Chapter 3

Modeling Treatflow Design Primitives and Assertions

This chapter identifies some of the workflow primitives used for healthcare workflow design. Primitives for structured workflow design are specified mathematically and observations are made by analyzing these specifications. In this chapter we present a verification algorithm which is used to study the correctness issues in a healthcare workflow design. Here we propose a modular approach to healthcare process specification. We have defined several operators to facilitate modular composition. Using these operators a medical practitioner can specify a treatment plan in an expression. We have also proposed rewriting rules to generate alternate equivalent treatment plans to facilitate decision making by doctors, patients as well as healthcare managers. In this chapter we analyze the domain and present a comprehensive view on genesis of exceptions and corresponding actions.

3.1 Introduction

Business objectives are met by successful executions of several business processes. A workflow is an ordered sequence of execution of business processes. It describes the exchange of process data, the control flow and the synchronization of process executions, the services provided by business processes as well as the system re-

1This work has been published in proceedings of sixth international conference on information and technology, CIT 2003, Bhubaneshwar, Tata McGrawHill Pages 91-96, 2003.
3This work has been published in proceeding of the 11th international conference on information technology, Bhubaneswar, Proc.IEEE Computer Society, pages 215-220, 2008.
quirements. Workflow system is regarded as an important means of increasing
the productivity of enterprises since it crosses organizational boundaries and facil-
itates the exchange of information among the users. Hence, workflow technology
is crucial to automate business processes and their re-engineering. The workflow
technology is applied to wide range of applications including office automation,
healthcare, insurance, manufacturing and e-governance. In order to facilitate the
use of this workflow technology, generic workflow management systems are being
developed. Thus workflow management systems are software applications that
support the design of workflows, their executions as well as monitoring.

A workflow designer follows different modeling approaches to model
business processes using software tools. In general, the workflow modeling ap-
proaches fall into two categories: structured and formal. Structured approach
follows a systematic way to compose business processes and the structured de-
sign is modular. Each module is represented by a graphic notation and modules
are connected by edges. Further a module can be decomposed into submodules or
atomic actions. In a workflow atomic actions are connected by directed edges. The
direction of an edge shows the control flow during workflow executions. Thus this
model provides a blue print of workflow system design. Such modeling approaches
are popular among software engineers because of their visual appeals. A designer
intuitively verifies workflow system design by going through design diagrams. In
contrast to the structured approach, a formal approach insists on mathematical
statements on business process behaviors and interactions. The use of a formal
approach helps the workflow designers to detect inconsistency, reduce ambiguity in
the development and reason about the system. Hence, the use of formal methods
is being seriously promoted among the designers of workflow systems.

Models for healthcare processes could be complex like the processes
observed in business domains. Modeling healthcare activities in one level could be
complex and cognitively loaded. Hence we propose a modular approach to healthcare process specification. We have defined several operators to facilitate modular composition. Using these operators a medical practitioner can specify a treatment plan in an expression. We have also proposed rewriting rules to generate alternate equivalent treatment plans to facilitate decision making by doctors, patients as well as healthcare managers. Though workflow technology is being used for automating enterprise activities but its uses in healthcare domain is at nascent stage. As healthcare workflow is directly deals with human life, therefore it must take precautions to undesired situations during its executions. In this chapter we analyze the domain and present a comprehensive view on genesis of exceptions and corresponding actions.

Following section discuss about Treatflow specification. The section 3.3 provides modeling of Treatflow activities. Section 3.4 presents Treatflow verification issues and with respect to that an algorithm is also presented. Modularization of treatment is discussed in section 3.5 and section 3.6 presents exception management in case of treatment. Finally we give a summary of the chapter.

### 3.2 Specifying a Treatflow

We proposed a model called Treatflow to specify a treatment by sequencing treatment activities as required to treat a patient. A doctor can make use of the modeling primitives to design a patient specific Treatflow. A Treatflow though similar to traditional workflow used in manufacturing domain still is augmented by special primitives viz. supportive, cooperative that are found useful to model activities in treating a patient. In addition, the traditionally known primitives viz. linear, parallel, split & merge, repeating, nested repeating and choice can also be used in modeling.
In this section specification of patient specific treatment activities in healthcare are discussed with the help of primitives. A hospital example is given below. In order to motivate readers we analyze Treatflow activities of a hospital and show that our primitives can be used to specify Treatflow.

**Case history:** A 60 year obese single old man with history of smoking and strong family history of CHD (Coronary Heart Disease). The patient notices sudden chest pain in early hours of morning. He rushes to the emergency department of a hospital. Administration and preliminary clinical examination by G.P. (General Practitioner) reveals that the person may be suffering from either cardiac ailment or acid peptic disease. The patient complains of shortness of breath and mild sweating and chest pain radiating to abdomen.

**Situation-1:** Routine blood samples are sent to the department of pathology for investigations e.g. CBP (Complete Blood Picture), Lipid Profile and ECG (Electro Cardiogram). Resting ECG shows changes in some leads. Trop-T (Troponin-T is a cardiac enzyme marker which shows injury to cardiac muscle at an earliest stage) test at the bed side is positive. The patient is shifted to cardiac ICU (Intensive Care Unit). The doctor prescribes dispirin 300mg tablet, clopidrogel 300mg tablet and 1 sorbitrate 10mg tablet and the doctor sends for cardiac enzyme profile. He is put in oxygen inhalation and continous cardiac monitoring. The patient is subjected to pre-operative checkup for operation in OT (Operation Theater) Pre-anesthetic checkup, peri and post operative recovery is done by anesthetist. After operation (Coronary angiogram) the patient is shifted to post-operative ward and is prescribed drugs and the patient is discharged after observing for an week.

**Situation-2:** ECG is normal. Then the patient is investigated in the line of Acid Peptic Disease (APD). The emergency doctor finds out an episode of binge drinking in the previous night. Intravenous ranitidine and anti spasmodic medication is
given. To counteract the side effects of disprin bolus dose of intravenous omeprazole and liquid sucralgel given. The patient improves and is observed for few days before discharging. When the symptoms are not abated, then upper-GI (Gastro Intestinal) endoscopy is done. If the upper-GI endoscopy is positive then prescribe drug for Duodenal Ulcer(DU). The patient improves and is advised a course of drugs for a period of six months.

The above scenario that takes place in a hospital can be specified in terms of Treatflow activities. These Treatflow activities can be specified in several ways namely linear, parallel, split & merge, repeating, nested repeating, cooperative and supportive etc. Further we present the analysis in structural form. We represent a Treatflow graph comprises of nodes and edges. In this case nodes are represented by rectangle. And split node and And merge node also represent with the help of a rectangle. Whereas in case of And split there is one incoming edge and more than one outgoing edges and in And merge there are more than one incoming edges and one outgoing edge. The same is true for Or split and Or merge, but the Or split and merge is represented with the help of a circle. But in case of linear it can have at most one incoming edge and at most one outgoing edge. The flow of execution is shown by directed edges.

3.3 Modeling Treatflow Activities

A graphical representation of a Treatflow in Figure. 3.1 shows the activities with respect to the said case study. Though pictorial description of treatment activities is easy to understand still not useful to reason about. For example, an analyst may be interested in tracing the completeness of a treatment using a given Treatflow. Particularly, the automation of a reasoning process requires process specification of treatment activities. In order to facilitate the process in this section we have
identified the types of activities and discuss on their concrete specification and behavioral correctness. Before discussing on individual category of primitive we present a generic syntax to describe a Treatflow activity. The syntax of a Treatflow activity specification is as follows:

**Treatment activity** :: <name>

**Status** :: {<active/waiting/start/end>}

**Attributes** ::

**Resource-required** :: {<name> <description>}

**Operation-time** :: {<time>}

**Observation** :: {<show(list-of-attributes)>}

**Hiding** :: {<hide(list-of-attributes)>}

**Record-attribute** :: {<name> <value>}

**Prec** :: {<predicates>}

---

**Figure 3.1: Treatflow Diagram**

The diagram shows the process flow of various activities and their interactions, such as administration, recording of symptoms, clinical exam, investigation, do-CBP, take-ECG, do-LipidProfile, collect-patient, diagnosis, do-MPD-Treat, take-drug, relief, no-relief, do-Endoscopy, observe, no-improve, improve, advice-drug, stop, do-CHD-Treat, do-Bedside-TopT, shift-to-cardiac-ICU, monitor-ECG, take-drug-CHD, do-Care-Enzyme, shift-to-ward, pre-operative-checkup, surgery, post-operative-checkup, shift-to-post-operative-ward, prescribe-drug.
Postc :: {< predicates > }
Safety-condition :: {<predicates> }
Sideeffect :: {<attribute,description>}
Patient-complaint :: {<attribute,description >}
Description :: {do <name> with <Prec> and <Postc> for <time> }
End

A unique name is given to each treatment activity. Once an activity is started, activity status gives a clear picture about activity either it is active or waiting before it has to come to an end state. An activity can assume different states e.g. active, waiting, start and end. Each attribute has a name and associated value(s). At different states during life cycle of an activity, its attributes can assume different values. In order to execute, an activity may need some resource which is represented by resource-required. Specific time period is allotted with the help of operation-time for the execution of an activity. Some treatment related data has to be recorded with the help of record-attribute while an activity is under execution.

During execution patient must be kept under observation. In order to start, an activity depends on satisfiability of Prec (pre-condition) and successful execution satisfies Postc (post-condition). Safety measure for each activity is taken into consideration with safety-condition. Patient-complaint has to be maintained for each activity in order to make change in the treatment. Each activity has to be executed with corresponding Prec and Postc for the specified period of time is described by description. Input and output attributes associated to an activity are also defined. Further we model a new idea on visibility of attributes to a patient. In a Treatflow activity, a patient is particularly directed to make note of some attributes specifying health parameters. And some are kept hidden for the
patient with the help of hiding attribute.

Later, it is shown that this specification of a Treatflow helps in analyzing and reasoning of an activity. For illustration we present an instantiation of an activity as follows.

**Treatflow-Activity** :: <Take-drug>

*Status* :: {<active>}

*Attribute* ::

*Resource-required* :: {<drug> <Disprin 300mg,Clopidrogel 300mg,Sorbitrate 10mg>}

*Operation-time* :: {<24 hours>}

*Observation* :: {<chestpain,sweating,change-in-ECG>}

*Hiding* :: {<ECG-report>}

*Record-attribute* :: [<ECG> <reading>]

*Prec* :: {<is-chestpain(yes)> ∧ <is-sweating(yes)> < change-ECG(yes)> }

*Postc* :: {<is-chestpain(no)> ∨ <is-sweating(no)> < change-ECG(no)> }

*Safety-condition* :: {<is-normal(BP)> }

*Sideeffect* :: {<headache,severe>}

*Patient-complaint* :: {<headache,severe>}

*Description* :: [do <Take-Drug> with <is-chestpain(yes)> ∧ <is-sweating(yes)> and <is-chestpain(no)> ∨ <is-sweating(no)> for < 24 hours>]

**End**

In order to model Treatflow, we have identified several execution control primitive. In this section each of these primitive is discussed with the help of an example.

- **Linear Activities**

  A linear chain depicts a serial execution of activities with a start and a end
activity. For example Administration, Record-symptom and Clinical-exam can be performed in sequence, with Administration and Clinical-exam as the start and the end activities as shown in Figure.3.2. These two end points provides a boundary between which activities involved in a linear chain gets executed. If a set s with activities (p,q,r) form a linear chain LC then the elements of the set are ordered. Thus

\[ \text{LC}(s) = p,q,r \]

Where p and r are start and end activities

\[ p \rightarrow q \rightarrow r \]

that q is succeeded by p and preceded by r.

Without loss of generality two features are introduced viz: Prec and Postc for each activity stating pre-condition and post-condition respectively.

For an activity p, p.Prec and p.Postc indicate pre-condition and post-condition respectively. For example if p and q are in sequence i.e (p,q) then

\[ p.\text{Postc} \Rightarrow q.\text{Prec} \]

Execution of an activity depends on satisfiability of pre-condition and successful execution satisfies post-condition. As the activities are executed serially pre-condition of an activity is satisfied by post-condition of its previous activity.

\[ [p.\text{Prec}]p \Rightarrow p.\text{Postc} \]
\[ [p.\text{Postc}] \Rightarrow q.\text{Prec} \]
\[ [q.\text{Prec}]q \Rightarrow q.\text{Postc} \]

Similarly an activity q is said to be linear to its preceding activity p when

\[ q.\text{StartTime} \geq p.\text{EndTime} \]
\[ p.\text{EndTime} \geq p.\text{StartTime} \]
and pictorially shown as $p \rightarrow q$

On execution of activity $p$, its status assumes Terminate, i.e. $p.\text{Status} = \text{Terminate}$ and the activity $q$ is started. Thus

$$p.\text{Status} = \text{Terminate} \implies q.\text{Status} = \text{Active}$$

![Figure 3.2: Linear Activities](image)

The **dynamic behavior** of linear activities $p \rightarrow q$ can be verified by making use of their pre- and post-condition specifications, i.e. $p \rightarrow q$ execution is correct if

$$p.\text{Postc} \implies q.\text{Prec}$$

when $p.\text{Postc}$ is the post-condition of the activity $p$ and $q.\text{Prec}$ is the pre-condition of the activity $q$.

- **Parallel Activities**

  In contrast to linear execution of activities, a set of activities can be carried out simultaneously for saving time or as required by a treatment procedure. A set of activities $P = \{p_i\}_{i=1}^n$ are said to be parallel then all of them are in active state in a given time $t$, i.e.

$$\overset{i}{\parallel} p_1 \parallel \overset{i}{\parallel} p_2 \ldots \parallel \overset{i}{\parallel} p_n \equiv \{ t \in T \mid \forall i \in n. \overset{i}{t} p_i.\text{State} = \text{Active} \}.$$  

Where $T$ is the time period spanning total execution period of all the activities, $\overset{i}{t} p_i.\text{State}$ indicates the state of activity $p_i$ at time $t$. A set of activities can be in parallel for a time duration. In Figure 3.3 two activities Take-drug-CHD and Do-CardiacEnzyme are specified as parallel, i.e. a patient has to go for CHD treatment and at the same time different treatment
activities like taking drug for CHD (Take-drug-CHD) and require test (Do-CardiacEnzyme) can be carried out. An activity \( p_i \in P \) obviously starts after the execution of its just preceding activity as discussed in case of linear activity. The interesting points in parallel treatment activities are conditional compatibility and non-race condition on resource requirements. The following rules are proposed for verification of dynamic behavior of parallel activities. If \( p_i \parallel p_j \), i.e., activities \( p_i \) and \( p_j \) are parallel treatments

\[
p_i \rightarrow p_j.Inv \land p_j \rightarrow p_i.Inv
\]  

(3.1a)

and

\[
p_i.Resource \cap p_j.Resource = \emptyset
\]  

(3.1b)

where \( p_i \rightarrow p_j.Inv \) means execution of \( p_i \) does not make \( p_j.Inv \) untrue, i.e., non-interference of treatments.

\( p_i.Inv \): health safety conditions associated to treatment \( p_i \)

\( p_i.Resource \): Resource required for treatment \( p_i \) (Note: Excludes sharable resources and resources that can be made multiple copies)

- **Split & Merge Activities**

Split and Merge are pseudo activities as these do not have tangible implementation. Rather these are the abstractions defined over a set of parallel activities. A set of parallel activities while originate concurrently we specify the point of origination as pseudo activities of sort 'Split' and common point...
they terminate is specified as a pseudo activity of sort 'Merge'. Each Split or Merge activity can be meaningfully labeled but do not have specific details as we see in case of an activity investigation in Figure. 3.4 as because a pseudo activity do not have an implementation. For speedy execution an

Figure 3.4: Split and Merge Activities

activity may split to several activities and they can run in parallel or in sequence and on their termination merge into an another activity. For example the process of Investigation is splitted into activities such as Do-CBP, Do-LipidProfile and Take-ECG and after execution again merged to the activity Collect-report where split and merge are the pseudo states. Let x,p,y are three activities being executed in sequence and activity p can split to $p_1, p_2, p_3$ activities that run in parallel. On execution of x,

$$x.Postc \Rightarrow p.Prec$$

Say p is split to three activities at time $ts$ so that start times (stimes) of an activity is $ts$ so as per definition

$$\text{split}(p) = p_1, p_2, p_3$$
$$\text{stime} (\text{split}(p)) = ts$$
$$\Rightarrow \text{stime} (p_1) = \text{stime} (p_2) = \text{stime} (p_3)$$
The activities $p_1 \ldots p_3$ are executed on parallel and may end at different times say end time (i.e. $etime$). For generality, let’s assume etimes are different. So

$$
etime(p_1) = t_{e1}$$
$$
etime(p_2) = t_{e2}$$
$$
etime(p_3) = t_{e3}$$

On completion they merge (their outputs are merged) to provide the output of $p$ and their cumulative post-conditions satisfy pre-condition of the succeeding process $y$. These definitions can be formally written as

A set of activities $\{p_i\}_{i=1}^m$ succeeding to an activity $x$, are said to be split activities when

$$t_{p_1} \parallel t_{p_2} \ldots \parallel t_{p_m}$$

and

$$x.\text{EndTime} \leq p_i.\text{StartTime} \quad \forall i = 1 \ldots m$$

An example of split activities can be seen in Figure. 3.4. Investigation activity is split into three implementation activities viz Do-CBP, Do-LipidProfile and Take-ECG which are carried out in parallel. The following properties of parallel activities qualify a merge point.

1. $p_i.\text{State} = \text{Terminate} \quad \forall i = 1 \ldots m$

2. $\forall i, p_i.\text{EndTime} \leq y.\text{StartTime}$

where $y$ is the activity that succeeds to the merge point. For example, on termination three parallel activities Do-CBP, Do-LipidProfile and Take-ECG and then Collect-report is carried out. So, the StartTime for the activity
succeeding to parallel activities should be greater than equal to the latest EndTime all at which all the parallel activities must have been terminated. 

**Dynamic behavior** of activities at split point should not be such that their pre-conditions contradicts to each other i.e

\[
\alpha \in p_i.Prec \implies \neg \alpha \notin p_j.Prec, i \neq j
\]  

(3.2)

where \(\alpha\) is a predicate and \(p_i.Prec\) is set of predicates of \(p_i\) as its pre-conditions. After split, the parallel activities satisfy the properties discussed before. The dynamic behavior of activities at merge point should satisfy the following condition. Post-conditions of these merging activities should not be in conflict i.e.

\[
\alpha \in p_i.Postc \implies \neg \alpha \notin p_j.Postc
\]  

(3.3)

Rules 1-3 can be used to verify dynamic behavior of parallel treatment activities.

• **Repeating Activities**

Alike looping in programming language, a set of activities may be executed repeatedly for some duration or until a given condition is satisfied. Repetitive execution of certain activities can also be found in healthcare workflow. For example, a patient may be advised to take antibiotics for three days or to undertake an exercise till the normality is achieved. An example can be seen in Figure. 3.5. This illustrates the need for modeling activities that are to be executed repetitively for specified time period (T) or until certain loop conditions (Lc) are satisfied. Formally a set of repetitive activities \(P = \{p_i\}_{i=1}^n\) with \(p_1\) the start activity and \(p_n\) the last should satisfy

\[
p_1.Prec \land p_i.State = Active_{i=1}^n \land \neg(p_n.Postc)
\]
The execution of a loop is said to be correct when

\[ P.\text{Postc} \implies P.\text{Lc} \]

i.e. The post-conditions due to set of activities P implies the satisfiability of loop condition of P i.e. P.Lc. The termination condition of a set of repeating activities can be specified by \( p_n.\text{Postc} \) i.e. post-condition of the last activity or by a time period T specifying that all the activities in P are to be executed T times.

The **dynamic behavior** of a set of repeating activities can be verified by the rules

- if \( p_1.\text{Prec} \) true then \( \forall p_i \in P, p_i.\text{State}=\text{Active} \)
- if \( p_n.\text{Postc} \) true then \( \forall p_n \in P, p_n.\text{State}=\text{Terminate} \)
- if No-of-Execution(P) = T then \( \forall p_i \in P, p_i.\text{State}=\text{Terminate} \)

No-of-Execution() is a function that returns the number of times P is executed.

Treatment for a particular ailment in general follow a generic procedure. Still it has to be person specific, as a patient may respond to some abnormal behavior based on his/her personal constitutions. A treatment needs to respond to such events. In order to enable Treatflow specifying such requirements we have added three primitives viz Nested Repeating, Cooperative and Supportive.
• Nested Repeating Activities

While a loop contains a set of activities, a nested loop contains a set of loops where loops are ordered from the inner-most loop to the outer-most loop. Conceptually the concept is similar to nested looping constructs found in a programming language. We have seen the utility of nested loop in Treatflow modeling as shown in Figure 3.6. For example, some people may develop cardiac behavior due to chloro-quinine treatment. In such case, chloro-quinine treatment is to be suspended for cardiac treatment. For a successful completion of the treatment, chloro-quinine treatment is to be resumed. Such a treatment pattern is modeled as nested repeating activities. Assume two activities $p_i$ and $p_j$ are nested and they form outer-loop and inner-loop respectively. The rules specifying such activities are as follows:

1. $\exists p_i \mid C-in(p_j,p_i)$
2. $\exists e \mid active-in(e,p_i) \land triggers(e,p_j)$
3. $\forall p_i \exists p_i.Postc$ or $p_i.T$

Figure 3.6: Nested Repeating Activities
The relation $C$-$in$ specifies that the activity $p_j$ is contained in the activity $p_i$. The first rule defines the outer and the inner loop. The second rule defines the event $e$ on being active or appearing during execution of $p_i$ may trigger execution of $p_j$. The third rule ensures termination of activities in nested loop. The **dynamic behavior** of nested repeating activities $p_i$ and $p_j$ when $p_i$ contains $p_j$ are:

1. $p_j$.State=$Active \Rightarrow p_i$.State=$Suspend$
2. $p_j$.State=$Terminate \Rightarrow p_i$.State=$Active$
3. $p_i$.startTime $<$ $p_j$.startTime and
4. $p_i$.endTime $>$ $p_j$.endTime

The above rule summarily describes that activity in outer-loop is suspended until the nested activity is terminated and then the execution of the outer loop is resumed. The temporal relations stated in the third rule discretized the execution time intervals for both the activities. Analysis of nested repeating activities exhibits mutually exclusion of the repeating activities.

- **Choice**

As in Figure: 3.7 from the activity of Diagnosis either the activity of Do-Endoscopy or Do-bedside-Trop-T can be carried out depending on the choice.
Let $p$ and $q$ are the activities to be executed. An activity $p$ can be decomposed into either $p_1$ or $p_2$ depending on the choice $c_1$ or $c_2$ respectively.

Before executing the activity $p$, the pre-condition of $p$ must be true. The activity $p$ is further refined to $p_1$ or $p_2$ under conditional choices. So for execution of $p_1$ and $p_2$ we need to extend pre-conditions as shown below.

\[
\begin{align*}
[p \text{Prec} \land c_1]p_1 & \Rightarrow p_1 \text{Postc} \\
[p \text{Prec} \land c_2]p_2 & \Rightarrow p_2 \text{Postc} \\
p \text{Postc} & = p_1 \text{Postc} \cup p_2 \text{Postc} \\
p \text{Postc} & \Rightarrow q \text{Prec} \text{ and } [q \text{Prec}] q \Rightarrow q \text{Postc}
\end{align*}
\]

The **dynamic behavior** for the choice activity is as follows.

As an activity succeeding to another activity with choice, it is possible to trace the choice path traversed during execution of the later process. Let $q$ succeeds to $p$ with choices $p_1$ and $p_2$ and the path through $p_1$ is traversed. Then according to definition we can infer the following.

\[
\begin{align*}
q \text{Prec} & = p \text{Postc} \\
p \text{Postc} & = p_1 \text{Postc} \\
\therefore q \text{Prec} & = p_1 \text{Postc}
\end{align*}
\]

Thus at $q$ it can be decided that the process is reached traversing the path passing through $p_1$.

- **Cooperative & Supportive Activities**

  A set of parallel activities are cooperative / supportive when they interact among themselves during their execution. There are two-way interactions among cooperative activities while one way interaction exists among
supportive activities as shown in Figure. 3.8. For example, the activity Operation-OT (Operation Theater) consists of two parallel activities viz. Anaesthetic-checkup and surgery. During operation two-way interactions exist to increase / decrease level of anaesthesia depending on the requirements demanded by surgery. Also depending on status due to Anaesthetic-checkup, Surgery is guided. The rules that govern such activities are as follows. When p and q are cooperative then at time(s) during their execution one requests another for some information m and eventually the later gets a reply from the former. The rule also tells that activities can not terminate before the scheduled pairs of Request-Reply events take place.

\[(p, q, \text{Cooperative}) \Rightarrow \{\exists t, t' \mid \text{Request}(p, q, m, t) \land \text{Reply}(q, p, m, t') \land (t' > t) \land (p, q, \text{Parallel})\}\]

A set of parallel activities are said to be supportive when one way interaction exists during their execution. An activity is termed to be supportive when it facilitates functioning of another activity. The concept of support means enabling pre-conditions or mitigating undesired effects due to an activity. CHD treatment is a good example of supportive activities. The CHD-treatment activity consists of activities like Do-CHD-Treat and Monitor-ECG . The information from Monitor-ECG supports Do-CHD-Treat. An activity p supports an activity q by passing information (Inform) m to q at different times.

![Figure 3.8: Cooperative and Supportive Activities](image-url)
during their execution.

\[(p, q, Supportive) \implies \{\exists t \mid Inform(p, q, m, t) \land (p, q, Parallel)\}\]

The **dynamic behavior** of supporting activities, say an activity q is supported by an activity p means

1. \(p.\text{Output} \implies q.\text{Prec}\)
2. \(p.\text{Output} \implies q.\text{Invariant}\)
3. \(p.\text{Output} \implies \neg q.\text{Use}\)

The specification of p and q activities should satisfy at least one of the above three rules to say that the activity p supports the activity q. The first rule says the output p implies pre-condition of q that is enabling of q. The second rule ensures the maintainability of invariants of q due to output of p. And the third rule tells the output of p makes the undesired side effects untenable. The above specification tells while two activities p and q are being executed concurrently and output due to execution of p satisfies invariants or pre-condition of q or disables \(q^c\) where \(q^c\) is an activity occurring in consequent to p.

### 3.4 Treatflow Verification

On composing a Treatflow it is required to verify it before making it operational as some mistakes might have crept in inadvertently while it being composed by a non-computer science professional. Mostly as we find in literature, workflow verification includes investigation on structural disorders [84]. They propose algorithms to detect errors prominently i.e lack of synchronization and deadlock. Whereas a graph-reduction based algorithm have been proposed [86]. As we have
already discussed, a graph G consists of nodes and edges. A Treatflow graph G
has the following characteristics. A Treatflow graph has one start node and at
least one stop node. For every And-split node there must be an And-merge node.
Figure 3.9 presents a simple Treatflow graph for a patient with general fever either
of kind viral or malaria fever.
A patient with fever on reaching a clinic goes through the registration- a necessity
for hospital administration and then the patient is clinically examined. Based on
this examination doctor intuitively decides a course of treatment either for viral
fever or malaria. For viral fever the necessary pathological tests are done and then
confirmatory treatment is initiated. Alternatively, if malaria treatment is decided then CBP (Complete Blood Picture) and MP (Malaria Parasite)
tests are to be done and then only corresponding confirmatory treatment is to be
initiated.

Because of the Or-node in the Treatflow in Figure. 3.9, the graph does have two
different instances Figure. 3.10(a) and Figure. 3.10(b) showing viral fever treat-
ment and malaria fever treatment respectively.

In case of Treatflow verification we have expanded the scope of verification keeping
the relevance of healthcare domain in view. The scope includes exploring two types
of disorders. We will explain these disorders with the help of pictorial descriptions.

- **Structural Disorder**:

  - **Incompleteness**

    Treatflow is considered as an ordered graph G(V,E) with start and
    stop nodes. Stating initiation and termination of a treatment the in-
    completeness of a treatment exists only if the graph has a path starting
    from the start node and that does not reach the stop node. The presence
Figure 3.9: Treatflow Graph

(a) Treatflow Instance for Viral Fever
(b) Treatflow Instance for Malaria Fever

Figure 3.10: Treatflow Graph for Fever Treatment
of such an incomplete path in a Treatflow indicates that the treatment is incomplete and so not well defined as shown in Figure 3.11.

---

**Lack of synchronization**

Lack of synchronization in a Treatflow occurs if an Or-merge node is traversed from more than one immediate parent node. Presence of such an error will cause repeat execution of a treatment leading from an Or-merge node. Of course such repetitions are not desirable and so a Treatflow must be corrected of such errors. In Figure 3.12 a patient on preliminary diagnosis of malaria fever treatment is advised to carry out CBP and MP tests simultaneously. And then based on the test report prescribe-drug activity has to be started. Though concurrent activities i.e CBP and MP tests are executed simultaneously, but due to the absence of synchronization the next succeeding activity i.e prescribe-drug will be invoked twice which is undesirable. At the time of flow
verification for this lapse must be identified.

- **Deadlock**

Deadlock in a Treatflow may occur when the progress of Treatflow execution depend on an event which is never happened before. Such a situation arises if an And-merge node is not traversed from all of its immediate parent nodes. In Figure 3.13 based on clinical examination, doctor has to decide either to go for viral fever treatment or malaria fever treatment. In both the cases corresponding pathological tests are being done and based on test report, drug for the particular treatment is prescribed. In such case either of the treatment is followed by Prescribe-drug, but not both. But if the execution of Prescribe-drug activity is depend on both the malaria fever treatment and viral fever treatment which is never possible then in such cases error may occur in
a treatment. At the time of Treatflow verification, such type of error must be identified.

Figure 3.13: Deadlock Disorder in a Treatflow Graph

- **Behavioral**

  Other than structural disorder, a Treatflow may have behavioral and temporal incorrectness that need to be identified and corrected for safe and cost-effective treatment. We have classified these disordered into

  - **Retention conflict**

    Some of the health caring activities have retaining effects. For example, once a tetanus injection is given, it has retaining effect for an year. In normal circumstances, say a diabetic patient is instructed to go for a blood test once in a month. So, for the patient such test has one month retention. Similar examples one could find for chemotherapy, x-ray or taking up some antibiotic drugs even. From this we are motivated to specify retention duration of an activity as a rd that gives the duration
for which the effects due to the activity is valid i.e with the given period
the activity need not be again executed to save time, money and even
for the safety of a patient.

Let $G'$ be an instance of a Treatflow graph $G$ with activity 'a' repre-
sented by node $v$ and the node has say two incoming edges. Suppose the
activity has retaining duration $T$. If the activity has been invoked for
the first time at time $t$ then it should not be invoked again for execution
before time $t+T$. For the second time say the node is in being visited at
time $t'$ and if $t' \leq t + T$ then the Treatflow had an error due to ignoring
retention. Retention conflict in a Treatflow Graph $G$ may occur if any
of its instance graph has a node with more than one incoming edges. A
healthcare activity like taking a drug or undergoing a pathological test
has its retentive capability. For example a drug once taken should not
be repeated within certain time period or a test once carried out on a
body should not be repeated within a time period. This is so because
the effects of an activity retains for certain period thus making repeat
of such activity unnecessary. This aspect of a Treatflow, we name as
Retention Conflict as shown in Figure:3.14.

- **Contextual conflict**

Two parallel or concurrent treatment activities are in contextual con-
flict if their respective pre-conditions are mutually exclusive. We con-
sider the case as shown in Figure. 3.15. In case of treatment in health-
care different types of tests i.e Complete Blood Picture and Malaria
Parasite can execute concurrently. All the pre-conditions for the exe-
cution of these test treatment activities are same. If Collect-report as
an activity is included along with those tests, then that results to a
contextual conflict because pre-condition of Collect-report activity and
pre-condition of testing activities are mutually exclusive to each other.

- **Expectation conflict**

Two parallel treatment activities are in expectation conflict if their respective post-conditions are mutually exclusive. As an example types of tests namely Complete Blood Picture and Malaria Parasite can execute concurrently. All the post-conditions for the execution of test treatment activities must be same that means report should be ready after execution of test treatment activities. Once all the test report has been done, then collect-report activity can be started as a succeeding activity. But in case Prescribe-drug is considered as one among the
concurrent activities as shown in Figure 3.16, then post-conditions of all these concurrent activities are not same which may result into an expectation conflict.

3.4.1 Verification algorithm

In order to verify a complete Treatflow graph, all the possible instances of Treatflow graph should be verified. After creating each instance verification should be done. The verification algorithm proposed in this section uses depth-first search and AO* algorithm in order to process Treatflow graph using AND-OR [86]. Definitions used in verification algorithm are as follows.

G: Original Treatflow graph
G': Part of graph G is traversed so far while executing the algorithm
visit_stack(vs) and or_split_stack(oss) and cycle_origin_stack(cos) Create_Instance
starts with the start node "S" in the Treatflow graph G'. While traversing all the child nodes of the non-visited nodes are linked to the graph along with their edges
and also marked those nodes as visited nodes. In each iteration of the algorithm, an instance of the Treatflow graph is created with the help of Create_Instance. After creating an instance, flow of Treatflow graph can be verified using Verify_Instance. Finally next instance can be created with the help of Init_forNextInstance. According to algorithm procedure Create_Instance creates an instance graph $G'$ from $G$ with a start node $S$. If child node(s) of $S$ is not visited in graph $G'$, then child nodes must be marked and linked to the graph. If the child node is an Or-split node then the leftmost unmarked child must be first marked and linked to the graph. Create_Instance procedure uses depth first search mechanism for traversal. Otherwise in all other cases all the child nodes are traversed, marked and linked to the graph $G'$.

Procedure Verify_Instance verifies the instance graph starting from the start node $S$. While traversing any of the non visited node are taken into consideration. In case of non-visited And-split node, pre-conditions of all the child nodes are to be stored. If pre-conditions are mutually exclusive, then the procedure
reports *Contextual Conflict*. Similarly for the case of an And-merge node, post-
conditions of all child nodes are to be stored. If any one of the post-conditions are
mutually exclusive to each other, then the procedure reports *Expectation Conflict*.

If the child node is already visited in case of an Or-merge node, then
the procedure *Verify_Instance* reports *Lack of Synchronization* for that child node.
In case of And-merge node if number of visits does not match with the number of
And-join edges, then the procedure reports *Deadlock* in that child node.

In any iteration for the case of And-merge node, the retention duration
for that node is stored as T and time for visiting that node must be recorded as
$t_i$. In case that And-merge node is already visited once and time for next visit
for that node is considered as $t_{i+1}$. If $t_{i+1} < t_i + T$ then the procedure reports
*Retention Conflict* at that node.

After completing the iteration if the last child node does not exist in the
set of terminating nodes TN, then the procedure reports *Incompleteness* for that
particular instance of the graph. If the instance graph is correct then it calls for
procedure *Init_forNextInstance* to create another instance. *Init_forNextInstance*
chooses the leftmost deepest Or-split node created by *Create_Instance*. In this
case one child has been taken into consideration. For the next instance, marking
are moved to the next child of the Or-split node for the next iteration. Thus
the proposed algorithm verifies the complete Treatflow graph by just verifying a
subset of the instance subgraph.

**Analysis of Algorithm**

While expanding an instance graph, each node in the graph is considered and
possible outgoing edges are linked at each step of the algorithm. The algorithm
repeats at algorithm 2 line number 1 to 19 till all the nodes are considered. Thus
in the process the algorithm always traversed the number of edges i.e less than the
total number of edges present in a Treatflow graph. Thus complexity of algorithm
Algorithm 1 Verify_Treatflow(G)
1: initialize visit_stack(vs)
2: initialize or_split_stack(oss)
3: initialize TN a set of terminating nodes
4: Push_aNode(vs,s)
5: Install_InstanceAt(G,G')
6: repeat
7: Create_Instance(G,G',vs,oss)
8: Verify_Instance(G')
9: Init_for_Next_Instance(G,G',vs,oss)
10: until empty_or_split_stack(oss)

Algorithm 2 Create_Instance(G,G',vs,oss)
1: while Not_empty(vs) do
2:   cn=pop_aNode(vs)
3:   if Not_Linked(cn,G') then
4:     Link_toGraph(cn,G')
5:     Mark(cn,G')
6:     if OrSplit(cn) then
7:       lcn=GetLeftmostUnmarkedNode(cn)
8:       Link_toGraph(lcn,G')
9:       Push_aNode(lcn,vs)
10:      Push_aNode(cn,oss)
11:     else
12:       if Not_Marked(cn,G') then
13:         Push_allchild(cn,G')
14:         Link_toGraph(cn,G')
15:         Push_allchildnode(cn,vs)
16:       end if
17:     end if
18:   end if
19: end while
Algorithm 3 Verify Instance(G')

1: Initialize InstanceVisitStack(ivs)
2: Initialize pre-conditionStack(precs)
3: Initialize post-conditionStack(postcs)
4: Initialize visitcount = 0 ∀ And Merge in G’
5: Initialize i=1
6: Retention_duration(rd)
7: initialize \( t_i \) for time_visit(cn)
8: initialize TN for set of terminating nodes
9: Label_Notvisit(G')
10: Push_aNode(s,ivs)
11: while Not_empty(ivs) do
12:  \( \text{cn} = \text{Pop_aNode}(ivs) \)
13:  if NotVisited (cn) then
14:     LabelVisit(cn)
15:     Push_allChild(cn,ivs)
16:  if visited(cn) \& OrMerge(cn) then
17:      SynchMiss_at(cn) ▷ Lack of Synchronization at child node cn
18:  if Not_visited(cn) \& AndMerge(cn) then
19:      \( T=rd(cn) \& t_i = t(cn) \)
20:      increment i
21:  end if
22:  if visited_AndMerge(cn) \& t_i = t(cn) then
23:     if \(( t_i < t_{i-1} +T) \) then
24:        Retention_Conflict_at(cn) ▷ Retention Conflict
25:     end if
26:  end if
27: end if
if AndMerge(cn) then
    IncrementVisitCount(cn)
    Labelvisit(cn)
    if AndSplit(cn) then
        for all_childnodes do
            Push_prec(childnode(cn), precs)
        end for
        if Not_equal_prec(childnode(cn), precs) then
            Contextual_Conflict_at(cn) \( \triangleright \) Contextual Conflict at cn
        end if
    end if
    if AndMerge(cn) then
        for all_incomingchildnodes do
            Push_postc(incoming_childnode(cn), postcs)
        end for
        if Not_equal_postc(childnode(cn), postcs) then
            Expectation_Conflict_at(cn) \( \triangleright \) Expectation Conflict at cn
        end if
    end if
end if
end if
end if
end while
if visited_AndMerge(cn) \( \wedge \) (VisitCount < CountAndMerge(cn)) then
    Deadlock_at(cn) \( \triangleright \) Deadlock at child node cn
end if
if cn \( \notin \) TN then
    Incomplete(G') \( \triangleright \) Incompleteness in path
end if

2 Create_Instance remains O(E).

In case of algorithm 3 Verify_Instance there is a loop in between line number 11 and line number 50. In this while block, all varieties of verifications are performed while visiting each node of an instance graph. Because of the line number 15, a node may be visited more than once through incoming edges, thus in the algorithm each edge of an instance graph is visited. The number of edges of an instance graph is always less than equal to a Treatflow graph. Hence, the complexity due to Verify_Instance also remains O(E).

The algorithm 4 Init_forNextInstance only performs certain initializa-
Algorithm 4  

\begin{algorithm}
\caption{Init\_forNextInstance\(G,G',vs,oss\)}
\begin{algorithmic}[1]
\State \(cn = \text{pop\_aNode}(oss)\)
\If{\text{NotEmpty}(oss)}
\State \(lcn = \text{GetLeftmostUnmarkedNode}(cn)\)
\State \(\text{Link}(lcn,cn,G')\)
\State \(\text{Push\_aNode}(lcn,vs)\)
\EndIf
\end{algorithmic}
\end{algorithm}

ions of variables like \(cn\) and \(lcn\) for creating next instance. Thus the complexity of the algorithm is negligibly a constant unit.

Algorithm Verify\_Treatflow has a loop in between line number 6 to 10 that contains a repeat loop executing the algorithms 2, 3 and 4. Based on the analysis made for these three algorithms the complexity for executing the repeat loop for single instance remains \(O(E)\). The number of times the repeat loop executed depends on the number of Or-split nodes and their branch factors. These branch factors at Or-split nodes exhibit a multiplying factor while creating instance graph. For example two Or-split nodes with branch factors say \(b_1\) and \(b_2\) will generate \(b_1 \star b_2\) instance graphs. In practice in a Treatflow graph, a number of Or nodes would be very much less than \(N\), the number of nodes in the graph and the average branch factor is usually a small number. These two factors contribute to worst case exponential complexity \(o(E^n)\). However the contribution due to these factors is much less than \(n^2\). If there are \(o\) number of Or-split nodes with average branching \(b\), then the contributing factor is \(o \star b\) which is a constant say \(c\). So, in practice the complexity of this verification algorithm is \(O(E^c)\) where \(c << n^2\).

3.5 Modularization of Treatment

Healthcare processes are complex because they involve many participants dealing with complex information [48]. A large system needs to be divided into blocks so
that the system becomes manageable, clear, modifiable and reusable. These blocks
known as modules are connected among themselves in such a way that module
correctness can be verified from the aspects of horizontal and vertical compositions
[50]. In order to manage a complex system, proposed an operational model viz.
MCPN: Modular Colour PetriNet is proposed which is composed of colored petri
net modules [72]. Though it is not suitable to model at one level only in order to
represent a large and complex service process, therefore [56] propose to model web
services using FSM modules. Refinement and modules of graph transformation
system is introduced in [51]. Here syntax of refinement is given by rule expression
which is based on rule names and rule composition operation symbol.

Now in practice, we observe that a complex health problem treatment
is modularized not only for administering the treatment but also to monitor and
to modify if necessary. This is also useful for administrative uses especially for
estimating cost and arranging resources. Health services are also being offered as
service packages. These trends motivate to plan a treatment in modular form. In
this work we focus on problems in specifying treatment modules and composition
of these modules to form a treatment plan termed as Treatplan. For composition
purpose we have defined several operators e.g. linear, parallel, choice, supportive
and cooperative and have shown that a prescription can be written as an expression
of modules. We also show that using operator properties, a treatment expression
can be rewritten to another treatment expression. Following the proposed method,
a tool can be developed that can assist a medico in prescription writing. The
safeguards specified in modules can alert medico, nursing staff as well as patients
on prescription and administration of wrong medications. These factors emphasize
the necessity of modularization of treatment process in healthcare domain.
3.5.1 Motivating example

Our previous work in last section defines primitives and proposes a method called *Treatflow* to model treatments as a flow of activities as found in workflows. It is observed that modeling a complex treatment i.e with multiple serious complaints could be complicated due to the large number of treatment activities and various order of executions. Hence we propose modularization of treatment plan. In order to motivate readers on modularization concept we would like to take the help of a case study shown below.

A male patient aged 40 yrs complains of increase in appetite, volume of urine and thirst. He also suffers from chest pain. After registration and assignment., the doctor has recorded those symptoms as hyperphagia, polyurea and polydipsia. After clinical examination and assessment of chest pain, the doctor prioritize the presumptive diagnosis of Coronary Heart Disease (CHD) or Acid Peptic Disease (APD). The doctor has made a probable diagnosis of Diabetes Mellitus (DM). Therefore simultaneously both in the line of DM as well as chest pain investigation started. Controlling blood sugar, Diet & exercise are the mainstay of treatment for DM. For chest pain the doctor recommends ECG, UGI endoscopy which helps in making a definite diagnosis of chest pain in CHD the principle of ABC (Airway, Breathing, Circulation) with drug therapy is followed.

These activities are to be arranged in a schematic way using the modeling primitives. Such a schema shows the way a treatment can be carried out, thus said a Treatflow. Different aspects like composing Treatflow activities and verifying a Treatflow are discussed in last section. Here, we would like to introduce the concept of modularization of treatment. For this case, the patient has to undertake a number of treatment activities for treatment like CHD (Coronary Heart Disease),
APD (Acid Peptic Disease and Diabetes). Modeling Treatflow for all these treatments becomes complex to design, verify and follow. This necessitates modular design of healthcare activities. From the stated case study, modules are identified and treatment plan is presented in graph form as shown in Figure 3.17. In next section we will specify the modules and operators for specifying a treatment plan in expression form.

### 3.5.2 Module Specification:

In order to motivate the reader with the help of hospital example given above, we discuss treatment module specification in case of treatment in a hospital. A treatment is a collection of treatment modules which can be represented as

\[
\text{Treatment} = \{TM_i\}_{i=1}^n
\]

whereas a Treatment Module is a set of Treatment Activities. That can be represented as
A specification of a treatment module needs to provide comprehensive view on a treatment module so that the specification will be useful for automation of Treat-flow activities and also enabling a practitioner and a patient to have a quick view on a treatment. Ideally, a treatment module needs to specify conditions at which it can be applied to a patient. In general, conditions specify health status of a patient that can be specified by health parameters and their values. This we say enabling-condition of a module. Similarly, a module should also project expected-condition i.e. health conditions that would result on successful treatment of the module. A treatment may have some limitations at which it is not applicable. These warnings should be recorded in module specification as limitation. Similarly advises are also to be specified to meet the exigencies that may happen during a Treatment Module execution. These are specified as caution in module specification. A module in treatflow-details refers to the name of a Treatflow that records the sequence of treatment activities that are to be executed during execution of the module. In treatment-duration, it records the days for which a module should be executed or the treatment must take place. In case a treatment requires execution of another treatment modules(s) the same should be specified in composed-of, it enlists the names of the modules that are included in a parent module. On instantiation of a module for a patient, we need also to record information on observed health conditions and the period for which a module instantiation has been executed in observed-conditions and treated-duration respectively. The syntax of a treatment module specification is as follows:

\[
\text{Treatment-Module} = \{TA_i\}^n_{i=1}
\]

\[
\text{enabling-condition} :: \{<\text{predicate}>\}
\]
Based on the example cited in Figure 3.17 we specify a module as follows:

**Treatment-Module** :: &lt;Diabetes-treatment &gt;

*enabling-condition* :: { &lt;is_hyperphagia(yes) &gt; ∨ &lt;is_polyurea(yes) &gt; ∨ &lt;is_polydipsia(yes) &gt; ∨ &lt;level(BS, ≥ 200mg %) &gt; }

*expected-condition* :: { &lt;level(90&lt;FBS&lt;110mg %&gt;) &gt; ∧ &lt;is_hyperphagia(no) &gt; ∧ &lt;is_polyurea(no) &gt; ∧ &lt;is_polydipsia(no) &gt; }

*composed-of* :: [ &lt;Diet &amp; exercise &gt; ]

*treatment-details* :: [ &lt;Treatflow_Activity_Diabetes &gt; ]

*limitation* :: { &lt; label(RBS)&lt;150mg %&gt; }

*caution* :: { &lt;”fasting should be avoided” &gt; }

*treatment-duration* :: { &lt;forever &gt; }

*observed-condition* :: { &lt;FBS&gt;&lt;nil&gt;}{ &lt;weight&gt;&lt;nil&gt; }

*treated-duration* :: { &lt;days &gt; }

End Diabetes_treatment
3.5.3 Module Composition:

This section presents a technique to synthesize a treatment that is a combination of several treatment modules. A combination of modules is achieved using composition operators such as linear, parallel, choice, support and cooperate. The rules for composition using these operators are introduced in this section.

- **Linear**

  A Treatment Module $TM_1$ is linearly combined with another treatment module $TM_2$ when the following rule is satisfied.

  \[
  \text{linear}(TM_1, TM_2) :: \\
  \{ \text{exec}((TM_1), \text{exec}(TM_2) \mid (TM_1.\text{exc} \Rightarrow TM_2.\text{enc})} \}
  \]

  From the stated example, enabling-condition (Enc) and expected-condition (Exc) for Administration and Clinical-examination modules are specified as

  Administration.Exc :: $< \text{available-patient-details(patient-id) \land assigned-doctor(patient-id) \land clinically-examined(patient-id)} >$

  Clinical-examination.Enc :: $< \text{available-patient-details(patient-id) \land assigned-doctor(patient-id) \land clinically-examined(patient-id)} >$

  In this case expected-condition of Administration module is equal to enabling-condition of Clinical-examination module. Therefore Administration and Clinical-examination modules can be composed linearly.

- **Parallel**

  Sometimes two or more independent treatment modules viz. $(TM_1, TM_2)$ can be run in parallel when simultaneously their enabling-conditions are implied by expected-conditions of their previous module $TM$. Obviously
such treatments are recommended for speedy recovery. So, formally a parallel composition of treatment modules is specified as: enabling-conditions of \((TM_1, TM_2)\) are simultaneously true.

Formally the rule for parallel composition is stated as:

\[
\text{parallel}(TM, TM_1, TM_2) ::
\{ \exists (TM_1, TM_2) \mid (TM\text{-Exc} \Rightarrow TM_1\text{.Enc} \land TM\text{-Exc} \Rightarrow TM_2\text{.Enc}) \}
\]

For example according to the case study a patient having symptoms of hyperphagia, polyurea, polydipsia and chestpain undergoes both Diabetes-treatment and Chestpain-investigation simultaneously. Because in such case Enc and Exc of all the modules like Clinical-examination, Diabetes and Chestpain-Investigation are specified as follows.

Clinical-examination.Exc:: (level(BS, ≥ 200mg %)) \land Is\_hyperphagia(yes) \land is\_polyurea(yes) \land is\_polydipsia(yes) \land Is\_chestpain(yes)

Diabetes-treatment.Enc:: (level(BS, ≥ 200mg %)) \land Is\_hyperphagia(yes) \land is\_polyurea(yes) \land is\_polydipsia(yes)

Chestpain-Investigation.Enc:: Is\_chestpain(yes)

As enabling-conditions of both these treatment modules are equal, therefore both in the line of Diabetes-treatment as well as Chestpain-Investigation has been started for the patient simultaneously.

• Choice

On administering a treatment module, the effect due to the module on patients may differ from person to person. And each type of expected-condition a separate treatment may be planned. This situation can be modeled by choice primitive and here we formally specify the same.

Let there be a module TM with disjunctive expected-conditions say \(Exc_1 \lor \)
$Exc_2 \lor \ldots \lor Exc_n$. 

If there exists a set of modules $TM_1, \ldots, TM_n$ with enabling-conditions $Enc_1 \ldots Enc_n$ such that $Exc_1 \equiv Enc_1 \ldots Exc_n \equiv Enc_n$ then we say that the modules $TM_1 \ldots TM_n$ can be composed succeeding to the module $TM$ with choice operator.

Graphically, the lines branching out from $TM$ in $TM_1 \ldots TM_n$ can be labeled with their respective conditions for the sake of clear cut visual understanding.

Formally the choice composition is defined as

$$
\text{choice}(TM, TM_1 \ldots TM_n) :: \\
\{ \exists TM_1 \ldots TM_n \mid ((TM.Exc_1 \Rightarrow TM_1.Enc) \land \\
(TM.Exc_2 \Rightarrow TM_2.Enc) \ldots \land \\
(TM.Exc_n \Rightarrow TM_n.Enc)) \} 
$$

Following the stated example it may be observed that Chestpain-investigation module may provide ECG changes suggestive of CHD, upper gastrointestinal endoscopy for APD. Based on each category respective modules viz. CHD treatment, APD treatment are to be invoked.

• Cooperative/Supportive

On execution of a treatment module $TM$, say two modules viz. $TM_1$ and $TM_2$ are enabled for execution simultaneously in such a way that their executional environments are inter-dependent. That is one module receives a resource from another and similarly the later needs another resource from the former. Such modules are said to be cooperative. In reality such situation arises for example while treating a CHD (Coronary Heart Disease) patient a doctor may choose to run ABC (Airway Breathing Circulation)
procedure and CHD treatment module as they support each other. Hence these two modules are said to be cooperative. A formal definition of two supportive modules is defined as.

**Cooperative**\((TM, (TM_1, TM_2))::\)

\[
\{ \exists \, TM, TM_1, TM_2 \mid TM.\text{Exc} \Rightarrow TM_1.\text{Enc} \land \\
TM.\text{Exc} \Rightarrow TM_2.\text{Enc} \land \\
\text{import}(TM_1, TM_2, v_1) \land \text{export}(TM_2, TM_1, v_1) \land \\
\text{import}(TM_2, TM_1, v_2) \land \text{export}(TM_1, TM_2, v_2) \}
\]

The above defines that the modules \(TM_1\) and \(TM_2\) are in parallel, health status \(v_1\) due to the module \(TM_2\) is used during execution of the module \(TM_1\). Similarly, health status \(v_2\) required by \(TM_2\) is collected during execution of \(TM_1\). This is modeled by two functions \textit{export} and \textit{import} as shown in the definition.

In case a module \(TM_1\) during its execution imports a variable from another module say \(TM_2\) but it does not import any variable from \(TM_1\) then we say \(TM_2\) supports \(TM_1\) which can be formally represented as

**Supportive**\((TM, (TM_2, TM_1))::\)

\[
\{ \exists \, TM, TM_1, TM_2 \mid TM.\text{Exc} \Rightarrow TM_1.\text{Enc} \land \\
TM.\text{Exc} \Rightarrow TM_2.\text{Enc} \land TM_2.\text{Exc} \Rightarrow TM_1.\text{Enc} \land \\
\text{import}(TM_1, TM_2, v_1) \land \neg \text{import}(TM_2, TM_1, v_2) \}
\]

For example during Diabetes-treatment, Diet&exercise treatment needs to be carried out. Hence a module for Diet&exercise treatment supports a Diabetes-treatment module.
3.5.4 Rewriting of a Treatment plan

In previous section we have introduced several operators that can be used to prescribe a treatment plan composed of several treatment modules. Compose operators correspond to their symbols as shown in Table 3.1. This indicates the sequence in which treatment is planned to proceed. But due to some problems like non-availability of resource like endoscope or certain drug, it may not be possible to run the module. In that case one needs to look for an alternative treatment plan. For the purpose, it is required to look for possible alternatives by exploring equivalent expressions to a given expression. A given expressions can be rewritten by using the properties of each composition operator. Some of the composition operators like linear, parallel, cooperative satisfy the associative property, whereas parallel and cooperative satisfy both commutative and transitivity property as shown in Table 3.2. Some of rewriting rules are listed below. With the help of which an equivalent treatment expression can be written.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear</td>
<td>→</td>
</tr>
<tr>
<td>Parallel</td>
<td>∥</td>
</tr>
<tr>
<td>Choice</td>
<td>⎌</td>
</tr>
<tr>
<td>Supportive</td>
<td>←→</td>
</tr>
<tr>
<td>Cooperative</td>
<td>≜</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Op/Prop</th>
<th>Commu</th>
<th>Assoc</th>
<th>Trans</th>
</tr>
</thead>
<tbody>
<tr>
<td>→</td>
<td>X</td>
<td>✓</td>
<td>X</td>
</tr>
<tr>
<td>∥</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>⎌</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>←→</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>≜</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
Rewriting Rules:

- **parallel to linear**
  
  Suppose a treatment expression is represented as:
  
  \[ TM_1 \rightarrow (TM_2 \parallel TM_3) \]
  
  If \( TM_2.\text{Exc} \Rightarrow TM_3.\text{Enc} \) and
  
  **Necessary condition**: \( TM_2.\text{Exc} \supseteq TM_1.\text{Exc} \)
  
  **Necessity**: Necessity of re-writing is that
  
  if \( \text{reqd-resource}(TM_2) \cap \text{reqd-resource}(TM_3) \neq \emptyset \)
  
  this may cause conflict for resource during execution. Hence if possible
  
  this parallel composition \( (TM_1 \rightarrow (TM_2 \parallel TM_3)) \) can be re-written as
  
  \[ TM_1 \rightarrow TM_2 \rightarrow TM_3 \text{ if } TM_2.\text{Exc} \supseteq TM_1.\text{Exc} \]
  
  **Re-write rule**
  
  \( (TM_1 \rightarrow (TM_2 \parallel TM_3)):: = TM_1 \rightarrow TM_2 \rightarrow TM_3 \text{ if } TM_2.\text{Exc} \supseteq TM_1.\text{Exc} \)

- **linear to parallel**
  
  Suppose a treatment expression is represented as
  
  \[ TM_1 \rightarrow TM_2 \rightarrow TM_3 \]
  
  If \( TM_2.\text{Enc} = TM_3.\text{Enc} \) and \( TM_2.\text{Enc} = TM_1.\text{Exc} \) and
  
  **Necessary condition**: \( TM_3.\text{Enc} \subseteq TM_1.\text{Exc} \)
  
  **Necessity**: The purpose of re-writing is that in case of emergency treatment modules like \( TM_2 \) and \( TM_3 \) should execute simultaneously, instead of one after another means \( \text{exec}(TM_2) \wedge \text{exec}(TM_3) \)
  
  Therefore linear composition of treatment modules can be re-written as
  
  \( (TM_1 \rightarrow (TM_2 \parallel TM_3)) \) if \( TM_3.\text{Enc} \subseteq TM_1.\text{Exc} \)
  
  **Re-write rule**
  
  \( TM_1 \rightarrow TM_2 \rightarrow TM_3:: = (TM_1 \rightarrow (TM_2 \parallel TM_3)) \text{ if } TM_3.\text{Enc} \subseteq TM_1.\text{Exc} \)
• support to linear

A treatment expression comprises of treatment modules using support as one of the composition operator represented as follows

\[ TM_1 \rightarrow (TM_2 \leftarrow TM_3) \]

If \( TM_1.\text{Exc} \Rightarrow TM_2.\text{Enc} \) and \( TM_3.\text{Exc} \Rightarrow TM_2.\text{Enc} \)

**Necessary condition:** \( TM_3.\text{Exc} \subseteq TM_2.\text{Enc} \)

**Necessity:** In case of supporting treatment module is short-lived and terminate immediately after execution, then if possible this supportive composition can be re-written as linear composition.

**Re-write rule**

\[ TM_1 \rightarrow (TM_2 \leftarrow TM_3) := TM_1 \rightarrow TM_3 \rightarrow TM_2 \text{ if } TM_3.\text{Exc} \subseteq TM_2.\text{Enc} \]

and \( \text{shprt-lived}(TM_3) \)

• supportive to parallel

\[ TM_1 \leftarrow TM_2 \equiv TM_1 \parallel TM_2 \]

• cooperative to parallel

\[ TM_1 \rightleftharpoons TM_2 \equiv TM_1 \parallel TM_2 \]

• choice to linear

\[ TM_1 \otimes (c_1 TM_2) (c_2 TM_3) \equiv TM_1 \rightarrow TM_2 \lor TM_1 \rightarrow TM_3 \]

Using the properties listed in the table an expression can be rewritten to an equivalent expression. The given expression is

\[(TM_1 \rightarrow TM_2 \rightarrow (TM_3 \parallel (TM_4 \otimes (c_1 TM_5) (c_2 TM_6)) \rightarrow TM_9) \land TM_7 \rightarrow TM_3 \land TM_8 \rightleftharpoons TM_5) \]
The above expression can be rewritten as

\[(T M_1 \rightarrow T M_2 \rightarrow (T M_3 \parallel (T M_4 \rightarrow T M_5)) \rightarrow T M_9) \land T M_7 \parallel T M_3 \land T M_8 \parallel T M_5 \]

\[\lor\]

\[(T M_1 \rightarrow T M_2 \rightarrow (T M_3 \parallel (T M_4 \rightarrow T M_6)) \rightarrow T M_9) \land T M_7 \parallel T M_3 \land T M_8 \parallel T M_5\]

This shows that from the given case study a treatment plan stated in an expression can be rewritten to an equivalent expression. On making use of module specification, in the following section we investigate on change and anomaly managements in healthcare domain.

### 3.6 Exception Management in case of Treatment

Healthcare treatment often requires human participations of doctors, patients and collaborating staff ranging from paramedical to administrative personnel. Though each of them play well defined roles still human actions are liable to lapses. Patient treatment involves life risks; hence healthcare workflow management should have means to deal with these lapses. This feature of a system is widely known as *exception management*. Exceptions are to be differentiated from failures. Failure means that a system does not work say due to program error, device failure, communication failure etc. Exceptions are sometimes called as semantic failures which may arise when activities cannot be executed as planned or do not meet the desired result. Exceptions are categorized as *known* or *unknown* interactions. An exception that is identified during system analysis is termed as known exception or otherwise unknown. A system is capable of processing known exceptions during
system analysis. The processing of these exceptions are explored and the system is designed accordingly. This kind of exception processing is widely found in workflow management systems in engineering domains or the domains where the roles and exceptions of the domain entities are well defined.

The management of exceptions in healthcare domain is strikingly different primarily due to variable behavior of patients even being affected by the same disease. This may result to discovery of new exceptions that are hither to not known. Hence a healthcare workflow management system must be in a position to deal with such undefined exceptions and to learn to handle with the same exceptions in future. In the following sub-section we have reviewed work related to exception management in healthcare domain. We focus on exception management in treatment process and present a comprehensive framework to deal with it. The proposed exception management process includes exception analysis, exception specification and processing of exceptions. These issues are elaborated in subsequent sub-sections.

Workflow describes the “normal behavior of a process where as an exception in case of workflow indicates “occasional behavior” [33] [60]. [28] proposed a methodology for modeling exception using activity graphs i.e WAMO (Workflow Activity Model) which enables the workflow designer in modeling not only current business process but also exception which may arise during execution. Thus a business process can be viewed as a composition of the core process and a collection of exception process. Guidelines are provided for specifying exceptional behavior in the three phases of exception handling that is detection, diagnosis and resolution. Behavior of an exception that represents deviations from a normal process, can be anticipated and handled accordingly [88]. Exceptions are specified using Chimera-Exc language which are specifically designed for expressing exception in WFMS [29][34]. In this formal properties of workflow application
with exceptions are discussed to indicate criteria for sound use of exceptions in workflow management.

There has not been enough work on exception management for healthcare workflow system. A work reported in [34] describes the usability of existing exception handling concepts in healthcare workflow systems. However, it suggests introduction of new techniques for exception handling. The suggested techniques include knowledge-based exception handling and dynamic exception handling. In general, it advocates that the existing exception handling techniques for workflow systems are mostly enough to manage exceptions in healthcare domain. As healthcare domain needs special attention to handle exceptions as life of patients are in stake, therefore the source of exceptions must include not only administrative entries but also patients, paramedical staff, doctors and resources. In nutshell, all entities that have a role in executing a treatment activity must be treated as a potential source of exceptions.

Though exception propagation in traditional workflow system is considered as hierarchical, we advocate a process that is not vulnerable to miss and delay. Again process based exception processing is traditionally considered. But, we are in opinion that this technique is computationally expensive with high latency time. Unknown exceptions are more expected in healthcare domain. And treatment to such an exception is exploratory. Hence, healthcare workflow management system should have capability to manage both unknown as well as exploratory treatment. In this section we make an attempt to evolve a holistic approach for handling exceptions in healthcare domain.

### 3.6.1 Exception in Treatflow

A Treatflow defines not only a sequence of treatment activities but also each treatment. For each treatment context i.e. pre-conditions and goal i.e. post-
conditions are defined. The scope of conditions for a treatment includes patient, staff as well as resources. Unsatisfiability of a condition raises an exception. In order to drive the point straight below we present an example. For example

A patient is admitted to the hospital with acute pain in abdomen. The patient is 40 yrs male having diabetes. He is having moderate fever and vomiting. The pain in abdomen is localized to right lower quadrant of abdomen. On admission necessary registration and investigation are being done. Then the patient receives injection for relief of pain in abdomen, vomiting and fever. His complete blood picture, urine examination and random blood sugar is sent to lab for analysis. He is kept N.B.M (Nil By Mouth). IV (Intra Venous) fluids are started to hydrate the patient. IV antibiotics are started to control the infection (if any). Simultaneously the patient is examined by duty doctor to prevent any complications. As soon as the blood reports arrive the doctor is intimated. The doctor after going through the reports tentatively diagnoses the case to be acute appendicitis. The case is referred to general surgeon. After examining the patient he instructs the duty doctor to control blood sugar and advises for ultrasound scanning (U/S) of whole abdomen, X-Ray chest, ECG, diabetologist opinion and anesthetists opinion for the appendectomy operation. After the control of blood sugar and anesthetist PAC (Pre Anesthetic Checkup) the O.T.(Operation Theater) time slot is fixed. The nurse is advised to carry out the formalities such as information to patient and relatives, operation risk is to be taken in writing, preparation of the parts etc. The patient undergoes operation and post operative recovery is uneventful. The patient is observed in POW(Post Operative Ward) for 24 hours. Then the patient is shifted to the MSW(Male Surgical Ward). On seventh day the sutures (Stitches) are removed and the patient is discharged.

For analysis, we will make use of the above example. And a systematic analysis follows in the following sections.
Figure 3.18: Exceptions in case of Treatflow
3.6.2 Exception Analysis

Usually there is a definite process defined to handle exceptions as seen in manufacturing domain and business domain. In case of healthcare, other than a generic way there could be specific method tailor made for a particular patient. Considering patient and process point of view analysis of exceptions has been done. In this section we analyze exception handling in healthcare domain with an aim to develop a process applicable in the domain.

A Treatflow means, for a particular ailment, same treatment can be applied to different patients suffering from the same ailment. Still, a treatment for a patient may proceed smoothly but for another patient with the same ailment may not go well through for patient specific reasons. Patient related exceptions can be either known or unknown type.

- **Known**

Known exceptions are predictable deviation from the normal treatment such as patient suffers from allergy or side-effect while undergoing a treatment. For example while administering drugs, doctor knows that patient may suffer from drug allergy which can be taken care of by mentioning these type of exception as known exception as shown in Figure. 3.18. For a system S, let there be a given treatment activity ta in a Treatflow with a set of treatment activities TA. There are specified exceptions that arise due to non-confirmation of any number of post-conditions or safety-conditions or confirmations of side-effect during the execution of the treatment, while system is aware of an exception say aware-of (excep), if and only if that is present in the list of exception exception-list.

This is formally represented as
aware-of(S,ta,excep) \equiv \{ ta \in TA \land \text{throws}(ta,excep) \\
| ((\text{not-true}(ta,\text{post-condition}) \lor \text{not-true}(ta,\text{safety-condition}) \\
\lor \text{has-sideeffect}(ta,\text{sideeffect})) \land \text{is-in}(excep,\text{exception-list}) \}

where \textit{aware-of} is a predicate with arguments system S, treatment activity \textit{ta} and exception \textit{excep} and \textit{throws} a function that system S uses to signal occurrence of an exception and predicates viz not-true, has-sideeffect and is-in have their usual meanings.

- **Unknown**

Unknown exception could be due to inconsistencies between a treatment process and biological process of a patient that is undergoing the treatment. A particular patient may get sideeffect while undergoing treatment regarding which doctor is not aware of from the beginning. For example while executing \textit{Prescribe-drug} as a treatment activity, due to some patient specific reason he/she may get sideeffect which is not-known to doctor. Therefore it is considered as unknown exception and needs to be handled by contacting doctors in case of emergency as shown in Fig.3.18. During execution of a treatment activity \textit{ta} within a Treatflow may throw an exception \textit{excep} which is \textit{not-aware-of} by the system S. This type of exception arises may be due to patient complaints or unsatisfied post-condition and which is not present in the list of exception \textit{exception-list}.

Which can be formally represented as :

not-aware-of(S,ta,excep) \equiv \{ ta \in TA \land \text{throws}(ta,excep) \\
| ((\text{not-true}(ta,\text{post-condition}) \\
\lor \text{has-patient-complaint}(ta,\text{patient-complaint})) \\
\land \text{is-not-in}(excep,\text{exception-list}))\}$
where \textit{not-aware-of} is a predicate with arguments system S, treatment activity ta and exception exce. \textit{throws} a function that system S uses to signal occurrence of an exception and predicates viz not-true, has-patient-complaint and is-not-in have their usual meanings.

A non-progressive state or a stalemate state arising during execution of a treatment triggers an exception. This situation is viewed as a failure of treatment process and such a failure of course, is to be attended. The action to deal with such situations is modeled as exceptions in treatment process. Considering the genesis of exceptions we categorize them into

- Resource
- Context & Goal
- Time
- Safety

perspectives. Exceptions of different perspectives are dealt in the following sub-sections.

\textbf{Resource Perspective}

Usually a treatment activity may require resources like a diagnostic report, doctor, nurse, operation theater etc. Non-availability of resources, on which treatment activities depend, may throw exceptions. For example the following treatment activity \textit{U/S-scanning} is not able to execute and throws an exception due to the unavailability of scanning machine as shown in Figure. 3.18 In a system S, among the set of treatment activities TA, a treatment activity ta needs a resource r from a list of resources say \textit{resource-list} for execution. Due to unavailability of resource
r, the treatment activity $ta$ may throw an exception $excep$ which can be formally represented as

$$\exists ta \in TA \mid \text{to-execute}(ta) \land \text{not-available}(\text{resource-required}(ta)) \Rightarrow \text{throws}(ta, excep)$$

where $\text{resource-required}$ a predicate with arguments treatment activity $ta$ and attribute $\text{resource-required}$ $\text{not-available}$ and not-true are the predicates having usual meaning.

In order to minimize nonavailability of resources and thus generation of exceptions, a healthcare workflow management system needs to have provision for resource management. In many areas of computer science like operating systems, distributed computing, the issue of resource management has been studied extensively. In this study we have not taken up this issue as our focus is on specification of treatment activities.

**Context and Goal Perspective**

Genesis of exceptions can be found on analysis of an activity in Treatflow. A treatment activity is taken up on a particular context with qualifying condition. For example, for blood sugar test a patient should not have breakfast on the test day. The qualifying condition $\text{no-breakfast-on-day}()$, in the context of blood sugar test can be thought of as a pre-condition. Obviously, a treatment activity is performed for a purpose and this is termed as the goal of the activity. On execution of the activity, it reached its goal state. The goal state of the activity is qualified with state conditions that are desired to be true at the goal state. These conditions are termed as post-conditions. Unlike pre-conditions that initiate an activity execution, there are cases where a set of conditions are required to be true during
execution of a treatment. For example, during an operation the pulse rate should be always within range 60-90 per minute. Any violation of pulse rate constraint in the context of operations generates exception. Here the constraint in the context of an operation can be thought as an invariant and any violation to an invariant causes exceptions. These cases are illustrated below by quoting activities shown Figure. 3.18. For example for \textit{operation} as a treatment activity blood sugar level must be < 140 mg % which may not be true for all patients. In those cases this type of treatment activity may throw an exception. Generally a treatment activity \( ta \) in a Treatflow checks for the pre-condition before execution. An exception \( \text{excep} \) arises by the treatment activity \( ta \) in case context is not satisfied for that treatment activity. It can be formally represented as

\[
\exists \ ta \in \ TA \ | \ \text{not-true}(ta,\text{pre-condition}) \\
\implies \ \text{throws}(ta,\text{excep})
\]

Similarly for example relief-of-pain may not be true after \textit{Administer-drug} as a treatment activity as in Figure. 3.18. Usually goal of a treatment activity must be true after execution. In some cases exceptions may arise if the goal of the treatment activity does not satisfy which can be formally represented as

\[
\exists \ ta \in \ TA \ | \ \text{execute}(ta) \land \text{not-true}(ta,\text{post-condition}) \\
\implies \ \text{throws}(ta,\text{excep})
\]

where \textit{execute} is a function where a system S uses to execute a treatment activity \( ta \).
Temporal Perspectives

Some treatment in Treatflow may have temporal characteristics e.g physiotherapy is to be done 2 times in a week, a drug for blood pressure control is to be taken at a specified time or operation theater is to be disinfected in periodic interval. Thus, for a treatment temporal constraints can be associated and a violation of such a constraint usually generates an exception. These cases are illustrated below by quoting activities shown Figure. 3.18 According to the figure all reports should have arrived in time as a result of post-condition of treatment activity Investigation. This treatment activity generates an exception which violates time constraint due to the reports not received on time. In a system S, operation-time is given to each treatment activity ta for its execution. An exception can be thrown by the treatment activity ta if it is not able to complete its execution within the specified period of time. It can be formally represented as

$$\exists ta \in TA \mid \text{under-exec}(ta) \land \text{not-intime}(ta, \text{operation-time}) \implies \text{throws}(ta, \text{excep})$$

where operation-time is the time for execution for treatment activity ta and not-intime is a predicate with the arguments treatment activity ta and operation-time

Safety Perspectives:

Some treatment in healthcare may have safety-condition characteristics e.g while Pencillin is administered during Administer-drug activity, a test dose intradermally should be done to know sensitibity lest severe anaphylactic reaction may
arise causing death in some of the cases. Thus, for a treatment safety condition can be associated and a violation of such a condition usually generates an exception. In a system \( S \), safety-condition is given to each treatment activity \( ta \) for its execution. An exception can be thrown by the treatment activity \( ta \) if it is not able to satisfy the safety-condition. It can be formally represented as

\[
\exists \, ta \in TA \mid \text{under-exec}(ta) \land \\
\text{violate}(ta, \text{safety-condition}) \lor \\
\text{reported}(ta, \text{side-effect}) \\
\implies \text{throws}(ta, \text{excep})
\]

where \( \text{under-exec} \) is for treatment activity \( ta \) under execution and \( \text{violate} \) and \( \text{reported} \) are predicates with the arguments among treatment activity \( ta \) with safety-condition and side-effect respectively.

Further, an omission of execution of a treatment in a Treatflow disturbs temporal sequence of treatment primitives. Such an omission is possible because a treatment primitive can be manually executed e.g. a patient skips a physiotherapy session. Again, failure of handling an exception in a defined time period should also throw an exception. Though this results to nesting of exceptions, but to manage complexity in exception processing we resort to procedural execution of exceptions. This aspect we deal in detail late during our description on exception processing. A treatment may generate several exceptions. Based on sources of exceptions we categorize them into different specializations. A given exception is could be one of the categories viz. Resource, Context & Goal, Safety and Temporal. In order to understand the types of exceptions in perspective of their resources, we have undertaken further analysis. For each perspective, we have defined several states and for each state we have specified a predicate that
Figure 3.19: Exception Categorization

is instrumental in throwing exceptions as shown in Table 3.3. For example for resource perspective, the states defined are *rna*, *rnsrbl* and *srnf* and these state acronyms stand for 'resource not available', 'resource not shareable' and 'shared resource not freed' respectively. For the *rna*, the predicate is Notavail(r) where \( r \in R \) means resource \( r \) not available.

Similarly for *rnsrbl* the predicate is Notsrbl(r) where \( r \in R \) means resource \( r \) not shareable and for *srnf* the predicate is Srbl(r) \( \land \neg \) free(r) where \( r \in R \) means shared resource not free.

For context and goal perspective, the states are *cns*, *agns*, *uwntg*, *udefg* and acronyms for these states are 'context not satisfied', 'anticipated goal not satisfied', 'unwanted goal' and 'undefined goal' respectively.

Predicate for *cns* is \( \neg (C) \mid C \in \text{context} \) means context \( C \) not satisfies.

Similarly for *agns* \( \neg (G) \mid G \in \text{goal} \) means not according to the anticipated goal.

For *uwntg* \( G \in \text{UG} \) where it is of type known exception and which is patient specific for example sideeffects and allergy etc.

But *udefg* is of type unknown exception which differs from patient to patient and is not pre-defined means not available. On presenting a comprehensive analysis on exception and their origins, in the next we would like concertize exception description with specification. And also discuss on exception handling issues. A treatment may generate several exception in context with the discussed perspectives. Treatment can throw exception of different types or may generate events as shown in Table 3.3 Apart from the origin of exceptional event, exceptional events
Table 3.3: Exception Analysis

<table>
<thead>
<tr>
<th>Exception Type</th>
<th>Type Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rna (Resource not available)</td>
<td>Notavail (r)</td>
</tr>
<tr>
<td>rnsrbl (Resource not shareable)</td>
<td>Notsrbl(r)</td>
</tr>
<tr>
<td>srnf (Shared resource not free)</td>
<td>Srlb(r) ∧ ¬free(r)</td>
</tr>
<tr>
<td>cns (context not satisfied)</td>
<td>¬(C)</td>
</tr>
<tr>
<td>agns (Anticipated goal not satisfied)</td>
<td>¬ G</td>
</tr>
<tr>
<td>uwntg (Unwanted goal)</td>
<td>G ∈ UG</td>
</tr>
<tr>
<td>udefg Undefined goal</td>
<td>G ∉ {G}</td>
</tr>
<tr>
<td>nct (Not completed within time)</td>
<td>¬ complete(TA,t)</td>
</tr>
</tbody>
</table>

and their corresponding actions is discussed in table 3.4. Actions for each exceptional event depend on the type of origin either known or unknown. Actions of each exceptional event specifically depends on event location i.e treatment activity(TA) level, multiple treatment activity in Treatflow(TM) or treatment module (TM) level. Again for each exception, an action is specified and each action may deal on a treatment or add few more treatments to the treatment plan.

3.6.3 Specification of Exception

Treatflow is a kind of workflow consisting of a sequence of Treatment activities where each activity is defined in 3.3 with pre-conditions and post-conditions. In the last section we have discussed on possible sources of exceptions. A Treatment activity may generate several exceptions and an exception could be of one of the types viz. rna, rnsrbl, srnf, cns, agns, uwntg, udefg and nct. We have discussed on each type in the last section. All these types of exceptions are of known types;
Table 3.4: Exceptional Events with Actions

<table>
<thead>
<tr>
<th>Exception Events</th>
<th>Exception Type</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>rna</td>
<td>Known</td>
<td>Allocate</td>
</tr>
<tr>
<td>rnsrbl</td>
<td>Known</td>
<td>Unlock/Getnew/Default</td>
</tr>
<tr>
<td>srnf</td>
<td>Known</td>
<td>Release/Pause</td>
</tr>
<tr>
<td>cns</td>
<td>Unknown</td>
<td>Pause/Default</td>
</tr>
<tr>
<td>cns</td>
<td>Unknown</td>
<td>Inform</td>
</tr>
<tr>
<td>agns</td>
<td>Unknown</td>
<td>Pause/Default</td>
</tr>
<tr>
<td>uwnfg</td>
<td>Unknown</td>
<td>Default</td>
</tr>
<tr>
<td>uwnfg</td>
<td>Unknown</td>
<td>Default</td>
</tr>
<tr>
<td>nct</td>
<td>Known</td>
<td>Pause</td>
</tr>
<tr>
<td>nct</td>
<td>Known</td>
<td>Inform</td>
</tr>
</tbody>
</table>

Table 3.5: Types of Actions

<table>
<thead>
<tr>
<th>Action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allocate</td>
<td>Assign resource</td>
</tr>
<tr>
<td>Pause</td>
<td>wait in the current treatment activity</td>
</tr>
<tr>
<td>Getnew</td>
<td>Get new resource</td>
</tr>
<tr>
<td>Unlock</td>
<td>Unlock resource</td>
</tr>
<tr>
<td>Default</td>
<td>Call a doctor</td>
</tr>
<tr>
<td>Release</td>
<td>Release resource</td>
</tr>
<tr>
<td>Inform</td>
<td>Inform patient</td>
</tr>
</tbody>
</table>

hence can be specified even before the system development stage. In addition to those we will add two more types called DefaultExcp and PatientExcp as shown in Figure 3.19. These two types of exception events are designed to deal with unknown situations. As we have told earlier, in case of healthcare workflow handling of unknown exceptions is crucial as one of such exception could be life threatening. We have dealt with unknown world using default reasoning. For each patient, a doctor can define a closed world specifying permissible values to all concerned life protecting parameters. As and when a defined health parameter assumes a value that is out of its defined range, Treatflow management system throws an exception of type DefaultExcp. This provision really does not take care of all possible unknown types of exceptions. This is simply because, even sometimes
doctors may fail to define a complete closed world that is safe for a patient. Or there could be a miss for human error. In order to take care of such situations we propose to have another type of exception called PatientExcp to allow a patient to create an exception of the type. As and when a patient feels uneasy can express health conditions to systems for help and to intimate the concerned doctor for help. Typically, an exception is identified by its unique identification number \( id \) and further described by its type, generator description, priority, condition and action. A formal description of exception is given as:

\[
\text{Exception} :: \{<\text{ExcepId}>\} \\
\text{Egenerator} :: \{<\text{TId}> <\text{TName}>\} \\
\text{Priority} :: \{<\text{int}>\} \\
\text{Condition} :: \{<\text{predicate}>\} \\
\text{Action} :: \{<\text{action-name}> <\text{action-parameter}> <\text{executor}> <\text{time}>\} \\
\text{End}
\]

Unique identification of an exception is considered for distinct accessibility as well as for clarity on dealing with exceptions. An exception is generated by an instance of a Treatflow being managed by Treatflow management system. At a given time, there could be several instances of Treatflows being operational. An instance while executing a treatment activity can throw an exception. In order to trace an exception we need to refer to the instance by \( TId \) and corresponding activity \( TName \) that has raised the exception. So, the field \( \text{EGenerator}::\{\text{TId},\text{TName}\} \). A type definition associated to an exception gives an idea on why the exception is generated. This explains who and why a given exception is generated. Before proceeding to take action for a given exception we have made a provision to doubly check on justification of action. And this has been formalized in field \( \text{Condition} \).
The field can have predicates that are to be satisfied for taking an action with respect to the exception. Similarly, the field action-name describes what action is to be taken in connection with an exception and who is the concerned entity/entities for doing that. It is formalized in subfield executor of Action. The subfield action-parameter further describes the resource an action may need. Latency for taking action is specified as a subfield to Action field. In nutshell, we follow the well known event-condition-action paradigm according to the exception specification discussed above. Here is an example of an exception stated as per the specification.

\textbf{Exception} :: \{<ER1>\}
\textbf{Egenerator} :: \{<TF1> <Do-U/S-scanning>\}
\textbf{Priority} :: \{<1>\}
\textbf{Condition}:: \{< to-exec(ta) \land not-available(resource-required(ta)) >\}
\textbf{Action}:: \{<Allocate-resource(ta,r)>\}
\textbf{Priority} :: \{<1>\}
\textbf{End}

On specifying exceptions, now we deal with the mechanism that is followed to catch exceptions and process them. We make use of throw-catch mechanism available with Java programming language.

\section*{3.6.4 Exception Processing}

Processing a treatment activity includes testing the satisfiability of pre-conditions, executing actions pertaining to the activity and the satisfiability of post-conditions. A class diagram in Figure 3.20 depicts the processing of exceptions of different types. These satisfiability tests throw exceptions. A class called SatisfyTreat-Condition with methods \textit{TestPreCond} and \textit{TestPostCond} respectively test pre-
Figure 3.20: Processing of Exceptions
conditions and post-conditions associated to an activity. This class instance throws exception events when the predicates in pre-/post-conditions are not satisfied. An exception event could be an instance of types of exceptions listed in Table 3.3. As we understand generation of exception is always asynchronous activity. The exception handler of the class instance SatisfyTreatCondition catches an exception and instantiates it (by PutExceptionLog class calling method StoreExceptionLog as shown in Figure: 3.20. The instance of an exception event is stored in a buffer ExceptionLog. This buffer is organized as a queue data structure. At one end the generated exception events are stored and at the other end of the queue an exception event is deleted for processing. So, at any given time ExceptionLog will have exception events that are to be processed. The logging of exceptions by Treatflow management system is shown in 3.21 A monitor class ScheduleException comes up in regular interval of time and generates a schedule of exception events. This
arranges exception events in descending order of priority. The time interval for activation of the monitor can be tuned so that the scheduling time and exception event processing together should be much less than the minimum of the time specified for execution of exception events. This is an ideal condition and sometimes hard to achieve for a system that mostly deals with very time sensitive treatments like open heart operations etc. The practical approach is to set the time less than the average of the time specified for all types of exceptions. The purposes of the scheduling is to provide maximum attention for high priority exception events. Of course, this will lead to stagnation of low priority exception events. This can be interpreted as a negligence of care to a patient which may further cause harm to the patient. In order to avoid such situations we propose to dynamically calculate priority that is proportional to the quantum of the time an exception is waiting on buffer for processing. A Java thread called ProcessException always haunts for exception events pending in queue ExceptionLog. It deletes an event from the queue and process it to take necessary action. As a nutshell processing of an exception includes evaluation of conditions and on succeeding to it, specified actions are to be taken. The thread is mainly composed of two methods called ProcessExceptCond and ProcessExceptAct to evaluate conditions and to take actions respectively. The predicates are to be seen in different prospectives viz. resource, context & goal, temporal and safety. For an example a predicate could be on patient health status as temperature \( \hat{=} 102c \). For evaluating this predicate, the method gets value for temperature of the patient (related to the instance of Treatflow) and compares with 102c. For this, obviously it’s required that healthcare management system is maintaining repository of all necessary information pertaining to each Treatflow instances under execution. So, the method has an interface to such a database for retrieving information required for the evaluation of predicates. On successful evaluation of all the relevant predicates associated to an exception; the Process-
\textit{sExceptCond} invokes the method \textit{ProcessExceptAct}. The invoked method based on types of exceptions may take actions as listed in Table 3.5. The generic actions are \textit{Allocate, Pause, Getnew, Unlock, Default, Release, Inform} etc. At a given time there could be several instances of Treatflow under execution. And any of them can throw an exception. All the exceptions generated by different instances of Treatflow are stored at ExceptionLog. So, the proposed exception handler of Treatflow management system treats all the exceptions globally instead of individually. The rationality of planning this way to provide equal opportunity for all the patients and to provide attentions to more needing patients.

### 3.7 Summary

Traditionally structured method is being used for system analysis and design. The reason for its popularity is due to the pictorial presentation of a system. However, the method lacks mathematical preciseness that can be used to prove the system mathematically. Formal method in software development helps in proving the system property mathematically.

The proposed work used some primitives viz, linear, split and merge, parallel, supportive, cooperative and choice that are basic to represent workflow. The uses of such primitives are shown in specifying workflow of a corporate hospital for treating patients. For each type of primitive other than pictorial representation, system behavior is also specified mathematically. Execution of each primitive is analyzed and shown that it is possible to make unambiguous inferences on execution of each workflow primitive. The analysis is implementable as it uses assertions as pre-condition and post-condition. A rule based system with assertions as rule can be used for verification and validation on workflow primitives pertaining to application domain. A business process can be synthesized as
a composition of workflow primitives. On reviewing works on modularization of workflows we have method to specify a treatment module and operators to compose a treatment. It is shown that a treatment prescription can be written as an expression of modules. As sometimes it may be necessary to explore all the possibilities in executing a treatment plan, we propose a method for rewriting a given expression by applying rules to rewrite.

Currently we are studying on correctness of a treatment expression written with modules. As modules at coarse level represent activities, we are interested in visualizing all possible control flows among defined activities. For the purpose, a modular expression is represented in a graph form and all possible paths are instantiated to study the Illegality especially incompleteness, deadlock and lack of synchronization. Here we focus on exceptional behavior of healthcare workflows and provide a comprehensive analysis on generation of such exceptions at the deviating behavior of a specified workflow. We also define actions that can be performed with respect to exceptions. The feasibility of this concept is known in a case study taken from healthcare domain.