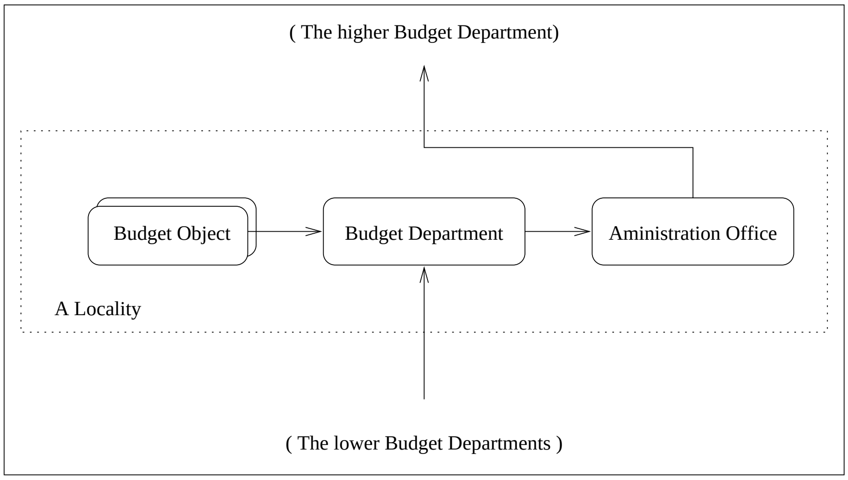

Figure 17: The Budget estimation process

The arrows define the directions of the procedures. Obviously, sometime there are exchanges: a unit and its upper together change its estimate to balance the other units at the same level. The picture points out the basic procedures of the process.

Budget Distributing Budget Distributing is the activity apportioning state budget. Based on the budget estimates which have been approved, the government offices distributes state budget to their responsible units. The process is carried out from the highest level to the lowest. After having the approved budget estimate, budget objects distribute state budget to its sub budget objects. The distribution has to fit to the given budget estimate of the object, see: figure 18.

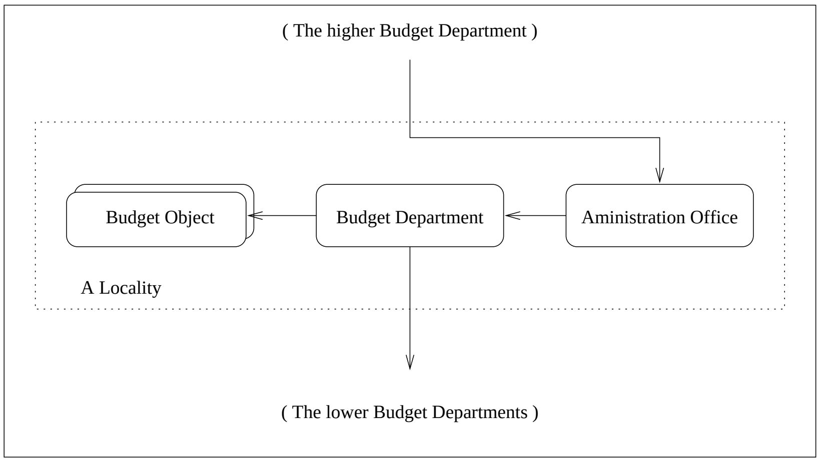

Figure 18: The Budget Distribution Process

In the budget controlling, there may be differences between plans and the reality. The approved budget estimated may be changed to fit practice. A budget department may change budget plans of its objects. A budget object can change the budget plan of its objects but the summary has to fit to the approved estimate.

Budget Collection State budget revenue resources are all the amounts of collection from collection activities of the country: taxation, contribution, aid, borrowing, etc. The collection amounts are managed in particular categories. An amount may be shared between several budget levels. Depending on the category, the budget law defines the rate to share collected amounts to different levels. In particular, in a budget level, there are some categories being determined to be collected for it only, the others need to be shared with the other levels. At present, we have the following kinds of sharing: province-centre, district-province and village-district-province. The percentages of the distributions are defined for particular collection categories in the Budget Law.

Budget Expenditure State budget expenditure consists of expense amounts for the country: social-economic development, ensuring defence, maintaining state activities, loan payment, etc.

For each budget levels are determined for particular expenditure categories. The budget estimate of a budget department shows its expenditure categories and the amounts it will be able to spend. The stage budget has to be spent in the correct category with a permitted value.

The budget of a level must not be used in the others except: a higher level could transfer budget to its lower ones when it delegates them to do work in its responsibilities or to balance the developments between local areas.

An expenditure amount has to:

1. be in the approved budget estimates;
2. be decided by the budget object itself;
3. be accepted by the responsible offices;

Budget Accounting The accounting process is to define status of budget implementation. It reflects the budget progresses and, in the other hand, provides for the budget controlling. The accounting data is also used for making statement reports and other processes.

Making statement reports The purpose of the process is to make the statement report for the whole country. The report is the summary of all the budget activities in the budget year. The process is carried out from the budget objects to the central budget department.

In a budget level:

1. The budget estimated objects make the statement for themselves and report to the upper object or to the budget department.
2. The budget department inspects, summarises statement reports and sends to the administration office.
3. The administration office inspects the statement report.
4. The budget department sends statement reports to the upper budget department.

We consider figure 19

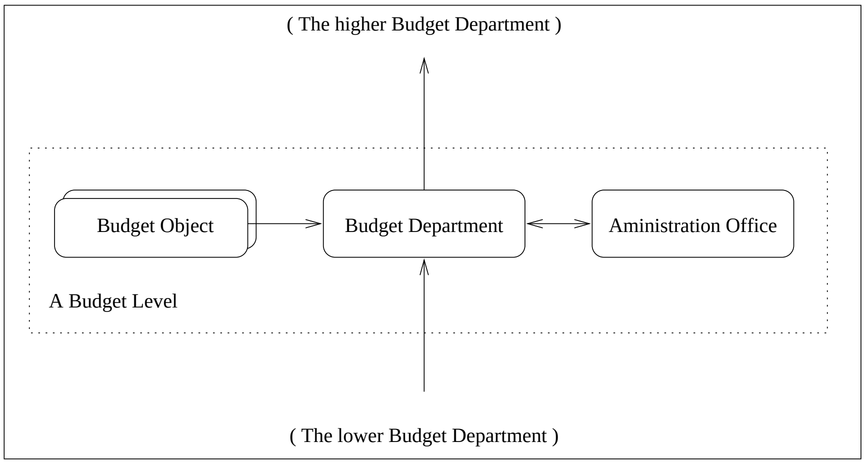

Figure 19: Making Statement Reports

4.2 Formal model

4.2.1 Specification of the basic concepts

The object T defines some basic types for the system. The Category type presents the classification of the State Budget. Each budget amount is defined to belong to a kind of collection or expenditure. A budget category may be divided into some other categories. The ($\leq$) operator presents the son-father relation between categories. Hid stands for budget holder identification; it is needed when the budget holder transacts directly with the budget offices.

object
  T :
    class
      type Category, Amount = **Int**, Hid

      value
        expenditure : Category,

        collection : Category,

        $\leq$: Category $\times$ Category $\rightarrow$ **Bool**,

        related : Category $\times$ Category $\rightarrow$ **Bool**
        related(c1, c2) $\equiv$ c1 $\leq$ c2 $\vee$ c2 $\leq$ c1

      axiom

```
    /* expenditure and collection are endependent */
    ~ related(collection, expenditure),

    /* all category belong to either expenditure or collection */
    ∀ c : Category • c ≤ expenditure ∨ c ≤ collection,

    /* there′s no common category between two unrelated ones */
    ∀ c, c1, c2 : Category • c ≤ c1 ∧ c ≤ c2 ⇒ related(c1, c2),

    /* category reflexive */
    ∀ d : Category • d ≤ d,

    /* category antisymmetric */
    ∀ d1, d2 : Category • d1 ≤ d2 ∧ d2 ≤ d1 ⇒ d1 = d2,

    /* category transitive */
    ∀ d1, d2, d3 : Category • d1 ≤ d2 ∧ d2 ≤ d3 ⇒ d1 ≤ d3
  end
```

A budget fund is simply defined in the BUDGET_FUND scheme. There are two main activities involving budget funds: Budget holders spend or pay amounts to them. When a budget office issues a stage budget to its budget holder, it need to ensure the fund is enough for the supply. That would be checked by the can_spend function.

```
scheme
 BUDGET_FUND =
  class
   type Fund = T.Amount

   value
    pay : T.Amount × Fund → Fund
    pay(a, f) ≡ a + f,

    spend : T.Amount × Fund ~→ Fund
    spend(a, f) ≡ f − a pre can_spend(a, f),

    can_spend : T.Amount × Fund → Bool
    can_spend(a, f) ≡ a ≤ f
  end
```

4.2.2 The Budget Plan

The SCHEDULE scheme defines the schedule of spending/collecting amounts for a certain budget category. The sum function computes the summary of two schedules. We also define communicative and associative properties of the function. The properties present the unordered process summarising schedules.

```
scheme
  SCHEDULE =
    class
      type Schedule

      value
        empty : Schedule,
        sum : Schedule × Schedule → Schedule

      axiom
        [sum_empty] ∀ i : Schedule • sum(empty, i) ≡ i,

        [sum_communicative]
          ∀ i1, i2 : Schedule • sum(i1, i2) ≡ sum(i2, i1),

        [sum_associative]
          ∀ i1, i2, i3 : Schedule •
            sum(sum(i1, i2), i3) ≡ sum(i1, sum(i2, i3))
    end
```

Separating categories is an outline characteristic of the budget planning. A budget plan defines schedules for categories independently. The FORM scheme defines the Form type as the collection of categories of a plan. No form includes nested categories: the corresponding schedules are independent.

```
scheme
  FORM =
    class
      type Form

      value
        empty : Form,

        is_in : T.Category × Form → Bool,

        ≤ : Form × Form → Bool
```

```
        f1 ≤ f2 ≡
          (
            ∀ c1 : T.Category •
              is_in(c1, f1) ⇒
              (∃ c2 : T.Category • is_in(c2, f2) ∧ T.(≤)(c1, c2))
          ),

        + : Form × Form → Form
        f1 + f2 as f
          post
            f1 ≤ f ∧
            f2 ≤ f ∧ (∀ f′ : Form • f1 ≤ f′ ∧ f2 ≤ f′ ⇒ f ≤ f′)

      axiom
        [empty_] ∀ c : T.Category • ∼ is_in(c, empty),

        ∀ c1, c2 : T.Category, f : Form •
          is_in(c1, f) ∧ is_in(c2, f) ⇒ c1 = c2 ∨ ∼ T.related(c1, c2)
    end
```

The budget plans are specified in the PLAN scheme. It has some functions to get form and schedule from a plan. The (≤) function checks if one plan can be a result after generalising an other. The function is useful in the budget planning processes: Budget objects or departments may change their plans, for example to make the budget estimate more comprehensive to report to the upper levels, or to make the budget distribution more detailed to provide to its units, but the original and the modified one have to satisfy the (≤) relation. The sum function summarises a plans.

```
scheme
  PLAN =
    extend extend SCHEDULE with FORM with
      class
        type Plan

        value
          empty : Plan,

          form : Plan → Form,

          is_in : T.Category × Plan → Bool
          is_in(c, p) ≡ is_in(c, form(p)),

          schedule : Plan × T.Category ~→ Schedule
          schedule(p, c) post true pre is_in(c, p),
```

```
summarise : Plan × Plan → Bool
summarise(p1, p2) ≡
  form(p1) ≤ form(p2) ∧
  (
    ∀ c : T.Category •
      is_in(c, p2) ⇒
      let
        cs =
          { c′ | c′ : T.Category • T.(≤)(c′, c) ∧ is_in(c′, p1) }
      in
        schedule(p2, c) = S.sum(schedule, empty, p1, cs)
      end
  ),

sum : Plan × Plan → Plan
sum(p1, p2) as p
  post
    form(p) = form(p1) + form(p2) ∧
    (
      ∀ c : T.Category •
        is_in(c, p) ⇒
        let
          cs1 =
            { c′ |
              c′ : T.Category • T.(≤)(c′, c) ∧ is_in(c′, p1)
            },
          sch1 = S.sum(schedule, empty, p1, cs1),
          cs2 =
            { c′ |
              c′ : T.Category • T.(≤)(c′, c) ∧ is_in(c′, p2)
            },
          sch2 = S.sum(schedule, empty, p1, cs2)
        in
          schedule(p, c) = sum(sch1, sch2)
        end
    )

object
  SP :
    class
      type Office = T.Category,  Report = Schedule,  Para = Plan

      value
        sum_report : Report × Report → Report = sum
```

```
        end,
      S : SUM(SP)

    axiom form(empty) ≡ empty
  end
```

4.2.3 Budget Objects

The BUDGET_HOLDER scheme specifies a budget object which is allowed to spend/collect state budget amounts (called a budget holder). The transaction type represents information of a budget transaction. The transact function does a transaction, the apply function imposes a budget distribution for a budget holder.

```
scheme
  BUDGET_HOLDER =
    class
      type
        Holder,
        Transaction ::
          category : T.Category
          amount : T.Amount ↔ re_amount
          info : Tran_info,
        Tran_info

      value
        transact : Transaction × Holder ~→ Holder
        transact(tr, bh) post true pre can_transact(tr, bh),

        can_transact : Transaction × Holder → Bool

      value
        apply : PL.Plan × Holder ~→ Holder,
        get : Holder → PL.Plan

      axiom [ get_apply ] ∀ d : PL.Plan, ob : Holder • get(apply(d, ob)) ≡ d
    end
```

A budget object may be a budget holder or itself consist of some other budget objects. The BUDGET_OBJ_STRUC scheme defines structure of a budget object. The is_in function checks if a budget holder with the given Hid belongs to a budget object and the get_holder returns the budget holder if its Hid belong to a given budget object. A budget object if not a budget holder, apart from sub-objects also has other information such as budget estimate, budget distribution... These are undefined and we provisionally call them Info.

```
scheme
  BUDGET_OBJ_STRUC =
    extend BUDGET_HOLDER with
      class
        type
          Object = BHolder | Compound,
          BHolder :: id : T.Hid  bh : Holder,
          Compound :: objects : Object-set  info : Info,
          Info

        value
          is_in : T.Hid × Object → Bool
          is_in(id, ob) ≡
            case ob of
              mk_BHolder(id′, bh) → id = id′,
              mk_Compound(os, i) →
                ∃ ob′ : Object • ob′ ∈ os ∧ is_in(id, ob′)
            end,

          get_holder : T.Hid × Object ~→ Holder
          get_holder(id, ob) ≡
            case ob of
              mk_BHolder(id′, bh) → bh,
              mk_Compound(os, i) →
                let ob′ : Object • ob′ ∈ os ∧ is_in(id, ob′) in
                  get_holder(id, ob′)
                end
            end
            pre is_in(id, ob)

        axiom
          ∀ id : T.Hid, ob1, ob2 : Object, cp : Compound •
            {ob1, ob2} ⊆ objects(cp) ⇒
            is_in(id, ob1) ∧ is_in(id, ob2) ⇒ ob1 = ob2
      end
```

The BUDGET_OBJ is an extend of BUDGET_OBJ_STRUC with functions describing budget planing processes. The processes are done in the whole system from the budget holders to the central budget department or in the contrary direction. In each budget department or budget object, there are some modifications between the receipt and sending plans (but they have to satisfy the ($\leq$) relation between plans). Because of this, there's no function presenting entirely a process. The estimate and apply functions for a given object are stages in the estimate and distribution processes as they proceed up (estimate) and down (distribution) the levels. The get function gets the distribution of a object. The is_done function checks if the distribution process is done for the object: budget distributions of its and its sub_objects are fitted.

```
scheme
  BUDGET_OBJ =
    extend BUDGET_OBJ_STRUC with
      class
        type Process == estimation | distribution

        value
          is_estimated : Object → Bool,

          estimate : Object ~→ PL.Plan
          estimate(ob) post true pre is_estimated(ob),

          apply : PL.Plan × Object → Object,

          get : Object → PL.Plan,

          is_done : Process × Object → Bool
          is_done(p, ob) ≡
            case ob of
              mk_BHolder(id′, bh) → true,
              mk_Compound(os, i) →
                case p of
                  distribution →
                    let estimate_sum = S.sum(SP.get, PL.empty, (), os) in
                      PL.summarise(get(ob), estimate_sum)
                    end,
                  estimation →
                    (∀ ob : Object • ob ∈ os ∧ is_estimated(ob)) ⇒
                    let get_sum = S.sum(SP.estimate, PL.empty, (), os) in
                      PL.summarise(estimate(ob), get_sum)
                    end
                end
            end

        object
          SP :
            class
              type Office = Object,  Report = PL.Plan,  Para = Unit

              value
```

```
            sum_report : Report × Report → Report = PL.sum,

            get : Para × Office → Report
            get(p, o) ≡ get(o),

            estimate : Para × Office → Report
            estimate(p, o) ≡ estimate(o)
          end,
        S : SUM(SP)

      axiom
        [get_apply_]
          ∀ d : PL.Plan, ob : Object • get(apply(d, ob)) ≡ d,

        /* any obj that includes an estimated obj must be an estimated one */
        ∀ ob1 : Object, ob2 : Compound •
          ob1 ∈ objects(ob2) ⇒ is_estimated(ob1) ⇒ is_estimated(ob2)
    end
```

4.2.4 Budget Departments

Based on the above scheme, BUDGE_DEP defines budget departments and their components. The BUDGET_DEP_STRUC defines the structure of the budget department. The BUDGET_DEP_COMP extends BUDGET_DEP_STRUC and defines basic components of a budget department. The budget BUDGET_DEP extends BUDGET_DEP_COM with definitions of the budget processes.

```
scheme
  BUDGET_DEP_STRUC =
    class
      type Department

      value
        deps : Department → Department-set,

        address : Department → M.Address,

        level : Department → M.Level
        level(d) ≡ M.level(address(d))

      axiom
        [hierarchic]
```

```
      ∀ d, d1 : Department •
        d1 ∈ deps(d) ⇒ M.(<)(address(d1), address(d)),

    [ unique_parent ]
      ∀ d1, d2, d3 : Department •
        d1 ∈ deps(d2) ∧ d1 ∈ deps(d3) ⇒ d2 = d3
  end
```

BUDGET_DEP_STRUC uses an object M that describes the addresses of the various officees and their relations. M is presented in appendix A.

```
scheme
  BUDGET_DEP_COMP =
    extend BUDGET_DEP_STRUC with
      class
        object BO : BUDGET_OBJ,  BF : BUDGET_FUND

        value
          objects : Department → BO.Object-set,

          is_in : T.Hid × Department → Bool
          is_in(id, d) ≡
            (
              ∃ ob : BO.Object • ob ∈ objects(d) ∧ BO.is_in(id, ob)
            ),

          get_holder : T.Hid × Department ~→ BO.Holder
          get_holder(id, d) ≡
            let ob : BO.Object • ob ∈ objects(d) ∧ BO.is_in(id, ob) in
              BO.get_holder(id, ob)
            end
            pre is_in(id, d)

        axiom
          /* one department doesn't have two holders with the same Hid */
          ∀ id : T.Hid, ob1, ob2 : BO.Object, d : Department •
            {ob1, ob2} ⊂ objects(d) ⇒
            BO.is_in(id, ob1) ∧ BO.is_in(id, ob2) ⇒ ob1 = ob2

        value
          fund : Department → BF.Fund,

          amount_pay : T.Category × T.Amount × Department → T.Amount,
```

```
    transact : T.Hid × BO.Transaction × Department ~→ Department
    transact(id, tr, d) post true pre can_transact(id, tr, d),

    can_transact : T.Hid × BO.Transaction × Department → Bool
    can_transact(id, tr, d) ≡
      is_in(id, d) ∧
      BO.can_transact(tr, get_holder(id, d)) ∧
      (
        T.(≤)(BO.category(tr), T.expenditure) ⇒
        BF.can_spend(BO.amount(tr), fund(d))
      )

  axiom
    [transact_]
      ∀ id : T.Hid, tr : BO.Transaction, d : Department •
        let
          d′ = transact(id, tr, d),
          c = BO.category(tr),
          a = BO.amount(tr)
        in
          if T.(≤)(c, T.expenditure) then
            fund(d′) = BF.spend(a, fund(d)) ∧
            get_holder(id, d′) = BO.transact(tr, get_holder(id, d))
          else
            let a′ = amount_pay(c, a, d), tr′ = BO.re_amount(a′, tr) in
              fund(d′) = BF.spend(a′, fund(d)) ∧
              get_holder(id, d′) = BO.transact(tr′, get_holder(id, d))
            end
          end
        end
end


scheme
  BUDGET_DEP =
    extend BUDGET_DEP_COMP with
      class
        value
          apply : PL.Plan × Department → Department,

          get : Department → PL.Plan,

          is_done : Department → Bool
          is_done(d) ≡
```

```
    let
      objs_plan = S1.sum(SP1.mk_report, PL.empty, (), objects(d)),
      deps_plan = S.sum(SP.mk_report, PL.empty, (), deps(d)),
      get_sum = PL.sum(deps_plan, objs_plan)
    in
      PL.summarise(get(d), get_sum)
    end,

  estimate : Department → PL.Plan
  estimate(d) ≡
    let deps_plan = S.sum(SP.mk_report, PL.empty, (), deps(d)) in
      PL.sum(deps_plan, objects_estimate(d))
    end,

  objects_estimate : Department → PL.Plan

axiom
  [get_apply_]
    ∀ p : PL.Plan, d : Department • get(apply(p, d)) ≡ p,

  ∀ d : Department •
    (
      ∀ ob : BO.Object • ob ∈ objects(d) ∧ BO.is_estimated(ob)
    ) ⇒
    objects_estimate(d) =
    S1.sum(SP1.mk_report, PL.empty, (), objects(d))

object
  SP :
    class
      type Office = Department,  Report = PL.Plan,  Para = Unit

      value
        sum_report : Report × Report → Report = PL.sum,

        mk_report : Para × Office → Report
        mk_report(p, o) ≡ get(o)
    end,
  S : SUM(SP),
  SP1 :
    class
      type Office = BO.Object,  Report = PL.Plan,  Para = Unit

      value
        sum_report : Report × Report → Report = PL.sum,
```

```
        mk_report : Para × Office → Report
        mk_report(p, o) ≡ BO.get(o)
      end,
    S1 : SUM(SP1)
  end
```

4.2.5 Hierarchical development

BUDGET_DEP_STRUC define the department structure recursively through the function *deps*. The axiom *hierarchic* and *unique_parent* show that there must be a tree structure and we can make this explicit by developing the type *Department* in BUDGET_DEP_STRUC_H

```
scheme
  BUDGET_DEP_STRUC_H =
    class
      type
        Department :: map : Department_map  address : M.Address,
        Department_map′ :: info : D_info  provinces : Province_map,
        Department_map = {| dm : Department_map′ • is_wf(dm) |},
        Province_map = M.Pid →m Province,
        Province :: info : D_info  districts : District_map,
        District_map = M.Did →m District,
        District :: info : D_info  communes : Commune_map,
        Commune_map = M.Cid →m Commune,
        Commune :: info : D_info,
        D_info

      value
        is_wf : Department_map′ → Bool
        is_wf(m) ≡
          dom provinces(m) = M.provinces() ∧
          (
            ∀ p : M.Pid •
              dom districts(provinces(m)(p)) = M.districts(p) ∧
              (
                ∀ d : M.Did •
                  d ∈ M.districts(p) ⇒
                  dom communes(districts(provinces(m)(p))(d)) =
                  M.communes(p, d)
              )
          ),
```

```
    deps : Department → Department-set
    deps(d) ≡
      { mk_Department(map(d), a) |
        a : M.Address • M.(<)(a, address(d))
      },

    level : Department → M.Level
    level(d) ≡ M.level(address(d))
  end
```

It is straightforward to show that BUDGET_DEP_STRUC_H impliments BUDGET_DEP_STRUC. The type *D_info* allows us to add the budget objects, funds and plans at each level. For examle, we develop BUDGET_DEP_COMP by defining

```
value
      objects : D_info → BO.Object-set,

      objects : Department → BO.Object-set
      objects(d) ≡
        case address(d) of
          M.general_address → objects(info(map(d))),
          M.province_address(p) → objects(info(provinces(map(d))(p))),
          M.district_address(p, d) →
            objects(info(districts(provinces(map(d))(p))(d))),
          M.commune_address(p, d, c) →
            objects(info(communes(districts(provinces(map(d))(p))(d))(c)))
        end
```

We treat funds and plans similarly

5 Treasury System

In this section, we provide an analysis and a step-wise formal model development of the Treasury Information System, which is an important subsystem of the Financial Information System (see [6] for more in detail) .

5.1 Narrative

5.1.1 Role and Structure of the Treasury System

The treasury system is an important subsystem of the Ministry of Finance, managing the State budget funds, mobilising capital for the State budget by issuing bills and bonds, performing payment for the budget holders following the plan which was approved. The role of the treasury system is described in figure 20 .

A production-business unit, either public or non-public, which is responsible to claim or pay budgetary amounts, depending on its business form and scale, is called a **customer** or **budget holder**.

The treasury system is organised into a hierarchy. The **State Central Treasury** is at the highest level and is located in the capital; a **Province Treasury** is at the middle level and is located in each province and a **District Treasury** is at the lowest level and is located in each district. Figure 20 illustrates this structure: All the Treasury departments at the three levels have to make revenue and expenditure accounting. The lower Treasury departments send their daily reports and monthly summary report on the revenue and expenditure situation to the upper levels. The State Treasury department synthesises all the reports from lower levels, makes a summary report daily and sends this **daily revenue and expenditure report** to the State budget department and the Finance Minister. Monthly, the State Treasury department sends the **actual revenue-expenditure report** to the State budget department. These reports help the State Budget department make a balance between revenue collection and expenditure, make a stock report and manage the state budget fund. Then it reports the **state budget management** situation to the Finance Minister in the following period. Conversely, the Finance Minister sends the **financial policy, or guidance information** such as issuing bonds, bills in order to mobilise capital for state budget to the State Treasury department, etc...

At the end of the year, based on the final reports which the budget holders and the Province Financial departments (they are in charge of synthesising local budgets) and the General Tax department send, the State Budget department makes a summary report of revenue-expenditure for the whole country. Then it submits the final summary report to the National Assembly for approval.

Based on the organisational structure of the functions and responsibilities of the treasury system,

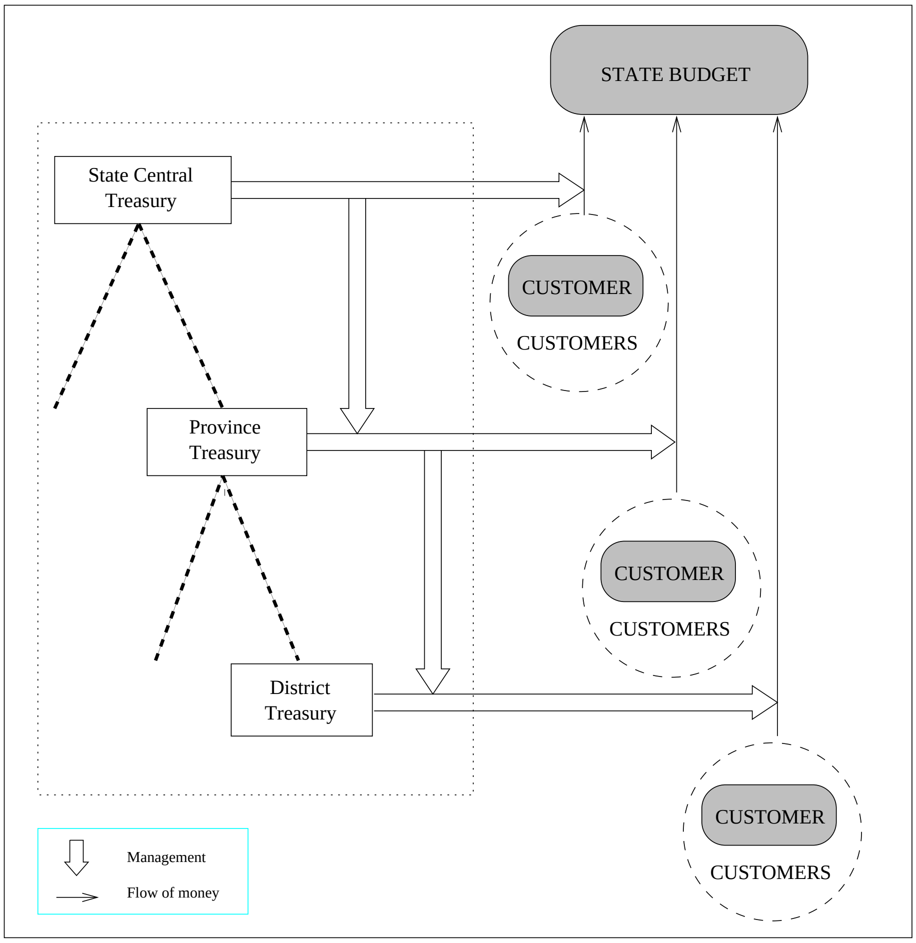

Figure 20: Role and Structure of the Treasury System

the activities of the treasury system are described in the figure 21.

The treasury system operates as "special bank" of all the budget holders. The budget holders have to open their accounts for receipts and disbursements of budgetary amounts following the approved planning.

Based on the approved expenditure plan which the State budget sends, the budget holders make their **detailed expenditure plans**, following each expenditure item in state budget nomenclature monthly or quarterly. They send this plan to the Treasury office where the budget holders make their payment transactions. When the budget holders go to the Treasury department to get their money for spending, it depends on the budget levels. If the budget holder belongs to the central budget, they go to the State Treasury department; if they belong to the local budget they go to the Province or District Treasury department to make their transactions. The Treasury department checks whether the expenses are in the approved plan, in the right expenditure items in the plan, and for the right object. If these expenses are valid the Treasury office performs payment for them.

The Treasury System is mainly focused on two objectives:

- Identifying the account incidence for each customer, i.e. each account required and amount in each customer.
- Administering the customer expenditure activities of budgeting.

5.1.2 Relationship between the Treasury system and other parts of the MOF

State Budget : allocative ceiling or release order amount for budget holder by Treasury System. Treasury System prepares treasury reports. These reports include: daily express account, actual monthly Revenue and Expenditure and End-year Financial Report. State Central Treasury makes these reports and sends them to the State Budget and Ministry of Finance.

Tax System : Treasury System supplies and compares tax collection data with each same level Tax System.

General department of Investment and Development : opens accounts in State Central Treasury, receives all allocation ceilings and release orders by account. Treasury System manages, supplies, compares the actual expenditure data with this system, see: figure 21

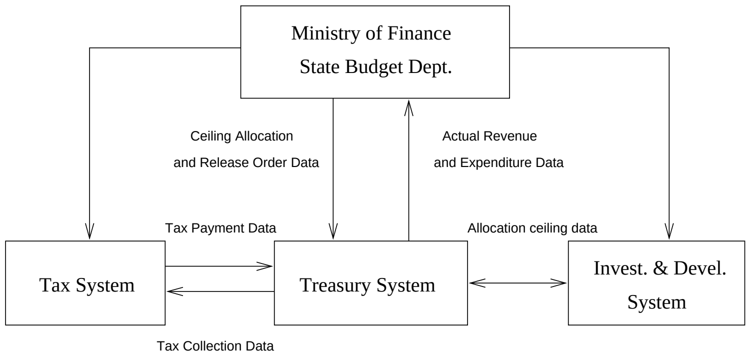

Figure 21: Relationship of the Treasury's Informations

5.1.3 Management of Treasury information

We can see that the treasury activities at all three levels have common functions such as management of customers of treasury accounting and manage bill/bond programs, the activities are described in figure 22:

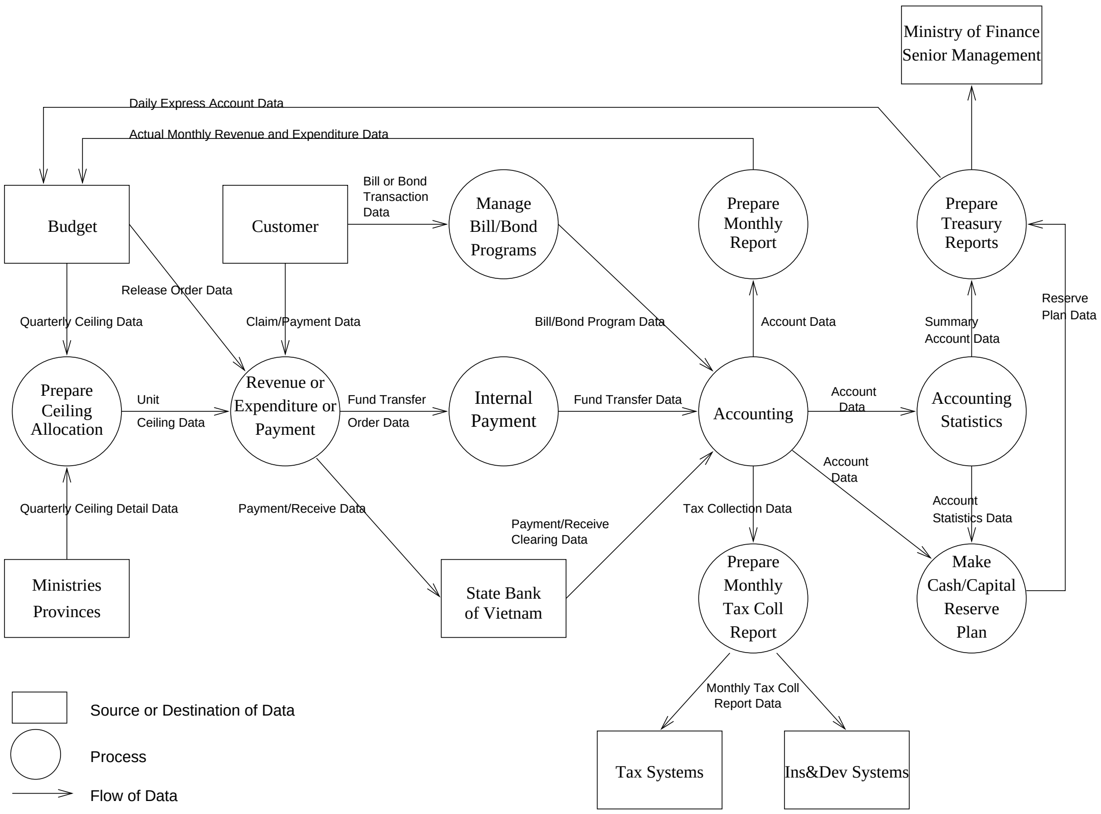

Figure 22: Abstract Information Dataflow Diagram

Management of Treasury information includes managing revenue and expenditure budgetary amounts.

- Manage revenue budgetary amount: see figure 23
- Manage expenditure budgetary amount: see figure 24

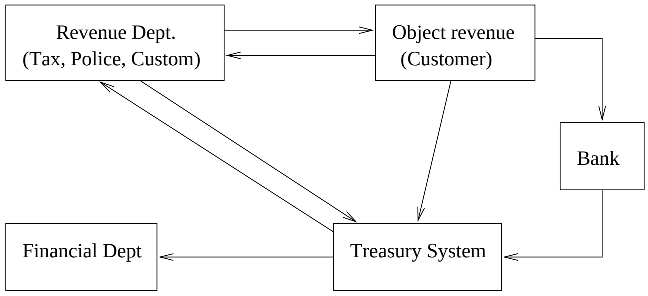

Figure 23: Abstract Information Dataflow Diagram of Budgetary Revenue

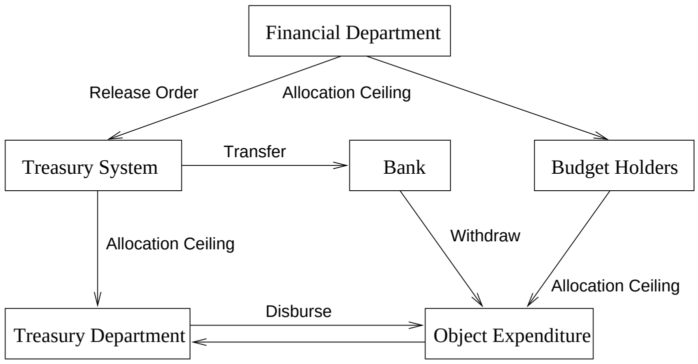

Figure 24: Abstract Information Dataflow Diagram of Budgetary Expenditure

5.2 Formal model

5.2.1 Budgets

The treasury's work depends on budget plans generated in the budget system [5]. Once budget plans are approved, detailed expenditure plans are made by budget objects and sent to the treasury. It is on the basis of these that the Treasury makes decisions about payments. So we shall reuse the modules SCHEDULE, FORM and PLAN from the budget system. This means we will also need to reuse the module TYPES.

scheme
  TYPES =
    class
      type Date, Category, Amount = **Int**, Money = Amount, Hid

      value

expenditure : Category,

collection : Category,

$\leq$: Category $\times$ Category $\rightarrow$ **Bool**,

related : Category $\times$ Category $\rightarrow$ **Bool**
related(c1, c2) $\equiv$ c1 $\leq$ c2 $\vee$ c2 $\leq$ c1,

$\leq$: Date $\times$ Date $\rightarrow$ **Bool**

axiom
/* expenditure and collection are independent */
$\sim$ related(collection, expenditure),

/* all category belong to either expenditure or collection */
$\forall$ c : Category $\bullet$ c $\leq$ expenditure $\vee$ c $\leq$ collection,

/* there′s no common category between two unrelated ones */
$\forall$ c, c1, c2 : Category $\bullet$ c $\leq$ c1 $\wedge$ c $\leq$ c2 $\Rightarrow$ related(c1, c2),

/* category reflexive */
$\forall$ c : Category $\bullet$ c $\leq$ c,

/* category antisymmetric */
$\forall$ c1, c2 : Category $\bullet$ c1 $\leq$ c2 $\wedge$ c2 $\leq$ c1 $\Rightarrow$ c1 = c2,

/* category transitive */
$\forall$ c1, c2, c3 : Category $\bullet$ c1 $\leq$ c2 $\wedge$ c2 $\leq$ c3 $\Rightarrow$ c1 $\leq$ c3,

/* date reflexive */
$\forall$ d : Date $\bullet$ d $\leq$ d,

/* date antisymmetric */
$\forall$ d1, d2 : Date $\bullet$ d1 $\leq$ d2 $\wedge$ d2 $\leq$ d1 $\Rightarrow$ d1 = d2,

/* date transitive */
$\forall$ d1, d2, d3 : Date $\bullet$ d1 $\leq$ d2 $\wedge$ d2 $\leq$ d3 $\Rightarrow$ d1 $\leq$ d3,

/* date total */
$\forall$ d1, d2 : Date $\bullet$ d1 $\leq$ d2 $\vee$ d2 $\leq$ d1
end

The fact that TYPES, PLAN and other schemes are shared between the budget and treasury systems means that we need to either make them objects global to both systems or make them parameters. In general, only modules expected to undergo no further development, i.e. easily translatable immediately into programming language code, should be made into global objects. So we will keep as global the object M from the budget system, that defines the structure of provinces and districts. We could do the same with an object T instantiating the scheme TYPES but have chosen to make it a parameter to show how such parameters are dealt with. The scheme PLAN, depending on SCHEDULE and FORM will certainly need more development. The structure of schedules, for example, still needs to be designed. So PLAN will be a parameter.

```
scheme
  SCHEDULE(T : TYPES) =
    class
      type Schedule

      value
        empty : Schedule,
        sum : Schedule × Schedule → Schedule,
        is_scheduled : T.Date × T.Amount × Schedule → Bool

      axiom
        [sum_empty] ∀ i : Schedule • sum(empty, i) ≡ i,

        [sum_communicative]
          ∀ i1, i2 : Schedule • sum(i1, i2) ≡ sum(i2, i1),

        [sum_associative]
          ∀ i1, i2, i3 : Schedule •
            sum(sum(i1, i2), i3) ≡ sum(i1, sum(i2, i3))
    end

scheme
  FORM(T : TYPES) =
    class
      type Form

      value
        empty : Form,

        is_in : T.Category × Form → Bool,

        ≤ : Form × Form → Bool
        f1 ≤ f2 ≡
          (
            ∀ c1 : T.Category •
```

```
          is_in(c1, f1) ⇒
          (∃ c2 : T.Category • is_in(c2, f2) ∧ T.(≤)(c1, c2))
      ),

    + : Form × Form → Form
    f1 + f2 as f
      post
        f1 ≤ f ∧
        f2 ≤ f ∧ (∀ f′ : Form • f1 ≤ f′ ∧ f2 ≤ f′ ⇒ f ≤ f′)

  axiom
    [empty_] ∀ c : T.Category • ∼ is_in(c, empty),

    ∀ c1, c2 : T.Category, f : Form •
      is_in(c1, f) ∧ is_in(c2, f) ⇒ c1 = c2 ∨ ∼ T.related(c1, c2)
  end
scheme
 PLAN(T : TYPES) =
  extend extend SCHEDULE(T) with FORM(T) with
   class
    type Plan

    value
      empty : Plan,

      form : Plan → Form,

      is_in : T.Category × Plan → Bool
      is_in(c, p) ≡ is_in(c, form(p)),

      schedule : Plan × T.Category ~→ Schedule
      schedule(p, c) post true pre is_in(c, p),

      summarise : Plan × Plan → Bool
      summarise(p1, p2) ≡
        form(p1) ≤ form(p2) ∧
        (
          ∀ c : T.Category •
            is_in(c, p2) ⇒
            let
              cs =
                { c′ | c′ : T.Category • T.(≤)(c′, c) ∧ is_in(c′, p1) }
            in
              schedule(p2, c) = S.sum(schedule, empty, p1, cs)
            end
```

```
      ),

    sum : Plan × Plan → Plan
    sum(p1, p2) as p
      post
        form(p) = form(p1) + form(p2) ∧
        (
          ∀ c : T.Category •
            is_in(c, p) ⇒
            let
              cs1 =
                { c′ |
                  c′ : T.Category • T.(≤)(c′, c) ∧ is_in(c′, p1)
                },
              sch1 = S.sum(schedule, empty, p1, cs1),
              cs2 =
                { c′ |
                  c′ : T.Category • T.(≤)(c′, c) ∧ is_in(c′, p2)
                },
              sch2 = S.sum(schedule, empty, p1, cs2)
            in
              schedule(p, c) = sum(sch1, sch2)
            end
        )

  object
    SP :
      class
        type Office = T.Category,  Report = Schedule,  Para = Plan

        value
          sum_report : Report × Report → Report = sum
      end,
    S : SUM(SP)

  axiom form(empty) ≡ empty
end
```

5.2.2 Accounts

We reuse the generic ACCOUNT module originally developed for the external debt management system [7]. We hide everything exported from ACCOUNT except the type *Account* and three functions that are "lifted" by being redefined at this level: *open*, *close* and *can_close*. Then we add the specific functions *transact* for recording amounts in a category, *can_transact* for checking *transact*'s applicability, and *cat_balance* for finding the balance in a category.

```
scheme
  TREASURY_AC(T : TYPES) =
    hide X, AC in
      class
        object
          X : class type TK == open | close | cat_trans(cat : Category) end,
          AC :
            extend ACCOUNT(T, X) with
              class
                value
                  balance_filter :
                    Date × (Transaction_kind → Bool) × Account → Money
                  balance_filter(d, f, trl) ≡
                    balance(d, ⟨ tr | tr in trl • f(kind(tr)) ⟩)
              end

        type
          Account = AC.Account,
          Date = T.Date,
          Amount = T.Amount,
          Category = T.Category

        value
          /* lifted functions */
          open : Date → Account = AC.open,

          close : Date × Account ⥲ Account = AC.close,

          can_close : Date × Account → Bool = AC.can_close,

          /* new functions */
          transact : Date × Category × Amount × Account ⥲ Account
          transact(d, c, a, ac) ≡
            if T.(≤)(c, T.expenditure) then
              AC.withdraw(d, X.cat_trans(c), a, ac)
            else
              AC.deposit(d, X.cat_trans(c), a, ac)
            end
            pre can_transact(d, a, ac),

          can_transact : Date × Amount × Account → Bool
          can_transact(d, a, ac) ≡ a ≥ 0 ∧ AC.is_open(d, ac),
```

```
          cat_balance : Date × Category × Account → Amount
          cat_balance(d, c, ac) ≡
            let
              f =
                λ k : X.TK •
                  case k of X.cat_trans(c′) → T.(≤)(c′, c), _ → false end
            in
              AC.balance_filter(d, f, ac)
            end
      end
```

5.2.3 Treasury

A treasury "object" has two components, an account and a plan:

```
scheme
  TREASURY_OBJ(T : TYPES, PL : PLAN(T)) =
    hide AC in
      class
        object AC : TREASURY_AC(T)

        type Object :: account : AC.Account  plan : PL.Plan

        value
          open : T.Date × PL.Plan → Object
          open(d, pl) ≡ mk_Object(AC.open(d), pl),

          close : T.Date × Object ~→ Object
          close(d, o) ≡
            mk_Object(AC.close(d, account(o)), plan(o))
            pre can_close(d, o),

          can_close : T.Date × Object → Bool
          can_close(d, o) ≡ AC.can_close(d, account(o)),

          transact : T.Date × T.Category × T.Amount × Object ~→ Object
          transact(d, c, a, o) ≡
            mk_Object(AC.transact(d, c, a, account(o)), plan(o))
            pre can_transact(d, c, a, o),

          can_transact : T.Date × T.Category × T.Amount × Object → Bool
```

```
    can_transact(d, c, a, o) ≡
      PL.is_in(c, plan(o)) ∧
      PL.is_scheduled(d, a, PL.schedule(plan(o), c)) ∧
      AC.can_transact(d, a, account(o))
  end
```

A department has a number of objects referenced by object holder identifiers (as used in the budget system). There seems no reason in the treasury system to preserve the hierarchical structure of budget objects, which is useful in the budget system for combining and distributing budget amounts.

```
scheme
  TREASURY_DEPT(T : TYPES, PL : PLAN(T)) =
    hide O in
      class
        object O : TREASURY_OBJ(T, PL)

        type Department = T.Hid →m O.Object

        value
          open : T.Date × T.Hid × PL.Plan × Department ~→ Department
          open(d, id, pl, dept) ≡
            dept ∪ [ id ↦ O.open(d, pl) ]
            pre can_open(id, dept),

          can_open : T.Hid × Department → Bool
          can_open(id, dept) ≡ ∼ is_customer(id, dept),

          close : T.Date × T.Hid × Department ~→ Department
          close(d, id, dept) ≡
            dept † [ id ↦ O.close(d, dept(id)) ]
            pre can_close(d, id, dept),

          can_close : T.Date × T.Hid × Department → Bool
          can_close(d, id, dept) ≡
            is_customer(id, dept) ∧ O.can_close(d, dept(id)),

          is_customer : T.Hid × Department → Bool
          is_customer(id, dept) ≡ id ∈ dom dept,

          transact :
            T.Date × T.Hid × T.Category × T.Amount × Department ~→
              Department
          transact(d, id, c, a, dept) ≡
```

```
      dept † [ id ↦ O.transact(d, c, a, dept(id)) ]
      pre can_transact(d, id, c, a, dept),

    can_transact :
      T.Date × T.Hid × T.Category × T.Amount × Department → Bool
    can_transact(d, id, c, a, dept) ≡
      is_customer(id, dept) ∧ O.can_transact(d, c, a, dept(id))
  end
```

We can now model the treasury system as a hierarchy of general, provincial and district departments:

```
scheme
 TREASURY(T : TYPES, PL : PLAN(T)) =
  hide D in
   class
    object D : TREASURY_DEPT(T, PL)

    type
     Department = D.Department,
     Treasury′ :: general : Department  provinces : Provinces,
     Treasury = {| t : Treasury′ • is_wf(t) |},
     Provinces = M.Pid →m Province,
     Province :: province : Department  districts : Districts,
     Districts = M.Did →m District,
     District :: district : Department

    value
     is_wf : Treasury′ → Bool
     is_wf(t) ≡
      dom provinces(t) = M.provinces() ∧
      (
       ∀ p : M.Pid •
        p ∈ dom provinces(t) ⇒
        dom districts(provinces(t)(p)) = M.districts(p)
      )
   end
```

We have not specified the functions available at this level for opening, closing and transacting: we shall see in the distributed system that they will become irrelevant. Neither have we specified the reporting functions, as these are very similar to those in the taxation system [4]

5.3 Further work

Instead of describing here the development of the tresury system into an imperative and concurrent form, we will consider in a separate section 7 the integration of the taxation, budget and treasury systems into a single, distributed system.

6 External System

The external information system is a simple perspective figure of the financial information system. In order to decompose it, we draw the following survey[7].

6.1 Survey for External Financial Information system

The process of developing the economy in Vietnam demands a huge foreign capital resource in order to upgrade the backward infrastructure and to industrialize . The fund is mobilised mainly from external loans, external aids and domestic source of private foreign currency. For using this capital efficiently, we need to build an integrated financial information system for external financial affairs. The external financial information system (EFIS) involves financial information relating to external entities or external budget.

6.1.1 Social entities of EFIS

As mentioned above, the social entities of financial information system (EFIS) involve the entities which take part in activities of the external budget system. They include:

- External aid Donors
- Aid Receivers
- External aid Receivers
- External Lenders
- Taxpayers and other revenue resources [8]
- State Bank
- State Lenders[9]
- State Borrowers
- Government

In this system, the Ministry of Finance has the following tasks:

[7]This section is quoted from [8]

[8]For example import taxes or fees for using airports and harbours are payable in foreign currency

[9]The Government is building a relevant policy in order to mobilise more efficiently the domestic source of private foreign currency

1. Guiding ministries, government offices to build the state external budget plan compounding the plan of receiving external aids and borrowing external debts.
2. Managing external currency, implementing the state external budget plan and making statement reports.
3. Managing external aid reception
4. Performing the state activities of implementing and managing the state external debts and loans
5. Managing the treasury of foreign currency which includes collecting and distributing external budget.

These duties are realised by

- External financial Department
- International relationship Department
- Aid reception and management Department.

In order to illustrate the state external financial system and activities of each entity, we consider the figure 25.

6.1.2 Component Systems

Based on the duties and the role of entities above, we can divide our system into the following subsystems, which are relatively independent.

- System of collecting and managing internal revenue resource
- Foreign currency budget system
- Aid reception management system
- External aid reception

1 System of collecting and managing internal revenue resource The internal revenue resource of foreign currency is collected from following sources

- Import-export Tax

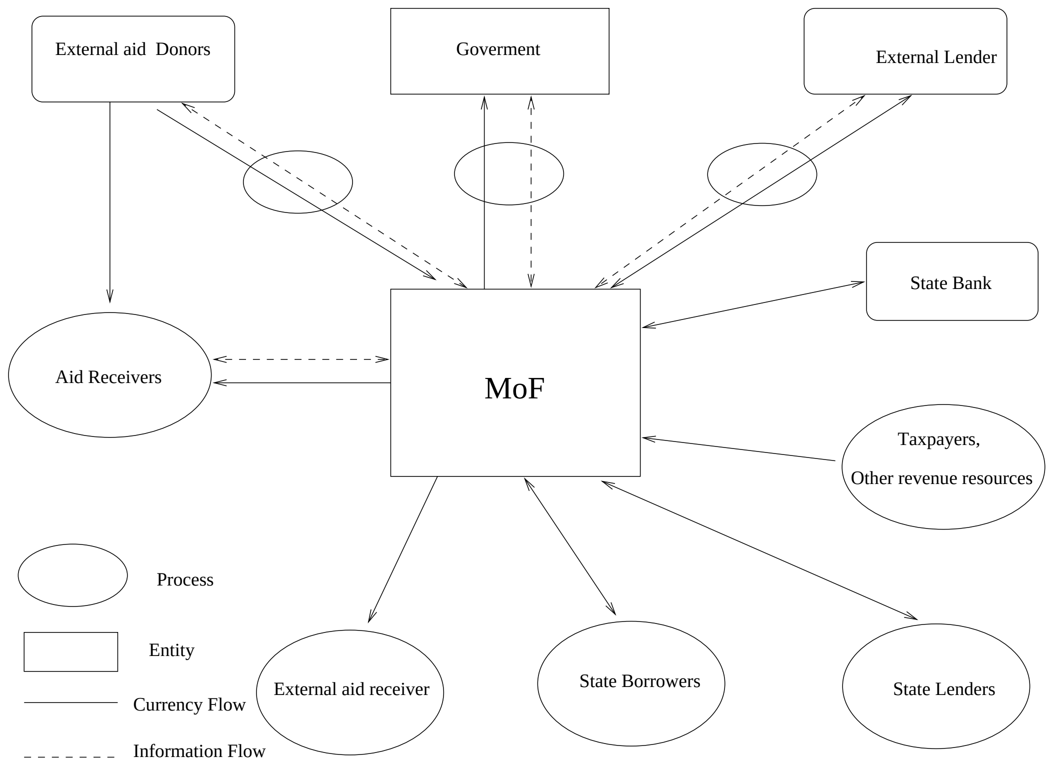

Figure 25: Data flow Diagram of External Financial System

- The fees payed by foreign currency such as: visa fee, fee for using harbours, airports, communication...
- Selling crude oil
- Export labour via the Government
- Domestic Lenders include individual people and economic organisations
  ...

The process of collecting and managing foreign currency is monitored by the external financial department. The foreign currency treasury is a part of the treasury system but relatively independent from the domestic currency treasury system and also managed by the external financial department. This information system is the same as the treasury system and can be described by figure 26.

The information of this system is processed like the tax system [3] and the treasury system. Therefore we omit the description in detail.

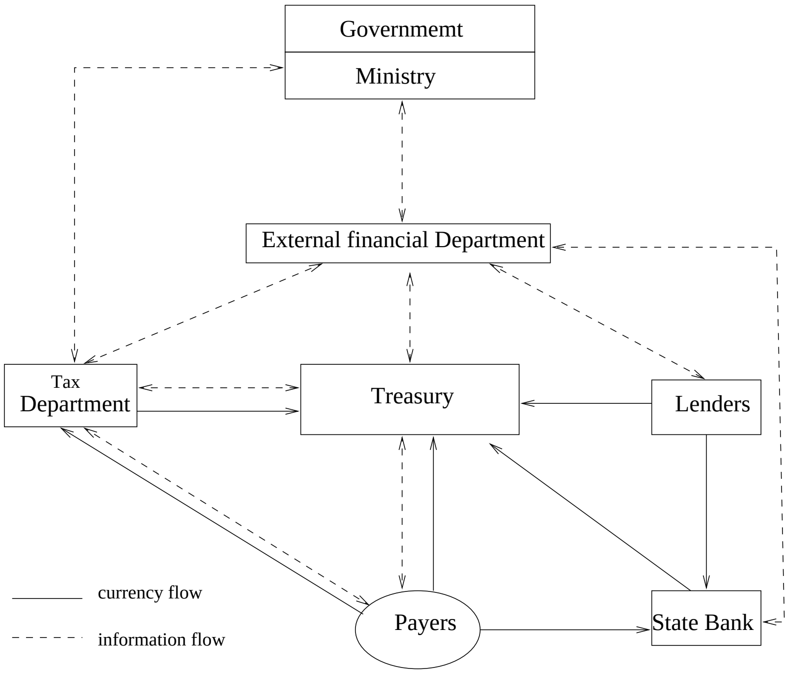

Figure 26: Data flow of collection process

2 Foreign currency budget system This system is a part of the budget system but managed by the external financial department. Cooperating with the general department of investment and development, the external financial department estimates the plan for collecting foreign currency, distributes expenditures and manages the process of realising it. Beside managing foreign currency budget, the external financial department builds the periodic statistic reports and sends them to the related entities. We refer the figure 27.

Like the description above, the information of this system can be processed as the budget system

3 Aid reception management system External aid is an important resource for investment and development in Vietnam. The MoF manages all kinds of external aid of foreign organisations via the Government. The MoF monitors the process of reception for each external aid and makes the periodic synthetic reports of aid reception situation to related entities.

The description in detail is presented in section 6.2.

4 External debt management system External debt is a crucial resource for investment and development in Vietnam. The information of debt situation has to be provided timely and accurately. This information system is described in detail in section 6.3.

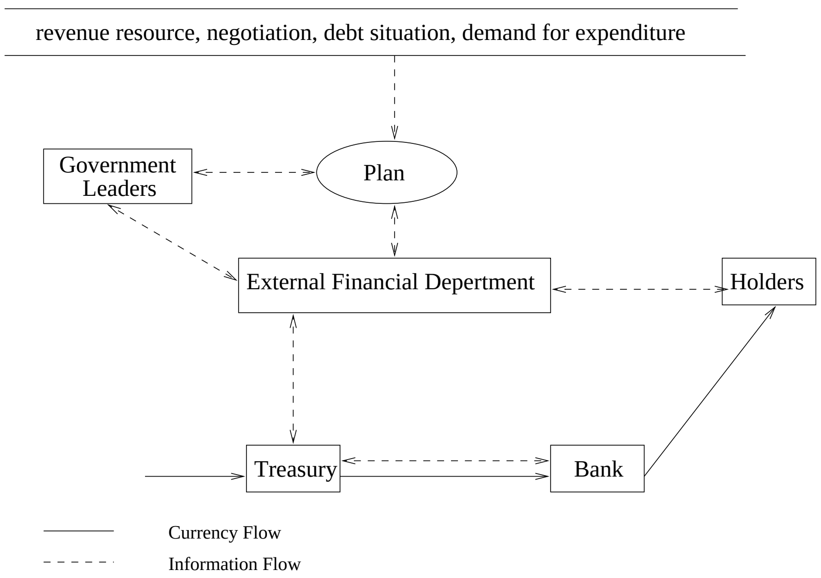

Figure 27: Data flow Diagram of External Budget System

Like the description above, in the next sections we concentrate on describing the information of the systems of external aid reception and external debts. The other duties can be considered similar to domestic budget system and treasury system.

6.2 External aid reception

External aids are the aids of foreign Governments (including the United Nations) and of the international organisations to Vietnamese organisations (aid Receivers) via the Government. They are based on contracts or conventions of Government and Aider.

In general, the aids of the non-government organisations are only small and with incomplete information. It is very difficult to estimate a plan, so for this kind of aid the MoF only monitors the reception situation. Its information can be reflected as a simple account.

Therefore, we concentrate on describing the information of aid reception for monitoring the aids of foreign Governments (including the United Nations) which is performed by the aid reception management department.

6.2.1 Narrative

Each external aid may be by goods or by money. Via negotiation, the Government has a reception plan for each aid, to which is attached a schedule and target (in detail) of aid.
Based on the plan, the aid receiver can receive external aid directly from the donor or indirectly via the Ministry of Finance. When receiving each aid, the Ministry of Finance distributes it to the receivers, makes a record account and books in the state asset register. If the receivers receive directly from the aider then they report to Ministry of Finance in order to account for it. Based on the monitoring, the Ministry of Finance makes periodic reports of the situation of receiving external aids.

When the aid is by money then it is sent to the treasury and distributed by the external financial department. The use of this amount is monitoring by the aid reception management department. The process can be described by figure 28.

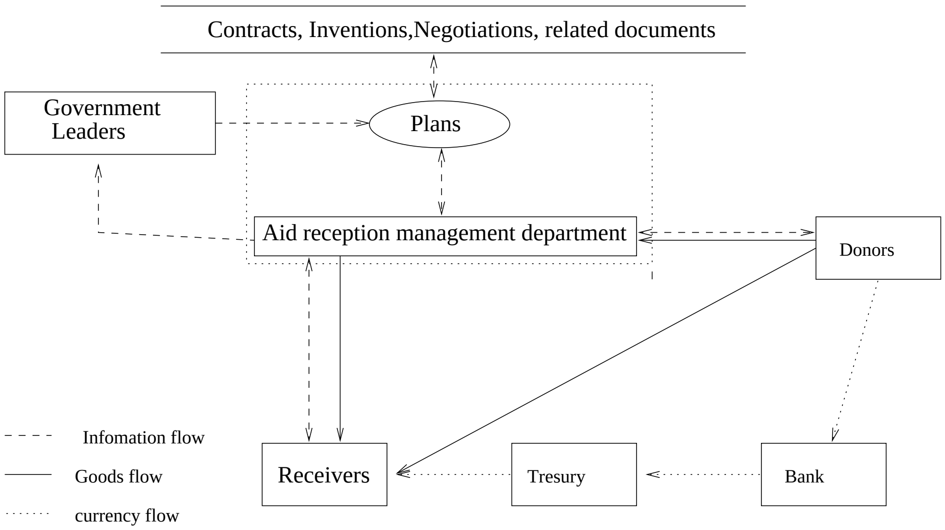

Figure 28: Data Flow of Aid Reception

1 Example The humane program of France[10] has an aid in order to upgrade the mine quarter of Quang Ninh province over two years. The aid plan is as follows

- Total aid amount : 20 millions FF
- Among this aid, there is 10 millions FF in order to buy new machines and train technical workers (attached a document in detail)
- 10 millions by goods including:

[10] this example is only imaginary

- 100 lorries
- 20 digging machines
- Other goods with attached list.

This plan has a detailed reception schedule. Based on this plan, The MoF makes the receipt account and periodic report of the reception situation. Composed with other aids, the MoF builds the periodic synthetic report.

2 Requirement Below, we specify the account record for monitoring the situation of external aid reception and the basic functions to make periodic reports.

A statistical record of external aids is a list of aid by terms (types of aids). Each aid by term will be called an aid term (abbreviated by aid); the example above is an aid term.

Based on the schedule of the plan account and the record of the reception account we describe the basic functions in order to build the synthetic and statistical report to the Leader and Government organisations.

For making the reports, we need the following information.

1. For every aid and a given period:
   - A detailed account record of the reception situation for each period.
   - A periodic sum for each kind of goods or money given and value equivalent (by Vietnamese Dong or USD).
   - Determining the state (planned, active or closed) of each aid at any given time.
   - The sum value and the kinds of good or money for each aid at any given time
2. For a record of aid list in each given period:
   - The kinds of goods or money received in any period.
   - The sum for each aid kind received in any period
   - The total sum value received in any period

6.2.2 Formal Model

Glossary *Transaction*: Record of reception of a kind of aid including receipt date, amount kind (kind name), amount, equivalent value and explanation ...
Date could be modelled as Nat (counting in day) but we choose to be more abstract. The date in the plan account is the due date.

Amount kind may be a tangible goods such as a lorry or a non tangible name for a group of goods or a currency name.
Amount is the money amount computed by Donor's currency (we will express by Nat).
Val indicates value equivalent of correspond amount of aid (this will be fixed either as Vietnamese Dong or USD).
Name: Name of each aid; often the name of the project or program including its key.
Explanation the negotiated target of the received aid including the code of any document attached.

1. **Aid Account** An aid account in a list of transactions. There are plan account and actual account but both are described the same:

```
scheme
  account =
    class
      type
        Amount = Nat,
        Amount_kind == USD | DM | _,
        Date,
        Transaction ::
          date : Date
          amount : Amount
          amount_kind : Amount_kind
          val : Nat
          explan : Explanation,
        Explanation,
        Account = {| trl : Transaction* • is_date_ordered(trl) |}

      value
        ≤ : Date × Date → Bool,

        is_date_ordered : Transaction* → Bool
        is_date_ordered(trl) ≡
          (∀ i1, i2 : Nat • {i1, i2} ⊆ inds trl ∧ i1 < i2 ⇒ date(trl(i1)) ≤ date(trl(i2))),

        account_sum : Account × Amount_kind → Nat
        account_sum(ac, ak) ≡
          case ac of
            ⟨⟩ → 0,
            ⟨tr⟩ ^ t →
              let a = account_sum(t, ak) in
                if amount_kind(tr) = ak then a + amount(tr) else a end
              end
          end,
```

val_sum : Account → **Nat**
val_sum(trl) ≡ **case** trl **of** $\langle\rangle \to 0$, $\langle \text{tr} \rangle$ ^ a → val(tr) + val_sum(a) **end**,

situa_account : Account × Date × Date → Account
situa_account(trl, d1, d2) ≡ ⟨ tr | tr **in** trl • d1 ≤ date(tr) ∧ date(tr) ≤ d2 ⟩,

head : Account × Date → Account
head(trl, d) ≡ ⟨ tr | tr **in** trl • date(tr) ≤ d ⟩,

period : Account $\stackrel{\sim}{\to}$ Date × Date
period(trl) ≡ (date(trl(1)), date(trl(**len** trl))) **pre** trl ≠ ⟨⟩,

sum : Account × Date × Date → **Nat**
sum(a, d1, d2) ≡ val_sum(situa_account(a, d1, d2))
end

2. Aid Term An aid term composed of a plan account; an actual account and other information of aid.We need to determine the kinds of aid involved, the state of aid at each date, plan sum and actual sum during any period .

scheme
aid_term =
class
object A : account

type
infor :: receiver : Receiver donor : Donor aid_info : Aid_info summary : Summary,
Receiver,
Donor,
Aid_info,
Summary,
Name,
State == planned | active | closed,
Aid ::
name : Name Plan : {| a : A.Account • a ≠ ⟨⟩ |} actual : A.Account info : infor,
Aid_kind = A.Amount_kind-**set**

value
aid_kind : Aid → Aid_kind
aid_kind(a) ≡ { A.amount_kind(tr) | tr : A.Transaction • tr ∈ **elems** Plan(a) },

state : A.Date × Aid → State
state(d, a) ≡

```
      if A.head(actual(a), d) = ⟨⟩ then
        planned
      else
        if
          ∀ b : A.Amount_kind •
            A.account_sum(Plan(a), b) = A.account_sum(A.head(actual(a), d), b)
        then
          closed
        else
          active
        end
      end,

    p_sum : Aid × A.Date × A.Date ~→ Nat
    p_sum(a, d1, d2) ≡ A.sum(Plan(a), d1, d2),

    actual_sum : Aid × A.Date × A.Date ~→ Nat
    actual_sum(a, d1, d2) ≡ A.sum(actual(a), d1, d2)
  end
```

3. Aids Aids is a list of aid terms with no duplicate names. For an aids, we determine the kind set, the sum for each aid kind and value sum for all the aids in any period.

```
scheme
  aids =
    class
      object A : aid_term

      type Aids = Aid_list,  Aid_list = {| al : A.Aid* • no_duplicate_name(al) |}

      value
        no_duplicate_name : A.Aid* → Bool
        no_duplicate_name(al) ≡
          (∀ i1, i2 : Nat • {i1, i2} ⊆ inds al ∧ i1 ≠ i2 ⇒ A.name(al(i1)) ≠ A.name(al(i2))),

        aids_kind : Aids → A.Aid_kind
        aids_kind(a) ≡ case a of ⟨⟩ → {}, ⟨b⟩ ^ t → A.aid_kind(b) ∪ aids_kind(t) end,

        sum_S : Aids × A.A.Amount_kind × A.A.Date → Nat
        sum_S(a, ak, d) ≡
          case a of
            ⟨⟩ → 0,
            ⟨ad⟩ ^ t → A.A.account_sum(A.A.head(A.actual(ad), d), ak) + sum_S(t, ak, d)
          end,
```

$$
\begin{array}{l}
\quad \text{val\_sum\_S} : \text{Aids} \times \text{A.A.Date} \times \text{A.A.Date} \rightarrow \textbf{Nat} \\
\quad \text{val\_sum\_S(a, d1, d2)} \equiv \\
\quad\quad \textbf{case } \text{a } \textbf{of } \langle\rangle \rightarrow 0,\ \langle \text{ad} \rangle \frown \text{t} \rightarrow \text{A.actual\_sum(ad, d1, d2)} + \text{val\_sum\_S(t, d1, d2)} \textbf{ end}, \\
\\
\quad \text{Close\_aid} : \text{A.A.Date} \times \text{Aids} \rightarrow \textbf{Nat} \\
\quad \text{Close\_aid(d, a)} \equiv \\
\quad\quad \textbf{card } \{ \text{ aid} \mid \text{aid : A.Aid} \bullet \text{aid} \in \textbf{elems} \text{ a} \wedge \text{A.state(d, aid)} = \text{A.closed } \} \\
\textbf{end}
\end{array}
$$

6.3 External debt

6.3.1 Narrative

By law, external debt is only used for developing and investing. Because the payment capacity of a developing country is limited, to borrow a debt is not easy. Based on the demand of investment and development in the country, the MoF estimates annual plans or long term plans of borrowing. In order to get a debt, the MoF must prepare a project with the aims in detail. Via negotiation with the Lender, they obtain a plan of receiving and amortising for the debt or debt term.

Based on plans, the MoF realizes borrowing and paying for all of state debts and also manages the state foreign currency source. The debts may be invested or re-lent.

The data flow of debt management can be described in figure 29.

1 Example We consider the following example [11]

A loan by Japan over 3 years to develop agriculture in the north delta of Vietnam. Total loan is 10 billion yen, which is disposed as follows:

- 2 billion yen by agriculture equipment (With a list attached in detail)
- 2 billion yen by chemical material (With a list attached in detail)
- The rest by yen (original money)

The debt will be amortised by crude oil or by money. A schedule for receiving and amortising has been negotiated and approved.

[11] This example is imaginary

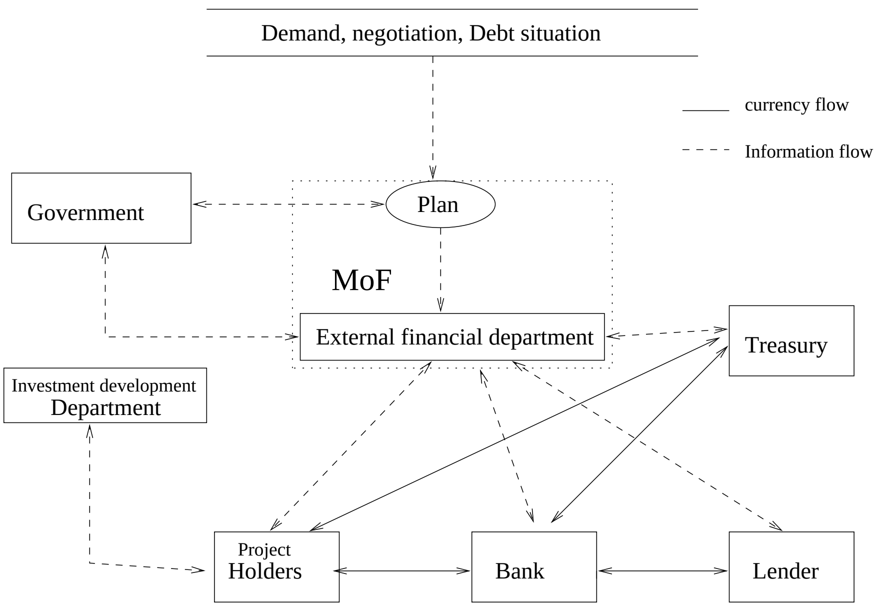

Figure 29: Data Flow of Borrowing and Amortising External Debts

There is no interest for the debt by goods; the money interest is computed by simple interest.

The goods received will be distributed to the project holders as original money. the money will be managed in the foreign budget system. Our information system must reflect the debt situation at each date and in each period.

2 Requirement In this section, we describe the following features which is only a development of [7] (therefore we omit some details)

- A record for each external debt and a list of debts.
- Present basic functions in order to build the periodic reports

. Differing from the aid reception, each transaction compounds a transaction kind (receipt, amortise—,etc.). The interest is under-specified. In order to build the reports, we specify the following information.

1. For every debt and a given period:
    - The detailed accounts (for both actual or in plan) of reception situation (receive account), amortisation situation (amortise account) and extract account in the period.

- Sums of original money and equivalent value which are received or amortised in the period (both actual or committed)
- The state and balance of the debt at any date

2. For a debt list and a given period:

- The equivalent sum for the received amount and the amortised amount in any period (both for actual and committed)
- The quantity of debt for each state at any date.

6.3.2 Model

1. Debt_account Compared with an aid account, for each debt account we have to compute interest, balance and necessary functions.

```
scheme
  debt_account =
    class
      type
        Amount = Nat,
        Date,
        Amount_kind = Goods_kind | Money_kind,
        Goods_kind,
        Money_kind == DM | F | USD | _,
        Explanation,
        Transaction ::
          date : Date
          kind : Transaction_kind
          amount_kind : Amount_kind
          amount : Amount
          quantity : Nat
          explan : Explanation,
        Transaction_kind == receipt | amortise | _,
        Account = {| trl : Transaction* • is_date_ordered(trl) |}

      value
        ≤ : Date × Date → Bool,

        val : Transaction → Nat,

        amortise_sum : Account × Date × Date → Nat × Nat
        amortise_sum(trl, d1, d2) ≡
          case trl of
            ⟨⟩ → (0, 0),
```

$\langle$tr$\rangle$ ^ t $\rightarrow$
let (a, b) = amortise_sum(t, d1, d2) **in**
if kind(tr) = amortise $\wedge$ d1 $\leq$ date(tr) $\wedge$ date(tr) $\leq$ d2 **then**
(amount(tr) + a, val(tr) + b)
else
(a, b)
end
end
end,

is_date_ordered : Transaction* $\rightarrow$ **Bool**
is_date_ordered(trl) $\equiv$
($\forall$ i1, i2 : **Nat** • {i1, i2} $\subseteq$ **inds** trl $\wedge$ i1 < i2 $\Rightarrow$ date(trl(i1)) $\leq$ date(trl(i2))),

interest : Account $\times$ Date $\rightarrow$ **Nat**,

receive_sum : Account $\times$ Date $\times$ Date $\rightarrow$ **Nat** $\times$ **Nat**
receive_sum(trl, d1, d2) $\equiv$
case trl **of**
$\langle\rangle \rightarrow$ (0, 0),
$\langle$tr$\rangle$ ^ t $\rightarrow$
let (a, b) = receive_sum(t, d1, d2) **in**
if kind(tr) = receipt $\wedge$ d1 $\leq$ date(tr) $\wedge$ date(tr) $\leq$ d2 **then**
(amount(tr) + a, val(tr) + b)
else
(a, b)
end
end
end,

balance : Account $\times$ Date $\rightarrow$ **Int**
balance(trl, d) $\equiv$
let (r1, r2) = receive_sum(trl, date(trl(1)), d) **in**
let (p1, p2) = amortise_sum(trl, date(trl(1)), d) **in** p1 − r1 − interest(trl, d) **end**
end,

period : Account $\rightarrow$ Date $\times$ Date
period(trl) $\equiv$ (date(trl(1)), date(trl(**len** trl))) **pre** trl $\neq \langle\rangle$,

goods_receive_sum : Account $\times$ Goods_kind $\times$ Date $\times$ Date $\rightarrow$ **Nat** $\times$ **Nat**
goods_receive_sum(trl, g, d1, d2) $\equiv$
case trl **of**
$\langle\rangle \rightarrow$ (0, 0),
$\langle$tr$\rangle$ ^ t $\rightarrow$
let (a, b) = goods_receive_sum(t, g, d1, d2) **in**

```
          if
            amount_kind(tr) = g ∧ kind(tr) = receipt ∧ d1 ≤ date(tr) ∧ date(tr) ≤ d2
          then
            (amount(tr) + a, quantity(tr) + b)
          else
            (a, b)
          end
        end
      end,

    goods_amortise_sum : Account × Goods_kind × Date × Date → Nat × Nat
    goods_amortise_sum(trl, g, d1, d2) ≡
      case trl of
        ⟨⟩ → (0, 0),
        ⟨tr⟩ ^ t →
          let (a, b) = goods_amortise_sum(t, g, d1, d2) in
          if
            amount_kind(tr) = g ∧ kind(tr) = amortise ∧ d1 ≤ date(tr) ∧ date(tr) ≤ d2
          then
            (amount(tr) + a, quantity(tr) + b)
          else
            (a, b)
          end
        end
      end,

    receive_account : Account × Date × Date → Account
    receive_account(trl, d1, d2) ≡
      ⟨ tr | tr in trl • kind(tr) = receipt ∧ d1 ≤ date(tr) ∧ date(tr) ≤ d2 ⟩,

    amortise_account : Account × Date × Date → Account
    amortise_account(trl, d1, d2) ≡
      ⟨ tr | tr in trl • kind(tr) = amortise ∧ d1 ≤ date(tr) ∧ date(tr) ≤ d2 ⟩,

    extract : Account × Date × Date → Account
    extract(trl, d1, d2) ≡ ⟨ tr | tr in trl • d1 ≤ date(tr) ∧ date(tr) ≤ d2 ⟩ pre d1 ≤ d2,

    head : Account × Date → Account
    head(trl, d) ≡ if date(trl(1)) ≤ d then extract(trl, date(trl(1)), d) else ⟨⟩ end
  end
```

2. **Debt_term** A debt_term is composed of a plan debt account; an actual debt account and other information of debt. For each Dept, we define the necessary functions.

```
scheme
  debt_term =
    class
      object A : debt_account

      type
        Debt ::
          name : Name
          money_kind : A.Money_kind
          plan : {| a : A.Account • a ≠ ⟨⟩ |}
          actual : A.Account
          info : Infor,
        Infor :: lender : Lender  summary : Summary  deb_infor : Deb_infor,
        State == planned | active | closed,
        Name,
        Summary,
        Lender,
        Deb_infor

      value
        actual_receive : Debt × A.Date × A.Date → A.Account
        actual_receive(deb, d1, d2) ≡ A.receive_account(actual(deb), d1, d2),

        actual_amortise : Debt × A.Date × A.Date → A.Account
        actual_amortise(deb, d1, d2) ≡ A.amortise_account(actual(deb), d1, d2),

        actual_receive_sum : Debt × A.Date × A.Date → Nat × Nat
        actual_receive_sum(deb, d1, d2) ≡ A.receive_sum(actual(deb), d1, d2),

        actual_amortise_sum : Debt × A.Date × A.Date → Nat × Nat
        actual_amortise_sum(deb, d1, d2) ≡ A.amortise_sum(actual(deb), d1, d2),

        actual_balance : Debt × A.Date → Int
        actual_balance(deb, d) ≡ A.balance(actual(deb), d),

        state : Debt × A.Date → State
        state(deb, d) ≡
          if A.head(actual(deb), d) = ⟨⟩ then
            planned
          else
            if actual_balance(deb, d) = 0 then closed else active end
          end,

        commited_sum : Debt × A.Date × A.Date → Nat × Nat
        commited_sum(deb, d1, d2) ≡ A.receive_sum(plan(deb), d1, d2),
```

```
    need_setle_sum : Debt × A.Date × A.Date → Nat × Nat
    need_setle_sum(deb, d1, d2) ≡ A.amortise_sum(plan(deb), d1, d2)
  end
```

3. Debts Debts is a list of debts (debt terms) with no duplicate name. For the given Debts, we define the necessary functions.

```
scheme
  debts =
    class
      object A : debt_term

      type Debts = Debt_list,  Debt_list = {| deb : A.Debt* • no_duplicate_name(deb) |}

      value
        no_duplicate_name : A.Debt* → Bool
        no_duplicate_name(deb) ≡
          (
            ∀ i1, i2 : Nat •
              {i1, i2} ⊆ inds deb ∧ i1 ≠ i2 ⇒ A.name(deb(i1)) ≠ A.name(deb(i2))
          ),

        actual_receive_sum : Debts × A.A.Date × A.A.Date → Nat
        actual_receive_sum(debl, d1, d2) ≡
          case debl of
            ⟨⟩ → 0,
            ⟨deb⟩ ^ t →
              let (a, val_a) = A.actual_receive_sum(deb, d1, d2) in
                let b = actual_receive_sum(t, d1, d2) in val_a + b end
              end
          end,

        actual_amortise_sum : Debts × A.A.Date × A.A.Date → Nat
        actual_amortise_sum(debl, d1, d2) ≡
          case debl of
            ⟨⟩ → 0,
            ⟨deb⟩ ^ t →
              let (a, val_a) = A.actual_amortise_sum(deb, d1, d2) in
                let b = actual_amortise_sum(t, d1, d2) in val_a + b end
              end
          end,

        quantity_closed : Debts × A.A.Date → Nat × A.Name*
```

```
quantity_closed(debl, d) ≡
  case debl of
    ⟨⟩ → (0, ⟨⟩),
    ⟨deb⟩ ⁀ t →
      let (n, names) = quantity_closed(t, d) in
        if A.state(deb, d) = A.closed then
          (n + 1, names ⁀ ⟨A.name(deb)⟩)
        else
          (n, names)
        end
      end
  end,

quantity_planned : Debts × A.A.Date → Nat × A.Name*
quantity_planned(debl, d) ≡
  case debl of
    ⟨⟩ → (0, ⟨⟩),
    ⟨deb⟩ ⁀ t →
      let (n, names) = quantity_planned(t, d) in
        if A.state(deb, d) = A.planned then
          (n + 1, names ⁀ ⟨A.name(deb)⟩)
        else
          (n, names)
        end
      end
  end,

quantity_active : Debts × A.A.Date → Nat × A.Name*
quantity_active(debl, d) ≡
  case debl of
    ⟨⟩ → (0, ⟨⟩),
    ⟨deb⟩ ⁀ t →
      let (n, names) = quantity_active(t, d) in
        if A.state(deb, d) = A.active then
          (n + 1, names ⁀ ⟨A.name(deb)⟩)
        else
          (n, names)
        end
      end
  end,

commited_sum : Debts × A.A.Date × A.A.Date → Nat
commited_sum(debl, d1, d2) ≡
  case debl of
    ⟨⟩ → 0,
```

```
        ⟨deb⟩ ⌢ t →
          let (a, val_a) = A.commited_sum(deb, d1, d2) in
            let b = commited_sum(t, d1, d2) in val_a + b end
          end
      end,

    need_setle_sum : Debts × A.A.Date × A.A.Date → Nat
    need_setle_sum(debl, d1, d2) ≡
      case debl of
        ⟨⟩ → 0,
        ⟨deb⟩ ⌢ t →
          let (a, val_a) = A.need_setle_sum(deb, d1, d2) in
            let b = need_setle_sum(t, d1, d2) in val_a + b end
          end
      end
  end
```

7 Combining and distributing separate subsystems

The financial system is a complex system which includes a number of subsystems and units. Each of these component system has its own functions and is managed and implemented independently. However, they all belong to the same "mother", have the same common task of financial management and, therefore, they need to communicate with each other.

In this section, we sketch the structure of the financial system and suggest its implementation as a distributed system communicating by messages[12].

7.1 Narrative

In the horizontal view, the financial system is divided corresponding to the administration levels of the various government professional offices. In each locality, there's a "small" financial system including several departments, each of them with specific financial functions. The departments at different levels, which have been determined to perform the same functions, are combined into professional subsystems. So, with the vertical view, the financial system can be considered as set of subsystems.

The way of viewing the system into professional subsystems is appropriate especially for analysing the information flow. Each subsystem has its own functions, data types and specific procedures of management. The different levels of the same subsystem have close relations. They may be

[12]This section is quoted from [9]

considered as a tree, with branches representing communication: the up direction is the reporting of information from a node to its upper unit, the down one is management and requesting of information.

Considered in the financial context, all the subsystems are tightly connected. The connections are not only at the top level: the components of the subsystems themselves need to communicate. For example, in a particular province, the treasury and tax departments need to exchange information to manage the taxpayers, or the budget department may ask the collection amount of the tax department for planning expenditure schedules for the locality. Besides the subsystems, the communication also involves some other particular units. For example the financial policy department has connections to all the subsystems for transmitting controlling information and getting reports from them, or the external financial department has to communicate to cooperate with the internal one.

In other documents we have analysed the financial information system through its professional subsystems, including the tax system [4], treasury system [6], budget system [5]. We also considered two single independent units in the system: debt and aid management forming the external financial system [8].

For the communication system we need to consider:

connection: The components need to communicate to others inside the same subsystem and also to others in in different subsystems. So a particular component has to be "visible" from outside its subsystem. This may cause the subsystem to be restructured.

data: The financial system is a complex system with a number of professional functions, so it has data of many types. As being organised to perform particular task, a subsystem can be considered to have a homogeneous data type for all the components. There will be a need to exchange more heterogeneous data when we link different subsystems.

geography: Each component can be seen as an independent object as it performs a particular professional function in a particular locality. Some other components may be physically close, some may be distant.

technology: Only some components may be able to exchange data electronically with others, and this situation may change with time as equipment is installed.

other: in the communication field, we may also need to look into aspects such as security, integrity, reliability, etc.

So the problem may be summarised as *distributing* each previously hierarchical subsystem so its lower components are visible, and also *combining* the distributed subsystems so that components in different subsystems can communicate. Because of the problems of geography and technology, we introduce a message system that allows them to communicate asynchronously. We want to transform from a situation like that pictured in figure 30 to one like figure 31

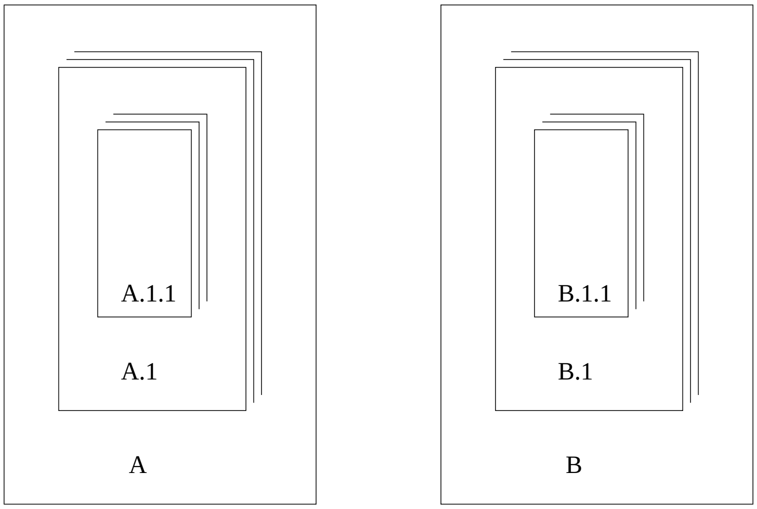

Figure 30: Separate hierarchical subsystems

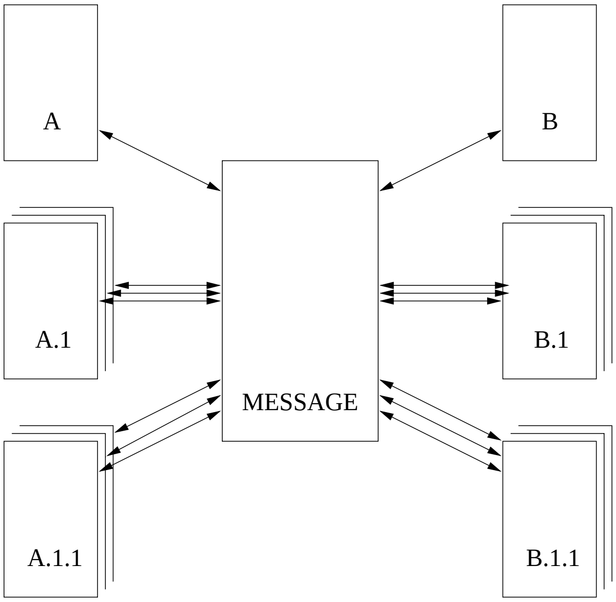

Figure 31: Distributed and combined subsystems

The message system can later be developed into a combination of a number of communication means between particular units. We will make no assumptions about this medium apart from no loss of messages, and expect particular protocols to be used at later stages of design to ensure this as far as possible. Such later developments would also handle the other problems like security.

An important aim in this transformation is to change the modules forming the hierarchical systems of taxation, budget etc. as little as possible and according to a standard method. We want to make the transformation reliable and predictable in its properties. This is especially the case because we will be working with concurrent modules (themselves obtained by the standard RAISE method [2] from applicative ones) and so we want very much to avoid doing proofs, which are hard with concurrent systems. We want to achieve "correctness by construction".

7.2 Formal Model

7.2.1 The message system

We will start by describing a generic message system. This will be parameterised by notions of *message* and *address* (of a component). We will also want to be able to identify responses to messages and we do this simply by numbering them. We will only need to make message numbers unique within a component; the combination of number and address will then uniquely identify a message globally.

A message with a zero number will be regarded as a *null* message, to be used to indicate failure in generating the response required, or unsuccessful communication.

```
scheme
  M_TYPES =
    class
      type
        Num = Nat,
        Address,
        Message_text,
        Message :: num : Num  sender : Address  text : Message_text

      value
        null : Message • is_null(null),

        is_null : Message → Bool
        is_null(m) ≡ num(m) = 0
    end
```

We define a MESSAGE system in terms of messages and addresses and a collection of buffers, one for each possible address. The buffers might be implemented as a collection of links between different offices, some electronic, some paper based.

```
scheme
```

```
MESSAGE(T : M_TYPES) =
  hide B in
    class
      object B[ a : T.Address ] : C_BUFFER(T{Message for Elem})

      value
        put : T.Address × T.Message → in any out any Unit
        put(a, m) ≡ B[ a ].put(m),

        get : T.Address → in any out any T.Message
        get(a) ≡ B[ a ].get(λ m : T.Message • true)
    end
```

See appendix for definition of C_BUFFER. It is possible to *get* a message satisfying a predicate, but the *get* in MESSAGE just gets the next message (returning *null* if there are none).

7.2.2 Additional modules

1. An in-tray We will supply each component with an "in-tray". This is a local buffer (another C_BUFFER) containing those messages for a component received from the message system. There are functions to put a message in the in-tray, to get the next one, and to get one with a specific number (a response to one sent with that number).

```
scheme
  IN_TRAY(T : M_TYPES) =
    hide IN in
      class
        object IN : C_BUFFER(T{Message for Elem})

        value
          put : T.Message → in IN.any out IN.any Unit
          put(m) ≡ IN.put(m),

          get_next_message : Unit → in IN.any out IN.any T.Message
          get_next_message() ≡ IN.get(λ m : T.Message • true),

          /* needs protection by time−out or iteration limit */
          get_message_by_number : T.Num ~→ in IN.any out IN.any T.Message
          get_message_by_number(n) ≡
            local variable result : T.Message := T.null in
              while T.is_null(result) do
                result := IN.get(λ m : T.Message • T.num(m) = n)
```

```
            end ;
            result
          end,

        init : Unit → in IN.any out IN.any write IN.any Unit
        init() ≡ IN.main()
    end
```

The function *get_message_by_number* has a comment that it needs protection by a time-out or a limit on the number of times it will iterate. If an expected message with the number being looked for has not yet arrived then, as written, it will loop indefinitely. In the final implementation extra code will be needed to avoid this, and it can indicate failure by returning a *null* message. Leaving this to final coding avoids trying to impose a particular strategy now. This must be done with care, because, for example, one could prove from the current definition that if it terminates the result cannot be null, while in practice it might be. Apart from a time-out or limit the loop itself would also be given a delay. This could be specified abstractly but in effect would be little more than a comment.

2. Message numbers The COUNTER module[13] generates message numbers for use when sending messages. Note that it can never return a zero message number (reserved for null messages).

3. Message coding and decoding As we remarked earlier, a particular subsystem, the tax system, say, will have its own types for data: identifiers for tax payers, report formats, etc. To avoid making such types global we will make each system have its own code/decode functions to convert the global messages into ones understandable locally. So the module CODE will be instantiated consistently within the tax system, but might be instantiated differently for a different system.

This coding may be very simple, such as unparsing data into text and parsing it again, or it might involve encryption for security.

```
scheme
  CODE(T : M_TYPES) =
    class
      type Message_text

      value
        code : Message_text → T.Message_text,
        decode : T.Message_text → Message_text
```

[13] see Appendix

```
    axiom [ decode_code ] ∀ mt : Message_text • decode(code(mt)) = mt
  end
```

4. A secretary We propose two versions of this module; either can be used by a particular component.

The first is a simple secretary whose task is simply to get messages from the message system and put them in the in-tray.

```
scheme
  SIMPLE_SECRETARY
    (
      T : M_TYPES,
      IN : IN_TRAY(T),
      M : MESSAGE(T),
      A : class value address : T.Address end
    ) =
    class
      value
        /* needs protection by delay in loop after null message */
        secretary : Unit → in any out any Unit
        secretary() ≡
          while true do
            let m = M.get(A.address) in
              if ∼ T.is_null(m) then IN.put(m) end
            end
          end
    end
```

The second is a skilled secretary whose task is to look for messages that are, say, standard requests for information like information about a taxpayer or a periodic report, to deal with these and to reply, putting others into the in-tray. *can_handle* is defined to depend on the sender as well as the text so that authorization can be checked.

```
scheme
  SKILL_SEC_PARM(T : M_TYPES, C : CODE(T)) =
    class
      value
        address : T.Address,
        can_handle : C.Message_text × T.Address → Bool,
```

```
      handle : C.Message_text × T.Address ⥲ in any out any C.Message_text

    axiom
      ∀ mt : C.Message_text, a : T.Address •
        handle(mt, a) post true pre can_handle(mt, a)
  end

scheme
  SKILLED_SECRETARY
    (
      T : M_TYPES,
      IN : IN_TRAY(T),
      C : CODE(T),
      M : MESSAGE(T),
      A : SKILL_SEC_PARM(T, C)
    ) =
    class
      value
        /* needs protection by delay in loop after null message */
        secretary : Unit → in any out any Unit
        secretary() ≡
          while true do
            let m = M.get(A.address) in
              if ∼ T.is_null(m) then
                let mt = C.decode(T.text(m)) in
                  if A.can_handle(mt, T.sender(m)) then
                    let res = C.code(A.handle(mt, T.sender(m))) in
                      M.put
                        (T.sender(m), T.mk_Message(T.num(m), A.address, res))
                    end
                  else
                    IN.put(m)
                  end
                end
              end
            end
          end
    end
```

Either secretary could be implemented in software or be performed manually. In the second case the specification acts as the assumptions about this manual part of the component.

It would have been possible to define a simple secretary as a skilled one for whom *can_handle* always returns false.

7.2.3 Adapting existing components

Much of the distributed system can be built with the existing modules from the hierarchical system together with developments of the additional generic modules described in the preceding section. But some adaptation is required. We will use the taxation system from [4] as an example.

1. Developing generic modules First we need to develop M_TYPES with the global notions particular to our system: in particular the subsystems involved and the definition of address. The module M (see appendix A) used in the budget system supplies locality (province, district and commune) addresses and we use it as the basis for the addresses in FIS_TYPES:

scheme
  FIS_TYPES =
    class
      type
        Num = **Nat**,
        Message_text,
        Message :: num : Num sender : Address text : Message_text,
        System == tax | treasury | budget | loans | aid,
        Address′ :: system : System locality : M.Address,
        Address = {| a : Address′ • is_address(a) |}

      value
        null : Message • is_null(null),

        is_null : Message → **Bool**
        is_null(m) ≡ num(m) = 0,

        is_address : Address′ → **Bool**
        is_address(a) ≡
          case locality(a) **of**
            M.general_address → **true**,
            M.province_address(p) → system(a) ∉ {loans, aid},
            M.district_address(p, d) → system(a) ∉ {loans, aid},
            M.commune_address(p, d, c) → system(a) = treasury
          end
    end

It is immediate that FIS_TYPES implements M_TYPES and so can be used as an actual parameter when M_TYPES is the formal.

Secondly, for each subsystem, we need to define the *Message_text* type for that system by developing the CODE module. In particular, we will need to allow for the functions *can_handle* and *handle* in SKILL_SEC_PARM to be defined.

For example, in the tax system we might want the secretary to handle the generation and despatch of reports, based on types defined in the tax system object RP:

```
scheme
  TAX_CODE(T : FIS_TYPES) =
    class
      type
        Message_text ==
          summarise_req(form : RP.Report_form) |
          summarise_res(report : RP.Report) |
          _

      value
        code : Message_text → T.Message_text,
        decode : T.Message_text → Message_text

      axiom [ decode_code ] ∀ mt : Message_text • decode(code(mt)) = mt
    end
```

In fact any global type objects like RP and the tax TYPES module should be included in TAX_CODE or at least given names different from such global objects in other systems. This is a little annoying as it may mean some (object prefix) renaming; this is a disadvantage of using global type modules in systems that end up as larger systems! — if they had been parameters there would be no such problem now.

Modules TREASURY_CODE etc. are defined similarly.

2. Adapting existing modules For each subsystem we will need only to adapt the top-level module at each level in the hierarchy (general, province, etc.). We will take the provincial tax department module for the tax system, CC_PTD, as an example. Here is the original:

```
scheme
  CC_PTD(RG : CC_REGIME) =
    hide PL, GR, DTDs, DTDids, CH, add_plan, change_form in
      class
        object
          PL : CC_PTD_PLAN,
          GR : CC_GROUP(RG),
```

```
  DTDs[ id : {| id : M.Did • id ∈ DTDids |} ] : CC_DTD(RG),
  CH :
    class channel mk_report : RP.Report_form,  report : RP.Report end

value
  DTDids : M.Did-set,

  mk_report : RP.Report_form → in CH.report out CH.mk_report RP.Report
  mk_report(rf) ≡ CH.mk_report ! rf ; CH.report?,

  init : Unit → write any in any out any Unit
  init() ≡
    ‖ { DTDs[ id ].init() | id : M.Did • id ∈ DTDids } ‖ PTD()
    ‖
    GR.init(),

  PTD : Unit → read any in any out any Unit
  PTD() ≡
    local
      channel done : Unit,  report : RP.Report

      variable temp : RP.Report

      value
        put_report : RP.Report_form → read any in any out any Unit
        put_report(rf) ≡
          (
            ‖
            { report ! DTDs[ id ].mk_report(RP.lower_level(rf)) |
              id : M.Did • id ∈ DTDids
            }
            ‖
            report !
              change_form
                (GR.mk_report(RP.lower_level(RP.lower_level(rf))))
          ) ;
          done ! (),
        collect_report : Unit → write temp in any Unit
        collect_report() ≡
          local variable finished : Bool := false in
            while ∼ finished do
              let rp = report? in temp := RP.summary(rp, temp) end
              ⌷
              done? ; finished := true
            end
```

```
            end
      in
        while true do
          let rf = CH.mk_report? in
            if RP.level(rf) = RP.PTD then
              temp := RP.empty(rf) ;
              (put_report(rf) ‖ collect_report()) ;
              CH.report ! add_plan(temp)
            else
              CH.report ! RP.empty(rf)
            end
          end
        end
      end,

    add_plan : RP.Report → in any out any RP.Report
    add_plan(rp) as rp′
      post RP.level(RP.form(rp′)) = RP.PTD
      pre RP.level(RP.form(rp)) = RP.DTD,

    change_form : RP.Report → in any out any RP.Report
    change_form(rp) as rp′
      post RP.level(RP.form(rp′)) = RP.DTD
      pre RP.level(RP.form(rp)) = RP.group
  end
```

We will call the adapted module DIST_PTD.

We need to do a number of things:

Add parameters The global module FIS_TYPES, the message system MESSAGE and, for the tax system, TAX_CODE need to be added as parameters.

```
scheme
  DIST_PTD
   (RG : CC_REGIME, F : FIS_TYPES,
    MSG : MESSAGE(F), C : TAX_CODE) = ...
```

Add a "self" address as a constant:

```
value
  self : F.Address
```

Add a counter and an in-tray :

```
object
  CTR : COUNTER(F),
  IN : IN_TRAY(F)
```

Replace the lower level object array The array of module CC_DTD, used to represent the district tax offices belonging to this province office will become a separate array at the top level and must be removed. The calls of functions within this array must be replaced by sending messages to the district tax offices. A convenient way to do this that leaves the rest of the province description unchanged is to define a "stub" module STUB_DTD that provides all the functions in CC_DTD used by CC_PTD (except *init*):

```
scheme
  STUB_DTD
    (
      F : FIS_TYPES,
      C : TAX_CODE(F),
      CTR : COUNTER(F),
      IN : IN_TRAY(F),
      MSG : MESSAGE(F)
    ) =
    class
      value
        province : F.Address,

        district : M.Did,

        mk_report : RP.Report_form → in any out any RP.Report
        mk_report(rf) ≡
          let
            sys = F.system(province),
            pid = M.pid(F.locality(province)),
            dest = F.mk_Address′(sys, M.district_address(pid, district)),
            num = CTR.get()
          in
            MSG.put
              (
                dest, F.mk_Message(num, province, C.code(C.summarise_req(rf)))
              ) ;
            let res = IN.get_message_by_number(num) in
              if F.is_null(res) then
                RP.empty(rf)
              else
```

```
                C.report(C.decode(F.text(res)))
              end
            end
          end
    end
```

Then STUB_DTD is used to make a replacement object array:

```
object
  DTDs[ id : {| id : M.Did • id ∈ DTDids |} ] :
    extend STUB_DTD(F,C,CTR,IN,MSG) with
    class
      axiom province = self, district = id
    end
```

Define can_handle and handle (renamed to avoid name clashes in the next step)

```
value
  prov_can_handle : C.Message_text × F.Address → Bool
  prov_can_handle(mt,a) ≡
    a = F.mk_Address′(F.system(self), M.up(F.locality(self))) ∧
    case mt of
      C.summarise_req(_) → true,
      _ → false
    end,

  prov_handle : C.Message_text × F.Address ~→ in any out any C.Message_text
  prov_handle(mt,a) ≡  C.summarise_res(mk_report(C.form(mt)))
  pre prov_can_handle(mt,a)
```

Add an object instantiating SKILL_SEC_PARM :

```
object
  P : class
    value
      address : F.Address = self,
      can_handle : C.Message_text × F.Address → Bool = prov_can_handle,
      handle : C.Message_text × F.Address ~→ in any out any C.Message_text
        = prov_handle
  end
```

Add an object instantiating SKILLED_SECRETARY :

object SEC : SKILLED_SECRETARY(F, IN, C, MSG, P)

Add the new servers to *init* This is the only function from CC_PTD that we need to change! We need to add the *secretary* and in-tray *init* function and remove the district *init* functions.

value
  init : **Unit** → **write any in any out any Unit**
  init() ≡ PTD() ∥ GR.init() ∥ SEC.secretary() ∥ IN.init()

Hide the new objects CTR, IN, P, and SEC

Most of this is very routine. The only real work is in deciding what messages can be handled automatically and writing the corresponding parts of the *handle* function. There will also be a need to write functions for sending or dealing with messages to or from other subsystems, whether handled automatically or not. These are inevitably new - they could not previously be specified within one subsystem. But the basic functions to support them - to receive a payment of tax, for example, will probably already exist. We only need to add the code to call the function in response to receiving the payment message from the treasury system and to acknowledge it.

7.2.4 The complete system

We generate new general and district tax office modules in the same way as we generated the new province tax office module in the previous section. Then we deal with the treasury system, the budget system, the loan and aid systems (the last two having only the general level).

The complete system then consists of a number of objects:

object
  /* global types */
  F : FIS_TYPES,
  /* message system */
  MSG : MESSAGE(F),
  /* the tax system */
  TXRG : CC_REGIME,
  TXC : TAX_CODE(F),
  TXG :
    extend DIST_GTD(TXRG, F, MSG, TXC) **with**
      class
        axiom

```
        self = F.mk_Address′(F.tax, M.general_address),
        DTPids = M.provinces()
    end
TXP[ pid : M.Province ] :
  extend DIST_PTD(TXRG, F, MSG, TXC) with
    class
      axiom
        self = F.mk_Address′(F.tax, M.province_address(pid)),
        DTDids = M.districts(pid)
    end
TXD[ (pid,did) : M.District ] :
  extend DIST_DTD(TXRG, F, MSG, TXC) with ...
/* the treasury system */
...
/* the budget system */
...
...
```

7.3 Conclusion

We have demonstrated that it is possible to take systems that are hierarchical, where lower levels are only accessible through higher levels, and distribute them into systems where the principal lower level modules (in this case those representing local offices within a professional system, such as the tax system) are accessible via asynchronous message passing.

The same technique can be applied to a family of systems, hierarchical to different degrees, sharing the same communication system, so that communication between arbitrary offices in different systems is possible.

The method relies on

- a number of standard generic modules like IN_TRAY and SECRETARY;
- a prescribed method involving minimal changes to the existing hierarchical modules
- the standard technique in RAISE for developing concurrent modules from applicative ones involving "server" processes and "interface functions" that allows concurrent invocation of the interface functions while ensuring lack of deadlock.

Such a method is amenable to quality control and should be reliable in practice. Attempting to prove formally that, for example, a report generated from the top of the tax system by summarising reports from lower levels will be the same in the distributed system (assuming no loss of messages) as in the hierarchical one would be prohibitively long. But there is instead a convincing argument of “correctness by construction”.

Creating the distributed system from a number of subsystems does mean that we need to ensure some uniformity of data types. The addressing system, in particular, must be common. There is also the need to agree data interchange types. We have made this as local as possible, so that types of data only transferred within the tax system, for example, do not need to be shared by other systems even though, in principle (but not in practice), messages containing such data could be sent to them. They would have no means of “decoding” such messages. Different subsystems will need to agree on formats for messages passed between them, but this is hardly surprising.

8 Conclusion

8.1 Implementation

As we have described above, FIS has a complicate domain characteristics composing a huge information volume. It is:

- Hierarchical
- Static and dynamic
- One-dimension and multi-dimension
- Tangible and intangible

The specifications presented here are very rigorous. Before any system or subsystem were implemented based on this specifications, we would need to regard financial policy and hardware available.

8.2 Future work

This document rigorously describes the Financial Information System, which would apply elsewhere, in other countries. In order to develop FIS to a management support system we need design a lot of tool for forecasting and optimally distributing state budget.

References

[1] The RAISE Language Group. *The RAISE Specification Language.* BCS Practitioner Series. Prentice Hall International, 1992.

[2] The RAISE Method Group. *The RAISE Development Method.* BCS Practitioner Series. Prentice Hall International, 1995.

[3] Do Tien Dung, Le Linh Chi, Nguyen Le Thu, Phung Phuong Nam, Tran Mai Lien, and Chris George. Developing a Financial Information System. Technical Report 81, UNU/IIST, P.O.Box 3058, Macau, September 1996. .

[4] Do Tien Dung, Le Linh Chi, Nguyen Le Thu, Phung Phuong Nam, and Tran Mai Lien. Taxation System: Narrative and Specification. Technical Report MoFIT/2, UNU/IIST, P.O. Box 3058 Macau, August 1996.

[5] Do Tien Dung. Budget System. Technical Report MoFIT/7, UNU/IIST, P.O. Box 3058 Macau, March 1997.

[6] Phung Phuong Nam and Chris George. Treasury System. Technical Report MoFIT/8, UNU/IIST, P.O. Box 3058 Macau, March 1997.

[7] Chris George. External Debt Management. Technical Report MoFIT/1, UNU/IIST, P.O. Box 3058 Macau, March 1996.

[8] Hoang Xuan Huan. External Financial Information System. Technical Report MoFIT/9, UNU/IIST, P.O. Box 3058 Macau, June 1997.

[9] Do Tien Dung and Chris George. Combining and Distributing the FIS. Technical Report MoFIT/10, UNU/IIST, P.O. Box 3058 Macau, June 1997.

A Other schemes

```
scheme
  AP_ACCOUNT(RG : AP_REGIME) =
    hide Queue, VL in
      class
        type
          Tran ==
            demand(T.Date, T.Period, T.Amount) |
            pre_demand(T.Date, T.Amount) |
            pay(T.Date, T.Period, T.Amount) |
            exemption(T.Date, T.Period, T.Amount) |
            over_settle(T.Date, T.Amount),
          Account =
            {| tr : Tran* •
              VL.date_ordered(tr) ∧ VL.over_amount(tr) > 0
            |},
          Queue = (T.Period × T.Amount)*,
          Bill ::
            debt : Debts  over_amount : T.Amount  new_demand : T.Amount,
          Debts = T.Period →m T.Amount,
          Receipt :: settlement : Settlements  over_settle : T.Amount,
          Settlements = T.Period →m T.Amount

        value
          demand :
            T.Date × T.Date × T.Period × T.Amount × Account × RG.Regime
            ~→
              Account × Bill
```

```
demand(d1, d2, p, a, ac, rg) ≡
  let o = VL.over_amount(ac), ds = VL.mk_debts(d1, ac, rg) in
    (
      if o = 0 then
        VL.add(demand(d2, p, a), ac)
      elsif a ≤ o then
        VL.add(pre_demand(d1, a), ac)
      else
        VL.add(demand(d2, p, a − o), VL.add(pre_demand(d1, o), ac))
      end,
      mk_Bill(if a ≤ o then (ds, o − a, 0) else (ds, 0, a − o) end)
    )
  end
  pre can_demand(d1, d2, ac),

pay :
  T.Date × T.Period × T.Amount × Account × RG.Regime ~→
    Account × Receipt
pay(d, p, a, ac, rg) ≡
  let
    q = VL.mk_pqueue(d, p, ac, rg),
    (a′, ac′, ss) = VL.pay(d, q, a, ac, [ ], rg)
  in
    if a′ > 0 then
      (VL.add(over_settle(d, a′), ac′), mk_Receipt(ss, a′))
    else
      (ac′, mk_Receipt(ss, 0))
    end
  end
  pre can_settle(d, ac),

exempt :
  T.Date × T.Period × T.Amount × Account × RG.Regime ~→
    Account × Receipt
exempt(d, p, a, ac, rg) ≡
  let
    q = VL.mk_equeue(d, p, ac, rg),
    (a′, ac′, ss) = VL.exempt(d, q, a, ac, [ ], rg)
  in
    if a′ > 0 then
      (VL.add(over_settle(d, a′), ac′), mk_Receipt(ss, a′))
    else
      (ac′, mk_Receipt(ss, 0))
    end
  end
```

```
    pre can_settle(d, ac),

  can_demand : T.Date × T.Date × Account → Bool
  can_demand(d1, d2, ac) ≡
    T.(≤)(VL.last_date(ac), d1) ∧ T.(≤)(d1, d2),

  can_settle : T.Date × Account → Bool
  can_settle(d, ac) ≡ T.(≤)(VL.last_date(ac), d)

object
  VL :
    class
      value
        add : Tran × Account → Account
        add(tr, ac) ≡
          if ac = ⟨⟩ then
            ⟨tr⟩
          else
            let tr′ = hd ac, ac′ = tl ac in
              if T.(≤)(date(tr′), date(tr)) then
                ⟨tr⟩ ^ ac
              else
                ⟨tr′⟩ ^ add(tr, ac′)
              end
            end
          end,

        mk_debts : T.Date × Account × RG.Regime → Debts
        mk_debts(d, ac, rg) ≡
          [ p ↦ a |
            p : T.Period, a : T.Amount •
              let o = owing(d, p, ac, rg) in o > 0.0 ∧ a = int o end
          ],

        over_amount : Account → Int
        over_amount(ac) ≡
          case ac of
            ⟨⟩ → 0,
            ⟨tr⟩ ^ t →
              case tr of
                over_settle(d′, a) → a,
                pre_demand(d′, a) → 0 − a,
                _ → 0
              end +
              over_amount(t)
```

```
    end,

  pay :
    T.Date × Queue × T.Amount × Account × Settlements ×
      RG.Regime
    →
      T.Amount × Account × Settlements
  pay(d, q, a, ac, ss, rg) ≡
    case q of
      ⟨⟩ → (a, ac, ss),
      ⟨(p, a′)⟩ ^ t →
        if a > a′ then
          pay
            (
              d,
              t,
              a − a′,
              add(pay(d, p, a′), ac),
              add_settlements(p, a′, ss),
              rg
            )
        else
          (0, add(pay(d, p, a), ac), add_settlements(p, a, ss))
        end
    end,

  exempt :
    T.Date × Queue × T.Amount × Account × Settlements ×
      RG.Regime
    →
      T.Amount × Account × Settlements
  exempt(d, q, a, ac, ss, rg) ≡
    case q of
      ⟨⟩ → (a, ac, ss),
      ⟨(p, a′)⟩ ^ t →
        if a > a′ then
          exempt
            (
              d,
              t,
              a − a′,
              add(exemption(d, p, a′), ac),
              add_settlements(p, a′, ss),
              rg
            )
```

```
      else
        (
          0,
          add(exemption(d, p, a), ac),
          add_settlements(p, a, ss)
        )
      end
  end,

add_settlements :
  T.Period × T.Amount × Settlements → Settlements
add_settlements(p, a, ss) ≡
  ss †
  if p ∈ dom ss then [ p ↦ ss(p) + a ] else [ p ↦ a ] end,

mk_pqueue : T.Date × T.Period × Account × RG.Regime → Queue
mk_pqueue(d, p, ac, rg) ≡
  let
    all_period =
      { p′ | p′ : T.Period • p′ < last_period(ac) },
    pq1 = mk_debt_queue(d, all_period, ac, rg),
    pq2 = mk_due_queue(d, {p}, ac),
    pq3 = mk_due_queue(d, all_period \ {p}, ac)
  in
    pq1 ⌢ pq2 ⌢ pq3
  end,

mk_equeue : T.Date × T.Period × Account × RG.Regime → Queue
mk_equeue(d, p, ac, rg) ≡
  let
    all_period =
      { p′ | p′ : T.Period • p′ < last_period(ac) },
    pq1 = mk_debt_queue(d, {p}, ac, rg),
    pq2 = mk_due_queue(d, {p}, ac),
    pq3 = mk_debt_queue(d, all_period \ {p}, ac, rg),
    pq4 = mk_due_queue(d, all_period \ {p}, ac)
  in
    pq1 ⌢ pq2 ⌢ pq3 ⌢ pq4
  end,

mk_debt_queue :
  T.Date × T.Period-set × Account × RG.Regime → Queue
mk_debt_queue(d, ps, ac, rg) ≡
  if ps = {} then
    ⟨⟩
```

```
  else
    let p = earliest_period(ps), o = owing(d, p, ac, rg) in
      if o > 0.0 then ⟨(p, int o)⟩ else ⟨⟩ end ^
      mk_debt_queue(d, ps \ {p}, ac, rg)
    end
  end,

earliest_period : T.Period-set ~→ T.Period
earliest_period(ps) as p
  post
    p ∈ ps ∧ (∀ p′ : T.Period • p′ ∈ ps ⇒ p < p′)
  pre ps ≠ {},

mk_due_queue : T.Date × T.Period-set × Account → Queue
mk_due_queue(d, ps, ac) ≡
  case ac of
    ⟨⟩ → ⟨⟩,
    ⟨demand(d′, p, a)⟩ ^ t →
      if ∼ T.(≤)(d′, d) ∧ p ∈ ps then
        ⟨(p, a)⟩
      else
        ⟨⟩
      end ^
      mk_due_queue(d, ps, t),
    _ → mk_due_queue(d, ps, tl (ac))
  end,

owing : T.Date × T.Period × Account × RG.Regime → Real
owing(d, p, ac, rg) ≡
  if ac = ⟨⟩ then
    0.0
  else
    let tr = hd ac, ac′ = tl ac, d′ = date(tr) in
      if ∼ T.(≤)(d′, d) then
        owing(d, p, ac′, rg)
      else
        add_interest
          (
            d′, d, extend_owing(owing(d′, p, ac′, rg), p, tr), rg
          )
      end
    end
  end,

extend_owing : Real × T.Period × Tran → Real
```

```
extend_owing(r, p, tr) ≡
  case tr of
    pay(d, p′, a) →
      if p ≠ p′ then
        r
      elsif a = int r then
        0.0
      else
        r − real a
      end,
    exemption(d, p′, a) →
      if p ≠ p′ then
        r
      elsif a = int r then
        0.0
      else
        r − real a
      end,
    demand(d, p′, a) → if p ≠ p′ then r else r + real a end,
    _ → r
  end,

add_interest : T.Date × T.Date × Real × RG.Regime → Real
add_interest(d1, d2, r, rg) ≡
  if T.(≤)(d2, d1) then
    r
  else
    let
      d2′ = T.previous(d2), rate = RG.penalty_rate(d2′, rg)
    in
      (1.0 + rate) ∗ add_interest(d1, d2′, r, rg)
    end
  end,

date_ordered : Tran* → Bool
date_ordered(trl) ≡
  case trl of
    ⟨⟩ → true,
    ⟨tr⟩ → true,
    ⟨tr1, tr2⟩ ^ trl′ →
      T.(≤)(date(tr2), date(tr1)) ∧
      date_ordered(⟨tr2⟩ ^ trl′)
  end,

last_date : Account → T.Date
```

```
        last_date(ac) ≡
          case ac of
            ⟨⟩ → T.start_date,
            add(tr, ac′) →
              case tr of
                demand(d, p, ac) → last_date(ac′), _ → date(tr)
              end
          end,

        last_period : Account → T.Period
        last_period(ac) ≡
          case ac of
            ⟨⟩ → T.start_period,
            ⟨tr⟩ ^ t →
              let p1 = period(tr), p2 = last_period(t) in
                if p1 < p2 then p2 else p1 end
              end
          end,

        period : Tran → T.Period
        period(tr) ≡
          case tr of
            demand(d, p, a) → p,
            pay(d, p, a) → p,
            exemption(d, p, a) → p,
            _ → T.start_period
          end,

        date : Tran → T.Date
        date(tr) ≡
          case tr of
            demand(d, p, a) → d,
            pre_demand(d, a) → d,
            pay(d, p, a) → d,
            exemption(d, p, a) → d,
            over_settle(d, a) → d
          end
      end
  end

scheme DATA = class type Data,  Key end

scheme
  AP_DATABASE
    (D : DATA) =
```

class
  type Database

  value
    /* generators */
    empty : Database,
    insert : D.Key × D.Data × Database → Database,
    remove : D.Key × Database → Database,
    /* observers */
    defined : D.Key × Database → **Bool**,
    lookup : D.Key × Database $\stackrel{\sim}{\to}$ D.Data

  axiom
    forall k : D.Key, db : Database •
      [defined_empty] defined(k, empty) ≡ **false**,

      [defined_insert]
        ∀ k′ : D.Key, d : D.Data •
          defined(k, insert(k′, d, db)) ≡ k = k′ ∨ defined(k, db),

      [defined_remove]
        ∀ k′ : D.Key •
          defined(k, remove(k′, db)) ≡ k ≠ k′ ∧ defined(k, db),

      [lookup_defined] lookup(k, db) **post true pre** defined(k, db),

      [lookup_insert]
        ∀ k′ : D.Key, d : D.Data •
          lookup(k, insert(k′, d, db)) ≡
            if k = k′ **then** d **else** lookup(k, db) **end**
            pre k = k′ ∨ defined(k, db),

      [lookup_remove]
        ∀ k′ : D.Key •
          lookup(k, remove(k′, db)) ≡ lookup(k, db)
            pre k ≠ k′ ∧ defined(k, db)
end

object
  M :
    class
      type Pid, Did, Cid, Level == center | province | district | commune

      value
        country : Pid $\underset{m}{\to}$ Did $\underset{m}{\to}$ Cid-**set**,

```
    provinces : Unit → Pid-set
    provinces() ≡ dom country,

    districts : Pid -~-> Did-set
    districts(pid) ≡ dom country(pid) pre pid ∈ provinces(),

    communes : Pid × Did -~-> Cid-set
    communes(pid, did) ≡
      country(pid)(did)
      pre pid ∈ provinces() ∧ did ∈ districts(pid)

  type
    Address′ ==
      general_address |
      province_address(pid : Pid) |
      district_address(dpid : Pid, did : Did) |
      commune_address(cpid : Pid, cdid : Did, cod : Cid),
    Address = {| a : Address′ • is_address(a) |}

  value
    is_address : Address′ → Bool
    is_address(a) ≡
      case a of
        general_address → true,
        province_address(p) → p ∈ provinces(),
        district_address(p, d) →
          p ∈ provinces() ∧ d ∈ districts(p),
        commune_address(p, d, c) →
          p ∈ provinces() ∧
          d ∈ districts(p) ∧ c ∈ communes(p, d)
      end,

    level : Address → Level
    level(a) ≡
      case a of
        general_address → center,
        province_address(_) → province,
        district_address(_, _) → district,
        commune_address(_, _, _) → commune
      end,

    /* true if first address is immediate descendant of second */
    < : Address × Address → Bool
    a1 < a2 ≡ a1 ≠ general_address ∧ up(a1) = a2,
```

```
      up : Address ~→ Address
      up(a) ≡
        case a of
          province_address(p) → general_address,
          district_address(p, d) → province_address(p),
          commune_address(p, d, c) → district_address(p, d)
        end
        pre a ≠ general_address
  end


scheme
  SUM_PARA =
    class
      type Office, Report, Para

      value
        sum_report : Report × Report → Report

      axiom
        [ commutative ]
          ∀ r1, r2 : Report • sum_report(r1, r2) = sum_report(r2, r1),

        [ associative ]
          ∀ r1, r2, r3 : Report •
            sum_report(sum_report(r1, r2), r3) =
            sum_report(r1, sum_report(r2, r3))
    end

scheme
  SUM(SP : SUM_PARA) =
    class
      value
        sum :
          (SP.Para × SP.Office → SP.Report) × SP.Report × SP.Para ×
            SP.Office-set
          →
            SP.Report
        sum(mk_report, empty, p, os) ≡
          if os = {} then
            empty
          else
            let o : SP.Office • o ∈ os in
              SP.sum_report
```

```
        (mk_report(p, o), sum(mk_report, empty, p, os \ {o}))
      end
    end
  end


scheme
 C_BUFFER
  (
   X :
    class
     type Elem

     value
      null : Elem • is_null(null),
      is_null : Elem → Bool
    end
  ) =
  hide V, split in
   class
    object
     V :
      class
       variable buff : X.Elem*

       channel put, get_res : X.Elem,  get : X.Elem → Bool
      end

    value
     main : Unit → in V.any out V.any write V.any Unit
     main() ≡
      while true do
       V.buff := V.buff ^ ⟨V.put?⟩
       ⌷
       let (el, e) = split(V.buff, V.get?) in
        V.buff := el ; V.get_res ! e
       end
      end,

     get : (X.Elem → Bool) → in V.any out V.any X.Elem
     get(f) ≡ V.get ! f ; V.get_res?,

     put : X.Elem → in V.any out V.any Unit
     put(e) ≡ V.put ! e,
```

```
      split : X.Elem* × (X.Elem → Bool) → X.Elem* × X.Elem
      split(el, f) ≡
        local
          variable
            before : X.Elem* := ⟨⟩,
            after : X.Elem* := el,
            result : X.Elem := X.null
        in
          while after ≠ ⟨⟩ ∧ X.is_null(result) do
            if f(hd after) then
              result := hd after
            else
              before := before ^ ⟨hd after⟩
            end ;
            after := tl after
          end ;
          (before ^ after, result)
        end
    end


scheme
  COUNTER(T : M_TYPES) =
    hide counter, count in
    class
      variable counter : T.Num := 0

      channel count : T.Num

      value
        main : Unit → out count write counter Unit
        main() ≡
          while true do counter := counter + 1 ; count ! counter end,

        get : Unit → in count T.Num
        get() ≡ count?
    end
```