FlexOr Technology Series

Monograph On
Jackson System Development

Gunnar Gotshalks

Original: 1994 May
Latest revision: 2015 May

Department of Electrical Engineering and Computer Science
Lassonde School of Engineering
York University
4700 Keele Street
North York, Ontario
M3J 1P3
gunnar@cse.yorku.ca
# Table of Contents

1 Overview .................................................................................................................................................. 1
  1.1 What are the problems in creating accurate models? ................................................................. 1
  1.2 How does JSD address the problems? ....................................................................................... 2

2 Development Method .......................................................................................................................... 5

3 Entity-Event Step .................................................................................................................................. 7
  3.1 Events .............................................................................................................................................. 7
  3.2 Entities ............................................................................................................................................ 8
  3.3 What to do ....................................................................................................................................... 8
  3.4 General considerations ................................................................................................................. 9

4 Entity Structure Step ........................................................................................................................... 11
  4.1 Structure diagrams and structure text ....................................................................................... 11
  4.2 Copy file design .................................................................................................................................. 17
  4.3 Line repacking design ................................................................................................................ 20
  4.4 Telegram analysis design ........................................................................................................ 21
  4.5 Other considerations .................................................................................................................. 25
  4.6 Handling complex entities ...................................................................................................... 29
  4.7 Structuring guidelines .................................................................................................................. 33
  4.8 Design for errors .......................................................................................................................... 33
  4.9 Summary of structure clashes ................................................................................................. 35

5 Initial Model Step ................................................................................................................................ 37
  5.1 System specification diagram ................................................................................................. 37
  5.2 Elaboration models .................................................................................................................. 39
  5.3 Merge problem .......................................................................................................................... 41
  5.4 Time grain markers ................................................................................................................... 44
  5.5 Process creation and deletion .................................................................................................. 45

6 Function step ......................................................................................................................................... 51
  6.1 Types of functions ...................................................................................................................... 51
  6.2 Miscellaneous considerations .................................................................................................. 53

7 System Timing step ................................................................................................................................ 53

8 Implementation Step .......................................................................................................................... 57
  8.1 Communications ......................................................................................................................... 57
  8.2 Reducing the number of processes .......................................................................................... 58
  8.3 Schedulers ....................................................................................................................................... 62
  8.4 Buffering ........................................................................................................................................ 64
  8.5 Reducing process memory requirements .................................................................................. 66
  8.6 Backtracking ............................................................................................................................... 71
  8.7 A variation on Backus-Naur Form ............................................................................................ 74
  8.8 Line justification: an example design ...................................................................................... 79
  8.9 Other considerations ................................................................................................................... 81
1 Overview

The primary purpose of system development is to build a model of a real world system. A definition of a model is the following statement.

\[ M \text{ is a model of a system } S, \text{ if } M \text{ can be used to answer questions about } S \text{ with accuracy } A. \]

If the model accurately reflects the real world, then the answers the model gives are applicable to the real world and we say the model is correct. If the model is inaccurate, then we get answers that are incorrect in that they are inapplicable to the real world\(^1\). All computer programs are models of the real world – the inputs are questions about the model and the outputs are the answer to those questions.

There are two fundamental types of models – the static and the dynamic. In static models time is irrelevant. For example, a database models a collection of objects and the relationships among them. One can ask questions about the objects and relationships but questions involving time cannot be asked; such as the following.

1. Was object A entered into the database before object B?
2. How many times has object A been modified?

Some databases such as a census do not even permit change. They are completely static. Other examples of static modeling are scene recognition, voice recognition and parsing textual information.

In dynamic models time is of central importance. We want to be able to ask the above questions and many others about the time ordering of events. Time is an important characteristic of many systems; the meaning of events may be different depending upon when they took place. In banks, accounts must be opened before monies can be deposited. A telephone rings before a person can answer. A sandwich is made before it can be eaten. The correct number of seconds must elapse before a pacemaker will jolt the heart. Rent is paid depending upon how much time has passed.

We are interested in dynamic models. JSD focuses on real-time modeling, and concurrent software where processes must communicate and synchronize with each other. JSD addresses most of the software development cycle – from specification to code. JSD specifications consist mainly of a distributed network of sequential processes that communicate by message passing and read only inspection of each other’s data (similar to your viewing a thermometer or clock). Model specifications are taken down to the program text level; hence, they are in principle executable. There is, however, typically are large mismatch between the resources specified by the model and those actually available, so the JSD implementation step transforms, without altering the design, the specification into one that will execute with the available resources.

1.1 What are the problems in creating accurate models?

Understand the real world domain

The first problem is having sufficient understanding of the aspect of the real world we wish to model. We cannot model all aspects of the real world but instead we must abstract those features that are of interest\(^2\). System requirements are an informal description of what questions and answers are desired. System specification is a formal description of what questions and answers are desired. Creating system specifications requires extensive domain knowledge.

---

\(^1\) All models are correct in the trivial sense that the answers always correspond to the questions according to the model. We are not, however, interested in arbitrary models but are interested in understanding the real world and making use of our knowledge of the real world. Consequently, we want models that accurately reflect our understanding of the real world.

\(^2\) The only one to one model of the real world is the real world itself. Consider having a one to one model of a house.
Evolution of models

The second problem is our understanding and perception of the real world changes over time. As our understanding changes so must our models change. A fundamental requirement of all systems is that they evolve over time. The questions we ask and the answers we expect to those questions will change. Consider, for example, taxation programs. As government budgets are passed tax rates and tax brackets change – changing answers to the same question. As deductions are created or taken away, as new types of taxes are created or old types are disposed of, the questions change.

Model implementation

The third problem is that a model must be implemented on available hardware using available software. The model specification must be translated into executable program text and we must be sure that the translation maintains the semantics of the specification. This is a non-trivial problem because we must match available resources to the model to meet space and time requirements.

A fundamental premise of JSD is that every object is modeled by its own process. JSD programs have a very large number of processes. For example, a bank has millions of customers. Each customer is modeled by its own process. There is one process for each account for each customer. Each elevator button for every elevator both inside the elevators and at each floor is a separate process. A mapping between the large number of processes and the small number of processors must be created – standard multiprocessing operating systems are not always adequate to the task.

The large number of processes frequently implies a large amount of redundant program text. The redundancy needs to be eliminated.

The objects modeled in systems persist for a long time. For example, a person can be a customer for a bank for 50 years and more. This means the corresponding process modeling the customer must run for the same period of time. A mechanism must be devised to remember, for many years, the state of each object and where they are in their process text.

1.2 How does JSD address the problems?

Understand the real world domain

A system developer will spend hundreds of hours in dialogue with the system users eliciting the requirements and creating the specifications. A major portion of the dialogue takes place at the beginning of system development but in JSD the dialogue continues until all user decidable issues have been resolved. The purpose of the dialogue is to resolve issues about what to model and what is an appropriate model of the system at every level of detail that directly corresponds with the real world. We must verify our models to show that they are accurate. The JSD method is conducive to building and verifying models. Some aspects of JSD can be formalized in CSP (Communicating Sequential Processes) and mathematical methods, including the use of automated theorem proving assistants in verification efforts.

Specifications are an abstract model of the system because they are logically correct but not directly executable. Specifications in JSD, however, are executable in principle assuming appropriate hardware and software resources.

Evolution of models

The JSD method separates abstract modeling into two phases. The first phase builds the model framework (superstructure or architecture) based on the structure and properties of real world objects and events. This part of the model is relatively static as it changes slowly over time. The second phase embeds functions within the framework, the specific questions we want answered. This part of the model is relatively volatile as it changes rapidly over time. Without a

---

3 This arises naturally as it is simpler to assume a unique process per object instance to reason about a system, than to have a single process modeling a set of objects. This is a natural division point between abstract and concrete models.
correct model it is arbitrarily difficult to write a given function and it is frequently impossible to predict the amount of difficulty in advance.

**Communicating processes**

In the past you have been asked and taught to write one process to solve one problem. Now you will learn that there are problems where it is difficult to write one process but relatively easy to write processes for sub-parts of a problem and then combine the sub-parts into one executable whole – a system in effect. For example, the notion of abstract data types show that properly designed classes and subprograms can simplify programming. Similarly, properly designed communicating processes can simplify programming a system.

When a program is partitioned into parts, the parts must execute as a unit. The parts must communicate – that is pass data among each other. In the case of subprograms, communication takes place through parameters and the returned value of functions.

How else can programs communicate? Programs communicate by reading and writing to files or queues (by analogy to people when they are sending letters or using E-mail), and by reading to and writing from shared memory locations (by analogy to people using a blackboard or bulletin board, if we take care to preserve or use time order of message writing when we are reading).

If we use communications methods, then we do not think of one process as a slave and another as the master. All processes (people) are equal and simply exchange messages. The notion of communicating processes leads to simpler system design and reduces the number and difficulty of changes we need to make when a system is modified.

**Model implementation**

Designing a system in JSD means to first construct an abstract model, and then construct a concrete model as a computer program. The implementation phase consists of the translation of the program text developed for the abstract model into executable program text, the concrete model.

The difficult part of the translation is to implement the communication channels among communicating processes and to map the large number of process unto a small number of processors. Techniques used are: use coroutines and process inversion (converting a process into a subprogram) to reduce the number of processes; use files, queues, common areas and subprogram parameters as communication channels to get efficient communications; where possible use standard operating systems for multiprocessing and where not possible writing custom process schedulers; use state-vector separation to combine many instances of process text into one copy, with the states stored in a database, to reduce the size of the executable system; use process dismemberment to split a process into smaller parts that may execute at different times or even in different environments (interactive and batch).
2 Development Method

Figure 2.1 shows the development steps using the JSD method. The leaf nodes are in general followed in a left to right order but feedback can occur at any time. Feedback often takes place between the “Specify model of reality” and “Specify system functions” in that some functions will require new entities and events to be modeled.

Figure 2.1: JSD steps in model construction.

Observe the bifurcation structure of the JSD method. At the top level, specification and implementation are separated with implementation being a single step. JSD distinguishes between specification (design) and implementation, while traditional methods distinguish between analysis and programming. In JSD all of the program text is developed in the specification step before the implementation step.

The specification step creates all the abstract program text because, until it is fully developed, parts of the system may be open to choices and different interpretations in the problem domain. Specifications are complete once all abstract modeling choices have been eliminated. In principle, an executable specification is created. In effect, the specification is carried along with the program. Proof, verification and testing go hand in hand with the creation of the specification.

The implementation step maps the program text onto the available software and hardware. No model program text is created. The only program text created is to realize the model, the glue to fit the parts together into an executable whole. Once an implementation structure is decided upon the translation proceeds in a mechanical manner. Because implementation usually means mapping many processes onto a single processor, the implementer has to worry about scheduling and may have to abandon general operating systems because they have too much overhead, lack of control or undesirable consequences. Scheduling decisions influence the type of system and its structure, hence scheduling is a part of design. Processes must communicate to work together.
The implementer may have to customize process communication because generally available methods are not sufficient.

Continuing with the bifurcation of the method at the specification level another split is made between the model framework (specify model of reality) and the questions the system will be asked (specify system functions). Specifying the functions deals first with the questions (functions) and then the timing considerations.

Specifying the model framework requires first modeling each entity type (describe abstract reality), then interfacing each object model with the real world and with each other (construct a realized model).

Specifying each entity type requires identifying the objects of interest and the events in which they can participate (entity event step), then specifying the time ordering of events for each object (entity structure step).

A rule of thumb is to defer what needs to be deferred. The ordering of the steps in JSD minimizes iteration by using the following guidelines

- Every decision is made explicit and fully documented at the time of the decision. That is the reason for developing program text as the specification is developed.
- Decisions are made in a logical order; later decisions are not preempted by earlier decisions. Contrast with top down approaches where top level major decisions are made before much of the system is understood.
- Descriptive decisions, the easier ones, are made early, the inventive decisions, the difficult decisions, are made later.
- Errors in design decisions are detected at the earliest opportunity before much costly additional work has to be thrown away.
- The most major decisions are orthogonal. They can be made independent of each other. This leads to more robust and more versatile system design, which in turn leads to easier evolution and more overall efficiency.

The remaining chapters discuss each of the JSD development steps in left to right order in Figure 2 – the primary order in which they are followed during development.

---

4 This is the major reason, in top down or functional design, for the design rule to throw away the first system. The building of the first system is used to learn and understand it, and then with hindsight the proper decisions can be made at the top levels of top down design. In JSD there is no top level to design down from, so there is no need to use hindsight to build a second system.
3 Entity-Event Step

In constructing a model the first decision is to decide what to include in the model and what to exclude from the model – determine the model boundary, Figure 3.1. Which in turn means determining what is inside and what is outside the boundary. Since the purpose is to model an aspect of the real world, the system under development is never included as a part of the model. The system under development is not what we want to model.

Stating what is inside or outside the model boundary describes what the model is about. Dynamic models are about objects and the actions in which they engage. The questions we want to ask are about the objects (static questions) or about the actions and results of actions (dynamic questions) in which they engage. Before we can ask the questions (design functions) we must know what objects and actions will exist in the model. If appropriate objects and actions exist, then adding functions will be easy. If appropriate objects and actions do not exist, then adding functions will be arbitrarily difficult.

In JSD we use the term entity when referring to an object. The terms event and action are used interchangeably. The precise term is event but as many people are used to anthropomorphic analogies the term action often feels more natural.

Entities are defined by the set of events in which they participate – no events, no entity. Entities have a lifetime, a history, based on the time sequence of events in which they participate. Since events characterize entities we first examine what we mean by an event.

3.1 Events

JSD events have the following properties.

- They occur at one instant in time. There is no notion of duration. A person can engage in the events start to eat and finish eating. But there is no corresponding event eating because eating takes time – namely the duration between the event start to eat and the event finish eating.

---

5 There is a correspondence with objected-oriented programming and JSD between abstract data types consisting of objects and operations on those objects and JSD entities and events.
• Events always take place in the real world, external to the model. Events that take place within the model are not considered.
• Events are atomic, indivisible. They have no substructure.
• Events can be common among entities.

Events are different from states. Objects are in a state between state defining events. For example a person is in the state of eating between the events start eating and finish eating. Another example is an elevator button that is in the state of being depressed between the press and release events. In JSD, if a particular state is important in a model, then the defining end point events must be a part of the model. Events can be as fine grained as necessary.

If some action in the real world takes a long time, we can consider the action as an event, if we treat it as instantaneous. By treating an event as instantaneous it becomes meaningless to ask questions such as when did the event begin and when did the event end. One can only ask when did an event occur relative to other events.

### 3.2 Entities

It is not sufficient to simply specify a set of events as defining an entity. A correctly defined entity in JSD has the following properties.
• Engages in events in a time-structured sequence. The events may be caused by the entity, such as enroll in a course, or experienced by the entity, such as hit by lighting.
• Exist in the real world and not be an artifact within the system.
• Be individually identified and distinguished from all other entities, even those of the same type.
• Participate in the same set of events over their entire lifetime. Entities cannot change types during their lifetime. The type of an entity is determined by the set of events in which it engages. A subset of the events in which an entity can engage is also thought of as the role the entity plays.

The first property defines a dynamic model – we can ask questions such as has the entity engaged in a particular event and how many times has it engaged in the event. Unless we can ask such time questions the entity does not belong in a JSD model. The second property makes sure that the model does not model parts of itself. The third property supports the first property in that if we cannot distinguish between entities then we either must treat them as a single collective entity or give up on time based questions, in which case JSD is an inappropriate modeling method.

### 3.3 What to do

**Gather raw data**

Analyze the aspect of the real world you are modeling and create a natural language description of what is to be modeled together with a list of questions or potential questions that are to be answered. Out of all the material collected through reading, thinking, interviews and discussion the following three lists are created. At this point be all-inclusive. It is better to err by including too much than by including too little. It is easy to throw away but difficult to include once some early decisions have been made.
• A comprehensive list of nouns – potential entities
• A comprehensive list of verbs – potential events
• A comprehensive list of questions and potential questions the model is to answer

---

6 Internal events are not considered in the entity-event step but they may arise in the function step.
Select entities and events

Associate sets of verbs with each noun. Every noun and associated verb set is thoroughly examined using the following as a guideline.

- Nouns are within the model boundary. They are of interest to model – necessary to answer the proposed questions – and are not a part of the system itself.
- The verbs associated with the nouns are proper events. All state like verbs are eliminated or replaced with defining pairs of events.
- The events are practically and economically detectable. If we cannot detect an event then we cannot model the action sequence of the corresponding entity, and we cannot ask questions involving the event.
- Entities are added if they are required to give a time order structure to events.
- Events are added to make other events meaningful.
- Be ruthless, every extra entity and event adds complexity to the system. Avoid synonyms. You must be confident that the entities and events are necessary to answer the desired questions.

Create the following document.

- For each proposed entity set document why it was accepted or rejected. Similarly document each event. This is useful for archive purposes when you forget later on and begin to wonder why some entities and events were rejected or accepted.

Document the selected entities and events

From the results of the previous step extract all the selected entities and events. The list is typically small relative to the size of the system – the bulk is typically in the input subsystem and in the functions. Document every entity by giving its name a general description followed by a detailed description of every event in which the entity participates.

3.4 General considerations

Entities are often complex and have different roles to play at different times. Sometimes each role is identified as different entities. For example a person may be an instructor at a university and a student at the same university. You have to be careful about whether these are to be treated as two different entities – thereby not being able to ask questions that require combining information about the two entities. Or, treated as one entity that engages in different action sets depending upon which role is being modeled. The following classifications should be considered.

Collective entities

It is often necessary to choose between modeling an individual, or the group to which the individual belongs, or even to model both the individual and the group with some common events. The events determine which way to go. And the events depend upon the questions to be answered.

Generic entities

Are different instances needed or not. Aircraft engines and other major parts of an aircraft are given serial numbers and are tracked. Others parts, such as ball bearings, are not distinguished or tracked over time. In a model of airplane parts, engines would be treated as individual entities of type engine. On the other hand, ball bearings would be treated as a generic type and only global properties of the ball-bearing type would be recorded and reported upon.

Common events

A common event is an event in which two or more entities participate – for example, a collision between two cars. Common events can occur between entities of the same type (such as two or more cars in a collision) or different types (an interview between a policeman and a car
Depending upon the role the entities play in the action, it may be necessary to give the same name or different names to the event.

It is completely up to you, the model builder, to decide whether or not to model common events. It depends upon the purpose of the model. You can treat common events as two distinct events but then will not be able to ask questions that combine the two entities. For example, in multi-car collisions an insurance company can regard the collision of each car as a separate event covered by a separate policy. It will then be difficult to find sets of cars that were in the same accident.

Sometimes apparently common events are distinct. Imagine a panel awarding prizes and a contestant winning, they appear to be the same event. But it may be possible that a third entity an appeal court, boss, chief judge may overrule the panel award and the contestant does not win. Writing a cheque is not necessarily the same event as making a payment, it occasionally happens that cheques bounce and no payment is made. Consequently, you have to carefully consider apparent common events to discover whether or not you want to model them as being common or separate.

As a rule of thumb, do not treat two actions as being common, if one can occur without the other. See the previous examples and consider the following three events – deliver (joint between company and customer), remove-from-warehouse (company only, customer not involved), give-to-customer (company not involved, only customer). Should deliver be considered a joint event or should there be two events remove-from-warehouse and give-to-customer which together indicate a delivery was made.

**Undetectable events**

While this is not usually tricky it is easy to let undetectable events slip by. Watch for things like get into and get out of an elevator. How do you know which person got in or out? How do you know whether they really did get in or get out?

**Entities, functions and costs**

The more entity types there are in a system the more complex and expensive it is, so try to keep the number of entity types down. Adding entities drive costs up. On the other hand, if entities are missing the associated functions will be difficult to provide. If such functions are anticipated in the near term, it may be more cost effective adding associated entity types at the model structure stage than adding them later.

Notice that no functions are introduced at this time. They are added later. We are building the framework for the system in which functions will be embedded.
5 Initial Model Step

The purpose of the initial model step is to connect the models (abstract description) of entities and events created in the entity-event step and the entity structure step to the real world. The abstract descriptions are regarded as sequential processes and their analogs in the real world are sequential processes so you connect a collection of sequential processes. Once the processes are connected, you have an executable system in principle. The system does not produce any output. The system engages the corresponding real world events with some nonzero time lag.

5.1 System specification diagram

The system specification diagram (SSD), along with suitable textual documentation, is the artifact that you create. The entities in the external world are referred to as level-0 processes. The corresponding model processes are referred to as level-1 processes. Any marsupial entities are also level-1 processes, since they are in the system and not in the real world. The processes are diagrammed with rectangular boxes labeled with the process name. Figure 5.1 shows the processes for customers and clerks in a store, with orders being a marsupial entity. In the SSD there is one box for each type of entity.

The process boxes are connected with either a data stream or state-vector connection, depending upon who initiates the message transfer between the processes.

5.1.1 Data stream connections

Data stream connections are shown in Figure 5.2. They represent an unbounded queue of messages. The sender is active and the receiver is passive. The sender creates the messages and determines when they are sent (send message). The receiver can read (receiver message) at any time but must read all the messages in the same order as they were sent.

---

1 The system is not directly executable on existing hardware and software due to a mismatch in characteristics – too many processes, too long an execution time, too much assumed buffering in the connecting communication channels. In the implementation step you resolve these mismatches by applying assorted mechanical transformations to achieve an executable system with desired characteristics.

2 At this point in system development you concern yourself with making sure the system is logically correct without consideration of timing, you will deal with timing in the timing and implementation steps.
There is a 1:1 connection between the real world entities customer and clerk and corresponding system entities. There is a 1:N connection between the customer and order processes and a 1:N connection between the clerk and order processes – the short vertical bars indicate the “N” part which can occur at either or both ends simultaneous, if you want N:1 or N:N connections. In all further examples of SSDs the model boundary is left out. Data streams are labeled with name within a circle. Marsupial entities are always connected to the corresponding level-1 entities.

Figure 5.2: Data stream connections for processes in Figure 5.1.

Data stream connection is the preferred connection between processes because messages can never be lost no matter what the relative speed of the sender and receiver processes. The sender is never blocked. The receiver is blocked only if there are no messages to read.

5.1.2 State-vector connection

A state-vector connection is shown Figure 5.3. The sender, a clock, is passive. The receiver, a person, is the initiator. The sender creates data at its own speed but it is the receiver who determines when to read the data (getsv – get state-vector – from a common message area). Because the receiver is active the following properties hold for state-vector connections.

1. Receiver may miss changes so must inspect sufficiently often. In Figure 5.3 the person must read the clock to determine at what time to act – for example when to leave to catch the train. If the person falls asleep or doesn’t come back from some other task soon enough they may miss the train.
2. Receiver may read the same state multiple times and must be prepared to reject the duplicates or unneeded states. In Figure 5.3, an anxious person will examine the clock every few seconds or minutes.
3. State must be consistent so must restrict getsv outside of the sender’s critical regions. Safest is to permit a getsv operation only when the sender is at a read, write, or getsv of its own. Can also use public and private state-vectors. The sender moves a private state-vector to the public message area as a single critical operation.
4. Sender can never block receiver unless the sender is in a critical region updating the state.
5. Sender is never blocked.

Figure 5.3: State-vector connection between a clock and a person. Person only knows the time by looking at the clock.
Entities may be connected with multiple message streams – all data stream, all state-vector or mixed. For example, an alarm clock has both a data stream and a state-vector connection to the user\(^3\).

State-vector connections are more difficult to implement correctly than data stream connections. In particular, model structures often need to be elaborated to take into account the loops that occur while a process waits for an event to occur – the event message is implied by a change in state.

### 5.1.3 Message stream structures

The structure of the sequence of messages can also be defined with JSP charts. This can be done for either data stream or state-vector connections. This is useful to verify that sender and receiver process structures correspond to the message stream structure and, hence, the sender and receiver correctly write and read the message stream. For example, Figure 5 shows a typical structure for a state-vector message stream.

![SV connection diagram](image)

Figure 5.4: A structure diagram for a state-vector connection.

### 5.2 Elaboration models

To correctly model a real world entity it is necessary to have a 1:1 correspondence of model events and real world events. The level-1 model must engage in every event that the real world entity engages in and must not engage in any extra events. The level-1 model structure must have the simplest possible structure that describes all possible event structures. Any other level-1 model would not be a correct model of the real world entity. It happens in some systems that you are not interested in a 1:1 model but interested in some elaboration of the model. That is you are interested in particular subsets of all possible sequences of a real world entity.

An elaboration model is a level-2 process connected, usually, by a data stream to the corresponding level-1 process. The level-1 process retransmits the messages of interest to the level 2 process; sometimes it is all the messages and sometimes it is a subset of the messages. For example, Figure 5.5 shows an SSD for a real world clock, a daily lunch at 12pm and an elaborated daily lunch at 12pm.

![SSD diagram](image)

Figure 5.5: SSD for the model processes in Figures 5.6 and 5.7.

\(^3\) Exercise: draw and describe an SSD for an alarm clock and a person interacting.
In Figure 5.6, a daily-lunch waits daily for 12pm. Because of the state-vector connection the level-1 process has to read extra not 12pm messages to detect the 12pm event.

![Diagram](image)

**Figure 5.6:** Structure elaboration for a daily lunch waiting for 12pm by inspecting a clock.

You may, however, be interested only in the 12pm events in Figure 5.6 but not in the non 12pm events. Figure 5.7 shows the structure of the level-2 process that only knows about the 12pm events. The Daily-lunch-1 process sends a message for 12pm events.

![Diagram](image)

**Figure 5.7:** Structure of a level-2 process corresponding to Figure 5.6.

Figure 5.8 shows the structure of daily-lunch-2 when all the events of Daily-lunch-1 are to be modeled but you also want to detect every second 12pm event. The Daily-lunch-1 process sends a message for every event it engages in.

The model step may show that additional actions are required to get a message to another process. For example in Jackson’s Daily Racket model a panel for selecting contest winners was required. The original events were “meet” and “award”, however, the panel needs a “disperse” event to get its decision out, otherwise there is no way of knowing when the panel is to send the message.

![Diagram](image)

**Figure 5.9:** Original level-1 model based on the entity-event step on the right, and on the left, the revised level-1 model structures for the panel after working on the model step. The disperse event was only detected at the model step when the designer hypothesized that a function reporting the number of awards made may be required.

---

4 It cannot send the message as an activity of meet because there would be no message on the first meeting. In addition, waiting for the next meeting would unduly delay the message.

5 An example showing that there is continuous feedback in developing a system, based on a continuous
5.3 Merge problem

The Figure 5.2 SSD of customer, clerk and order processes illustrates the data stream connection merge problem\textsuperscript{6}. How does the order process read the data streams? If it tries to read from an empty stream CO, it could be blocked forever waiting for a message that never comes; meanwhile clerk messages are being ignored on the stream ClO. The converse can happen with

\textsuperscript{6}There is no merge problem with state vector connections. The receiver reads the state vectors in any order it needs to without getting blocked because it is impossible to not have a message to read.
an empty CIO stream. Even when both streams have messages, in what order are the streams to be read: read until one is empty, read alternately, or what other pattern.

In any significant system the merge process will appear many times. It is up to you, the designer, to specify a solution to each merge problem. The following is a summary of possible solutions.

5.3.1 Fixed merge

Read data in a predetermined order among the input connections. If no message is on a data stream, then wait until one appears. This solution is feasible when messages arrive frequently enough that there is no significant blockage and the fixed order is acceptable. An example would be answering a collection of telephones on a talk show. The phones could be answered in a fixed sequence, usually there are enough callers that all the lines are busy all the time and the order in which callers are accepted is unimportant. Presumably the lines would be allocated in the same sequence as callers dialed in to prevent blockage at an unused line.

Figure 5.10 shows a typical SSD for a fixed merge.

![SSD for a fixed merge](image)

Figure 5.10: SSD for a fixed, data and periodic merge.

Figure 5.2 shows an SSD for a fixed merge at the order-1 process. Accompanying documentation would describe the merge and ultimately, the program text for the order event structure would specify a fixed merge.

5.3.2 Data merge

Need messages on all input channels. You need to “prime the pump” by reading the first message on all channels. The merge process selects one of the messages by a data value in the message, processes the message and reads the next message on the channel. The process repeats forever. The data merge is the basis of the sort-merge algorithm. As long as messages arrive frequently enough on all channels this solution works well. A typical data value to use is a time stamp so messages are processed in time order. An example could be a magazine contest where entries come in from various sources and the earliest correct responses are awarded prizes.

Figure 5.10 shows a typical SSD for a data merge.

Figure 5.2 shows an SSD for a data merge at the order-1 process. Accompanying documentation would describe the merge and ultimately, the program text for the order event structure would specify a data merge.

5.3.3 Periodic merge

The major problem with both the data merge and fixed merge solutions is getting blocked on empty input channels. The periodic merge alleviates this problem by inserting dummy messages (periodically) into each message channel, thereby guaranteeing input channels are empty for a bounded length of time, and then use data merge. A typical example is batch processing. At processing time a dummy message is inserted into each stream to force the merging and processing of all real data in all streams. Discarding it processes the dummy message.

Figure 5.10 shows a typical SSD for a periodic merge.

Figure 5.2 shows an SSD for a periodic merge at the order-1 process. Accompanying documentation would describe the merge. The customer and clerk processes would have additional events added so dummy messages could be sent periodically (see the section on time-
5.3.4 Rough merge

Rough merge avoids blocking by first testing for the presence of a message on an input channel. If a message exists, read the message. If there is no message, try another channel. The channels are often tested in a fixed (polling) order.

![Figure 5.11: A typical SSD for a rough merge](image)

Figure 5.11: A typical SSD for a rough merge

Figure 5.12 shows an SSD specifying a rough merge for the Order-1 process.

![Figure 5.12: A rough merge of CO and ClO channels into Order-1.](image)

Sometimes it is desirable to specify a separate process for the merge. Figure 5.13 is an example. The input to the merge process is indicated as a fixed merge. The documentation will specify the use of look ahead reads if a rough merge is used.

![Figure 5.13: SSD showing a separate merge process for order input.](image)

The problem is ensuring fairness as processing order is heavily dependent upon channel speeds. May starve some connections by polling from the “top” after each read, if the top few connections have rapidly arriving messages. Can also be unfair by not reading a particular channel often enough. For example, cycling through in a strict polling sequence could alternate data on two channels when the data from one of the channels should, occasionally, be read more frequently than the other channel; two customer messages may come in rapid succession without an interleaving clerk order. Rough merge always has indeterminacy and a degree of unfairness.
One to many, 1:N, and many to many, N:N, connections are always a rough merge. Thus, in Figure 5.13 the CO streams from each customer are rough merged into one customer stream before being merged by process Merge-C-C1. It is not meaningful, useful or possible to specify any other type of merge.

Rough merge is the usually merge solution. You, as designer, specify, in the timing step, the degree of unfairness that will be tolerated.

5.4 Time-grain markers

JSD is based on the time ordering of events and particular times are themselves events. After all, in most systems we need to schedule and count time. For example, you want to produce reports at the end of the day when the store is closed so closing time is an event. In other cases you may want to keep track of rental time in days or hours. In JSD there is no global clock, nor is it necessary to model or show a clock. Time is introduced by inserting time-grain markers periodically into a data stream. Documentation describes the nature and purpose of each time-grain marker stream. Time-grain markers are always rough merged with other data streams.

Figure 5.14 shows a structure chart of a process to count rental time and a partial SSD in which the process would be embedded.

---

7 Many, if not most, merges in real life are rough merges. Consider examples such as: who gets through a door when multiple people arrive simultaneously; who gets at the checkout counter; are you too late to return a library book; what time is on your parking lot ticket.
5.5 Process creation and deletion

Process creation and deletion is left as an implementation matter. Some processes are fixed for the duration of the system – e.g. elevators. Others vary, such as the number of customers, and number of orders. New level-1 processes, such as customers, are created outside of the system but are easily modeled. Marsupial entities, such as orders, and level-2 elaboration processes are created by the system – parents do the creation.

Deletion time is very hard to detect and is usually handled outside of the system such as archive time. Essentially, if no further communication will take place then the process can be deleted. But even if it can be shown that all of the processes data streams will cease to have messages either input or output, it can have its state-vector read at any time. Even if a process reaches its end of text, a state-vector read could take place. Consequently, a process cannot delete itself because it is unaware of all of the external circumstances that may use a state-vector read. Of course, if it can be known, by some mechanical means, that absolutely no further communication will take place of either data steam or state-vector type, then the process could be deleted by an automatic process.
6 Function step

After the initial model step you have, in principle, an executable system that models the real world but it does not have any output – it does not answer any questions. The model does have, however, the necessary structure in which to embed appropriate output functions. The purpose of the function step to add the question answering mechanism – add the output.

6.0.1 Functions are processes

Functions are always considered as processes not subprograms. They exist for the life of the system. Functions must be capable of computing cumulative effects – effects across many invocations of the function. Always consider output as the set of all reports over the lifetime of the system. For example, a monthly cheque writing function for a payroll system produces not just one set of cheques with the process being run once month, but is a process that produces a potentially unbounded number of sets of cheques, at the rate of one set each month. Another example is a bank balance enquiry function. The function is a process that reads a sequence of enquiries and produces a sequence of balances, not a function that is run once for each enquiry. Even the function of turning an elevator light on and off as the elevator passes a given floor is a lifetime function. If the process ends up being simple it can be implemented as a subprogram but it is designed as a process.

6.0.2 Model structure does not change

The model process structure can never change because of a function. The structure reflects the real-time ordering of events in which the model can engage. Functions can never affect real events. For model processes you can only add variables and operations to maintain information required for a function. For example: (1) in a customer process you can keep a count of how many times the customer engaged in a particular event; (2) keep a running total of some data item in an input message (running total of withdrawals and deposits at a bank); (3) keep track of whether an object is rented or not.

6.1 Types of functions

6.1.1 Embedded functions

Sometimes functions are so simple that they can be embedded in a model process. That is the output statements are placed within the process text. For example cumulative effects can be handled in the model process such as keeping a running balance of withdrawals and deposits and output a message whenever the balance becomes too small or large. Another example is the control of elevator lights in elevator models, they are turned on and off by embedded functions as the elevator model receives floor arrival messages. Figure 6.1 shows FO as the output data stream of an embedded function in the Model-1 process.

![Figure 6.1: SSD for an embedded function.](image)

Multiple instances of the model process imply multiple instances of the function process. There is one function process for each instance of the model process.

Until specified otherwise, each embedded function has its own output stream.
6.1.2 On request functions

On request functions must be independent processes because the request message must be merged with the data used by the function. The request message is not an event recognized by the model process. Requests are rough merged with data streams. Figure 6.2 shows a typical example. The function SR reports on all the data the process received since the last request.

![Figure 6.2: SSD showing rough merge of request with model data as input to a function.](image)

Requests are fixed merged with state-vector streams. Figure 6.3 shows a typical example. The function SR reports the current value of state variables in the process Model-1; e.g. current account balance.

![Figure 6.3: SSD showing fixed merge with state-vector data as input to a function.](image)

TGMs are used for periodic requests, and used for delay and timing.

6.1.3 Feedback functions

Feedback or interactive functions are possible. Feedback is normally rough merged. Example is depositing the interest on a bank account, Figure 6.4. A feedback function introduces internal events; therefore you must use structure elaboration to introduce a level-2 process connected to the level-1 process. The connection from the level-1 process echoes all the messages from the input. These are rough merged with function events. The level-2 process then describes the structure of the combined set of events.
6.2 Miscellaneous considerations

Functions can be connected to process using either state-vector or data connections. Choice depends on who has the initiative. Alarm clock, Figure 6.5, has both types of connections. A person can look at the clock to get the time (state-vector) and may hear the alarm for notification of a particular time (data stream connection). A heavy sleeper needs an alarm; a light sleeper needs a clock. Turning the alarm on and off is an example of turning a data stream on and off.

6.2.1 Text pointer

Never ever rely on the text pointer as a part of the state-vector for determining the state of a process modeling an entity. Always use variables to record the state and either transmit the value of the variable on a data stream connection to the function or have the function use a state-vector connection to read the variable. An example is how to know when an object is in a rented or available state – the model process has text statements \texttt{rented:=’yes’} and \texttt{rented:=’no’}, and the function reads the value of ‘rented’.
6.2.2 Tricks and design

Cleverness in design and tricks should be avoided. They are a sign you are not implementing the specification but implementing something equivalent. Usually this causes problems when the specifications must be adhered to more closely or the equivalence becomes false through a change somewhere else.

JSP (Jackson system programming) is used to structure functions. Their structure depends upon the sequence of messages they read and write.

6.2.3 Mutual exclusion

Mutual exclusion can be handled with data stream connections. Figure 6.6 shows how mutual exclusion is handled. All merges must be fixed. The data streams are read/written in the order indicated from top to bottom.

![Diagram of mutual exclusion](image1)

Figure 6.6: An SSD for exclusive use of a resource by multiple requestors.

Figure 6.7 shows a more complex example with multiple elevators accessing multiple buttons for requests. To ensure elevators do not interfere with each other, you want to have only one elevator at a time inspecting a button, receiving the state deciding what to do and issuing a promise or nopromise message to service the button request.

![Diagram of mutual exclusion](image2)

Figure 6.7: An SSD for multiple requestors and multiple resources.
6.2.3 Merging TGM streams

Merging TGM streams with other input streams cannot do timing. If there are delays in reading the TGM stream then a sequences of TGMs may be read too fast since they are already in the buffer. TGMs may accumulate while a process is waiting for input from some other stream in a fixed merge. Instead there are two other solutions.

The first solution is to specify a clock process with its state being read by the timing process – sufficiently often! – to determine a time interval or particular time. Figure 6.8 shows the SSD. The input to the clock is a regular pulse, which the clock can read without delay, insuring correct timing, since that is its only task. This solution permits AnyProcess to continue working on other tasks while waiting for an appropriate time or delay interval.

![Figure 6.8: SSD for reading a clock to determine timing.](image)

If the process has nothing to do or it is imperative for the process to be delayed from doing anything then the second solution in Figure 6.9 is recommended. The clock, dedicated to a single process, not only reads timing signals but also counts the appropriate interval after a begin message before output of the end message. This solution avoids the busy wait of the first solution.

![Figure 6.9: An SSD for delay timing.](image)

6.2.4 Indeterminacy

Indeterminacy is always a part of modeling. It is not a disadvantage. Full determinacy is based on implementation not specification. Time periods are used as convenient approximations. The timing step is used to control the amount of tolerable indeterminacy, which in general can never be reduced to zero. Only tighter and tighter control can be achieved at the expense of reducing concurrency and introducing overhead.

6.2.5 Access paths and databases

Assume databases contain required state-vectors. Can always design processes to specify the ordering and selection of the appropriate state-vectors. These would be used in the implementation step as necessary.
6.2.6 Sorting

Must be aware of delay in output when input needs to be grouped for sorting.
4 Entity Structure Step

JSP, which stands for Jackson Structured Programming\(^1\), uses a graphical notation, called structure diagrams, to describe the structure of an object in terms of its component objects. In JSD, the objects of interest are the sets of events in which entities engage and we use JSP structure charts to specify the order of events for each entity.

While we use the same descriptive language to describe objects as we use to describe programs, to make it easy to transform entity structures into programs, the purpose is to structure the world, not the program. Jackson says it best

> “Although we are using notations which, to some readers, are familiar from programming, it must always be remembered that we are describing the real world, not the programs which make up the system. Nor are we describing the structure of a database, nor the structure of the system itself. We are simply describing the order in which each entity performs or suffers the actions in which it participates. As we will see later, when we discuss the implementation step, one of the large benefits of the kind of model we build in JSD is the comparative ease with which we can transform it into a set of runnable programs; it is not an accident that we describe the ordering of real world events in constructs that are also used in programming” \(^2\)

Jackson’s basic premise is that the control structure of a program, consisting of sequences, choices and loops – should correspond to the structure of its input and output – the sequences of objects, choices of objects and loops over objects. This is a useful rule for most problems that process data or message streams as in JSD. The rule does not help where internal data structures are being modified or used, such as inserting into a binary tree, graph traversal, or sorting.

4.1 Structure diagrams and structure text

JSP uses a graphical notation\(^3\) to describe the structure of an object in terms of its component objects. The substructure is a refinement of the upper level. The only methods of refinement are to partition an object into a sequence of sub-objects, a choice of sub-objects and iteration over sub-items. Structure diagrams are the equivalent of regular expressions with named sub-expressions. Structure text, a form of pseudocode, is the textual equivalent of structure diagrams. We present a variation of what Jackson proposed as it is possible to be more flexible than he suggested.

Structure diagrams are used to show overall structure and not details. Details are described in documentation and in structure text. In particular, the semantic actions and conditions for choice and iteration are indicated in structure text.

4.1.1 Sequence structure diagrams

An object can be composed of a sequence of sub-objects. In Figure 4.1, “Sequence item” is composed of “sub item 1”, “sub item 2” and “sub item 3”, in that order. In a structure chart down means component or subpart, while left to right means sequence order.

---

\(^1\) See [JACK78].

\(^2\) [JACK83] page 84. The very last phase, about program constructs does not say much. There are only three ways of denoting the structure of objects and computations – sequence, choice and loop. Anything else is just a variation.

\(^3\) Equivalent to naming subparts of regular expressions
For example, in Figure 4.2, a freight train is composed of an engine, coal car, freight cars and caboose (at least they used to have these components).

Figure 4.2: JSP structure chart for a freight train – version 1

Sequence structure text

The sequence structure text corresponding to Figure 4.1 is shown below. Comments, using C++ style, indicate where semantic actions are inserted. Sequence-item, sub-item-1, sub-item-2, etc. are the labels from the corresponding structure chart. The read statements are a part of the standard structure text. The assertions should be true if the sequence correctly represents the structure it is modeling.

```
read message  // Must always have read the next message to process.
    // This read could be done by the previous item in the structure or
    // be the first read for the process.
    // Insert semantic actions to take place before sub-item semantic actions.
Sequence-item seq
    sub-item-1:
        assert message is sub-item-1
        // Insert semantic actions for sub-item-1.
        read message
    sub-item-2:
        assert message is sub-item-2
        // Insert semantic actions for sub-item-2.
        read message
    sub-item-3:
        assert message is sub-item-3
        // Insert semantic actions for sub-item-3.
        read message  // Read ahead for the next item in the structure
Sequence-item end
    // Insert semantic actions to take place after all sub-item semantic actions
```

4.1.2 Iteration structure diagram

An object may be composed of a repetitive sequence of a sub-item. Iteration is used when we cannot list all sub-objects in a sequence – we may not even have an explicit number of sub-objects. JSP iteration is an iteration of zero or more sub-objects; specific numbers are given in
accompanying documentation. Figure 4.3 shows a schematic representation of iteration. Figure 4.4 is an example that shows the collection object “freight cars” consists of zero or more objects called “a freight car”.

![Figure 4.3: Schematic for JSP loop structure chart.](image)

**Iteration structure text**

The iteration structure text corresponding to Figure 4.3 is shown below. Comments, using C++ style, indicate where semantic actions are inserted. Collection and Repetitive-item are the labels from the corresponding structure chart.

```cpp
read message
   // Insert semantic actions to take place before the loop.
Collection itr while (condition)
   Item-item:
       assert message is repetitive Item-item
       // Insert semantic actions for Item-item.
       read message
Collection end
   // Insert semantic actions to take place after the loop
```

4.1.3 **Choice structure diagram**

An object may consist of one of a choice objects. In Figure 4.5 “Conditional item” consists of one of the items “Choice 1”, “Choice 2”, or “Choice 3”.

![Figure 4.5: Choose one of three items](image)

For example, we could have different types of freight cars in a train, as in Figure 4.6. Thus, in Figure 4.6, a freight car can only be one of “flat bed”, “oil tank” or “refrigerator”.

![Figure 4.6: Zero or more instances of a freight car makes the object “freight cars”.](image)
In a choice construct there is no implied order. The construct simply states that one of the choices must occur. Changing the order of the choices does not change the meaning of the structure. The JSP diagram in Figure 4.7 is identical meaning to the diagram in Figure 4.6.

Figure 4.8 shows how a conditional or optional choice is diagrammed.

Choice structure text
The choice structure text corresponding to Figure 4.5 is shown below. Comments, using C++ style, indicate where semantic actions are inserted. The choices are deterministic, in that the condition is known in advance and are mutually exclusive. Conditional-item, choice-1, choice-2, etc. are the labels from the corresponding structure chart.

```cpp
read message // Insert semantic actions to take place before choice semantic actions.
conditional-item set
    alt (choice-1)
        assert message is choice-1 // Insert message actions for choice-1.
        read message
    alt (choice-2)
        assert message is choice-2 // Insert semantic actions for choice-2.
```
When we do not know the condition in advance we need to use backtracking, which we indicate with different keywords as shown below. The `posit` keyword indicates the current state is to be saved on a stack. The `quit` keyword indicates the state is to be restored from the top of the stack, the stack top is dropped, and processing advances to the next posit or end of conditional, whichever occurs first. The `exit` keyword indicates that the state is to be dropped from the top of the stack and the processing advances to the end of the conditional.

```
read message
alt (choice-3)
  assert message is choice-3
  // Insert semantic actions for choice-3.
  read message
conditional-item end
  // Insert semantic actions to take place after choice semantic actions
```

```
if happy (one of the assumptions was true) then
  // Insert semantic actions to take place when a choice was made.
  // Usually pop the backtrack stack and exit to parent indicating happiness
else unhappy (all assumptions were false)
  // Insert semantic actions to take place when no choice could be selected.
```
4.1.4 A complete example

Figure 4.9 shows a more refined definition of a train, than in Figure 4.2.

The following is the pseudo code corresponding to Figure 4.9.

```plaintext
read message
   // Insert semantic actions to take place before a freight train.
Sequence-item seq
   engine:
      assert message is engine
      // Insert semantic actions for engine.
      read message
   coal-car:
      assert message is coal car
      // Insert semantic actions for coal car
      read message
   freight-cars:
      assert message is flat bed, oil tank, refrigerator, or caboose
      // Insert semantic actions to take place before the iteration
      freight-cars itr while (message ≠ caboose)
         assert message is flat bed, oil tank or refrigerator
         // Insert semantic actions to take place before choice semantic actions.
      a-freight-car set
         alt (refrigerator)
            assert message is refrigerator
            // Insert semantic actions for refrigerator
            read message
         alt (flat bed)
            assert message is flat bed
            // Insert semantic actions for flat bed
```
read message

alt (oil-tank)
assert message is oil-tank // Insert semantic actions for oil-tank
read message
a-freight-car end // Insert semantic actions to take place after choice semantic actions
freight-cars end // Insert semantic actions to take place after the loop
// Read ahead for the next item in the structure done at end of freight-cars

caboose:
assert message is caboose // Insert semantic actions for caboose
read message // Read the end of file
Sequence-item end // Insert semantic actions to take place after all a-freight-train semantic actions

4.2 Copy file design
The next few sections use JSP to show how we structure program input and output and how the corresponding programs are constructed.

4.2.1 Character by character copy
We are asked to design a program to copy a file character by character. The structure of the input is a sequence of characters and the output is also a sequence of characters. The exact count is not known so we have a loop over characters. The input and output structures, shown in Figure 4.10, correspond. The two structures can be merged into a single process in a trivial manner.

open input, output
read ch
while ch <> EOF
  write ch ; read ch
end
write ch
close input, output

Figure 4.10: Copy problem – version 1

4.2.2 Copy line by line
We are asked to copy files containing lines and characters within lines. The program in Section 4.2.1 solves the same problem but in this design it is easy to modify the program to count characters and lines. Figure 4.11 shows the input and output structure charts.
4.2.3 Copy by character type

Yet another design specifications calls for a program to distinguish among the types of characters appearing in the file. We can modify the program to do things such as count different types of characters as well as lines. Figure 4.12 shows the JSP structure chart.

4.2.4 Count characters per line

Suppose we want to modify the previous program to count characters per line, instead of copying the file. The input has the structure as defined in Section 4.2.3 but the output has the structure shown in Figure 4.13. We have a correspondence at the upper levels (line and EOF). At the lower level, count in the output corresponds to both contents and EOL. For each instance of line in the input structure there is one instance of count in the output structure. Modifying the program in Section 4.2.3 becomes a trivial exercise. Similarly, modifying the program in Section 4.2.2 is simple because of the correspondence between input and output (both have lines defined). On the other hand modifying the program in Section 4.2.1 is more difficult because line is not defined for input; there is no correspondence between input and output.
4.2.5 Copy which permits word counting

Modifying the programs presented in the earlier sections to count words is difficult because the designs do not have the ‘word’ object designed into them. Before we can get such a program we need to define a structure for input which contains words, see Figure 4.14.
The following program text corresponds with the JSP structure chart in Figure 4.14.

```java
open input
read ch
while ch <> eof
  while ch <> eol
    case ch of
      space : read ch
        while ch = space do read ch end
      alpha : write ch ; read ch
        while ch = alpha do read ch end
    end case
  end
  read ch
end
close input
```

The word counter becomes obvious to write because we have the correct structure in which to embed the counting semantics.

**Exercise:** Embed the appropriate semantics into the previous program to output the count of characters on each line with one count per line.

These examples show that program design is strongly influenced by the design and structure given to the input and the output. With a complete description of the input and output, it is easy to add functionality.

### 4.3 Line repacking design

The problem is to write a program to compress an input file by filling output lines with as many words as possible up to the fixed maximum length of an output line. Output lines are to have no leading or trailing spaces. Inter-word spaces are to be reduced to one character.

Apparently the input and output have the same structure as shown in Figure 4.15, except the output forces a word on a line and there is no choice since space and word alternate.

![Figure 4.15: Repacking problem JSP structure charts](image)
There is a problem, however, because the line sizes are different. This is a subtle structure clash because superficially it seems that a line is a line is a line, so it is easy to miss the problem and deal with it in a superficial manner that will lead to problems when it is time to modify the program. We cannot write the program with a single loop structure because the terminating condition for the “contents” input loop is different from the terminating condition for the “contents” output loop. If we use an analogous solution to the copy problem we need two interleaved loops – one for input and one for output – each one with a different boundary condition, a boundary clash. How do we structure the program?

The solution is to write two communicating processes. The input process reads the input file, produces words and sends the words to the output process. The output process receives words from the input process and writes the output file. Each process has its own loop. The end of the message stream is indicate by the null word. The following two processes show the solution.

**Input program**

```plaintext
open input ; read aChar
while aChar <> EOF
{ while aChar <> EOL
  { switch aChar
    { Space : read aChar ; read aChar
      while aChar != (EOL or Space)
        aWord = aWord || aChar ; read aChar
      send aWord
      Output the built word.
    }
  read aChar
  Finished one input line
}
send '' ; close input
Signal no more words
```

**Output program**

```plaintext
open output
receive aWord
while aWord <> ''
{ write aWord ; receive aWord
  while aWord <> '' and enoughSpace
    { write space ; write aWord
      receive aWord
    }
  write EOL
  Another word on the line
}
write EOL
End of an output line
}
close output
No more words
```

For an implementation of the above see “Repack Implementation as Coroutines and Inversion” in the Example Designs. This is a famous problem in that it was used in various papers in the late 70’s and early 80’s, invariably with excessively complex solutions. Compare the above solution with the text formatting solution in [DROM85] in the references.

### 4.4 Telegram analysis design

This larger program using communicating sequential processes was studied in detail in the mid 1970’s. Various solutions and methods of arriving a solution were proposed. The designs used structured programming with top down design. In many of the published examples errors were
found. Jackson showed that using JSP with communicating processes it was straightforward to produce a correct, understandable and verifiable design. The following is the specification.

An input file consists of text for a sequence of telegrams. The text is stored in blocks of length BLOCK_LENGTH, with the last character of each block being the EOB character. Blocks contain zero or more words and one or more spaces between words. Spaces may or may not precede the first word in a block or follow the last word in a block. Words do not span block boundaries. A telegram ends with the word ZZZZ. The text terminates with an empty telegram. The last block in the file has the character EOF as its first character and no telegram text.

An output file analyzing the input file is to be produced. An example output report file is the following.

Telegram Analysis
Telegram 1: 15 words of which 2 are oversize
Telegram 2: 111 words of which 0 are oversize
Telegram 3: 22 words of which 6 are oversize
End of telegram analysis

An oversize word contains TooMany or more characters.

### 4.4.1 Telegram analysis solution

This section shows only the structure charts. The corresponding documented program text is too long to include here; see the section Telegraph JSP design report, in the Example Design section of the readings.

Figures 4.16, 4.17 and 4.18 show the important structures in the telegram problem. The telegram problem has a structure clash between the input file of blocks (Figure 4.16) and the telegram file of words (Figure 4.17). The structure clash arises because blocks are not neatly nested within telegrams and telegrams are not neatly nested within blocks.
The report structure (Figure 4.18) and telegram structure (Figure 4.17) correspond because for each telegram there is exactly one report line. The two structures could have been merged. It is obvious in the telegram structure where the operations of initializing, incrementing and writing the report line occur. It is a simple matter to insert these operations at the correct locations in the control structure corresponding to the telegram data structure. The report header line is written at the beginning of the program and the trailer line is written at the end.

One solution to this problem is to design at least two communicating programs. The input program that reads a sequence of blocks, extracts words and sends them to the output program. The output program reads a sequence of words, forms telegrams, counts words and writes the output report.

Even though the report structure and the telegram structure do not clash a designer would separate these structures into two communicating programs. The reason being to simplify each structure, at a slight increase in extra processing time, with a view that future modifications would be easier to do. It is a good design to separate each major data structure into a process or abstract data type.
Our solution has three communicating programs.

1. The input program reads a sequence of blocks, extracts words and sends them to the telegram analysis program.
2. The analysis program reads a sequence of words. Counts the number of words and oversize words in each telegram. Each pair of counts is sent to the report program.
3. The report program reads pairs of counts and produces the required output line.
4.5 Other considerations

4.5.1 Multiple inputs

Draw a structure chart for each input structure. If the inputs are read sequentially, then combine
the structure charts as a sequence at the upper level. If the structures are read in an interleaved
pattern, and if there is structure correspondence, interleave the structures in one structure
diagram. If there is no structure correspondence, then each input structure becomes a separate
communicating process and you must design a marsupial entity, see Section 4.7.1, that
interleaves the intermediate structures sent by each of the corresponding input processes.

4.5.2 Internal data structures

In many programs there are one or more internal data structures that must be designed.

4.5.2.1 Example of a calendar print program

An example is the calendar print program; see the Sections Calendar Print Problem, Calendar
Print Program Two Process Design, and Calendar Print Program Three Process Design, in the Example
Design section of the readings.

The input is a year for which to print the calendar. The output is a printed calendar. Since
the input is a single item it automatically corresponds with the output so it looks like only one
process is required. There is, however, an internal structure consisting of the canonical calendar
independent of the form in which it is printed. There is often a structure clash between such
internal structures and the input structure (not in the calendar case), and output structures (there
is in the calendar case). Good design of programs with structure clashes require multiple
processes to resolve the structure clashes, so be on the lookout for all relevant object structures,
even hidden ones!

4.5.2.2 Example of an integration program

Here is a mathematical program to show that JSP is not restricted to text processing or
business applications. The problem is to integrate the function \( y = f(x) \) between \( a \) and \( b \).
Integration is equivalent to computing the area under the curve between \( a \) and \( b \). The solution is
to compute a series of approximations to the area and stop when successive approximations are
close enough.

![Figure 4.19: Integration – first approximation.](image-url)
Figure 4.19 shows the first approximation as a single rectangle with the top left corner touching the curve.

The second approximation, Figure 4.20, shows two rectangles, each half the width of the rectangle in the second approximation, where the top-left corners touch the curve.

Each approximation doubles the number of rectangles by halving the width. At all times the top-left corner of the rectangles touches the curve. Figure 4.21 illustrates the situation for the n+1 approximation.

As the rectangle widths get smaller the approximation gets better. Eventually the improvement is not worth further computation and we stop.
Each approximation deals with a sequence of terms, where each term is the area of a rectangle. The computation itself deals with a sequence of approximations – a sequence of sequences of terms. Figure 4.22 gives the JSP structure chart for the internal data structures in the solution.

![Structure Chart](image)

Figure 4.22: Integration – structure chart

Now we are ready to write the corresponding program text. The following is a straightforward translation of the structure chart in the context of the problem.

**Start of program**

Compute the first sequence

```plaintext
read a, b               // The integration limits
, epsilon              // The error tolerance
w = b - a                // First rectangle width
n = 1                    // Approximation counter
curr = last = f(a)*w     // First sequence
```

Outer loop computes the remaining sequences

```plaintext
while n < 1 or abs(last-curr) > epsilon
n += 1 ; w = w/2          // Next approx. New rectangle width
aPoint = a                // Starting point for the sequence
last = curr ; curr = 0    // Remember last approx. Prep for next
```

Inner loop computes the current sequence of terms

```plaintext
while aPoint < b
   curr += f(aPoint)*w
   aPoint += w
end
end
```

We have the approximate answer.

```plaintext
print curr , n , abs(last-curr)
```
End of program

Once we have a program we can analyze it for efficiency. In this problem there is a great deal of re-computation of terms – each sequence re-computes all of the terms in the previous sequence. Figure 4.23 illustrates the situation.

\[
\begin{align*}
\text{Approx} & \quad a \quad a+w/8 \quad a+2w/8 \quad a+3w/8 \quad a+4w/8 \quad a+5w/8 \quad a+6w/8 \quad a+7w/8 \\
A0 & \quad x \\
A1 & \quad x \\
A2 & \quad x \\
A3 & \quad x \\
\end{align*}
\]

Figure 4.23: Integration terms computed in each approximation

We can reduce the computation effort by a factor of 2 by considering only the sequence of new terms. Because we have a correct structure for this problem it is easy to modify the previous program to produce the following more efficient program. The program structure remains the same. What we do is change the target operations to take into account our efficiency observation.

Start of program

\begin{verbatim}
read a, b           // Nothing changes for the first sequence.
                   , epsilon
w = b - a
n = 1
curr = last = f(a)*w

Outer loop computes the remaining sequences.

while n < 1 or abs(last-curr) > epsilon
    n += 1 ; lw = w ; w = w/2   // Remember the previous width
    aPoint = a + w             // Half way into first rectangle
    last = curr ; curr /= 2    // Start with half of previous area

Inner loop computes the current sequence of terms. Do new terms only.

    while aPoint < b
        curr += f(aPoint)*w
        aPoint += lw        // Advance by old width
    end

We have the approximate answer.

    print curr , n , abs(last-curr)
\end{verbatim}

End of program

4.5.3 Elaborations

The first structure chart that you construct is the simplest one that captures the essence of the sequence of events in which the entity engages. In Figure 4.24 the left hand machine structure chart shows that a machine is bought and then used and repaired. In the right hand structure chart is an elaboration that shows there are multiple uses between each repair. The left hand structure chart is the one that describes the messages that come from the real world. The right hand structure chart is an interpretation (elaboration) of the message stream that makes it possible to add a function to count uses between each repair.
4.6 Handling complex entities

Attempting to construct a single structure chart for an entity may lead to excessively complex or impossible structure charts. Such cases arise when there is an interleaving of events due to marsupial entities, different entity roles and premature termination of entities. These situations are described in the following sections.

4.6.1 Marsupial entities

Marsupial entities are entities that are not in the real world but arise out of a variation of multithreading structure clashes. Multithreading implies that the entity is concurrently participating in an event sequence where subsequences of events have a clear but arbitrarily interleaved substructure. Each such subsequence represents an entity distinct from a parent entity arising as consequence of the activities of the parent. Marsupial entities cannot exist by themselves; they must have a parent.

Typical examples are bank accounts for a customer and customer orders at a store. In the former case a customer may have multiple accounts and can engage in events in an arbitrary order of accounts. In this case an account is a marsupial entity that has a clearly structured event sequence – open must occur before deposit or withdraw. But drawing a single structure chart for a customer that includes all possible ordering of account activity while maintaining the ordering structure for each account is impossible. Consequently, accounts are a marsupial entity type. The customer actions are now a choice of actions on any of the accounts. Similarly a customer may have multiple concurrent orders at a store. Each order is a marsupial entity.

The example shown in Figures 4.25 and 4.26 show a marsupial entity account that arises from the interaction two entities – a bank customer who deposits and withdraws money and the bank that adds interest to the account and charges a fee for the account.
Figure 4.25: Customer and bank activity shown as separate structures.

Figure 4.26: Structure of the marsupial entity account arising from the customer and bank structures in Figure 4.25.

4.6.2 Multiple roles

An entity may have more than one role during its lifetime. Each role has a clear structure but the roles have an arbitrary interleaving during the entity lifetime. It is impossible, or extremely unwieldy, to have a single structure chart showing all possible interleavings while simultaneously preserving the structure of each role. Again a structure chart is created for each role and the parent entity’s structure is an arbitrary choice of actions across roles.

For example a person could be both a student and an instructor at a university. Figure 4.27 shows greatly simplified structure charts for the roles of student and instructor. Figure 4.28 shows the choice structure that combines the two roles. Jackson gives an example of a soldier having course and career roles\(^4\).

\(^4\) [Jack83] pages 103 to 107.
The program would have processes for Person, Student and Instructor structure charts. The Person process would interface with the world.

### 4.6.3 Premature termination

For robust systems we must accept the case of premature termination of an entity. For example a customer may leave the country or die without engaging in the final close event for each bank account. A soldier may leave the armed services before attaining the highest rank and/or completing all courses. In each case the model entity will run forever waiting for the next event. The model entity will never reach the end of its structure.

Figure 4.29 shows how to handle premature termination for the instructor structure shown in Figure 4.27. To handle premature termination the structure chart of the entity is augmented with a choice at the upper level. One choice describes the normal event sequence. The other choice describes the prematurely terminated sequence. Semantically, when premature termination occurs, backtracking takes place over the entire history of events backing out of the normal event sequence. The entire event sequence is “replayed” in the premature termination choice without distinguishing events. Finally the entity engages in the termination events.
It may appear to be inefficient to back out of the normal structure and replay the events in the premature termination structure but you are only describing the semantics of premature termination, not the execution. At implementation time you make use of beneficial and neutral side effects to avoid most, if not all, of the implied program execution. Most of the execution time, depending upon the side effects, is taken up in the termination events and not in backtracking.

4.6.4 Composite structures

When entities participate in common actions and you want to model that commonality, you create a marsupial entity that models the event sequences of all common events. You not only produce a structure chart for each participating entity type but also produce a structure chart for the marsupial entity. The marsupial entity's structure chart is a composite structure that includes the structure of all events of the entities participating in common events. You must verify that composite structure does not restrict or augment the event sequences of the parent entities.

Checking a composite structure

When you have created a composite structure from a set of other structures it is good to verify the correctness of the composite structure. Correctness is verified by following the following rules to remove all but a selected base component of the composite structure.

1. Select one of the base structures, say BS, within the composite structure.
2. Replace with null (–) every leaf of the composite structure that does not occur in the base structure BS. This leaves only and all events in which the designated parent can engage.

---

Apply the following transformations to the composite structure until it is as simple as possible. The transformations do not change the constraints on the ordering of the leaves of the structure.

- Remove a null leaf that is a part of a sequence.
- Remove a null leaf that is a part of a selection which itself is the part of an iteration.
- Replace a selection of one part by its part.
- Replace a sequence of one part by its part.
- Remove one of two identical parts of a selection.
- Remove one of two identical iterations that are consecutive parts of a sequence.
- Replace a sequence of one part by its part.
- Remove one of two identical iterations that are consecutive parts of a sequence.
- Reduce two levels of iteration to one; that is, replace B with C where A is an iteration of B and B is an iteration of C.

If the base structure BS is correctly interleaved within the composite structure, then the result of step 3 should be the correct structure diagram for the structure BS.

Applying the above rules to Figure 4.26 for either the customer or the bank you obtain the corresponding structure in Figure 4.25.

### 4.7 Structuring guidelines

One structure chart is created per entity; exceptions occur with interleaving clashes due to marsupial entities and multiple roles.

- Create a minimal structure that represents all the valid event orders. Elaborations are done later in the model step. Figure 4.24 shows an unelaborated sequence of use and repair events that may take place at a rental store and an elaborated structure that enables the storeowner to gather statistics on how frequently a machine requires repair. Jackson gives an example of buying, binding and using library books.

- Minimize visual clutter in structure charts. Do not add constraints to the charts. Put all constraints and additional information in an accompanying narrative description. The purpose of the charts is to highlight the structure, not give the details.

- Consider the entire lifetime of an entity – initial conditions are frequently forgotten. In Figure 4.1 we have the buy event even though rent and repair are the events of interest to the owner. Jackson gives the example of an elevator starting at the lowest floor because that is where the elevator would be installed and connected to the cables and the controller.

### 4.8 Designing for errors

How are programs designed when there is the potential for error in their input? Simple, the error structures become a part of the definition of "acceptable input". Error structures are alternatives to correct input that are included in the initial structure charts. See “A Solution to the Problem of Correcting Erroneous Input to the Telegram Analysis Problem” in the Example Designs. If error structures are not included, then the structure of the program does not match the structure of the input. The program is not a valid model of the input and consequently the program is incorrect with respect to the specifications. Furthermore, it is impossible to detect errors in the input and to embed the appropriate semantic actions in the program.

There are two types of errors that can occur in event sequences as follows.

1. The sequence of messages is a true reflection of real world events but it is only correct with respect to an error free model.
2. The sequence of messages is a false reflection of real world events.

---

An example of a type 1 error is a customer at an automatic teller machine (ATM) pressing arbitrary buttons instead of a "correct" sequence of buttons. The ATM program should read the sequence of button sequences, detect error sequences and react appropriately.

To handle this type of error the structure chart must provide for all possible "error sequences" so the program can correctly identify and react to such errors. At this point in building the model, you work on the assumption that all input messages representing events are true. Structure charts handle correct and incorrect error sequences based on this assumption.

An example of a type 2 error is a sequence of messages arriving over a network. Some messages may be lost because a part of the network goes down or buffers overflow. The sequence of messages sent is correct, while the sequence of messages received is in error.

To handle this type of error an input subsystem, Figure 4.30, is built to interface the real world with the model. The input subsystem is developed, using JSD, to correct, as best it can, false real world input streams into true model input streams. Essentially, the input subsystem converts a sequence of messages the model cannot interpret into a sequence of messages the model can interpret.

![Diagram](image)

Figure 4.30: Showing where the input subsystem fits in.

It is your choice as the designer as to what should be in the model and what should be in the input subsystem.
4.9 Summary of structure clashes

The following table presents a summary of the types of structure clashes, or correspondence problems, that occur in practice.

<table>
<thead>
<tr>
<th>structure clash</th>
<th>Example problems</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>ordering clash</td>
<td>printing matrix transpose; calendar printing</td>
<td>read in; reorder; write out</td>
</tr>
<tr>
<td>structure clash</td>
<td></td>
<td></td>
</tr>
<tr>
<td>boundary</td>
<td>line repacking</td>
<td></td>
</tr>
<tr>
<td>general</td>
<td>telegram analysis</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Processes communicate maximum structure that corresponds (working from the bottom up)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>a) asynchronously – file, channel buffer (queue)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>b) synchronously – coroutines, message sending</td>
<td></td>
</tr>
<tr>
<td></td>
<td>c) program inversion</td>
<td></td>
</tr>
<tr>
<td></td>
<td>i) one process is the master the rest are subprograms</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ii) have a scheduler with every process a subprogram.</td>
<td></td>
</tr>
</tbody>
</table>

| multi-threading | Example problem: boat rental report |
|                | Solution: separate process for each thread |
|                | a) duplicate program text |
|                | b) one program text with state vector extraction – data base |

Types of programming difficulties
- communications – which method to use
- backtracking – which choice to take
7 System Timing step

The system timing step specifies and documents the timing constraints for the model, including the specification and documentation of the degree of fairness for rough merges – since rough merges are dependent upon the relative timing of messages. Documentation is informal. It is a set of rules to be referred to during the implementation step, leaving freedom to have batch or interactive implementation depending only upon timing constraints.

In the system timing step you address problems such as the following.

- How up to date is the model. Some system outputs must be produced within a limited time following arrival of inputs. If certain information is needed then how up to date does each process have to be to provide the information: within 2 seconds; 5pm previous day; 9am next day; customer information within 3 seconds and account information from the previous day. Elevator model must take into account traveling speed and time between detection of floor arrival and when motor must be stopped so elevator stops at the floor and not above or below.
- Level-1 processes must execute getsv from level-0 at a minimum frequency. Elevator button must be read quickly enough to detect the close and open circuit of a user pressing the button – at least every 20ms.
- Processes whose state-vectors are examined must be sufficiently up to date.
- TGM records must be read within certain amount of time after being written.
- Rough merge must be tightly synchronized.

Fairness in rough merges depends on the merge algorithm and the relative speeds of the input processes. Suppose both clerks and customers can affect orders, Figure 7.1. If the message streams CO and ClO are batched then an alternating merge can be unfair. Consider having 2 customer messages early in the day and 1 clerk message later in the day. An interactive system can also be unfair. Customer messages may be delayed in transmission with the clerk message arriving first.

![Figure 7.1: Rough merge of order messages from customers and clerks.](image)

How much unfairness can be tolerated? Rough merges always require careful consideration and in many cases the system must be designed to tolerate a lot of unfairness. In some cases you must specify very close tolerances. For example, you may want to produce a listing of closing balances at the end of each day from customer accounts. One solution is Figure 7.2. When the TGM arrives at Lister it loops over requesting a balance and reading a balance, and eventually produces the list.

---

8 Only the relative time is important not the absolute time. As we must process faster and faster the less tolerant we are of message transmission delays.
For other design considerations it may not be wise to have Account-1 read a series of requests. Instead you may need the design in Figure 7.3 where TA is the time at which Account-1 will transmit the set of balances and TL is the time at which the lister will list the balances it has read. The overall structure of Account-1 and Lister is simpler.

A problem occurs when TL is merged in the Lister not all of the balances may have been written by Account-1 resulting in an incomplete list. A synchronization process is introduced to output TL when the account balances have all been written. The solution is in Figure 7.4. The Sync process reads the time to start the listing then issues a send balance message to every account on TA. The Sync process waits for all the TA channels to be empty – indicating the account process has read the message and written the balance. Finally, the Sync process sends the start-listing message on TL.

---

9 Notice the horizontal line in the process box. It indicates that the process is introduced only for synchronization purposes.

10 Yet another instance where timing must be considered. The balance must be written “immediately” upon reading the TA message. This is a specification for the implementation step that the read-write pair should ideally be uninterrupted; i.e. a critical region. A weaker condition, providing for more concurrency, is for the TL message to not be written between a read-write pair.
**Figure 7.4:** SSD listing account balances using a synchronization process.

Figure 7.5 shows a general case of a process R that synchronizes two other processes P and Q through the messages it sends to P and Q. Typically R waits for one of the streams DP or DQ to be empty before sending a message on the other stream.

**Figure 7.5:** Process R synchronizes processes P and Q.
Notes
8 Implementation Step

By the time you get here the specification and program text for all processes has been written as a part of the previous steps. While in principle the system is executable, the appropriate match of hardware and software is unlikely, and the timing constraints are informal and must be enforced through appropriate scheduling of processes and communications.

The following lists the major problems still to be solved.

- Dealing with communication between processes. When a program is partitioned into parts, the parts must execute as a unit. The parts must communicate – that is pass data among each other. We can use either synchronous or asynchronous communication methods.
- Sharing processors among processes. Usually there are several orders of magnitude more processes in a system than there are processors – thousands to millions of customers but 1 to 10 processors. For each process we must ensure the following.
  - Sufficient CPU time for each process when it needs it.
  - Enough memory for storing all the program text and state-vectors. May need to dismember, overlay, or use virtual memory.
- Implementing state-vector and data stream connections. We do not have infinite buffers and cannot read variables local to a process. Solution is to transform (mechanically) process program text taking the following into account.
  - Transform to fit the machine rather than redesign to fit existing implementation structures.
  - Do make use of existing facilities – after the transformation is made. Otherwise reinventing existing solutions. Point is not to constrain design due to available implementation facilities.

The next sections address these problems.

8.1 Communications

What do we communicate? This has already been decided in the design stage. In JSD, processes exchange the largest corresponding unit, working from the bottom up, in the output structure chart for the sending process and the input structure chart for the receiving process.

8.1.1 Asynchronous communication

People communicate at their convenience. Each person can work independently of others on different tasks or the same task. People communicate asynchronously when using mail utilities. Each person can read and compose messages independently of the others. A person does not have to sit at a terminal or mailbox waiting for a message while another person composes a message. The only restriction is that a person cannot read a message before it is sent.

The same thing can take place with processes. One process can send a message and another process can receive the message. The two processes work independently and continuously, as long as the reader does not get ahead of the writer. The usual order of reading and writing follows the FIFO discipline. Communication occurs through a buffer, which can be a file, a queue, or a channel. We leave the implementation up to the operating system or network.

For robust communication the primitives sendMessage, receiveMessage, sendAnswer and receiveAnswer are used. There are two communication channels, one for messages and the other for answers. Message and answer objects are the same so they are recycled by treating the send message, receive message, send answer, receive answer as a loop.
8.1.2 Synchronous communications

People communicate cooperatively at the same time and deal with the same topic or task when they are in meetings, using a telephone or attending a class. Communication only takes place by mutual consent and agreement. Each party must receive when others are sending. Only one party can send at a time.

In the case of processes we say they rendezvous, the receiver and sender wait for each other at the communication point and when both are ready they exchange data. After communication takes place the processes may carry on working independently of each other.

8.2 Reducing the number of processes

The simplest implementation scheme is to have one processor per process. In general, this is not feasible because of the excessive number of processors and communication lines required; it is just too expensive. Besides while the processes have very long life times – decades for customers and elevators – they do not need much processing time so timesharing is a solution. General-purpose timesharing operating systems may work for some systems, especially small ones. For example using multitasking in Ada with message passing. But general-purpose operating systems are not designed to handle millions of processes and they may not have desired scheduling and timing properties, consequently, it becomes your responsibility to schedule the processes.

You could write or customize an operating system scheduler to handle the unique needs for a particular system but this is frequently not a good technique to use. Not only because of the difficulty of creating a specific custom operating system but also because of the inherent inefficiency of using a general-purpose scheduler in time to switch processes and in time to deal with inter-process communication.

There are two major ways of reducing the number of physical processes – coroutines and inversion.

8.2.1 Coroutines

The communicating processes are designed in the same way as designing asynchronous communicating processes. The communication primitives send and receive, are replaced as follows.

send(message) becomes messageArea = message ; resume receiver
receive(message) becomes resume sender ; message = messageArea

The coroutines communicate through a common messageArea that can hold one message and is global to the coroutines.

Initial conditions set up a state vector for the initial process instructions for sender and for the receiver. The sender is resumed; that is, its state vector is loaded into the CPU registers and the process executes.

At a resume statement the current process’ state vector is saved and the resumed process’ state vector is loaded so that it will continue execution. The effect is similar to a return from a subprogram except that state of the ‘returning’ program is saved in a fixed memory area instead of being discarded as in a normal subprogram return (see Figure 8.1).

For an example implementation see Appendix 8, “Repack design and implementation”. The appendix shows a coroutine implementation the design of the line repacking problem in Section 4.4.
8.2.2 Program inversion

Program inversion is the technique by which you both reduce the number of physical processes and schedule the processes (analogous to writing a customized operating system scheduler).

Inversion is used when the operating system or programming language does not support either coroutines or send and receive operations, or when the implementation of send and receive operations are too inefficient and do not give the programmer the fine grain control they need in critical applications.

Process inversion means transforming a communicating process into a subprogram. The communication channels are replaced by one subprogram parameter for each channel on which the process is communicating. The parameters substitute for the communication channels. When the caller puts a value into a parameter and a subprogram is invoked this is equivalent to the caller sending a message. When a subprogram executes and reads a value from a parameter, this is equivalent to receiving a message from the caller. By analogy, when a subprogram puts a value into a parameter and returns to the caller, this is equivalent to sending a message to the caller. When the caller resumes execution after a return from a subprogram and reads a value from a parameter this is equivalent to receiving a message from the subprogram. The subprogram is invoked once for each message sent to the process the subprogram is modeling. The subprogram is invoked once for each message sent by the process the subprogram is modeling.

Since only one message at a time can be transmitted for each subprogram call, it is necessary for the subprogram to return back to the caller between messages. This is equivalent to the corresponding process being interrupted and resumed between each message it sends or receives.

To simulate an interrupt the subprogram must execute a return statement. To simulate a resume the subprogram must have the equivalent of a bookmark that you use when you resume reading a book. A bookmark in a program is a statement label. The subprogram stores in the static variable, nextStatement, the resumption point (bookmark location), just before returning.
to the caller. On entry to the subprogram the first instruction executed is “goto nextStatement” where nextStatement contains the label of the instruction following the last return statement executed by the subprogram.

An extra parameter “reason for next call” is also necessary because the caller of the subprogram has to know into which parameter to put data when the process (subprogram) is to receive a message and from which parameter to extract data when the process (subprogram) is to send data.

Rough merges are usually implemented with inversion. If the order of the data is acceptable, you do not need a scheduler but can invert directly with respect to the process reading the merged stream. Otherwise need a scheduler to create an acceptable order.

Variables that must exist across subprogram calls are declared static.

Example

Figure 8.2 shows a SSD for a telegram analysis problem. Figure 8.3 shows a SID (System Implementation Diagram) where processes U, A and R have been inverted with respect to a scheduler.

A prototypical process inversion

Assume process P is communicating with process Q along channel A (receives messages) and channel B (sends messages) – see Figure 8.4.
We are going to modify the stand-alone process P into a subprogram that is callable from a modified process Q. Figure 8.5 is the diagrammatic description of the situation.

![Figure 8.5](image)

Figure 8.5: Inversion of process P with respect to process Q. One line represents the inversion. Each of the other lines represents a channel of communication.

To process P add the following subprogram header – one parameter for each channel plus one to describe along which channel the message transfer is taking place.

```c
void function P( aChannelAmessage , aChannelBmessage , reasonForNextCall)
```

Insert, in process P, the following statement as the first executable instruction, where `nextStatement` is a static variable initialized to `label0` (a label on the first executable instruction in the original program).

```c
goto nextStatement; label0:
```

Replace, in process P, each instance of `receive(aChannelAmessage)` with the following statements – they cannot be a subprogram. The `reasonForNextCall` encodes the fact that the caller must put data into the parameter `aChannelAmessage` so the subprogram can simulate receiving a message. The subprogram then uses the data by copying the data in the parameter into a local variable.

```c
reasonForNextCall = receiveOnChannelA
nextStatement = labelxxx
return
labelxxx: localvariable = aChannelAmessage
```

Replace, in process P, each instance of `send(aChannelBmessage)` with the following statements – they cannot be a subprogram. The `reasonForNextCall` encodes the fact that the caller must use the data in the parameter `aChannelBmessage` so the subprogram can simulate sending a message. Notice the interrupt occurs before the data is put into the parameter. This occurs because the caller must be notified that on the next call the data will be sent. This tactic simplifies the conversion because the program does not have to be analyzed to see what the next message will be.

```c
reasonForNextCall = sendOnChannelB
nextStatement = labelxxx
return
labelxxx: aChannelBmessage = localVariable
```

The following statements must be appended to the process whenever it would terminate execution – they cannot be a subprogram. The program text stores the reason `Done` to indicate to the caller that the subprogram should not be called anymore. The extra label and return are inserted in case the caller does invoke the subprogram again. If that happens the subprogram is equivalent to a null operation.
reasonForNextCall = Done
nextStatement = labelxxx
return
labelxxx: return

In process Q the following initialization statement is inserted so that it is executed before the first send or receive message from process P. The purpose is to establish the reason for the first message to be transmitted between the processes.

P(aChannelAmessage, aChannelBmessage, reasonForNextCall)

For each receiveMessage in process Q (a send in process P) the following statements must be substituted.

P(aChannelAmessage, aChannelBmessage, reasonForNextCall)
localVariable = aChannelBmessage

For each sendMessage in process Q (a receive in process P) the following statements must be substituted.

aChannelAmessage = localVariable
P(aChannelAmessage, aChannelBmessage, reasonForNextCall)

8.3 Schedulers

When communicating programs are inverted to become subprograms, the designer can choose to invert any process with respect to any other process. Thus, for example in the telegram problem (see Section 4.5 and Figure 8.6) the designer could make any one of the three processes the main program and invert the others with respect to it.

Once a process is inverted it becomes possible to use it in new situations making the process more useful. Since there is no reason to favour one process over another one, it makes sense to invert all the processes in the program and write a scheduler to call each process at appropriate times to simulate their communications. Figure 8.7 illustrates the situation.

Figure 8.6: Some of the alternative inversion choices for the telegram program implementation – unblock process, U, analyze process, A, and report process, R.

---

1 This is the fundamental reason why packages, abstract data types and classes in object-oriented programs are created.
Figure 8.7: Using a scheduler for the telegram problem implementation – unblock process, U, analyze process, A, and report process, R.

There are two basic types of schedulers. They are described in the following sections using the telegram problem as an example.

### 8.3.1 Telegram problem processes

Let us assume that the telegram problem (see Section 4.5) is solved with the following three processes and that each process has been inverted.

**Process 1**  
`unblock(word, reasonForNextUnblockCall)`  
The input process reads blocks and sends words to process 2 along the word channel.

- `reasonForNextUnblockCall == sendWord`  
  send a word on word channel
- `done`  
  no more words to transmit

**Process 2**  
`analyze(word, count, reasonForNextAnalyseCall)`  
The analyze process receives words from process 1. Analyses the word stream and produces a word count and an oversize count for each telegram. The counts are sent to process 3.

- `reasonForNextAnalyseCall == receiveWord`  
  get a word on the word channel
- `sendCount`  
  put a count on the count channel
- `done`  
  no more words are expected

**Process 3**  
`report(count, reasonForNextReportCall)`  
The report program receives counts from process 2 and produces the Telegram analysis report.

- `reasonForNextReportCall == receiveCount`  
  get a count on the count channel
- `done`  
  no more counts expected.

### 8.3.2 Pipeline scheduler

The pipeline scheduler can only be used in pipeline processes such as in the telegram program. The scheduler is constructed by considering the inverted process call order from back to front (report process to unblock process). The construction is straightforward for a pipeline. There is no need to analyze the different message transmission rates for each channel. The structure is analogous to polling backwards along the pipeline, except that there is no need to verify that sender is ready as that is implicit in the looping structure. Extension to longer pipelines is obtained by nesting an additional while loop for each process in the pipeline.
Do the initialization call for each process.

unblock(word, reasonForNextUnblockCall)
analyze(word, count, reasonForNextAnalyseCall)
report(count, reasonForNextReportCall)

Scheduler loop

while reasonForNextReportCall == receiveCount
{ while reasonForNextAnalyseCall == receiveWord
{ unblock(word, reasonForNextUnblockCall)       // Move a word from process
    analyze(word, count, reasonForNextAnalyseCall) // to process 2.
}  
analyze(word, count, reasonForNextAnalyseCall)  // Move a count from process
2   report(count, reasonForNextReportCall)         // to process 3.
}  

8.3.3 General purpose scheduler

This scheduler works for any inter-process communication structure. The polling order would be
adjusted to minimize the times polling is done by ordering the poll by decreasing message
frequency. Here it is necessary to verify that the send and receive side of a channel are both
ready to engage in a transmission. Extension to additional channels is straightforward.

Do the initialization call for each process.

unblock(word, reasonForNextUnblockCall)
analyze(word, count, reasonForNextAnalyseCall)
report(count, reasonForNextReportCall)

Scheduler loop

while ( (reasonForNextUnblockCall != done)
or (reasonForNextAnalyseCall != done)
)  
{ if (  (reasonForNextUnblockCall == sendWord)
    and (reasonForNextAnalyseCall == receiveWord)
    )
{ unblock(word, reasonForNextUnblockCall)       // Move a word from process
    analyze(word, count, reasonForNextAnalyseCall) // to process 2.
}  
else if ( (reasonForNextAnalyseCall == sendCount)
    and (reasonForNextReportCall == receiveCount)
    )
{ analyze(word, count, reasonForNextAnalyseCall)  // Move a count from process
   report(count, reasonForNextReportCall)         // to process 3.
}  

8.4 Buffering

How are data stream connections implemented? Can use shared memory or buffers. In some
cases buffering is forced due to the nature of the system.

In section 8.2 it was assumed that senders and receivers are always synchronizable – in a
network of communicating processes there always exists a schedule of message transfers such
that no more than one message is waiting on any one channel, all channels will be processed (no starvation) and the computation will not become blocked (no deadlock).

The synchronizable assumption is not, unfortunately, universally true. There exist networks such that some communication channels will have to buffer a sequence of messages before the first message in the sequence can be processed. A real life example occurs when a clerk is away for a while doing other tasks, in the meanwhile their input bin continues to accumulate messages representing work to be done.

A sufficient condition that buffers are required is that processes engage in simultaneous communications with other processes and the input from the other processes must be merged in a non-alternating manner. For example, consider the following problem. An unbounded sequence of increasing integers is to have integers in the odd positions have 30 added to them and integers in even positions have 8 added to them. The resulting sequence is to be printed in ascending order with duplicates removed.

The most straightforward solution is to have a network of communicating sequential processes as shown in Figure 8.8. The solution arises by recognizing that the input consists of two interleaved sequences each of which requires different processing. Process S splits the input stream. Processes P30 and process P8 add 30 and 8, respectively, to the numbers in each stream. The subsequences are merged to produce one output stream in process M. Finally process D removes duplicates. Process D could be combined with process M but we keep each process as simple as possible in the original design.

No matter which communication method we use to implement the above processes we need to have buffers for channels 3 and 4, or channels 1 and 4, or channels 2 and 3. We need one buffer for each subsequence to easily handle the merge.

In Figure 8.9 data from P30 and P8 is collected in the buffers for channels 3 and 4. Whenever one of the buffers is full the scheduler repeatedly polls process M sending buffered data until one of the buffers is empty. Then back to polling P30 and P8 until a buffer is full and polling M until a buffer is empty. This repeats until EOF is sent along channels 3 and 4 to process M. P30 and P8 call S to read an integer along channels 1 and 2 respectively. M calls D whenever it wants to send data.

---

3 It is in the implementation step (once all the program text for the processes has been created) that we decide how to implement the process communications and modify the program text accordingly.
Questions – What if we do not have unlimited buffering for data streams; have to worry about speed of processes. Limited buffering can lead to deadlock.

8.5 Reducing per process memory requirements

8.5.1 State-vector separation

In many problems there are many identical processes except for the state of each process. An example is dealing with a set of customers in a company, bank or other institution. A separate process models each customer. Figure 8.10 shows the situation.

Each customer process consists of program text plus a state-vector (such as account balance, address, where they are in a customer life cycle, etc.). The program text is identical for all customers but the state-vector is unique to each customer. Maintaining a copy of the program text for each customer is too expensive in space and modification effort. As a consequence, the
program text and state-vector are separated. Only one copy of the state-vector is maintained for all customers.

The set of state-vectors is maintained in a database. When the customer process is invoked by the scheduler, the process is either passed the state-vector (the scheduler reads the database); see Figure 8.11, or the process is passed the customer id and the customer process reads the state-vector from the database; see Figure 8.12. In either case the customer process uses the state-vector to continue executing the program text for the customer at the appropriate point. This is analogous to resuming an inverted process except that the equivalent to nextStatement and static variables are maintained in a database instead of variables in the subprogram.

Figure 8.11: A SSD showing state-vector separation from program text for customer processes. The scheduler reads and writes to the database. The scheduler sends the state-vector to the customer process and receives a modified state-vector when the process returns.

Figure 8.12: A SID showing state-vector separation from program text for customer processes. The customer process reads and writes directly to the database. The scheduler sends the customer id to the customer process.

8.5.2 Process dismemberment

Process dismemberment means creating copies of the program text and removing unneeded text from each copy so each copy does a different subset of work of the original process. Dismemberment must be along structural boundaries. Each subprocess is a substructure of the original. Can do this if the message streams can be cleanly partitioned because data streams are also dismembered. Dismembered data streams are input and output into dismembered processes.
Dismemberment is into disjoint sets. Each member of the set is run separately. Works as long as the following holds.

- Members can run in structure sequence.
- Members are logical subunits of the original process.

For dismemberment need buffering to retain data between runs.

Even schedulers can be dismembered. This causes a lot of work in analyzing effects on response time and faithful adherence to the order in which data must be processed.

Each dismembered process is a stand-alone process that can be inverted with respect to other processes and with respect to schedulers.

Can dismember state-vectors, this gets us into database design. Cannot do database design until we have produced the model and considered implementation.

In SIDs dismembered processes and data streams are indicated with suffixed names. For example process P could be dismembered into P-a and P-b, while data stream CO could be dismembered into CO-a and CO-b.

Dismembered processes are not always inverted with respect to other processes. They may be run as batch programs where the process text is executed once from beginning to end. Figure 8.13 is an example of process P invoking the dismembered batch process Q-b.

![Figure 8.13: SID showing batch execution from the beginning to the end of Process Qb.](image)

**Example**

Consider the simple system in Figure 8.14 where we have a single process P transforming data stream A into data stream B. For every AGRP in A there is one BGRP in B. BM is the total of the integers in AMR. BT is the total of integers in ATR. BW is the total of BT and BM. M and T are time grain markers merged into the A stream.

---

Figure 8.14: SSD of a process P transforming structure A into structure B.

The structure of the program is the following.

```plaintext
Process P
  GroupLoop: loop forever
    BW = 0
    AGMgroup: BM = 0 ;
      loop read A
        exit when M
        BM += AMR ; BW += AMR
      endLoop
      write BM
    end AGMgroup
    AGTgroup: BT = 0
      loop read A
        exit when T
        BT += ATR ; BW += ATR
      endLoop
      write BT ; write BW
    end AGTgroup
  endGroupLoop
Endprocess P
```

Suppose that input records A arrive frequently, the M record arrives at 9am Monday and the T record at 9am Thursday. So far with JSD we have assumed that P will execute continuously either on its own processor or be timeshared in someway. Suppose we do not want P to execute continuously but instead want a batch program to be run once a week on Thursday mornings.

Dismembering the program text of P means splitting it into the manual procedure (P-a) to run the batch program on Thursday mornings and the batch program itself (P-b) – the body of the GroupLoop. The input to the batch program is the name of appropriate saved input file.
The manual procedure consists of instructions to the operator.

At or shortly after 9am every Thursday run program P-b using the latest generation of file A. Give the output to the fussbudget manager.

Dismembering the input stream A means saving the input data in a data file – the send process writes to a file instead of the data channel. Every T record causes a new version of the input file to be created (A-b). Dismembering the output stream B means removing the paper from the printer after the run (B-b).

Dismembering the state-vector is trivial as there is nothing to do. There is no need to remember state-vector values between runs. The SID appears in 8.15.

It turns out that management wants to have the BM and BT records as soon as possible after the M and T times but continuous running of P is not feasible. The program P must be dismembered into three parts. The first is the manual procedure (P-c) to run two batch programs – one on Monday morning and one on Thursday morning. The second program (P-d) is the batch program to compute BM and a partial result for BW. The third program (P-e) is the batch program to compute BT and finish computing BW.

The manual procedure consists of instructions to the operator.

Every Monday morning run the program P-d using the latest generation of the input file A creating the intermediate file BW. Give the output to the fussbudget manager. Every Thursday morning run the program P-e using the latest generation of input files BW and A. Give the output to the fussbudget manager.

Dismembering the input data stream A means changing its feeder program to write new generations of the A file at every M (A-d) and T (A-e) record. Dismembering the output B means removing the paper after each run (B-d and B-e).

Dismembering the state-vector means saving the intermediate value BW, the only part of the state-vector that needs to be remembered across runs. The SID appears in Figure 8.16.
8.6 Backtracking

Backtracking can occur in any problem where there is a choice to be made as to what is the next substructure within a larger structure – the ‘O’ boxes in a JSP structure chart. The only solution to backtracking problems is to try each alternative in turn. If an alternative fails, then back up to where the choice was made and try a different choice. Backtracking requires recording the current state before a choice is made. If the choice is a good one, then the saved state is thrown away. If the choice is a poor one, then the saved state is used to restore the system to the last choice point so that a different choice can be tried. For example consider Figure 5.11 it is difficult to decide when to stop looping on leader, recognize the double zero or double one, so how can a program be written.

In many problems, and all those describable with JSP structure charts, it is possible to remove nondeterministic choices\(^5\) – a case where one does not know which choice to make to one where which choice to take is determined. A deterministic solution is easy to implement but it may be too large; have too many states – boxes in structure charts. In converting from a nondeterministic solution to a deterministic one the number of states can grow from \(N\) to \(2^N\) (that is why the deterministic description is more complex than the nondeterministic one). The only other alternative is use backtracking and accept the overhead of saving and restoring the state of a computation, and accept the overhead of taking a wrong choice by throwing away a computation and starting again.

---

\(^5\) It is possible to create a JSP structure chart for any nondeterministic finite state machine. Any nondeterministic finite state machines can be converted to deterministic finite state machine. Deterministic finite state machines are equivalent of JSP.
8.6.1 When to save and restore backtracking states

In saving the state it is important, for efficiency reasons, to remember that state changes may be one of three types as described in the following list.

- Harmful – in this case it is necessary to save and restore the state on backtracking. An example could be a counter that must be reset to an initial value before each alternative is tried and the counter is modified while an alternative is being tried.

- Neutral – in this case the decision to save and restore the state depends upon ease of implementation and efficiency criteria. Normally, such states would not be saved and restored unless making an exception is more costly than saving and restoring. An example here is the value of temporary variables in a computation. If each alternative initializes such variables no matter what, then either saving and restoring the state, or not saving and restoring the state of such variables will not change the computation.

- Beneficial – in following an alternative, partial computations may be made which would have to be redone in other alternatives, if backtracking were to take place. In such cases it would be foolish to throw away the result and recompute it. The state of such variables would never be saved and restored upon backtracking. In fact, in some problems, additional state variables are used to keep track of beneficial side effects to make alternatives faster to check out. An example would be the reading of an input stream where two different alternatives are identical for the first N tokens and then differ. There is no point in rereading and reanalyzing the first N tokens if backtracking forces us to the second alternative.

It may be thought that the common lead part could be factored out. But an examination of JSP diagrams will often show that the structure is different and so the program with the common lead part factored out would not reflect the required structure. Future changes could be difficult to implement even though the current implementation is superficially a correct model.
8.6.2 An example of backtracking implementation

The program shown below is an implementation\(^6\) of how the Leader or Double choice in Figure 1 could be handled in the problem of recognizing strings with at least one pair of consecutive 0s or 1s.

What we need is to buffer the input. In this problem we need to keep at least two characters in memory. If we see a 0 then we need to look ahead to see if the next character is a 0. If it is a 0, then we recognize a Double, otherwise we need to backtrack and recognize more Leader. Correspondingly, we need to do the same thing if we see a 1.

Variables:

- `buffer[N]`: A buffer of \(N\) characters, \(N \geq 2\). The buffer is used as a wrap-around queue.
- `cc`: Pointer to the current character in the buffer.
- `btp`: Backtrack point in the buffer. Points to the first character of a potential Double.
- `eb`: Pointer to the end of the buffer. The last character in the buffer.
- `found`: True if a Double has been found. False otherwise.
- `ch`: A single character of input. Used to loop over the Trailer.

Start of program

```c
cc = 0; found = FALSE  /* Assume not found at first. */
```

Read in the first two characters of the string. We assume a robust `getChar` operation that continually returns EOF on every call, if we are at the end of file, and does not fail. Loop over the input string.

```c
buffer[0] = getChar; buffer[1] = getChar; eb = 1
```

while (not found) and (buffer[cc] <> EOF)

Hypothesize a double 0. While the Leader is the left-most option and seemingly needs to be checked first, it is the absence of a Double that keeps us in Leader. Thus, Double must be tested for and if not found we have Leader.

```c
btp = cc  /* Save the state */
if buffer[cc] = 0
{ cc = (cc+1) mod N  /* Advance to the next character */
  if buffer[cc] = 0 { found = TRUE }
  else { cc = btp }  /* Backup, restore state */
}
else { cc = btp }  /* Backup, restore state */
```

Hypothesize a double 1. Notice that with alternatives after the first one we must check the found condition. If we have not found a suitable choice, then try the next alternative. Otherwise terminate the hypothesis structure.

\(^6\) For backtracking problems it is easier to use an extended form of BNF grammar. Section 8.7, “A variation on Backus-Naur Form” describes one such grammar and how backtracking can be implemented with context free grammars.
Implementation Step

```c
if not found
    { btp = cc                   /* Save the state */
        if buffer[cc] = 1
            { cc = (cc+1) mod N        /* Advance to the next character */
                if buffer[cc] = 1 { found = TRUE }
                else { cc = btp }        /* Backup, restore state */
            }
        else { cc = btp }          /* Backup, restore state */
    }

Hypothesize the Leader

    if not found
        { cc = (cc+1) mod N    /* Advance one leader character */
            eb = (eb+1) mod N    /* Prepare to read the next input character */
            buffer[eb] = getChar
        }

}  // End of the while loop

We either have an error, or Trailer

    if found
        { ch = getChar        /* Loop over the Trailer */
            while ch <> EOF { ch = getChar }
        }
    else /* There is an error. The input does not contain a Double */

8.7 A variation on Backus-Naur Form

The definition of a Backus-Naur Form like notation is the definition of a **meta language**. We are defining a language the purpose of which is to specify other languages. What we do is add a bit of **syntactic sugar** to the formal definition of a grammar. With **syntactic sugar** we introduce new notational forms that help the user to take short cuts or abbreviate the formal definitions to make grammars easier to define. The new notation neither adds nor deletes any expressive power of the language of specifying grammars.

The notation is used to describe the syntax of context free structures. The virtue of the notation is that it uses the ASCII character set and can be typed in machine-readable form. Consequently, it is possible to construct tools that automatically convert the BNF-like notation to executable programs. The major changes from standard BNF is to simplify recursive definitions by introducing looping constructs, and to simplify the specification of optional items.

8.7.1 Basic notation definitions

We present here a table of symbol definitions for the notation. An extended annotated example of the use of the notation is given in Section 6.4.2. In this notation we restrict ourselves to context free grammars.

"..." is a literal string. The double quotes indicate the boundary of the string and are not a part of the string. The string must appear exactly as indicated within the quotes including the distinction between upper and lower case characters. The double quote character does not occur as part of the string. Use single quoted literal strings to designate a double quote – e.g. "'".

'...' is a literal string. The single quotes indicate the boundary of the string and are not a part of the string. The string must appear exactly as indicated within the quotes including the distinction between upper and lower case characters. The single quote
character does not occur as part of the string. Use double quoted literal strings to designate a single quote – e.g. "'".

Where there is no ambiguity a literal string can simply represent itself without the quotes. Quotes are required only to enclose special symbols that are meta characters – characters in the notation – or other special characters such as space and tab.

<...> is the name of an entity, a nonterminal. Thus, <FlexOr file>, <unnamed section>, <named section> are the formal names for a ‘FlexOr file’, an ‘unnamed section’ and a ‘named section’. The angle brackets indicate that the entity has further structure that is defined in other rules.

::= means ‘is defined as’.

[...] means the contents, ..., are optional; the contents may occur zero times or once.

(...) means the contents, ..., must occur exactly once; the contents is not optional.

+ means to loop. It is used as a prefix to ‘(...)’ and ‘[..]’. The use of + in this way is a deviation from standard mathematical notation where + is used as a superscript suffix. The decision to use a prefix form lies in the concept that it is easier for people to understand that a loop exists by stating this at the beginning of the loop rather than at the end.

+[...] means the contents, ..., must occur zero or more times. Note that we do not need the Kleene star operator, *, to indicate zero or more. The combination of the + meaning loop and optional for ‘[..]’ gives us the desired meaning.

+(...) means the contents, ..., must occur one or more times.

white space consists of blanks, tabs and end of line characters. It is used to separate the components of a rule. It also denotes the ‘and’, or concatenation, operation when separating two entities.

, the comma is used as a choice operator. It means ‘or’.

~ the tilde character is used to mean negation.

; terminates a rule.

8.7.2 An Example of the notation

This section presents an extended example of using the BNF-like notation defined in Section 2.1. Grammatical notations can be used to describe data structures, parsing, recognition or output of text, or a computation to be carried out. We are going to describe one type of breakfast, <breakfast>, that a person may wish to eat. Hence we want to develop a rule or definition for <breakfast>.

<breakfast> ::= ... ;

AND – the sequence operator

‘And’ represents the sequence operation. It indicates that the parsing, recognition, output or computation of its operands to the right of the operator is done after the parsing, recognition or output of the operands to the left of the operator. The ‘and’ operation is denoted by the juxtaposition of symbols – white space can be used as a separator to improve readability. In the following a breakfast entree consists of bacon and eggs. In a computational sense, we have the bacon first and the eggs after.

<breakfast> ::= <bacon> <eggs> ;
Implementation Step

OR – the choice operator

‘Or’ represents choice. It indicates that the parsing, recognition, output or computation of its operands can be any single one of its operands. We use parenthesis or brackets to indicate the scope of the choices. In our notation, we use two types of scope designators because when a choice exists we want to distinguish between ‘must choose’ and ‘do not have to choose’. The following shows two examples of choice.

\[
\begin{align*}
\text{<bread>} & ::= ( \text{toast, roll, croissant} ) ; \\
\text{<bread>} & ::= [ \text{toast, roll, croissant} ] ; \\
\end{align*}
\]

The above defines a sub-rule for our breakfast. We want to be able to select one of three types of bread. The first rule says we must have one of the bread types – for example the waiter will bring bread to the table and we will be charged for it. The second rule says that we may have one of the bread types, but we may choose to have no bread at all – if we do not choose a bread type the waiter will not bring us any bread and we will not be charged for it.

We do not require the use of quotes because there is no ambiguity in the terms ‘toast’, ‘roll’ or ‘croissant’.

NOT – the negation or absence operator

Not represents the specific absence of something. It is used to indicate that the parsing, recognition, output or computation of its operand would be an error, it is not permitted or it should not happen. In its place anything else can be done. Its main use is to simplify grammar rules by not forcing us to list all but a few options. The following states that a bread covering can consist of anything but jam.

\[
\text{<bread covering>} ::= \sim \text{jam} ;
\]

Zero or more – an iteration operator

To have a reasonable computational model it is necessary to have iteration included. In our extended BNF we have an iteration construct that permits us to loop zero or more times. In effect it says that we may parse, recognize, output or compute the construct within the brackets zero or more times in sequence. In the following, we are saying a person may have as many cups of coffee as they want, including having no coffee at all.

\[
\text{<coffee>} ::= +[ \text{cup} ] ;
\]

We have the effect of zero by using the brackets meaning optional and the loop is indicated by the +.

One or more – an iteration operator

In our extended BNF we have an iteration construct that forces us to loop at least once. In effect it says that we must parse, recognize, output or compute the construct within the parenthesis at least once in a sequence. In the following, we are saying that a person must have at least one cup of coffee but may have as many as they want after that.

\[
\text{<coffee>} ::= +( \text{cup} ) ;
\]

We have the effect of once by using the parenthesis meaning must choose and the loop is indicated by the +.

A larger example

In our rather restricted menu a person may choose to have juice. If juice is chosen, then it must be one of orange, grapefruit or tomato. The person must have eggs and must select one of ham, bacon or sausage. Bread is optional and if bread is selected, then the person may select a sweet covering of jam, jelly or honey. If bread is not selected then the person cannot have a sweet covering. For a beverage the person may have nothing or an unlimited supply of tea.

\[
\begin{align*}
\text{<breakfast>} & ::= [ \text{<juice>} ] \text{eggs} ( \text{bacon, ham, sausages} ) \\
& \quad [ \text{<bread>} [ \text{<sweet covering>} ] ] \\
& \quad +[ \text{“cup of tea”} ] ; \\
\text{<sweet covering>} & ::= ( \text{jam, jelly, honey} ) ;
\end{align*}
\]
<juice> ::= ( orange, grapefruit, apple );
<bread> ::= ( toast, roll, croissant );

8.7.3 Programming semantics

For each of our language constructs we need to indicate the semantics that corresponds to it. Since backtracking is involved we will need to be able to save and restore the state of the computation. The state is problem dependent but must contain a representation for anything that must be recomputed if backtracking is to take place. Items having detrimental effects must be recomputed, while items having neutral effects may or may not need to be recomputed (efficiency or ease of definition hold sway). Items having beneficial side effects should not be recomputed. Note that the state includes the flag \texttt{looped} described later. The following three actions are required.

- saveState – save the state of the computation on a stack.
- restoreState – restore the state from the top of the stack.
- dropState – remove the state from the top of the stack.

For the looping constructs it appears to be necessary to have a counter in the state to count the number of times the loop has been taken. But we only need a Boolean flag, \texttt{looped}, which is true if we have looped at least once and false otherwise. The flag is required to distinguish the semantics between '+'[...] and '+'(...). The following three operations are required.

- set \texttt{looped} to true.
- set \texttt{looped} to false.
- test the state of \texttt{looped}.

At each point in our computation we need to know if we have parsed, recognized, output or computed a particular construct. Thus we have a happy flag. ‘happy’ is true if we have successfully parsed, recognized, output or computed the previous construct and false otherwise. ‘happy’ is not part of the state. The entire computation uses only a single happy indicator.

- set happy to true.
- set happy to false
- test the state of happy.

We need an exit operation to exit a local construct. A local construct may be a rule (if we are not within brackets or parenthesis), may be a choice set (we are within brackets or parenthesis) or may be an option within a choice set (need to abandon a choice and try another one). We examine each of our language constructs in turn.

and – whitespace

When we are at the space between constructs we need to determine if the computation is to continue.

\begin{verbatim}
if happy then continue
else if in a choice set then advance to end of the choice
   else exit the rule
fi
\end{verbatim}

or –,

We are at the end of a choice.

\begin{verbatim}
if happy then advance to the end of the choice set
else restoreState ; happy ← true ; advance to the next choice
fi
\end{verbatim}
not

The semantics takes place at the end of the construct.

\[
\text{happy } \leftarrow \neg\text{happy}
\]
Reverse the state of happy. One of ‘and’, ‘or’, ‘]’ or ‘)’ still occurs after the construct and will take over with their usual semantics.

Optional – […]
At the ‘[’

\begin{align*}
\text{enter the construct} \\
\text{saveState} \quad \{ \text{prepare for backtracking} \}
\end{align*}

At the ‘]’

\begin{align*}
&\text{if } \neg\text{happy then restoreState} & \{ \text{None of the options was chosen} \} \\
&\text{happy } \leftarrow \text{true} & \{ \text{Force happiness, effect of optional} \} \\
&\text{fi} \\
&\text{dropState} \quad \{ \text{Complement ‘[’} \\
&\text{exit the construct}
\end{align*}

Must choose – (...)
At the ‘(’

\begin{align*}
\text{enter the construct} \\
\text{saveState} \quad \{ \text{Prepare for backtracking} \}
\end{align*}

At the ‘)’

\begin{align*}
&\text{if } \neg\text{happy then restoreState fi} & \{ \text{None of the options was chosen}\} \\
&\text{dropState} \quad \{ \text{Complement ‘(’} \\
&\text{exit the construct}
\end{align*}

Zero or more – +[...]
At the ‘+ [’ – beginning of the construct

\begin{align*}
\text{enter the construct} \\
\text{saveState} \quad \{ \text{Prepare for backtracking} \}
\end{align*}

At the ‘]’

\begin{align*}
&\text{if happy then dropState} & \{ \text{Forget previous state} \} \\
&\text{go to beginning of construct} & \{ \text{Save new state} \} \\
&\text{else restoreState} \\
&\text{dropState} \quad \{ \text{Complement the ‘[’} \\
&\text{happy } \leftarrow \text{true} & \{ \text{Force happiness, effect of optional} \} \\
&\text{exit the construct} \\
&\text{fi}
\end{align*}

One or more – +(...)
At the ‘+(’ – beginning of the construct

\begin{align*}
\text{looped } \leftarrow \text{false} \quad \{ \text{Assume did not loop} \} \\
\text{enter the construct} \\
\text{saveState} \quad \{ \text{Prepare for backtracking} \}
\end{align*}

At the ‘)’

\begin{align*}
&\text{if happy then dropState} \quad \{ \text{Forget previous state} \} \\
&\text{looped } \leftarrow \text{true} \quad \{ \text{Indicate looped at least once} \}
\end{align*}
8.7.4 Special entities

To simplify various rules it is desirable to name ASCII characters that have no easily
distinguishable graphic representation and to name useful collections of characters. The
following table lists the names we are using.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alpha</td>
<td>any alphabetic character</td>
</tr>
<tr>
<td>AlphaNum</td>
<td>any alphabetic character or decimal digit</td>
</tr>
<tr>
<td>Control</td>
<td>any control character</td>
</tr>
<tr>
<td>Digit</td>
<td>any decimal digit</td>
</tr>
<tr>
<td>Eof</td>
<td>the end of the file</td>
</tr>
<tr>
<td>Eol</td>
<td>the end of a line</td>
</tr>
<tr>
<td>Graphic</td>
<td>any printing character except space</td>
</tr>
<tr>
<td>Lower</td>
<td>any lower case alphabet character</td>
</tr>
<tr>
<td>Punctuation</td>
<td>any graphic except alphabetic or digit</td>
</tr>
<tr>
<td>Space</td>
<td>the space character</td>
</tr>
<tr>
<td>Upper</td>
<td>any upper case alphabet character</td>
</tr>
<tr>
<td>Whitespace</td>
<td>any white space characters except for end of line</td>
</tr>
</tbody>
</table>
In a properly documented design each rule and symbol would as fully described as any other design or program text.

File ::= +[ BlankLine , Paragraph ] Eof ;
BlankLine ::= [ SpaceGroup ] Eol Eol;
SpaceGroup ::= +( Space );
Paragraph ::= ( OneLineParagraph , MultilineParagraph ) ;
OneLineParagraph ::= [ SpaceGroup ] Word +[ SpaceGroup Word ] [SpaceGroup ] Eol Eol ;
Word ::= +( Graphic ) ;
MultilineParagraph ::= FirstLine +[ MiddleLine ] LastLine ;
FirstLine ::= [ SpaceGroup ] Word +[ SpaceGroup Word ] [ SpaceGroup ] Eol ~Eol ;
MiddleLine ::= [ SpaceGroup ] Word +[ SpaceGroup Word ] [ SpaceGroup ] Eol ~Eol ;
LastLine ::= [ SpaceGroup ] Word +[ SpaceGroup Word ] [ SpaceGroup ] Eol Eol ;

8.8.3 Implementation notes

To handle backtracking each line would be read into an array. This could be done during the execution of the first alternative, or each line is read as a unit at each end of line. (Remembering to prime the pump by reading the first line when the file is opened). The state to remember contains the following items.

- current character pointer
- failure pointer, where to back up to
- reason for failure

The second and third items can be used for efficiency if rescanning is not wanted. They can be used to factor out the common lead part of many of the objects. If such a choice is made then additional items need to be kept track of such as the start of the nonblank part of a line.

To make an executable program out of the design it is necessary to embed semantic actions into the EBNF rules and to supply the necessary support routines. The following two rules show the semantic actions necessary to write the one-line paragraphs to standard output.

File ::= initializeInput +[ BlankLine , Paragraph ] Eof ;
OneLineParagraph ::= initializeALine [ SpaceGroup ] Word +[ SpaceGroup Word ]
[SpaceGroup ] Eol Eol writeOneLineParagraph ;

The following C++ code defines the semantic actions to be provided by the implementer. Many semantic actions dealing with input of characters are provided in the included library. In particular the semantic actions for the special entities in Section 2.4 are defined.

```c++
#include "/cs/fac/include/grammar.support"
int sl; // point to character that starts a line
void initializeALine() {
    sl = ccp; } // Remember the current character position
initializeInput() // Defined in the above support file.
```

7 For example the following is a semantic action that parses the end of line character.

```c++
void Eol() { if (character[ CCP ] == ' \n ' ) nextChar(); else happy = False ;
where character[ CCP ] is the current character to examine and nextChar() advances the CCP pointer appropriately.
```
Provide the write a line operation to output the one line paragraph. The characters are stored in a ring buffer, character, so the temporary character pointer, tcp, wraps around the buffer. It starts at the character pointed to at the beginning of the line and terminates when it points to the current character, ccp; this character is yet to be recognized and is not a part of the line.

```cpp
writeOneLineParagraph() {
    for ( int tcp = sl ; tcp != ccp 
        ; tcp = (tcp + 1) % CharacterBufferSize )
        cout << character[tcp]; }
```

We need a main program to invoke the first rule and return the state of happiness – in Unix shells 0 means ok so we return the complement "not happy".

```cpp
int main () { File() ; return !happy; }
```

Finally, we need to define the state to save for backtracking and the backtrack operations. The minimum state to save is the current character pointer. To keep the state stack, we use an array. The save and restore operations copy variables. Drop state is pop the stack and not use the popped value.

```cpp
struct State { int ccp; }
State stateSpace[100];
int stateLevel = -1;
void saveState() {
    stateLevel++; stateSpace[stateLevel].ccp = ccp; }
void dropState() { stateLevel--; }
void restoreState() { ccp = stateSpace[stateLevel].ccp; }
```

### 8.9 Other considerations

The implementation program text is always generated from the design program text using mechanical translations. Never change the design!

Include initial conditions for all processes. For inversion run every process until it communicates giving the reason for the first communication, then invoke the process within a loop.

When scheduling, deadlock and starvation must be prevented.

If a process is looping over getsv operations it needs a way of interrupting itself. Can be periodic – count calls. Another method is to alternate the getsv with reading a time grain data stream and invert with respect to the process supplying the data stream. That way there is one getsv for each data item and we can control the frequency of getsv. In a multiprocessing system can assign priorities to the processes.

The scheduler should read merged input streams otherwise a blocked process would block the entire system.

The more streams we invert on the more opportunity for the scheduler to take control and the more fine-grained scheduling can be done.

May need extra processors for long processes such as sorts.

Adding TGM streams and inverting with respect to them introduces time slicing. The scheduler can then loop over the processes periodically. Must make sure that enough processor time is available for all processes plus scheduler. Make sure that zero time – time between invocations – is not too large.