SOFL : A Formal Engineering Methodology for Industrial Applications*

Shaoying Liu†, A. Jeff Offutt‡, Chris Ho-Stuart*, Yong Sun⋆, Mitsuru Ohba†
†Hiroshima City University, Japan
‡George Mason University, USA
*Queensland University of Technology, Australia
⋆The Queen's University of Belfast, UK

Abstract

Formal methods have yet to achieve wide industrial acceptance for several reasons. They are not well integrated into established industrial software processes, their application requires significant abstraction and mathematical skills, and existing tools do not satisfactorily support the entire formal software development process.

We have proposed a language called SOFL (Structured-Object-based-Formal Language) and a SOFL methodology for system development that attempts to address these problems using an integration of formal methods, structured methods and object-oriented methodology. Construction of a system uses structured methods in requirements analysis and specifications, and an object-based methodology during design and implementation stages, with formal methods applied throughout the development in a manner that best suits their capabilities.

This paper describes the SOFL methodology, which introduces some substantial changes from current formal methods practice. A comprehensive, practical case study of an actual industrial *Residential Suites Management System* illustrates how SOFL is used.

Keywords: Structured Methods, Object-Oriented Methodology, Formal Methods, Data Flow Diagrams, Formal Languages.

1 Introduction

Formal methods have not been used in industry largely because it is difficult to apply them in practical settings [1, 2]. There are a number of reasons for this. First, the application of formal methods requires high abstraction and mathematical skills to write specifications and conduct proofs, and to read and understand formal specifications and proofs, especially when they are very complex. A software engineer must make a significant commitment to learn and become proficient at the necessary skills. Second, existing formal methods do not offer usable and effective *methods* for use in well-established industrial software process. Many texts on formal methods focus on notation, but are not well suited for helping practitioners apply the method in a practical development process. With isolated exceptions, formal methods are still largely perceived as an academic invention divorced from real applications. A third problem is expense. Experience has shown that adding formal methods to a development process can incur significant additional costs, but that when formal methods are fully integrated

*Work is supported in part by the Ministry of Education of Japan under Joint Research Grant-in-Aid for International Scientific Research FM-ISEE (08044167) and by Hiroshima City University under Hiroshima City University Grant for Special Academic Research (International Studies) SCS-FM (A440).

into a development process and costs measured over the full life cycle, costs may actually decrease. However, introduction of radically new processes requires a very expensive initial outlay, which most companies cannot afford given the constraints of schedule, budget, and labor. Although the quality of their systems is important, most companies need to keep time as their primary priority to meet the demands of the market. This view is shared by other researchers in the formal methods community [3]. Finally, effective tool support is crucial for formal methods application, but existing tools are not able to support a complete formal software development process, although tools supporting the use of formal methods in limited areas are available [4, 5, 6].

To make formal methods more practical and acceptable in industry, some substantial changes must be made.

1.1 Adapting Formal Methods for Industry

This paper proposes changes to software process, notation, methodology, and support environments for constructing systems. Figure 1 illustrates changes in the software process. First, formal methods often assume that a formal specification of the system under development should be completed before it is implemented. This is impractical. Some requirements must be obtained and recorded in the specification before design and implementation, but others are better captured during design and/or implementation. For this reason, we divide user requirements into two parts, to be obtained in different stages. The first part is the user's *primary* functional requirements. It is important to have a functional specification reflecting complete primary requirements before designing the software because this serves as a contract between the developer and the user, and as a firm basis for subsequent development. For critical systems, requirements for critical properties such as safety and security are part of the primary functional requirements. Primary requirements should be consistent and unambiguous. The second part is *secondary* requirements for the system, such as its background tasks, non-critical functions, and some quality aspects. These may include the interface layout, usability, and efficiency. Secondary requirements can be captured during design and implementation with techniques such as prototyping.

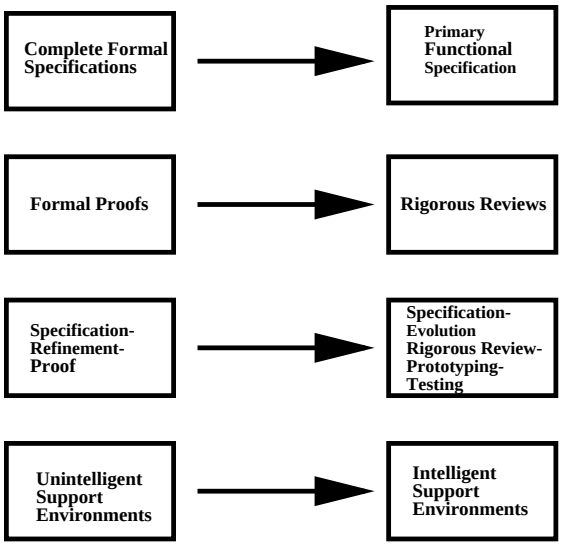

Figure 1: Changes in the software process

The second software process change follows from the observation that it is expensive and difficult to perform formal proofs. We suggest replacing formal proofs with a system of *rigorous reviews*.

The purpose of rigorous reviews is to ensure the internal consistency of specifications at different levels, to validate the specification against user requirements, and to ensure that designs or programs satisfy their requirements specifications or designs. Rigorous reviews should be based on formal proof principles, but must be easier to conduct. While most existing review methods tend to focus on error detection in design or programs, such as Parnas and Weiss' *Active design reviews* [7] and Fagan's *design and code inspections* [8], Knight's *phased inspection* tends to ensure that the product being inspected possesses required properties [9]. Rigorous reviews may not be as convincing as formal proofs for ensuring correctness, but a sound and practical review technique may be automatically supported, thereby reducing time and labor costs. A review should also take into account the risks and costs of failure; and either justify lack of full formal proof; or else advise that formal proof be taken for highly critical specifications. In any case, reviews should be backed up by rigorous testing.

The third process change concerns prototyping and testing activities. These should not be replaced by formal methods, but each should be used as appropriate, for appropriate purposes (more details on this point will be given later). An *evolutionary* approach is more practical than formal refinement for systems development [10], because the software development process is an engineering activity, rather than a purely mathematical process. Therefore, we need both mathematical and engineering technology to achieve quality and productivity. For example, formal specifications can serve as a foundation to generate test data [11, 12, 13].

Most existing tools and support environments can help reduce the workload of specification construction and formal verification, but are not able to guide the developer through the entire development. This may be because it is hard to give a theoretical foundation for construction of intelligent support environments, or because the formal methods are not sufficiently mature to be supported in that way. However, as formal methods are more difficult to use than informal methods, their acceptance requires intelligent support environments. There are many tools available to support some kinds of manipulation of formal specifications, but most are focused primarily on manipulating the notation rather than intelligent support of real software development processes. Some work has been done in this direction [14].

Initial steps towards tool development for SOFL have been made in our ongoing project FM-ISEE[1] [15]. The most difficult issue is how to develop a sufficient set of rules in a knowledge base for support environments to guide and to assist developers through systems development in general.

With regard to notation, purely mathematical notations do not scale well for large complex systems and are difficult for engineers to read and understand. We suggest that an appropriate graphical notation with precise semantics should be used as a formal notation to help model the high level architecture of systems, because it is more readable and can help indicate the higher levels of structure. Formal mathematical notation may then be used to define components of the system. Notation for connectives should be based on natural language rather than symbols (for example, using **and** rather than $\wedge$, **or** rather than $\vee$) to enhance readability. This also makes it easier to enter notation using the standard keyboard. Many existing formal notations (such as Z [16] or VDM-SL [17]) use mathematical symbols that cannot be directly entered using the standard keyboard. A support tool can alleviate this problem, but users may still find it inconvenient to have two different notations; one that can be entered from the standard keyboard and another for display of specifications. Natural language should also be a complement to formal notation in specifications to facilitate their interpretation. Effective formal specification requires a balanced mix of graphical notation, natural language, and mathematical notation, as indicated in Figure 2.

Formal methods alone are not sufficient to cope with the complexity of system development. An

[1] FM-ISEE stands for Formal Methods and Intelligent Software Engineering Environments. This project is an international collaboration among Hiroshima City University and Kyushu University of Japan, The Queen's University of Belfast of the U.K., George Mason University of the U.S.A., and Queensland University of Technology of Australia.

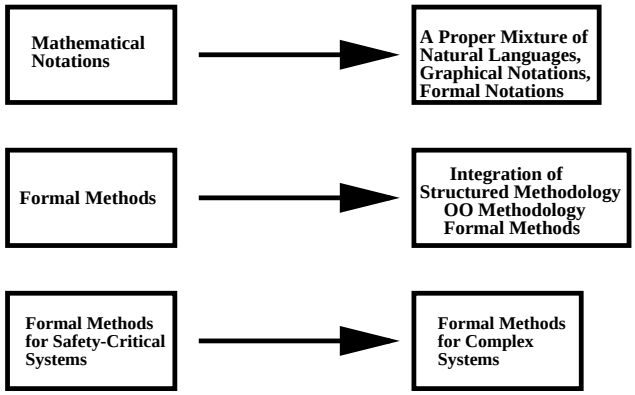

Figure 2: Changes in notation, methodology and application domain

appropriate integration of structured methods, object-oriented methodology, and formal methods may combine the advantages of those three approaches to improve the quality and efficiency of the software development process. Since this is one of the most important issues addressed in this paper, it is discussed in detail in section 1.2.

It is hoped that these changes will mean that formal methods do not need to be limited to safety-critical systems, but can be applied to other complex computer systems as well.

1.2 Integration of Methodologies

Two approaches have been adopted to integrate formal methods with structured methods. One is to use the Yourdon or the DeMarco approach to construct a data flow diagram and its associated data dictionary, and then to refine the data flow diagram into a formal specification by defining data flows and bottom level processes with the formal notations. The examples of this approach include Semmens and Allen's work on integrating Yourdon's method and Z [18], Bryant's work on Yourdon's method and Z [19], Fraser's work on data flow diagrams and VDM [20], and Plat and his colleagues' integration of data flow diagrams and VDM [21]. The other approach is to incorporate traditional data flow diagram notation into a formal specification language to provide a mechanism for structuring system specifications and a graphical view for the system specification. In this way, data flow diagrams are treated as part of formal specifications. The examples of this approach include Liu's work on designing the *Formal Requirements Specification Method* and the *Structured and Formal System Development Language* in which DeMarco data flow diagrams are combined with VDM [22, 23].

To integrate formal methods and object-oriented methodology, formal methods are usually used to specify the functionality of operations defined in classes. Examples of this approach include VDM++ [24] and Object-Oriented Z [25].

The above research attempts to make structured methods, object-oriented methodology and formal methods more usable. However, to the best of our knowledge, no previous effort has been made to integrate all three approaches. We believe that they are complementary approaches to systems development, and that all three should be integrated for maximum benefit.

To support the changes in formal methods and to overcome the deficiencies mentioned above, we propose a language called SOFL (Structured-Object-based-Formal Language) and the SOFL methodology. SOFL integrates structured methods based on extended and formalized data flow diagrams, object-oriented methodology, and VDM-SL formal notation. The motivation for SOFL is that existing

languages are not well suited to supporting our proposed changes to the software process, notation and methodology, and that existing approaches for integrating different methodologies do not give a consistent and systematic combination of the different kinds of notations that can be used to help understand, refine, verify, and implement specifications.

SOFL supports the use of structured methods for requirements analysis and specification in two ways. First, by providing appropriate abstraction and functional decomposition facilities. Second, by providing a usable mechanism for communication between the developer and the user to validate the specification. It can help a project team distribute tasks, increasing software productivity. On the other hand, when class hierarchies and inheritance are used appropriately, they increase information hiding and component re-usability.

This is different from the classical object-oriented approach in which data structures are designed to reflect real world objects. There are three difficulties with using an object-oriented approach for requirements analysis. First, it can be difficult to identify useful objects when the developer is not familiar with the application domain. Some graphical notations exist [26], but they only help to express classes, objects and their relations, not to identify objects and classes. Second, even if objects are identified in early stages, the specific demands from the rest of the system may still be unclear, so it may not be clear exactly what are appropriate methods and how they should be defined. Structured methods provide a way to help identify useful objects and their necessary methods by decomposing and evolving high level specifications to lower level specifications. Another difficulty concerns communication between the developer and the user. Users often prefer to think in terms of tasks rather than objects (especially if not well trained in object-oriented methodology).

SOFL uses structured methods for requirements analysis and specification and an *object-based* approach for design and implementation. During both the structured and object-based development of the system, formal methods can be applied to provide high quality specifications and verifications of various levels of the system. SOFL is intended to allow flexibility in the completeness of specifications, thereby allowing system developers to balance the benefits that can be obtained by using formal methods and the convenience that can be gained by using informal approaches (within the SOFL framework). For example, if developers cannot write a complete specification for a component (such as a process in a data flow diagram) at one level, they can partially specify the component by giving incomplete pre and/or postcondition (in an extreme case, both pre and postcondition of the component might be **true**) with details to be given more completely at a lower level of specification.

The application scope of SOFL proposed in this paper is limited to non-concurrent software. Although there are parallel constructs in the extended data flow diagrams used in SOFL, they are treated as a nondeterministic ordering rather than concurrency. Extensions to deal with concurrent and real-time systems development is under way in another joint project.

The major contributions of this paper are:

1. Establish the SOFL methodology based on changes to software process, notation and methodology, with guidelines for developing a software system from the beginning.
2. Define and improve the SOFL language from its initial description [27] by describing its syntax and informal semantics.
3. Present a case study of a system for managing residential suites to demonstrate the capability and usability of SOFL.
4. Propose important topics for future work in the field.

The remainder of the paper is organized as follows. Section 2 describes the software process using SOFL. Section 3 describes the SOFL methodology for building software systems, illustrated with the

Residential Suites Management System case study. In section 4, we evaluate SOFL in the light of the case study. Finally, section 5 gives conclusions and outlines future research.

2 Software Process Using SOFL

Software process is an area that has received much attention from many researchers [28]. The software process advocated for SOFL is a specialized Waterfall development model [29], as shown in Figure 3. The essence of this process is that software system development using SOFL consists of two phases: *static development* and *dynamic development*. Each phase includes capturing of requirements. Static development refers to the activities that produce a program, and dynamic development refers to the activities that improve the program until it satisfies user requirements. For this software process to work, the architecture derived in the static phase must be able to accommodate changes made in the dynamic phase.

Static development is divided into four stages: *preliminary requirements analysis*, *formal requirements specification*, *design*, and *implementation*.

The first activity of static development captures user requirements as completely as possible. Usually such requirements are written in natural language, possibly coupled with special terminology of the application domain used in an informal manner. Once the developer reaches a general understanding of the problem, static development proceeds to formal requirements specification. In this stage more detailed requirements analysis is carried out by building a formal specification (shown as Requirements specification *n* in Figure 3). This specification reflects the primary user requirements derived through successive evolutions, each of which transforms an abstract specification to become more concrete (e.g. Requirements specification *1*, Requirements specification *2*). Evolution steps may involve the usual notion of formal refinement, and may also involve introducing additional constraints required in the final solution. Hence a specification during the evolution process is considered to provide functional constraints on the final concrete specification rather than a complete description.

The third stage of static development is design, which transforms the final formal requirements specification to a design specification. The design is reached by successive evolutions that transform an abstract design to a more concrete one.

The final stage of static development is implementation, in which a program is constructed to realize the design. This involves realization of the major executable components, definition of the concrete data structures, and implementation of any minor auxiliary structures or functions that may be assumed in the design.

It is important to verify the design and program against the requirements specification and design through rigorous reviews.

Dynamic development is a process of discovering dynamic features of the system, verifying the system against the user's actual needs and acquiring more requirements by means of prototyping and testing. Prototyping is effective for elucidating the user's requirements for features that cannot easily be obtained by formal analysis, such as the human-computer interface and timing requirements. Formal methods can be useful to identify what should be prototyped under what constraints in a system. Prototyping can also be used for risk analysis as in the Spiral model [30], but the use of formal methods can improve the quality of prototypes.

Testing should be conducted to verify the system against both the informal user requirements (system testing) and the end-user (acceptance testing). It helps narrow the gap between the informal user requirements and the formally implemented system and the gap between the informal user requirements and the user's actual needs.

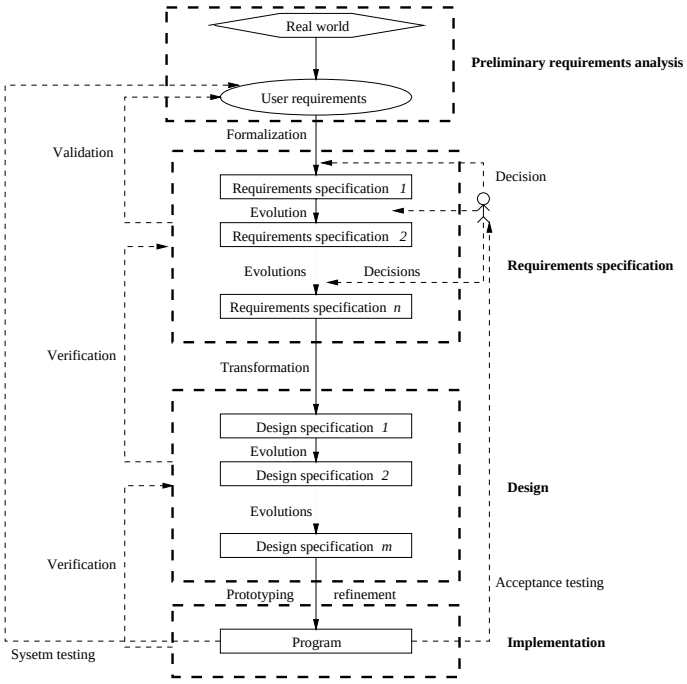

Figure 3: Static and dynamic development process

Static development and dynamic development may interact with each other in practice. That is, when static development reaches a certain level, some parts of the current system specification can be prototyped and tested, possibly in parallel with further static development of the rest of the current specification. Also, after some parts of the specification are prototyped and tested, it may be necessary to capture more requirements and design more specification components, which leads to further static development.

3 SOFL Methodology for Building Software Systems

Constructing a formal software system using SOFL consists of three separate activities: requirements specification, design, and implementation. Requirements specifications and designs are described using the SOFL specification language, while implementation realizes the design using the SOFL implementation language.

A SOFL specification is a hierarchical condition data flow diagram that is linked with a hierarchy of specification modules, as shown in Figure 4. The diagram describes data flow between components, and constrains the flow of control by the data flow operational semantics [31]. The specification modules precisely define the functionality of the components in the diagram.

The implementation is a hierarchy of implementation modules (or i-modules for short), as illustrated in Figure 5. An i-module is a collection of declarations of constants, types, classes, state variables, procedures, and functions. A SOFL program always has an initial i-module called **program** that must contains a procedure **main** from which the entire program starts execution.

When using the SOFL methodology, engineers construct the initial condition data flow diagrams and specification modules, then use decomposition, evolution, and transformation to construct an object-based design from the structured requirements specifications, and finally transform the design

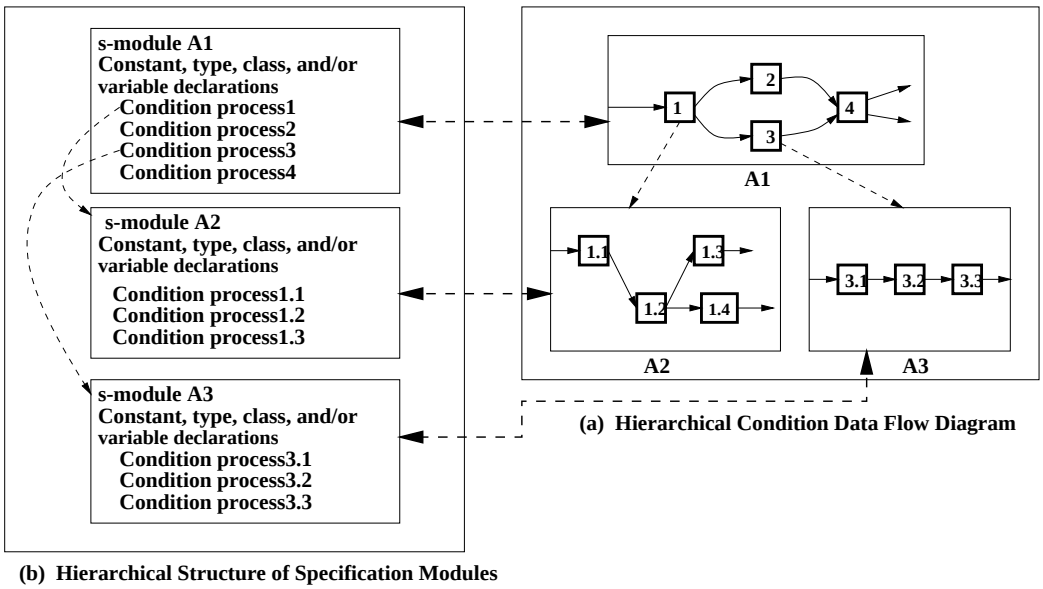

Figure 4: The structure of a SOFL specification

to programs.

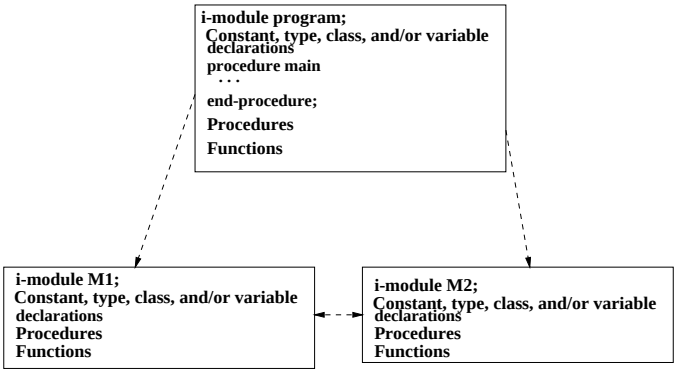

Figure 5: The structure of a SOFL implementation

3.1 Background of Case Study

A case study involving an actual *Residential Suites Management System* has been conducted by the first author. It is used to illustrate the SOFL methodology.

A local Residential Suites company manages various businesses and resources. The business includes front desk services, room services, telephone services, report services, sports club services, finance, and security. The resources include 500 apartments (200 single, 100 double, 100 twin, 30 deluxe, 50 suite, and 20 maisonette), two shops, two restaurants, and a swimming pool. The company uses computers to support the activities at the front desk, but there is no computerized system to support the entire management. The manager of the Residential Suites cooperated with us in developing a specification of a software system.

Specification started with an interview with the Residential Suites manager to establish the user requirements. This initial contact was to gain a basic understanding of the business and some relevant documents on the management system. The manager has no knowledge of programming, but understands the management system (the application domain). After preliminary requirements analysis, an abstract high level agreement was reached on the function of the potential management system. The major functions of the software system are: Reservation, Check in and check out, Room services, Telephone services, Report services, Finance, Sports club services, and Security.

Through the analysis, it was found that the Residential Suites management has significant differences from a normal hotel. For example, the front desk personnel can only allow customers to check in if they have made a reservation in advance. Additionally, security requires that this reservation must be made through a contracted organization. Also, customers must stay for at least seven days and no more than one year. Daily room services are not provided unless specifically requested by customers, and if they are, an additional charge is levied. The Residential Suites management system can be thought of as being between a normal hotel system and an apartment rental system.

3.2 Construction of Condition Data Flow Diagrams and Specification Modules

3.2.1 Condition Data Flow Diagrams

A *condition data flow diagram* (CDFD) is a directed graph consisting of data flows, data stores, and condition processes. A data flow is labeled with a typed variable that represents a packet of data transferred between condition processes. A data store is a variable of a specific type to represent data at rest. A condition process is like a process in DeMarco data flow diagrams [32], but with pre and postconditions that specify its functionality. Figure 6 shows the major components used in CDFDs and Figure 7 explains the meanings of condition processes with various combinations of input and output data flows.

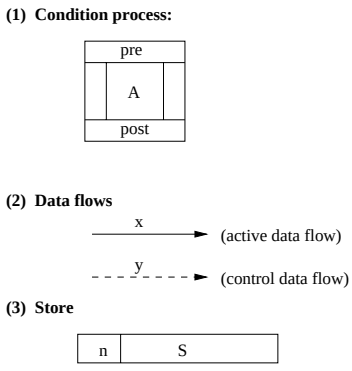

Figure 6: The components of CDFDs

Figure 8 shows the top level CDFD for the Residential Suites Management System. The information manager accepts requests for reservations from customers (res-req), requests for reports (rep-req), and check-out requests from customers (room-no). It produces daily reports for the manager (d-rep), and bills for customers (check-out-bill).

There are four important distinctions between CDFDs and DeMarco and Yourdon data flow diagrams:

1. A CDFD has an operational semantics [31]. It not only describes how components are statically connected, but also how they dynamically interact with data flows. Consider the condition pro-

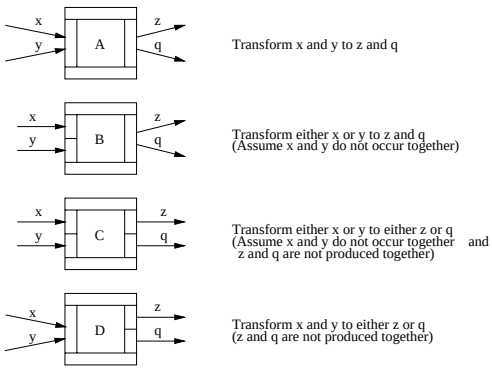

Figure 7: Condition processes with input and output data flows

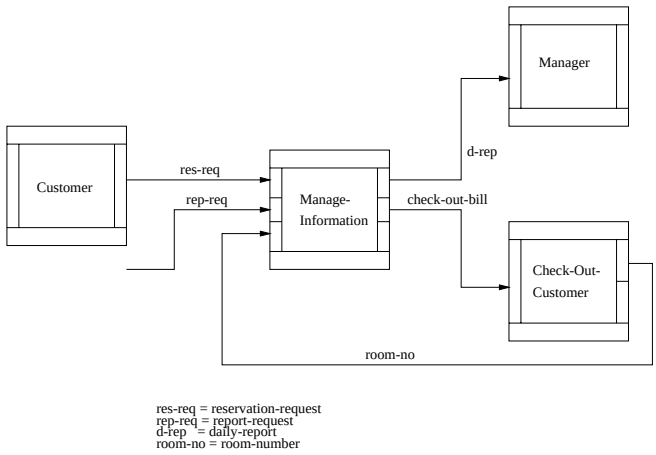

Figure 8: The top level CDFD of the requirements specification

cesses Manage-Information and Check-Out-Customer of the CDFD in Figure 8. By the operational semantics, the condition process Manage-Information can *fire* given any one of the inputs res-req, rep-req and room-no; and may generate either a d-rep, check-out-bill or non-output (indicated by the output port with no data flow). *Firing* of condition process Check-Out-Customer requires data on the flow check-out-bill, and either does not generate output or generates a room-no.

2. A classical data flow diagram represents the static structure of the system and a given process appears only once in a strictly nested hierarchy, while a CDFD represents the dynamic structure of the system and an appearance of a process in the CDFD represents a use of the process. The same process may be used in different parts of a specification to process different data.

3. In classical data flow diagrams, a data flow only transfers data from sources (e.g. process, external process, or data store) to destinations (similar). In CDFDs, a data flow indicates both data transfer and system control.

4. CDFDs may use *conditional nodes* (analogous to *diamond* in flowcharts) to express alternative data flows depending on the values in a data packet. They also use connecting points to avoid crossing of data flows, as shown in Figures 9 and 10. Classical data flow diagrams have no such nodes.

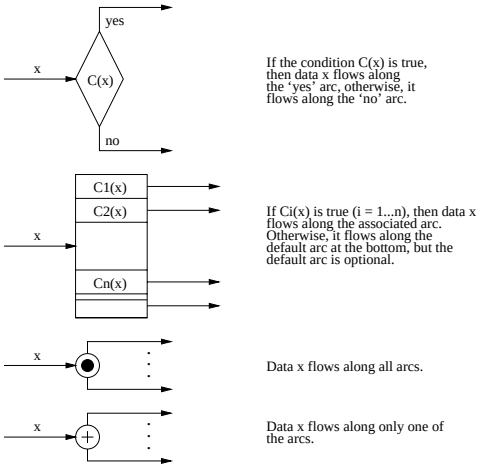

Figure 9: Conditional and diverging data flows

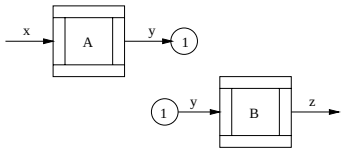

Figure 10: An example of connecting point

Drawing a CDFD helps to describe an overall picture of the system structure. It provides a foundation for defining components in the associated specification module (called an *s-module*), and for improving structure during the formal development. The s-module and the CDFD are complementary in three respects: (1) the CDFD describes the relation between condition processes in terms of data flows while the module describes the precise functionality of the condition processes in terms of their inputs and outputs, (2) the CDFD describes the dynamic structure of the system while the module provides a static definition of all its components, and (3) the CDFD provides a graphical view of the system at the current level while the module supplies the details of the system in a textual form.

3.2.2 Specification Modules

One important issue is how to define the s-module associated with a CDFD. We use the following four rules:

Rule (1) Each data flow, except those between data stores and condition processes, and each data store in the CDFD corresponds to a state variable in the module. State variables are introduced with the keyword **var**.

Data stores will be accessed directly from condition processes as external variables, and hence no separate variable is needed to hold data in transition to or from a store; however data in transition between processes will be held in a data flow variable.

Rule (2) For each condition process, a **c-process** definition is supplied, which may have a name (which is the same in the diagram and in the textual specification), input and output parameters for incoming and outgoing data flows to other processes, external variable references for data stores, a

pre and postcondition to define functionality, a decomposition, and a comment.

The syntax for parameter lists is intended to reflect the structure of inputs and outputs given in the diagram.

- The input ports (respectively output ports) from top to bottom in the graphical condition process are defined from left to right in the input (respectively output).
- Different ports are separated by the symbol |.
- Each connection to an input (or output) data flow in the graphical notation is defined as a (formal) parameter with the appropriate type. The order of parameters in the textual specification corresponds to the top to bottom ordering in the graphical notation.
- A port with no associated data flow is supplied with a dummy variable that has a special type **void**. Such a variable can be defined or undefined; although when defined it has no useful value.

For example, the definition of condition process Manage-Information in Figure 8 is outlined as follows:

c-process Manage-Information(res-req: Reservation | rep-req: ReportRequest | room-no: **nat**)
d-rep: DailyReport | check-out-bill: CheckOutBill | dummy: **void**
pre $Pre_{\text{Manage-Information}}$(res-req, rep-req, room-no)
post $Post_{\text{Manage-Information}}$(res-req, rep-req, room-no, d-rep, check-out-bill, dummy)
decomposition ...
comment ...
end-process;

where Reservation, ReportRequest, DailyReport, and CheckOutBill are types defined in the specification module of the CDFD in Figure 8, given later in this section, and **nat** is a built in type representing natural numbers.

The parameters res-req, rep-req, room-no, d-rep, and check-out-bill can have different labels from the data flows, since they are linked to the diagram by their order of appearance. The variable dummy represents the empty port of the graphical condition process. Pre and postconditions are predicates. A precondition can refer only to variables corresponding to input data flows and input external variables, and postconditions can refer to all variables.

SOFL uses a VDM style semantics, where variables can be bound to a value of their type, or else they can be **nil** , which is the generic value meaning *unavailable* or *non-value*. The predicate **bound(**x**)** is **true** when x is bound to any useful value, but **bound(nil)** is **nil** .

Here are some possible preconditions for the condition process Manage-Information in Figure 8:

- **true**
  This indicates no specific precondition at all.
- **bound**(res-req) **or bound**(rep-req) **or bound**(room-no)
  This means that either res-req, rep-req or room-no has a value. This precondition is guaranteed by the operational semantics.
- **bound**(res-req) **or** P(rep-req, room-no), where P(rep-req, room-no) is a predicate depending on either rep-req or room-no but not on res-req
  This precondition needs to be checked against the sources of rep-req and room-no; the condition process only has well defined behavior for either a rep-req or room-no that satisfies P.

The postcondition can also have many forms. A **true** postcondition asserts nothing about the values output by the process. A more interesting postcondition is

```
if      bound(rep-req)
then    P2(rep-req, d-rep)
else    if room-no > 5
        then P2(room-no, check-out-bill)
        else bound(dummy)
```

If rep-req is available, then d-rep is generated and will satisfy P1(rep-req, d-rep). If room-no is available and greater than 5 then check-out-bill is generated and will satisfy P2(room-no, check-out-bill), and otherwise the dummy variable is bound, indicating activation of the port with no outgoing data flow. Since dummy is a variable of type **void**, the only thing that can be said about it is whether or not it is bound.

Note that complete description of pre and postconditions also requires consideration of the port structure of a process. For example, it is an implicit precondition of the process Manage-Information that only one of res-rep, rep-req and room-no will be bound. This cannot be expressed directly since VDM predicates are *monotonic*, meaning that they cannot assert that a value is unbound. However, if P(res-req, rep-req, room-no) is the explicit precondition, then the full precondition is P(res-req, **nil** , **nil**) **or** P(**nil** , rep-req, **nil**) **or** P(**nil** , **nil** , room-no). Similar rules apply for the postcondition.

The sample pre and postconditions given are well formed in the sense that taking into account the port structure does not alter the meaning of the predicates. A check for consistency between conditions and port structures is usually straightforward, and so we will not consider this subtlety further in this paper.

Rule (3) Define a store as an **rd** (read only) external variable for a condition process if there is a data flow from the store to the condition process but no data flow from the condition process to the store in the CDFD.

Define a store as an **wr** (write and read) external variable for a condition process if there is a data flow from the condition process to the store in the CDFD.

For example, the condition process A and B in Figure 11 are defined in the s-module as follows:

```
c-process A()
  ext    wr s: Type
  pre    Pre_A(s)
  post   Post_A(~s, s)
end-process;

c-process B()
  ext    rd s: Type
  pre    Pre_B(s)
  post   Post_B(s)
end-process;
```

where ~s and s represent the value of s before and after *firing* (execution) of condition process A.

Rule (4) An omitted precondition or postcondition is taken to be **true**.

Condition processes without input (or output) data flows can be used to represent an interface between the software system and its external operating environment.

By applying the above four rules, we construct a specification module for the CDFD in Figure 8 as follows:

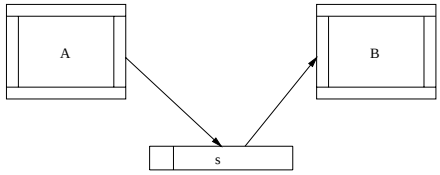

Figure 11: Condition processes connected to a store

s-module Residential-Suites-System : Figure 8;

type

Reservation = **composed of**
name: **string**
address: **string**
telephone: **nat**
period: Period
room-type: RoomType
date: Date
end ;

Period = Date * Date; /* check-in day, check-out day */

Date = **nat** * **nat** * **nat**; /* day, month, year */

RoomType = {SINGLE, DOUBLE, DELUXE, SUITE, MAISONETTE}

DailyReport = **composed of**
shift-report: ShiftReport
income-report: IncomeReport
events-report: **string**
end ;

ShiftReport = **map** Name **to** (Time * Time); /* associate name with (start time, end time) */

Name = **string** ;

Time = **nat** * **nat**; /* hour, minute */

IncomeReport = **composed of**
total: **real**
income-list: **map string to real**; /* **string** is for the description
of reasons for charge, **real** is for the amount of the charge. */
end ;

CheckOutBill = **composed of**
way-of-payment: **string** /* e.g. Cash, Visa, MasterCard */
room-no: **nat**
check-out-date: Date
remarks: **string**
signature: **string**
end ;

ReportRequest = {SHIFT, INCOME, EVENTS, ALLREPORT};

var

res-req: Reservation;

rep-req: ReportRequest;

check-out-bill: CheckOutBill;

```
  d-rep:    DailyReport;
  room-no:    nat;
inv
  forall[x inset Reservation |    diff(x.period(1), x.date) <= 1 and diff(x.period(2), x.period(1)) >= 7 and
                                  diff(x.period(2), x.period(1)) <= 366];
/* Any reservation must be made at least one day before check in and any customer must stay at the Suites at least
seven days and no more than one year. */
  function diff(day1: Date, day2: Date): nat
  == undefined /* A precise definition of this function is left for design or implementation. */
  end-function;
  c-process Customer()res-req: Reservation
    post          res-req.name <> " " and res-req.address <> " " and res-req.period
                  <> ((0, 0, 0), (0, 0, 0))
    comment
                  A reservation request output by this condition process must provide
                  at least a name, address and period of intended stay.
  end-process;
  c-process Manage-Information( res-req: Reservation | rep-req: ReportRequest | room-no: nat)
  d-rep: DailyReport | check-out-bill: CheckOutBill | dummy: void
    pre                 true
    post                if bound(rep-req)
                        then bound(d-rep)
                        else
                        if bound(room-no)
                        then bound(check-out-bill)
                        else true
    decomposition       Manage-Information-Decom1, Manage-Information-Decom2
    comment
                        The input rep-req is used to produce the output d-rep; room-no is
                        used to produce check-out-bill; and res-req is used to produce noth-
                        ing. More detailed definition of the functionality cannot be properly
                        expressed at this level due to the lack of information.
  end-process;
  c-process Check-Out-Customer()room-no: nat | dummy: void
    pre           true
    post          true
    comment
                  No specific requirements are given for the functionality of this condi-
                  tion process at this level. It is left open for later extension.
  end-process;
  c-process Manager(d-rep: DailyReport)
    pre   true
  end-process;
end-module;
```

3.3 Decomposition

3.3.1 Decomposition Rules

A complete specification is a structured hierarchy of CDFDs, in which a condition process at one level is decomposed into CDFDs at a lower level. The decomposition of a condition process defines how its

inputs are transformed to its outputs. The decomposition must conform to the constraints given by the high level condition process specification. Such a decomposition is not only a definition of the high level condition process, but also a possible extension in functionality. This allows SOFL to support functional abstraction at high levels and detailed implementation at lower levels.

There are three of important distinctions between SOFL decomposition rules and those for classical Data Flow Diagrams:

(1) A lower level CDFD must have all the input and output data flows of its higher level condition process, but it is also allowed to have additional input and output flows.

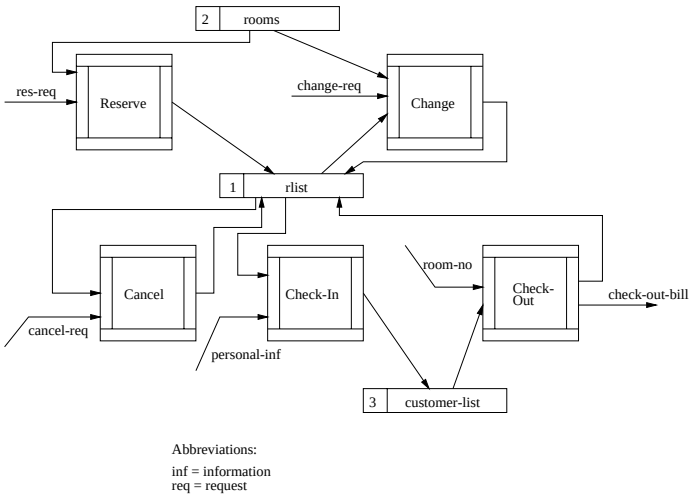

Figure 12: Part of decomposition of Manage-Information

Any additional data flows in the lower level CDFD are considered to be internal state variables of the high level condition process. These additional data flows have one open end; the other end is not given. An advantage of this feature is that a developer can concentrate on the most important issues and leave the job of determining the exact source or destination to design or implementation. For example, condition process Manage-Information in Figure 8 has input and output data flows res-req, rep-req, room-no, d-rep, and check-out-bill, but its decomposed CDFDs in Figures 12 and 13 have the additional input and output data flows change-req, cancel-req, personal-inf, income-inf, shift-inf, events-inf, shift-rep, income-rep, and events-rep.

Experience in the case study suggests that this rule is more effective than Yourdon's rule [33] in obtaining abstraction and encapsulation of data and condition processes. An engineer is able to focus on the most important problems at each level, which reduces the complexity of each level of the hierarchy.

(2) A condition process may be decomposed into a number of lower level CDFDs.

This is especially useful when a condition process has to be decomposed into a lower level *disconnected* CDFD. In that case, a sensible way to organize the disconnected CDFD is to treat each connected part as one CDFD and to build their corresponding s-modules separately. For example, the condition process Manage-Information in the top level CDFD in Figure 8 is decomposed into three CDFDs. Two of these are given in Figure 12 and 13, respectively, and another one is omitted for brevity. The s-module for the CDFD in Figure 12 is given in Appendix A.

(3) CDFDs can directly use the constants, types, classes, and variables declared in the higher level s-modules they are derived from. In other words, the scope of the constants, types, classes, and

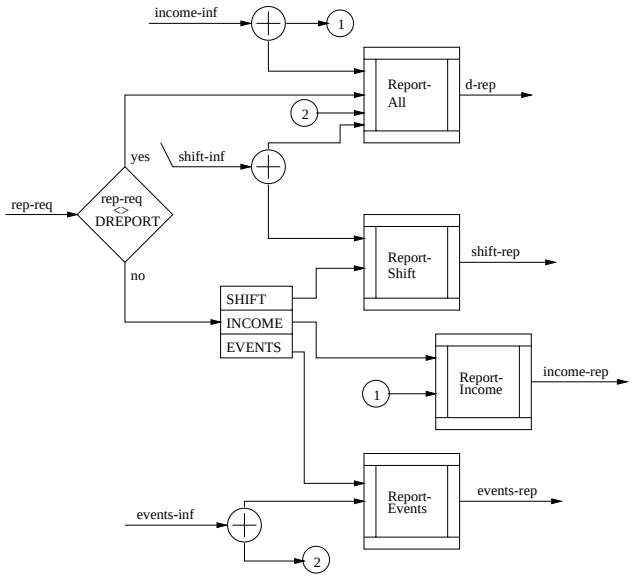

Figure 13: Part of decomposition of Manage-Information

variables declared in an s-module is the module itself and all its *descendent* s-modules. This reduces unnecessary duplication of the declarations of the same constants, types, classes, or variables in the hierarchy of CDFDs.

3.3.2 How to Use Decomposition

There are two useful approaches to constructing the hierarchy of CDFDs and their s-modules. One is *module-first* and the other is *CDFD-first*.

In *module-first* decomposition, a designer draws a CDFD, then defines all components of the CDFD in its s-module. This has several advantages.

(1) It helps clarify possible ambiguity of data flows, data stores, and condition processes in the CDFD.

(2) Defining an s-module may lead to improved understanding and consequent improvements to the CDFD before proceeding further.

(3) It helps to identify which condition processes need decomposition.

SOFL offers the following criteria for choosing the process to be decomposed.

- If the functionality of a condition process is too complex to be specified by pre and postconditions, then it should be decomposed.
- If no clear relation can be given between input values and output values of a condition process (that is, pre and postconditions are not clearly known), then it should be decomposed.
- If the relation between input values and output values is not *functional* (that is, giving a unique output for a given input), then it is useful to consider decomposing this condition process.

  Of course, in practice it can be useful to work with incomplete specifications, either because a non-deterministic solution is satisfactory or because of practical constraints of time and/or cost. Hence this criterion should be tempered with flexibility based on the situation and the developer's engineering judgment.

Experience on the case study suggests that the module-first approach is especially effective when the developer is not familiar with the application domain. Formalizing the components of the CDFD *forces* the developer to gain more understanding of the CDFD. However, a disadvantage of this approach is that the developer does not get a full picture of the overall system before formalization, and may therefore do some unnecessary work that needs to be repeated once more understanding is gained.

In *CDFD-first* decomposition, after drawing a CDFD the designer decomposes those processes that will require further detail. This approach has the advantage of helping the developer obtain an overall picture of the system and the relations between components within and between levels before proceeding with the full formal definition. However, it can be difficult to decide which condition processes should be decomposed. Our experience shows that this approach is effective when the developer knows the application domain well, or has experience developing software systems for similar application domains.

Both approaches were used in this case study as deemed appropriate, and we consider both approaches to be necessary in development of a complex system.

3.4 Evolution

In addition to decomposition, development of the CDFD hierarchy requires *evolution*. Evolution refers to changes in the structure or functionality of a condition process or CDFD.

- A condition process is decomposed by constructing a corresponding lower level CDFD to refine the process. Evolution of a condition process or a CDFD changes its structure or functional definition to make a new version of the current specification.
- Decomposition extends a specification in a top-down fashion while evolution improves it in a horizontal direction, as shown in Figure 14.

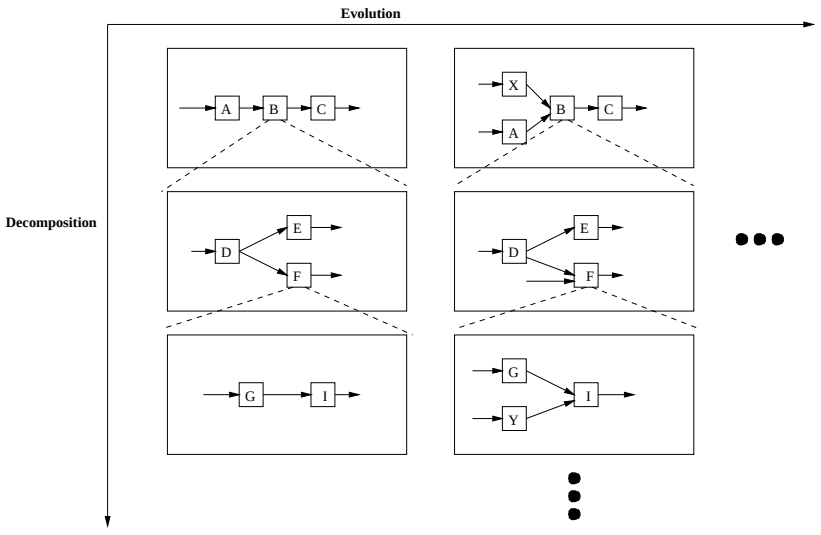

Figure 14: Illustration of decomposition and evolution

The following approach is an effective way to use evolution and decomposition when constructing a specification:

- Decomposition and evolution are interleaved, but decomposition usually comes first and then evolution steps are made if necessary.
- When decomposition of a condition process reveals the need for a change to the condition process itself, then an evolution of the condition process and/or the associated CDFDs needs to be carried out. The result of the evolution is another hierarchy of CDFDs that properly reflects the decomposition relation between high level condition processes and their decomposed CDFDs.

By carrying out decomposition and evolution, we transform the top level CDFD to a final specification consisting of the CDFDs in Figures 8, 12, 13, and 15 (other parts of the specification are omitted for brevity).

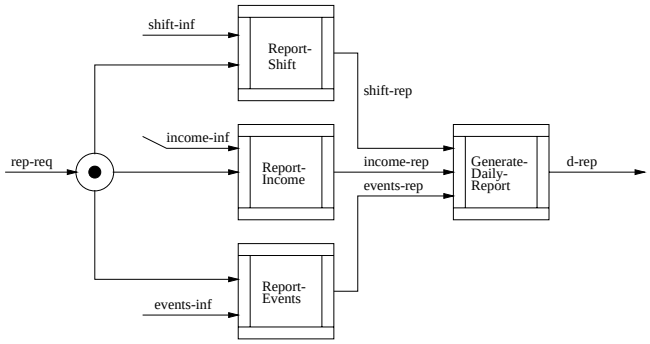

Figure 15: Decomposition of Report-All

3.5 Transformation from Structured Requirements Specifications to Object-based Design

To derive a design from the requirements, software engineers must develop algorithms and a software architecture. They should help achieve quality attributes such as reliability, readability, reusability, information encapsulation, and maintainability. This is necessarily a creative process, but one that can be supported by appropriate guidelines. The SOFL methodology offers the following four guidelines:

(1) Design the system as a hierarchy of CDFDs by following the hierarchy of CDFDs in the requirements specification.

This does not necessarily mean there should be a strict one-to-one correspondence. Some changes may be advisable to improve the quality of design. There may also be processes in the requirements specification that represent real world entities rather than parts of the software system, such as Customer in Figure 8. In many cases several levels of CDFDs in the requirements specification correspond to one level of design. For example, the CDFD in Figure 16 in the design specification of the Residential Suites Management System is derived from the CDFDs in Figures 8 and 12 in the requirements specification.

(2) If a condition process in the requirements specification is directly used in the design, then its formal specification (by pre and postconditions) should be a refinement of its formal requirements specification (i.e. they satisfy the laws of refinement [34]).

(3) If several condition processes access or change some data stores and provide services for these stores, then use a class declaration that includes those data stores as its attributes, and implement

the condition processes as methods on this class. If condition processes accessing other stores need to use either the data stores defined as attributes or condition processes defined as methods on the current class, then a subclass of the current class can be created to include those other data stores as its attributes and those condition processes accessing the data stores as its methods.

Experience with the case study shows that using the structured method for requirements specification helps to identify what condition processes need to access or change what data stores, and to identify the relations between different data stores and between the condition processes that access or change those stores. A thorough understanding of these relations can facilitate the building of class declarations and a class hierarchy in the design. This class hierarchy aids data and function encapsulation, but the foundation of design remains the CDFD hierarchy. Condition processes may then use the services (methods) provided by the classes.

The CDFD in Figure 12 demonstrates this. The condition processes Reserve, Cancel, and Change access the two data stores rlist and/or rooms, while Check-In accesses both rlist and customer-list. Condition process Check-Out also uses the same data stores as Check-In, but in addition it generates an output data flow check-out-bill. By considering this situation and applying the above guideline, we create a class hierarchy in the design specification module Top-Design, that consists of three classes: CustomerReservation, CheckInCustomers, and CheckOutCustomers, where CheckInCustomers is a subclass of CustomerReservation and CheckOutCustomers is a subclass of CheckInCustomers.

By considering these class declarations and the requirements specification, the top level CDFD is constructed in Figure 16, and its associated s-module Top-Design in the design specification is given below. Note that condition process Reserve-Services takes an object r-object of class CustomerReservation as its input and generates a new object r-object of the same class. Other condition processes like Check-In-Service and Check-Out-Service also process objects (e.g. ci-object, co-object) of other classes. The dashed directed lines represent *control data flows* that carry no real data for condition processes to consume, but can cause firings of condition processes.

We must also consider the user interface. There are no specific requirements for the layout of the interface in the primary requirements because this is not part of the functional requirements. However, it is an important part of the system to the users, and they will probably make suggestions for improvement after seeing a prototype. Requirements for functions like Interface in Figure 16 can be considered secondary requirements.

s-module Top-Design: Figure 16;

type

...

class CustomerReservation;

rlist: ReservationList;

rooms: Rooms;

method Reserve(res-req: ReservationRequest)

ext	**wr** rlist: ReservationList
	wr rooms: Rooms
pre	Pre_{Reserve}(res-req, rlist, rooms)
post	$Post_{\text{Reserve}}$(res-req, ~rlist, ~rooms, rlist, rooms)
comment	
	...

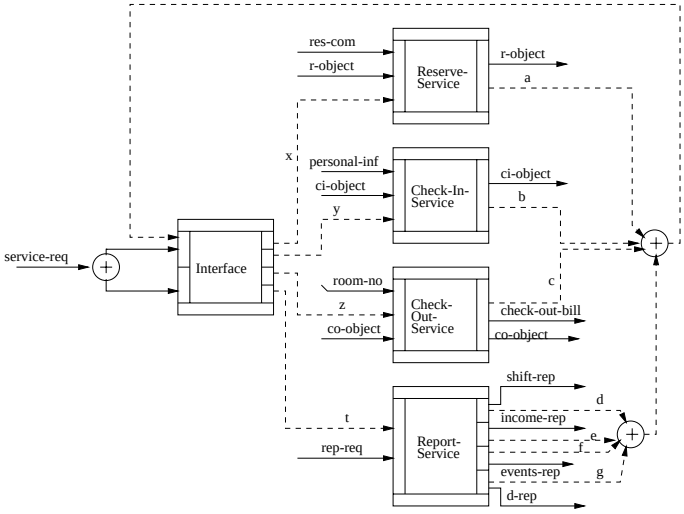

Figure 16: Top level CDFD of design

method Change(change-req: ChangeRequest)
ext **wr** rlist: ReservationList
wr rooms: Rooms
pre Pre_{Change}(change-req, rlist, rooms)
post $Post_{\text{Change}}$(change-req, ˜rlist, ˜rooms, rlist, rooms)
comment
...

. . .

end-class;

class CheckInCustomers / CustomerReservation; /* CustomerReservation is a super class of CheckInCustomer */

customer-list: CustomerList;

method Check-In(personal-inf: PersonalInformation)
ext **rd** rlist: ReservationList
wr customer-list: CustomerList
pre $Pre_{\text{Check-In}}$(personal-inf, rlist, customer-list)
post $Post_{\text{Check-In}}$(personal-inf, ˜customer-list, rlist, customer-list)
comment
...

end-class;

. . .

var

d-rep: DailyReport;

r-object: CustomerReservation;

ci-object: CheckInCustomers;

co-object: CheckOutCustomers;

check-out-bill: CheckOutCustomers;

c-process Reserve-Service(res-com: Choice, r-object1: CustomerReservation, x: **void**)
r-object2: CustomerReservation, a: **void**
pre $Pre_{\text{Reserve-Service}}$(res-com, r-object1, x)
post $Post_{\text{Reserve-Service}}$(res-com, r-object1, x, r-object2, a)
. . .
end-process;
...

end-module

(4) If several condition processes access or change data stores and provide services for these stores, then another object-based design technique can be used, creating a class declaration for each data store. The store declaration in the requirements specification becomes a class declaration in the design. Some of the condition processes can be implemented as methods on each class. This approach usually requires further changes to the structure of condition process specifications, making the external variable representing a store into an input and output parameter of the process. Choosing which approach to use for generating classes is a design judgment made for each application.

3.6 Transformation from Designs to Programs

As described in the beginning of section 3, a SOFL implementation is a hierarchy of i-modules. I-modules interact with each other by procedure, function, or method calls; and also by shared data stores. An i-module that calls another i-module must include the signatures of procedures, functions or methods that are used. This structure is similar to that of C++ programs, but SOFL programs are at a higher level of abstraction, since the same abstract data types (sequences, sets, maps, composite objects, etc) are used in i-modules and in specifications. This facilitates the transformation from design to program, and the verification of the program against its design.

From the SOFL point of view an implementation is a program because it is executable (given a uitable compiler); but it can also serve as a detailed design for guiding implementation in a lower level programming language like C or C++.

Two approaches can be used to carry out the transformation. One is *direct transformation* and the other is *object-oriented transformation*. The two approaches share most of the rules, but differ in how they organize the declarations of procedures that are derived from condition processes in the design.

3.6.1 Direct Transformation

Direct transformation means that the hierarchy of CDFDs and associated s-modules is mapped to an equivalent hierarchy of i-modules. The specific guidelines are:

(1) Transform the top level s-module to the start i-module **program** and every other s-module to an i-module.

(2) Implement the top level CDFD of the top level s-module as the **main** procedure in the start i-module.

(3) Transform each condition process specification to a procedure declaration and each function specification to a function declaration. The following rules map input and output data flows of a condition process to procedure parameters:

- Map an input data flow to a **value** parameter of the procedure.
- Map an output data flow to a **var** parameter of the procedure.

- If an input has the same name and type as those of an output data flow, then map both flows to one **var** parameter of the procedure.

(4) Transform design classes and methods directly to implementation classes and methods. If there is no create method explicit in the design, also supply a create method to create the initial object of the class.

The semantics of a method in the specification is slightly different from the corresponding method in the implementation. The former is a function that returns a new object of the same class, but is implemented as a method that changes the state of a given object.

(5) For each open input and output data flow of a CDFD, create a procedure to obtain or to output the corresponding values at the program interface (for example, by using **read** and **write** statements within the procedure).

(6) For a condition process that is not further decomposed, declare a procedure in the i-module that corresponds to the CDFD in which the process is used. Otherwise, declare a corresponding procedure whose body is derived from its decomposed CDFD.

(7) If a condition process is decomposed into a number of CDFDs, merge all those CDFDs to one before transforming it to an i-module.

For example, we transform the top level CDFD in Figure 16 and its s-module to the top level i-module (giving just an outline for brevity) as follows:

i-module program;

type

...

class CustomerReservation;

rlist: ReservationList;

rooms: Rooms;

method create();

...

method Reserve(res-req: ReservationRequest);

...

method Change(change-req: ChangeRequest);

...

method Cancel(cancel-req: CancelRequest);

...

end-class;

class CheckInCustomers / CustomerReservation;

customer-list: CustomerList;

method create();

...

method Check-In(personal-inf: PersonalInformation);

...

end-class;

class CheckOutCustomers / CheckInCustomers;

bill: CheckOutBill;

method create();

...

method Check-Out(room-no: **nat**);

...

end-class;

procedure main

... /* declarations */

Algorithm(Reserve-Service(...), Check-In-Service(...), Check-Out-Service(...), Report-Service(...))

end-procedure;

procedure Reserve-Service(...);

...

end-procedure;

procedure Check-In-Service(...);

...

procedure Check-Out-Service(...);

...

procedure Report-Service (...) / RS-module

end-module;

In the procedure **main**, Algorithm(Reserve-Service(...), Check-In-Service(...), Check-Out-Service(...), Report-Service(...)) denotes an algorithm in which procedures Reserve-Service, Check-In-Service, Check-Out-Service, and Report-Service may be called. **procedure** Report-Service (...) / RS-module at the end of the module is a specification of the procedure Report-Service (at program level) and its detailed declaration is given in another i-module called RS-module (details omitted).

3.6.2 Object-Oriented Transformation

In object-oriented transformation, the object-based structure of the design is transformed to a hierarchy of classes (with some auxiliary procedures such as the **main** procedure). The transformation rules are as for the direct transformation, with the following differences:

- For each CDFD in the design, create a class that defines all the state variables in the CDFD as the class attributes.
- Transform each condition process used in the CDFD to a method in the class created from the CDFD.
- If a condition process is decomposed into a lower level CDFD, then the body of its transformed method should be an implementation of the lower level CDFD by calling the methods that are defined on the class of the lower level CDFD.

Let us again take the top level CDFD in Figure 16 and its s-module as an example of this approach. It is transformed to the program:

i-module program;

type

...

class ResidentialSuites;

... /* Attribute declarations that corresponding to all the global variables in the design s-module of the CDFD in Figure 16. */

method create(...);

...

method Reserve-Service(...);

...

method Check-In-Service(...);

...

method Check-Out-Service(...);

...

method Report-Service (...);

substate: RS-class;

Algorithm1(substate.create(...), ...)

end-method;

end-class;

class CustomerReservation;

...

procedure main

state: ResidentialSuites;

Algorithm(state.create(...), state.Reserve-Service(...), state.Check-In-Service(...), state.Check-Out-Service(...), state.Report-Service(...))

end-procedure;

end-module;

In the procedure **main** of this top level i-module, we declare a variable state of class ResidentialSuites and use its methods in the body of **main**, i.e.:

Algorithm(state.create(...), state.Reserve-Service(...), state.Check-In-Service(...), state.Check-Out-Service(...), state.Report-Service(...)).

Condition process Report-Service in the CDFD in Figure 16 is decomposed into the CDFD in Figure 17, which is transformed to the class RS-class. Thus, a variable substate of class RS-class is declared and its potential methods are used in the body of method Report-Service, i.e.:

Algorithm1(substate.create(...), ...).

The direct and object-oriented transformation approaches are not necessarily the only possible ways to transform a design into an implementation. Some applications may not need to use any classes at all, depending on the particular application and the judgment of the developer. The SOFL constructs are flexible enough to support a range of strategies.

4 Evaluation of SOFL

The case study shows that the SOFL methodology is practical for realistic systems development, and that the SOFL language is capable of dealing with applications in a convenient fashion. The advantages of SOFL are as follows.

First, SOFL helps reduce the complexity of system development. Drawing a hierarchy of CDFDs helps

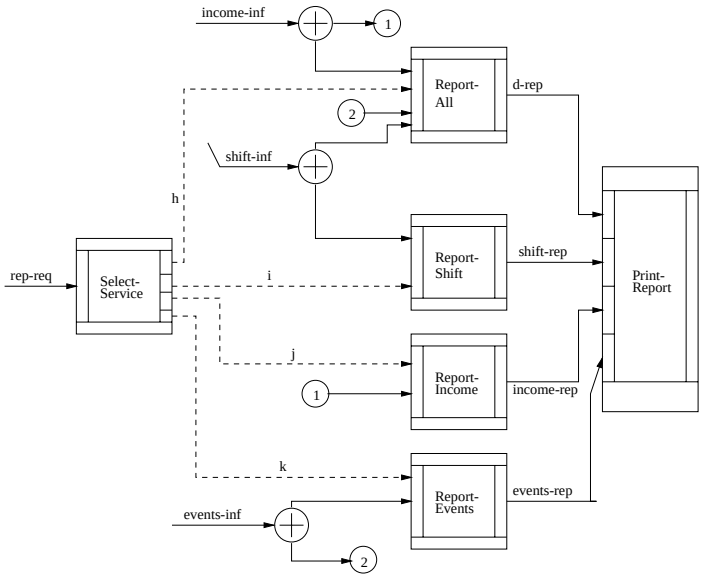

Figure 17: Decomposition of Report-Service

obtain an overall picture of the system, which provides a foundation for determining what components need to be defined in associated specification modules. Writing formal specifications helps to resolve ambiguity and obtain more accurate requirements; but it is important to have a preliminary structure and data to organize formalization. Condition data flow diagrams play an important role here. Giving a formal specification for data and condition processes helps to resolve ambiguity, to discover the necessary auxiliary data or functions (since they are used in the formal specification), and to improve the structure of CDFDs. Hierarchical CDFDs provide a comprehensible structure, which allows the developer to trace condition processes or data in the specifications (both requirements specification and design) during the decomposition, evolution, modification, and transformation stages. The graphical notation allows much more efficient search than is possible using textual specification modules alone, especially when the specification becomes larger and more complex.

Second, CDFDs in requirements specifications seem to be more comprehensible than either informal or formal texts alone, because they clearly show how data flows between condition processes that match the users' domain knowledge. This helps us explain the specification to the user for validation. The manager of the Residential Suites has no knowledge about programming at all, but he is fully aware of the management system. After carrying out a high level requirements analysis and drawing the high level CDFDs (without formalization yet), we had an interview with the manager and found that it was easy for him to understand the diagrams with our explanation. Of course, CDFDs alone are not sufficient to convince him that we are modeling the right things because he may not know what exactly each component means. To this end, formal definitions of data and condition processes in s-modules play an important role as well. Not only do they help us discover and understand actual functions of processes in the Residential Suites, but they also provide a foundation for interpreting the meaning of components during validation of the requirements specification.

Third, the SOFL language offers flexibility for construction of requirements specifications. A specification need not be in a strict hierarchy of CDFDs, although this is recommended. The developer can model data and functions in a real world system by drawing and specifying each CDFD separately. This may help avoid artificial abstraction and decomposition that do not appropriately reflect the situation in the real world system. This may also facilitate communication between the developer and

the user for validation of the specification. This experience is also shared by Hall [35].

Fourth, the CDFDs and class constructs in specification modules make the SOFL language capable of supporting an object-oriented approach from the very beginning of requirements specification. If the developer has experience in using object-oriented methods to model real world systems in similar application domains, then taking this approach from the very beginning may have benefits. In that case, object-oriented analysis and specification can be done as follows. Try to identify useful objects first (which is assumed not to be difficult for an analyst experienced in the application domain) and then for each object, build a CDFD in which internal data flows and stores and condition processes are treated as attributes and methods, as in Figure 12. Communications between objects are through method references. If a condition process (or method) needs to use other condition processes in a different CDFD corresponding to another object, then this condition process is decomposed into a sub-diagram of that CDFD that includes only those condition processes and their connections to be used by the high level condition process (or method).

Finally, SOFL facilitates transformation from requirements specifications in a structured style to designs in an object-based style and from designs to programs in the appropriate style. This is due to the CDFDs and their s-modules, which provide a guide, traceability, and precision for design; and because the requirements specification gives a clear picture of the entire system, including data and operations and their relationships. For example, transforming the CDFD of the requirements specification in Figure 8 and 12 to the CDFD of the design in Figure 16 has benefited from these features.

Despite the above advantages of SOFL, we find that lack of effective tool support for SOFL may be a barrier for its application to large scale projects. During specification and design, it becomes critically important for assurance of system quality and productivity that developers are able to efficiently draw and modify CDFDs, write specification modules, and check their consistency. It is necessary to investigate how SOFL can be applied to improve management, cost estimation, risk analysis, and communication in a software project.

5 Conclusions

5.1 Contributions

This paper makes two major contributions. The SOFL language and methodology are proposed, with attendant changes to the software process; and some guidelines for developing a software system are provided. To demonstrate the capability and features of SOFL, we present a case study of an industrial system for managing Residential Suites and evaluate the result.

The SOFL methodology includes two major components: a software development process and a specific method for building software. Software development is divided into two phases: static development and dynamic development, and each phase involves capturing different kinds of requirements.

Static development is divided into four stages: preliminary requirements analysis, requirements specification, design, and implementation. Preliminary requirements analysis extracts high level informal requirements from the user, while requirements specification formalizes and develops the requirements. Formalization is an iterative evolutionary process, with evolution making an abstract specification more concrete. In the design stage, the final formal requirements specification is transformed into a formal design through successive evolutions. Implementation is a transformation from the final design (specification) to a program written using SOFL implementation modules. Such a program can serve for prototyping and as a detailed design for the developer to conduct a more efficient implementation using another programming language if necessary.

Dynamic development is a process of discovering dynamic features of the system, verifying the system against the user's actual needs and acquiring more requirements by means of prototyping and testing. Prototyping is effective for obtaining requirements that may not be clear to the user, and that cannot be effectively obtained via formal methods. Testing and inspection are practical approaches for verification and validation.

The specific method for building software system includes (1) the construction of condition data flow diagrams and associated specification modules, (2) decomposition, (3) evolution, (4) transformation from structured requirements specifications to object-based designs, and (5) transformation from designs to programs. It emphasizes that structured methods can be effective for requirements specification while object-based methods can be used for design and implementation. It also gives guidelines for each of these five activities.

5.2 Active and Future Research

We are actively pursuing further research in several directions: (1) application of SOFL to a more complex system in industry, (2) rigorous review techniques, (3) program testing based on formal specifications, (4) automated transformation from CDFDs and condition process specification to programs, and (5) development of an intelligent support environment for SOFL based on an existing graphical prototype.

Application of SOFL to complex industrial systems can provide further data on the effectiveness of the SOFL methodology, and address open questions. How can SOFL be used by a project team in the industrial environment? How does the use of SOFL impact project management and communication? How does SOFL affect software validation, verification and maintenance? The first author is currently cooperating with a local electric corporation to investigate how SOFL is applied to develop a railway crossing controller.

We have already started research on rigorous review techniques, and have two ongoing projects. The first is to generate test data for inputs and outputs to and from a condition process. If a situation is found where the precondition is true but the postcondition is false for all the test data, then it implies that some bugs may exist in the specification (although it is not certain). The results of tests can also be used for validation purposes, to confirm whether the specification is consistent with the user requirements. Also, the process of developing tests can help identify inefficient specification definitions or expressions.

The second approach is to provide relations between different parts of the specification and proof obligations for the developer to check. This checking activity cannot be fully automatic, but can be supported by a software tool. For example, when checking whether a CDFD is consistent, we need to ensure that the precondition of any condition process is implied by the the postcondition of preceding condition processes. When checking whether a CDFD is consistent with its high level condition process specification, we need to show some implication relations between the pre and postconditions of the CDFD and those of its high level condition process. These proof obligations can be automatically provided by a tool, but the checking of the obligations has to be done by the developer.

In addition to rigorous review techniques, we are conducting research on program testing based on SOFL specifications [13]. We generate test data from logical expressions (preconditions and postconditions) using established rigorously defined criteria. We have made progress on test data generation rules and criteria, but have not yet provided tool support.

Automated transformation of CDFDs and condition process specification into programs is another interesting research area. We have investigated techniques for automatically transformating implicit pre and postconditions to programs [36]. Since it is generally impossible to construct a system that

will transform all formal specifications into an executable program, we take a semi-automatic approach in which specifications are automatically transformed to abstract programs that contain some unexecutable operations, and the unexecutable operations are manually transformed to lower level programs under the support of a tool.

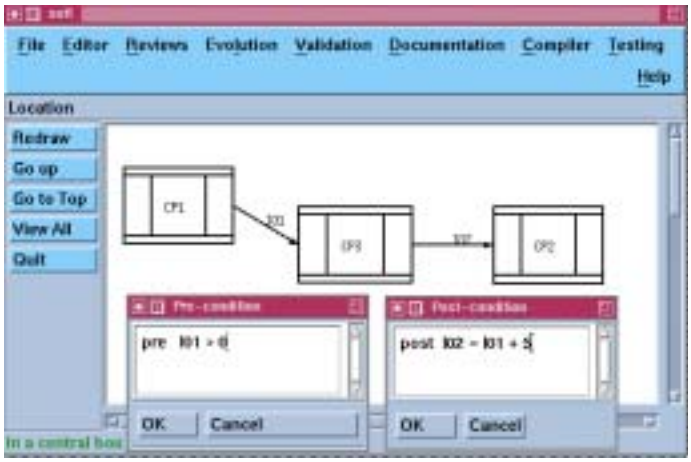

Figure 18: Graphical user interface

The final project is trying to develop an intelligent software engineering environment to support the SOFL language and methodology. The main concept is to build a knowledge base that contains rules for *guiding* the developer to conduct activities such as constructing CDFD, decomposing, and transforming [15]. We have developed a prototype in C under X Windows and Unix on Sun workstations that provides a graphical user interface and basic support for the construction of SOFL specifications. The prototype includes services for drawing condition process boxes and data flows between condition processes, redrawing diagrams, deleting components of diagrams, resizing diagrams, moving components, generating an outline specification module for each CDFD, and accessing precondition and postconditions of each condition process by mouse clicks on the corresponding boxes in the diagram. Figure 18 shows the layout of the graphical user interface and a simple CDFD with the precondition and postconditions of the condition process CP3 given in its associated specification module. We are improving the graphical user interface and support tool, and expanding it to support syntax and type checking of specification modules.

6 Acknowledgments

We would like to thank Professor M. Clint and D. Crookes of The Queen's University of Belfast of the U.K. for their constructive comments on the initial version of this paper, and research student Glenn Evans for the implementation of the prototype of the SOFL graphical user interface. We would also like to express our special gratitude to three anonymous referees for their critical and constructive comments and suggestions on the revision of this paper. This work is supported in part by the Ministry of Education of Japan under Joint Research Grant-in-Aid for International Scientific Research FM-ISEE (08044167) and by Hiroshima City University under Hiroshima City University Grant for Special Academic Research (International Studies) SCS-FM (A440). This work is also supported by a Visiting Research Fellowship (for Shaoying Liu) and a NIDevR grant (for Yong Sun) from The Queen's University of Belfast of the U.K., and by the National Science Foundation under grant CCR-93-11967 (for Jeff Offutt). We thank these organizations for their support.

References

[1] Shaoying Liu, Victoria Stavridou, and Bruno Dutertre. The Practice of Formal Methods in Safety Critical Systems. *The Journal of Systems and Software*, 28:77–87, 1995.

[2] Dan Craigen, Susan Gerhart, and Ted Ralston. Formal Methods Reality Check: Industrial Usage. In *FME'93: Industrial-Strength Formal Methods*, pages 250–267, Odense, Denmark, April 1993. Springer-verlag.

[3] Daniel Jackson and Jeannette Wing. An Invitation of Formal Methods. *IEEE Computer*, pages 16–30, April 1996.

[4] The VDM-SL Tool Group. Users Manual for the IFAD VDM-SL tools. Technical Report IFAD-VDM-4, The Institute of Applied Computer Science (IFAD), December 1994.

[5] A. Bloesch, E. kazmierczak, and M. Utting. The Sum Environment in Ergo: A Tutorial. Technical Report 96-22, Software Verification Research Centre, The University of Queensland, September 1996.

[6] Judy Crow, Sam Owre, John Rushby, Natarajan Shankar, and Mandayam Srivas. A Tutorial Introduction to PVS. In *WIFT '95: Workshop on Industrial-Strength Formal Specification Techniques*, April 1995.

[7] Dave L. Parnas and David M. Weiss. Active Design Reviews: Principles and Practices. In *Proceedings of the 8th International Conference on Software Engineering*, pages 215–222, Aug. 1985.

[8] M. E. Fagan. Design and Code Inspections to Reduce Errors in Program Development. *IBM Systems Journal*, 15(3):182–211, 1976.

[9] J. C. Knight and E. A. Meyers. An Improved Inspection Technique. *Communications of the ACM*, 36(11):50–61, Nov. 1993.

[10] Shaoying Liu. Evolution: A More Practical Approach than Refinement for Software Development. In *Proceedings of Third IEEE International Conference on Engineering of Complex Computing Systems*, pages 142–151, Como, Italy, September 8-12, 1997. IEEE Computer Society Press.

[11] P. Stocks and D. Carrington. A Framework for Specification-based Testing. *IEEE Transactions on Software Engineering*, 22(11):777–793, November 1996.

[12] E. Weyuker, T. Goradia, and A. Singh. Automatically Generating Test Data from a Boolean Specification. *IEEE Transactions on Software Engineering*, 20(5):353–363, May 1994.

[13] A Jeff Offutt and Shaoying Liu. Generating Test Data from SOFL Specifications. *Journal of Systems and Software*, (to appear).

[14] A. Bloesch, E. Kazmierczak, P. Kearney, and O. Traynor. Cogito: Methodology and System for Formal Software Development. *International Journal on Software Engineering and Knowledge Engineering*, 5(4):599–618, 1995.

[15] Shaoying Liu. Formal Methods and Intelligent Software Engineering Environments. Technical Report HCU-IS-95-006, Hiroshima City University, 1995.

[16] Antoni Diller. *Z: An Introduction to Formal Methods*. John Wiley & Sons, 1994.

[17] John Dawes. *The VDM-SL Reference Guide*. Pitman, 1991.

[18] L.T. Semmens, R.B. France, and T.W.G. Docker. Intergrated Structured Analysis and Formal Specification Techniques. *The Computer Journal*, 35(6), 1992.

[19] A. Bryant. Structured Methodologies and Formal Notations: Developing A Framework for Synthesis and Investigation. In *Proceedings of Z User Workshop*. Oxford 1989, Springer-Verlag, 1991.

[20] M.D. Fraser *et al.* Informal and Formal Requirements Specification Languages : Bridging the Gap. *IEEE Transactions on Software Engineering*, 17(5):454–466, May 1991.

[21] Nico Plat, Jan van Katwijk, and Kees Pronk. A Case for Structured Analysis/Formal Design. In *Proceedings of VDM'91, Lecture Notes in Computer Science*, volume 551 of *Lecture Notes in Computer Science*, pages 81–105, Berlin, 1991. Springer-Verlag.

[22] Shaoying Liu. A Formal Requirements Specification Method Based on Data Flow Analysis. *The Journal of Systems and Software*, 21:141–149, 1993.

[23] Shaoying Liu. Internal Consistency of FRSM Specifications. *The Journal of Systems and Software*, (2):167–176, May 1995.

[24] E.H. D'urr and J. van Katwijk. VDM++, A Formal Specification Language for Object Oriented Designs. In *Conference Proceedings Tools Euro'92 in Technology of Object-Oriented Languages and Systems, Tools 7*. Prentice Hall International, 1992.

[25] S. R. L. Meira and A. L. C. Cavalcanti. Modular Object-Oriented Z Specifications. In C. J. van Rijsbergen, editor, *Workshop on Computing Series*, Lecture Notes in Computer Science, pages 173–192, Oxford, U.K., 1990. Springer-Verlag.

[26] P. Coad and E. Yourdon. *Object-Oriented Design*. Yourdon Press Computing Series, Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

[27] Shaoying Liu and Yong Sun. Structured Methodology + Object-Oriented Methodology + Formal Methods: Methodology of SOFL. In *Proceedings of First IEEE International Conference on Engineering of Complex Computer Systems*, pages 137–144, Ft. Landerdale, Florida, U.S.A., November 6 - 10, 1995. IEEE Computer Society Press.

[28] Karen E. Huff. Software Process Modeling. In Alexander Wolf and Alfonso Fuggetta, editors, *Software Process*, volume 5 of *Trends in Software: Software Process*, pages 1–24. John Wiley & Sons., 1996.

[29] W.W. Royce. Managing the Development of Large Software Systems. In *Proceedings of IEEE WESCON*, pages 1–9, 1970. Reprinted in Thayer, R.H. (ed.) IEEE Tutorial on Software Engineering Project Management, 1988.

[30] Barry W. Boehm. A Spiral Model of Software Development and Enhancement. *IEEE Computer*, pages 61–72, May 1988.

[31] Chris Ho-Stuart and Shaoying Liu. An Operational Semantics for SOFL. In *Proceedings of The 1997 Asia-Pacific Software Engineering Conference (to appear)*, Hong Kong, 1997. IEEE Computer Society Press.

[32] Tom DeMarco. *Structured Analysis and System Specification*. Yourdon Inc., 1978.

[33] Edward Yourdon. *Modern Structured Analysis*. Prentice Hall International, Inc., 1989.

[34] C. Morgan. *Programming from Specifications*. Prentice-Hall International(UK) Ltd., 1990.

[35] Anthony Hall. Using Formal Methods to Develop an ATC Information System. *IEEE Software*, 13(2):66–76, March 1996.

[36] Shaoying Liu and Chris Ho-Stuart. Semi-Automtic Transformation from Formal Specifications to Programs. In *Proceedings of the International Conference on Engineering of Complex Computer Systems*, pages 506–513, Montreal, Quebec, Canada, October 21-25, 1996. IEEE Computer Society Press.

Appendix A

The specification module for the CDFD in Figure 12:

s-module Manage-Information-Decom1 : Figure 12;

type

ChangeRequest = **composed of**
name: **string**
address: **string**
period: Period
room-type: RoomType
end ;

CancelRequest = **composed of**
name: **string**
address: **string**
end ;

PersonalInformation = **composed of**
name: **string**
age: **nat**
arrival-date: Date
address: **string**
nights-of-stay: **string**
nationality: **string**
passport-no: **nat**
end;

CustomerList = **map** PersonalInformation **to** CustomerStatus;

CustomerStatus = **composed of**
room-no: **nat**
room-type: **nat**
deposit: **nat**
rate: **nat**
check-in-out: {IN, OUT}
end ;

ReservationList = **map** Reservation **to** Rooms;

Rooms = **set of** Room;

Room = **composed of**
room-no: **nat**
room-type: RoomType
end ;

var

change-req: ChangeRequest;

cancel-req: CancelRequest;

```
personal-inf: PersonalInformation;

room-no: nat;

rlist: ReservationList;

customer-list: CustomerList;

rooms: Rooms;

 c-process Reserve(res-req: Reservation)
  ext                wr rlist: ReservationList
                     rd rooms: Rooms
  pre                res-req notin dom (rlist) and
                     is-possible(res-req.room-type, res-req.period, rlist, rooms)
  post               let r = obtain(res-req, rlist, rooms) in
                     rlist = override (~rlist, {res-req -> r})
  decomposition      i-module
  comment
                     This condition process takes a new reservation request and makes a reservation
                     if the required room is available for the required period.
 end-process;

 function is-possible(room-type: RoomType, period: Period, rlist: ReservationList, rooms: Rooms): bool
  ==    exists[   r inset rooms | r.room-type = room-type and
                  not exists[c inset dom (rlist) | is-overlapped (c.period, period)
                  and rlist(c) = r]
        ]
 end-function;

 function obtain(res-req: Reservation, rlist: ReservationList, rooms: Rooms)rm: Room
  pre     is-possible(res-req.room-type, res-req.period, rlist, rooms)
  post    let vacancy = {y : Room | y.room-type = res-req.room-type and
          not exists[c inset dom (rlist) | is-overlapped(c.period, res-req.period) and rlist(c) = r]}
          in
          rm inset vacancy
 end-function;

 function is-overlapped(p1: Period, p2: Period)re: bool
  pre     true
  post    if day(p2(1)) >= day(p1(1)) and
          day(p2(2)) <= day(p1(2)) or
          day(p2(2)) >= day(p1(1)) and
          day(p2(2)) <= day(p1(2))
          then re = true
          else re = false
 end-function;

 function day(p1: nat)re: bool
  ==    undefined
 end-function;

/* The function day is undefined here and is left for implementation */
```

c-process Change(change-req: ChangeRequest)
ext **wr** rlist: ReservationList
rd rooms: Rooms
pre **exists**[r **inset dom** (rlist) | r.name = change-req.name **and**
r.address = change-req.address] **and**
is-possible(change-req.room-type, change-req.period, rlist, rooms)
post **let** x = get(change-req.name, change-req.address, rlist) **in**
rlist = **override** (~rlist, {x -> ~rlist(x)})
decomposition **i-module**
comment
The precondition requires that the person to make a change on the reservation must exists on the reservation list and there is vacancy as well. The postcondition describes that a required change is made on the reservation list.
end-process;

function get(name: **string** , address: **string** , rlist: ReservationList) re: ReservationList
pre **true**
post **let** r : **dom** (rlist) | r.name = name **and** r.address = address **in**
re = r
end-function;

c-process Cancel(cancel-req: CancelRequest)
ext **wr** rlist: ReservationList
pre **exists**[r **inset dom** (rlist) | r.name = cancel-req.name **and**
r.address = cancel-req.address]
post **let** x = get(cancel-req.name, cancel-req.address, rlist) **in**
domdl ({x}, rlist)
decomposition **i-module**
comment
...
end-process;

c-process Check-In(personal-inf: PersonalInformation)
ext **rd** rlist: ReservationList
wr customer-list: CustomerList
pre **exists**[r **inset dom** (rlist) | r.name = personal-inf.name **and**
r.address = personal-inf.address]
post **let** x = get(personal-inf.name, personal.address, rlist) **in**
let cus = getm(CustomerStatus) **in**
let y = modify(cus, room-type -> personal-inf.room-type, check-in-out -> IN)
in
customerlist = **override** (~customer-list, {x -> y})
]
decomposition **i-module**
comment
The person who check in the residential suites must have a reservation in advance. The condition process adds this person to the checked in customer list.
end-process;

c-process Check-Out(room-no: nat)check-out-bill: CheckOutBill

ext	**wr** rlist: ReservationList **wr** customer-list: CustomerList
pre	**exists**[cus **inset rng** (customer-list) | cus.room-no = room.no **and** cus.check-in-out = IN
post	**let** cus: **rng** (customer-list) | cus.room-no = room-no **and** cus.check-in-out = IN **in** customer-list = **rngdl** (˜customer-list, {cus}) **and** **exists**[p **inset dom** (customer-list) | customer-list(p).room-no = room-no **and** rlist = **domdl** ({p}, ˜rlist)] **and** check-in-out = produce-bill(cus, customer-list, rlist)
decomposition	**i-module**
comment	The precondition requires that the person to check out be on the check in customer list. The postcondition imposes the following three requirements: (1) the check out customer is removed from the checked in customer list. (2) the check out customer is removed from the reservation list. (3) the check-out-bill is generated by applying the function "produce-bill" whose definition is omitted.

end-process;

end-module;