Temporal Logic based constraint verifier for Business Process Models

Chethana Rudradevaru

Master of Science
Computer Science
School of Informatics
University of Edinburgh
2004

(Graduation date: Jan 2005)

Abstract

The huge cost involved in the deployment of a business process model often puts the emphasis on simulations to detect the inconsistencies and flaws in the designed business process models. The central components of business process models are the interaction model and intraprocess model. The interaction model concentrates on the communication aspects or the information exchange between the different processes within a business process while the intraprocess model focuses on the information flow within a business process. In practical design, the interaction model is more error prone as it concerns several interacting processes communicating with each other and be it automated or manual entity deployed for each of the interacting processes, the chances of errors due to misunderstanding of the responsibility boundaries often creep in. It is important to avoid conditions like live-lock and deadlock, so that processes terminate in a timely fashion. The hurdles that are imposed on the progress of the processes are the constraints which ought to be satisfied so that subsequent information exchanges can take place for the process to eventually terminate. Faulty deployments can also cause constraints to be bypassed which could cause considerable loss in terms of business value. Hence it is important to verify the designed model with respect to the satisfaction of the constraints. The aim of this project is to implement such a verification system which allows the determination of constraint satisfaction by simulating the designed business process interaction model as agent interactions. End-to-end automation of the business process modelling, simulation and verification activities has not been attempted in this project. Manual intervention has been used in various important phases and deficiencies in each of the activities which could pose a hindrance to automation have been identified.

Acknowledgements

I would like to thank my supervisor Dr.Dave Robertson, first of all for accepting to supervise me in an unknown venture, being the source of light in what seemed to me a dark path, patiently answering all my queries and providing an all-round support during the entire course of this project.

I also thank Dr.Jessica Chen-Burger for her time in explaining the process models.

I would like to extend my heartfelt gratitude to my beloved parents Smt.Jaya and Sri.Rudradevaru and, my sister Dr.Shilpa Rudradevaru for their unconditional love and support in all walks of my life. Without their encouragement this course would have remained a dream forever.

I wish to thank all my well-wishers who trusted and assisted me in this endeavour.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Chethana Rudradevaru*)

Table of Contents

1 Introduction **2**

1.1 Business Process Modelling . . . 3

1.1.1 Business Process Models (BPM) . . . 4

1.2 Overview and chapter organisation . . . 5

2 Identification of constraints **6**

2.1 Introduction . . . 6

2.1.1 Temporal Logic and models of time . . . 6

2.1.2 Constraints . . . 8

2.2 Sufficiency of temporal constraints . . . 10

2.2.1 Well-formed-formula . . . 10

2.2.2 Test for sufficiency . . . 11

2.3 Constraint List . . . 15

2.3.1 Translation scheme . . . 15

2.3.2 Constraint definitions . . . 16

3 Process Communication Model Description Language - PCMDL **17**

3.1 Introduction . . . 17

3.2 Fundamental Business Process Modeling Language . . . 18

3.3 Unified Modeling Language . . . 18

3.4 PCMDL . . . 19

3.4.1 Notations and semantics for basic PCMDL elements . . . 19

3.4.2 Notations and semantics for PCMDL connector elements . . . 21

3.4.3 Formal representation of PCMDL . . . 23

3.5 Formal definition of properties 26
3.6 PCMDL Model Execution 29
3.7 Illustration 29

4 Agent Simulation 35
4.1 Introduction 35
4.2 LCC protocol 35
4.2.1 LCC Syntax 37
4.2.2 Clause Expansion 38
4.3 Simulator output 39
4.4 Illustration 41
4.4.1 Protocol Specification 41
4.4.2 Simulation Output 43
4.5 Mapping PCMDL to protocol specification for the simulator 45
4.5.1 Problems in general automated translation 50

5 Implementation for Constraint Verifier 51
5.1 Introduction 51
5.2 Concepts of time model 51
5.3 Constraint language and Annotations 52
5.4 Preprocessor 54
5.5 Constraint Specification 54
5.6 The Interpreter 55
5.7 Verifier output 62
5.8 Illustration of constraint specification and verification 62

6 Evaluation and Discussion 64
6.1 Introduction 64
6.2 Supply chain management (SCM) - Order call off 64
6.2.1 Process Description 64
6.2.2 PCMDL 65
6.2.3 Protocol Specification 65
6.2.4 Properties for verification 66

6.2.5 Constraint Specification 68
6.2.6 Evaluation and Discussion 70
6.2.7 Conclusion 74
6.3 Software Quality Management - Change management process 75
6.3.1 Process Description 75
6.3.2 PCMDL 75
6.3.3 Protocol Specification 76
6.3.4 Assumptions 77
6.3.5 Properties for verification 78
6.3.6 Constraint Specification 80
6.3.7 Evaluation and Discussion 83
6.3.8 Conclusion 88
6.4 Tender invitation and award 89
6.4.1 Process Description 89
6.4.2 PCMDL 89
6.4.3 Protocol Specification 90
6.4.4 Assumptions 92
6.4.5 Properties for verification 93
6.4.6 Constraint Specification 94
6.4.7 Evaluation and Discussion 96
6.4.8 Conclusion 100

7 Conclusion and future directions **102**
7.1 Future Directions 103

A Appendix A - Business Term Glossary **105**

Bibliography **107**

Chapter 1

Introduction

The expansion and liberalisation of the international markets has resulted in the delayering and downsizing of organisations, with decentralised semi-autonomous business units [22]. This has led to businesses being geographically and internationally distributed. Added to this structural change of the organisations, the adoption of ideologies like the Total Quality Management has shifted the traditional product driven focus to customer driven focus with increased importance on the quality of service delivered to the customer. This in turn has resulted in organisations focusing more on the core business processes which cross departmental boundaries. A *Business Process* is a collection of activities designed to produce a specific output for a particular customer or market [29][1]. Organisations are structured with respect to an overall business objective creating *Horizontal organisations* and requiring management across organisations [22]. Hence the design or redesign of any business process should address the distributed nature of organisations which adds complexity to the design in terms of the interactions and coordination involved among the different units to achieve the common goal of the designed business process. Usually the cost factor involved with the deployment of the designed business process is very high. Hence it is important to ensure that the design is correct before deployment. One solution to this would be to build conceptual models of the business process and use them for process diagnosis.

A business process evolves constantly as it is subject to changes. There are several fac-

[1] A few other jargons in the business realm - Business Process management and Workflow management are defined in the appendix A with further elaboration of Business Process definition

tors contributing to the initiation of a business process change. These include changes in the regulation and policies, changes in business priorities, changes required because of Business Process Re-engineering (BPR) and many more. The time to effect the change should be as minimal as possible to retain the competitive edge which is a critical factor in achieving business success. A model driven approach to create, deploy and manage business processes is one of the solutions to address the rapid process change requirements. Business Models also aid analysis of the business processes to identify areas of improvement.

1.1 Business Process Modelling

Process modelling which was initially used in the manufacturing sector of the industry has gained patronage in the service sector also because of the similarity and repetitiveness in the service tasks performed[6]. Business Modelling based on formal approaches provide the precision in the translation of design to implementation without leaving any scope for the design to be interpreted differently if implemented by different implementors. Also formal representations allow for better analysis of the designs to identify the process improvements that can lead to increased efficiency, profitability and effectiveness. Formal representation of the processes also forms the basis for future automation of tasks which make up a business process.

Representative business process modelling methods are described in the *Handbook of Organisational Processes* [31], Workflow Reference Model [14], Process Interchange Format (PIF) [18], Process Specification Language [28], Integration DEFinition Language (IDEF3) [16], Integration DEFinition Language (IDEF0) [1], Unified Modeling Language (UML) Activity diagram (extension) [27] Event Driven Process Chains (EPC) [23] and Petri-Nets [32].[6]

The focus of many traditional workflow management systems are limited to process design and enactment with little emphasis on diagnosis [3]. Bolstering the modelling technique with the power of simulation can benefit the design activity to a great extent. Simulation provides valuable insight into the strengths and weaknesses of design. It

also assists in evaluating a process for predicted changes without actually changing the business process in operation.

1.1.1 Business Process Models (BPM)

A model supporting business processes using methods, techniques and software to design, control, enact and analyse operational processes involving humans, organisations, applications, documents and other sources of information is a *Business Process Model*.[3]. Though the word *model* is used commonly to mean a description of something, an elevated meaning of this is given in [15]. As per this we say A is a model of B, if A and B share some important properties that we are interested in. The business process model should thus reflect accurately the properties of the business process that we attempt to model. For reasons mentioned earlier, BPMs should have formal foundation. The BPM should also appeal to the business process designers which in most cases are the managers, who are not concerned about the technicalities of formal representation. This could be achieved by using a visual modelling technique underpinned by a formal representation.

In one view, BPM can be categorised as the *Business Models* and the *Process Models* [5]. A business model is more abstract than the process model with prime focus on *what* the system is to achieve while the process model is more realistic in describing *how*. A business model is concerned with the value exchanges among the business partners while the process model focuses on the operational and procedural aspects of business communication. Thus the process model is a *dynamic model*. In our work we focus on the process models.

A business process typically comprises of sub processes and interact with other business processes. Interaction between different processes forms a vital part of any business. Business interactions can be either a one-to-one interaction or a multi-party interaction. Business interaction represents the external manifestation of internal business processes [33]. Dependencies exists between the interactions and have to be executed in a constrained manner. A process model allows us to model the permissible se-

quences of business interactions and the constraints on the tasks that can be performed as part of the interactions. Interaction logic is regulated by specific business rules. The body of rules governing the information exchange shapes the interaction in accordance with specific patterns. A business rule may specify a *check* on the data relationship or the *value set* for a business entity or an *action* that must be performed [15]. All of these act as constraints on the interaction. A designed process model should therefore be verified for the satisfaction of such constraints before live deployment.

1.2 Overview and chapter organisation

In this project we build a constraint verifier based on temporal logic that detects the satisfaction of properties of designed process models and uses agent simulation. We scope the project to address linear process interactions and temporal constraints. By linear process interactions we mean that at any point of time only one of the interacting process is active. As we focus on temporal constraints, we develop and use models to represent the temporal relations used in business process definitions. To be able to categorise constraints in diverse modelling domains we use *safety, liveness, correctness, deadlock* and *termination* properties applicable to other standard models

The modelling of temporal dependencies is dependent on the time model. We identify a time model and the types of constraints that is to be supported by the verifier in *Chapter 2*. We design a modelling language for describing the process interactions and describe the verifiable properties in *Chapter 3*. We simulate designed models with a simulator described in *Chapter 4*. In *chapter 5* we present the details of implementation of the verifier. Use of verifier for sample real-life scenarios is given in *Chapter 6*. In *Chapter 7* we summarise our conclusion and future directions.

Chapter 2

Identification of constraints

2.1 Introduction

In this chapter we describe our choice of technology and identify the types of constraints that we wish our target system to verify. Conceptual models like the BPM are greatly influenced by the representation of time. A process embodies a temporally sequential coordinated series of stages linked together as a cohesive unit. The domain of BPM encompasses a variety of business vertical industries like the telecom, finance, utilities etc.,. In order to facilitate precise specification of the constraints irrespective of the field of study we choose logic. Since our emphasis is on the temporal aspects of the process models and since standard logic takes no cognizance of time related propositions we choose to use temporal logic.

2.1.1 Temporal Logic and models of time

We use temporal logic to specify the process model properties and to verify that these properties are preserved by the designed model. *Temporal logic* is a type of logic that is used for reasoning about properties changing with time. Most of the discussion that we present in this section regarding the time models and temporal operators are derived from *Handbook of Logic in Artificial Intelligence and Logic Programming* [9] and *Software Blueprints, Lightweight uses of logic in conceptual modelling* [26]. The

first need is to establish a temporal framework, a model of time itself and then to superimpose on this a set of structures representing the various kinds of changes that can occur in time. The structure we will superimpose on the time model is the business process model. Broadly there are two types of time models *independent-time models* and *dependent-time models*. In an *independent-time* model, time exists independently of any change. In the *dependent-time* model, time has no independent existence and is defined entirely in terms of an antecedently given notion of change. There are different structural models of time - *linear* (one future), *branching* (many possible futures) *or circular* (infinitely branching), *discrete or continuous* (tense), *bounded or unbounded* (finitely or infinitely stretching into past or future). In the *forward branching time* model each point has a unique past but more than one future. The different futures of a given point can be thought of as representing different possibilities. The passage of time involves narrowing down of possibilities as more and more of the 'fluid' indeterminate future becomes crystallized into a fixed determinate past.[9]. Discrete structures have built-in notion of duration, the duration of time from *t1* to *t2* is simply the number of instants between these two times. We choose to use simplistic model of the passage of time which is linear and the time points are dependent on the events in the superimposed process model.

There are two kinds of logic that have been used to express time-dependency of information : modal logic and first order logic.[11]. Modal logic approaches capture naturally the relative position of formulae with respect to an implicit current time by talking about past, present and future. On the other hand, first order logic (FOL) approaches naturally support absolute positions of formulae along the time line by making time explicit. With temporal modality it is possible to reason with concepts like eventuality and necessity.[26].

To be able to describe about time, the use of logic has been extended with temporal operators or *'logical connectives with temporal interpretation'*[26]. We use a small subset of the operators listed below for our work in this project.

- $\diamond$ read as *'eventually'* : $\diamond$P asserts that P is true sometime in the future or at least once in the future. This is a form of existential quantification over time. This is

a unary operator.

- $\rightarrow$ read as *'implies'*. p $\rightarrow$ q represents a conditional if *p* then *q*. This is a dyadic operator.

- $\Box$ read as *'always'* : $\Box$P asserts that P is true now and at all future times. This is a form of universal quantification over time. This is a unary operator.

- $\bigcirc$ read as *'next state'* : $\bigcirc$ P asserts that P will be true at the next instant in time after the initial time. This is a unary operator.

- *S* read as *'Since'* : *qSp* asserts that *q* has been true since the time *p* was true. This is a binary operator

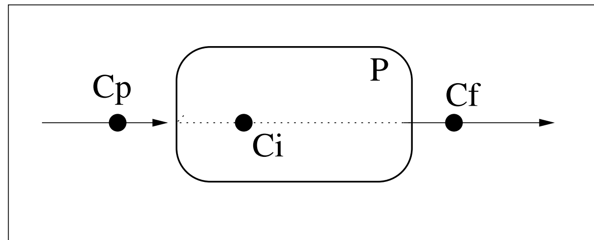
- *U* read as *'Until'* : *qUp* asserts that *q* will be true until the time when *p* will be true. This too is a binary operator

In our work we will use the $\diamond$ and the $\rightarrow$ operators to state and verify the satisfaction of constraints.

2.1.2 Constraints

As per the IDEF9 method, *a* **Constraint** *is a relationship that is maintained or enforced in a given context* [20]. *Relationship* is an abstract association or connection that holds between two or more conceptual objects. A constraint is a special kind of relationship that is restricted or compelled to exist under a given set of conditions. We identify such constraints between business processes and in relation to time. A constraint is said to *hold* in a given context when the relationship is maintained in that context. In order to verify whether a constraint holds for a process using modal temporal logic, the intuitional points of *current time* reference would be at the start of the process, while the process is in progress and after the process has ended. We choose to represent the start of a process P as *'start(P)'*, end as *'end(P)'* and constraint as *'C'*. To make a distinction between constraints at the different points of reference we denote *'Cp'* as the constraint before the start of a process called *Prior Start Constraint*, *'Ci'* as the constraint within the process, *In-Process Constraint* and *'Cf'* as the constraint

at the end of the process, *Process-end Constraint*. The diagram below illustrates the identified constraints with respect to a process P.



Constraints

The temporal logic representation for the patterns depicted above are :

[G1] $Cp \rightarrow \diamond$ ***start(P)***

[G2] ***start(P)*** $\wedge$ ***Ci*** $\rightarrow \diamond$ ***end(P)***

[G3] ***end(P)*** $\rightarrow \diamond$ ***Cf***

Further we restrict our scope by making the assumptions given below

[A1] A process once started always terminates.

[A2] Once a process ends, the subsequent process (if any) will be executed.

[A3] A process is never instantaneous.

[A4] A constraint once satisfied remains true thereafter.

The corresponding logical representations are

[G4] ***start(P)*** $\rightarrow \diamond$ ***end(P)***

[G5] ***end(P1)*** $\rightarrow \diamond$ ***start(P2)***

[G6] ***start(P)*** $\rightarrow \neg$***end(P)***

[G7] ***satisfied(C)*** $\rightarrow \Box$***satisfied(C)***

The statements with numbering **[Gn]** are used to denote the patterns that we intend to include in our system. We will call this set as the *'Included Pattern List'* .

2.2 Sufficiency of temporal constraints

In this section we check to see if the constraints that we chose by intuition for the $\diamond$ operator are sufficient to express all the practical scenarios. First we give the rules for well-formedness and then conduct the sufficiency test on the permutations of {start(P), end(P), C, $\rightarrow$, $\diamond$} constituting different patterns.

2.2.1 Well-formed-formula

A well-formed-formula(wff) of sentential logic is any expression that accords with the following rules. [4]

1 *Atomic sentence* - A sentence letter standing alone is a wff.
Definition : The sentence letters are the atomic sentences of the language of sentential logic.

2 *Negation* - If ϕ is wff, then the expression $\neg\ \phi$ is also a wff.
Definition : A wff of this form is known as a Negation, and $\neg\phi$ is known as the negation of ϕ.

3 *Conjunction* - If ϕ and ψ are both wffs, then the expression ($\phi \wedge \psi$) is a wff.
Definition : A wff of this form is known as the Conjunction. ϕ and ψ are known as the left and the right conjuncts respectively.

4 *Disjunction* - If ϕ and ψ are both wffs, then the expression ($\phi \vee \psi$) is a wff.
Definition : A wff of this form is known as a Disjunction. ϕ and ψ are the left and the right disjuncts respectively.

5 *Conditional* - If ϕ and ψ are both wffs, then the expression ($\phi \rightarrow \psi$) is a wff.
Definition : A wff of this form is known as the conditional. The wff ϕ is known as the antecedent of the conditional . The wff ψ is known as the consequent.

6 *Biconditional* - If ϕ and ψ are both wffs, then the expression ($\phi \leftrightarrow \psi$) is a wff.
Definition : A wff of this form is known as the Biconditional. It is also sometimes known as an equivalence.

7 Nothing else is a wff.(as per [4])
As for the temporal logic operators or connectives we include an additional rule stated below.

8 No modal operators can appear before the '$\rightarrow$' operator meaning the antecedent of the conditional cannot have modal operators. We make this rule because we want the evaluation of our temporal expressions to be anchored at a specific instant of time

2.2.2 Test for sufficiency

In order to ascertain that we have accounted for all the possible valid combinations of the chosen logical operators, we consider the different permutations of the five representations $\diamond$, $\rightarrow$, *start(P)*, *end(P)* and *C* with the logical connective 'and' ($\wedge$). We then eliminate those propositions which are syntactically malformed as per the rules listed in section 2.2.1 and evaluate the need for inclusion of the remaining propositions into our *Included Pattern List* in section 2.1.2. The eliminated propositions are each labelled [En] where 'n' is an identifying integer while the included propositions are labelled [Gn] and are in boldface. While evaluating we make use of the commutative property of the conjunction connective to reduce the list of propositions to be evaluated.
In the remainder of this section we present the discussion for those propositions which are syntactically correct.

$$[\text{E1}]\ C \wedge start(P) \rightarrow \diamond end(P)$$

This proposition corresponds to the In-Process Ci constraint which is already included in *Included list of patterns*.

$$[\text{E2}]\ C \wedge end(P) \rightarrow \diamond start(P)$$

The proposition that at the time of reference, if a constraint holds and the process ends, then sometime in the future, the process will start is not sound. For any process, the start of a process must precede its termination. Hence we eliminate the above

proposition and also all the others which are syntactically correct but have end(P) and $\diamond$start(P). These are listed below.

$$C \rightarrow end(P) \wedge \diamond start(P)$$
$$end(P) \rightarrow C \wedge \diamond start(P)$$
$$end(P) \wedge C \rightarrow \diamond start(P)$$

Next we consider,

$$\text{[E3] } C \rightarrow start(P) \wedge \diamond end(P)$$

The above proposition is that if a constraint holds then it implies that a process starts and will eventually end. We can derive a new pattern to indicate that if a constraint holds then it marks the start of a process which will eventually end. However we can restrict this to mark just the start of a process as we already have the pattern G4 which indicates the eventual termination of a process once it has started. This would allow us to represent constraints where we would like the process to begin with some initial condition. It may not be necessary that the condition holds before the process could begin. Let us call this as the *Start-Point* constraint Cs. There is a subtle difference between the constraint Cs and the Cp constraint in G1. The Cp constraint does not allow the process to start if it is not satisfied, while the Cs constraint does not put such a restriction on the start of a process. If we check for the satisfaction of the constraint at an instance just before the start of the process P, Cp *must* hold while it is not necesssary that the constraint Cs holds. Similarly the difference between the Cs and the Ci constraint of G2 is that Ci allows a constraint to be satisfied after the process has started and not necessarily at the start of the process. Ci is more lenient in allowing the constraint to be satisfied over an interval of time corresponding to the interval when the process is active and after the start of the process. The next question that needs to be addressed with regard to the above proposition is whether we need such a subtle distinction in the business process constraint specification? Probably yes. Let us consider an example where we model the visa application and grant process. Document verification is one of the sub-processes of the visa application process. The requirement is that at the time of verification the bank account balance of the applicant should have some minimum balance. We can represent the verification process with the

Cs constraint on the minimum balance. One might argue that the verification process itself can be elaborated in which case the minimum balance constraint can be specified as a Ci constraint on the verification process. We feel that it is a matter of choice of the modeler as to how he chooses to express the requirement. In order to allow the stricter Cs specification we choose to add the following proposition to our *Included list of patterns*.

[G8] $\boldsymbol{Cs \rightarrow start(P)}$

We now consider,

[E4] $start(P) \wedge end(P) \rightarrow \diamond C$

The above proposition can be derived from G3 and G4. Using G4 the start and end of the process can be established. Further, comparing the above proposition with G3 helps us to recognize the fact that this proposition is just the G3 pattern with start of a process also taken into consideration rather than just the end. Hence we feel that this is redundant. More over, with both start and end at the point of reference this represents an instantaneous process which we ignore by virtue of our assumption in [A3].

[E5] $start(P) \rightarrow C \wedge \diamond end(P).$

This gives a very lenient proposition that if we just know that a process has started then we can imply that the condition holds and eventually the process will come to an end. This proposition shifts the significance from the constraint to the process itself. This would be useful in the cases where we would like to indirectly derive the satisfaction of a constraint where it might be difficult to establish the constraint satisfaction directly. For example, when modeling business processes from the domains such as defense and health care which are bound by policies guarding the data privacy, it might not be possible to check the validity of a constraint directly. In such cases we can resort to the use of above proposition. Precaution should however be exercised in applying this type of verification rule to appropriate business processes as it could otherwise incorrectly indicate the satisfaction of the constraint. Since we already have the proposition G4 to reason about the start implying the eventual termination of a process, we will include the proposition in our list of patterns with just the start(P) and the constraint C, which we will choose to call Cz. This is the dual of the Cs constraint in G8.

$$\textbf{[G9]}\ \textbf{\textit{start(P)}} \rightarrow \textbf{\textit{Cz}}$$

Next we consider the proposition,

$$\text{[E6]}\ \textit{start(P)} \rightarrow \textit{end(P)} \wedge \diamond C$$

This proposition exemplifies an instantaneous process where the process starts and ends at the same instant. If we choose not to consider the entire proposition but consider only part of it with just the start of a process implying the eventual satisfaction of a constraint, then the argument that we putforth to E5 can be used here as well, with the difference being the time when the constraint holds is in the future rather than the time at which the process starts. We include the following to our list of patterns as *After-Start Cv constraint*.

$$\textbf{[G10]}\ \textbf{\textit{start(P)}} \rightarrow \diamond\textbf{\textit{Cv}}$$

The last of the patterns with the logical *and* operator,

$$\text{[E7]}\ \textit{end(P)} \rightarrow \textit{start(P)} \wedge \diamond C$$

The interpretation of the above is that if we know that a process has ended then we can derive the start of a process and the constraint being satisfied eventually. The above proposition is also related to instantaneous processes. end(P) $\rightarrow \diamond$C has occured in our list as Process-end Cf constraint in G3.

We have encountered several occurrences of instantaneous processes in our above analysis. Hence we devote a small part of our discussion in summarising a few observations of the instantaneous processes. An instantaneous process can be defined as a process that takes no time for its execution. This, we feel is highly impractical especially in the business process domain. We believe that any business process is characterized by more than one activity and each activity takes 'some' time for its completion. Since we believe that every process takes some time, the start and end of a process cannot happen at the same instant of time. If we know a process has ended then it must have

started sometime in the past. It was this observation that prompted us to include the assumption A3 about instantaneous processes that we presented in section 2.1.2. [1]

2.3 Constraint List

In this section we give the entire list of patterns that we aim to incorporate in our constraint verifier. Here we include the initial list that was given in section 2.1.2 along with the patterns that were uncovered in our sufficieny test in section 2.2.2.

2.3.1 Translation scheme

In order to be able to define the various constraints we use the following translation scheme.

before(A,B) - A occurs before B.
holds(C,T) - Constraint C holds at time T.
holdsin(C,P) - C holds in process/process role P.
start(P,T) - Process P starts at time T.
end(P,T) - Process P ends at time T.
satisfied(X) - Constraint X was satisfied.
Cp(C,P) - C is the Prior-start constraint for the process P.
Ci(C,P) - C is the In-Process constraint for the process P.
Cf(C,P) - C is the Process-end constraint for the process P.
Cs(C,P) - C is the Start-Point constraint for the process P.
Cv(C,P) - C is the After-start constraint for the process P.

If a constraint 'holds', it does not necessarily mean that the constraint was 'satisfied'. In order to 'satisfy' a constraint, in addition to 'holding', the temporal ordering requirements for the constraint type should be satisfied. This is elaborated in the definition of

[1] We could use this to introduce a new pattern to express the restriction that a process should have started before it can end. The following are the various forms of this statement. $end(P) \rightarrow \neg(\diamond start(P))$
$end(P) \rightarrow \overleftarrow{\diamond} start(P)$
$\overleftarrow{\diamond}$ operator above symbolizes *sometime in the past* notion.

each of the constraints that is given in the next section. However we have left out the definition of Cz constraint as it is the dual of Cs constraint.

2.3.2 Constraint definitions

Cp constraint : Also called Prior-Start constraint, this indicates that the constraint should hold before the process for which it is specified has started.

$$satisfied(Cp(C,P)) \rightarrow holds(C,T1) \wedge start(P,T2) \wedge before(T1,T2)$$

Ci constraint called an In-Process constraint, this requires the constraint to hold after the process has started and before the process has ended.

$$satisfied(Ci(C,P)) \longrightarrow holdsin(C,P).$$
$$holdsin(C,P) \longrightarrow holds(C,T1) \wedge start(P,T2) \wedge end(P,T3)$$
$$\wedge\ before(T2,T1) \wedge before(T1,T3).$$

Cf constraint : Also called Process-end constraint, this signifies that the constraint will be satisfied sometime after the process has ended.

$$satisfied(Cf(C,P)) \rightarrow holds(C,Tc) \wedge end(P,Te) \wedge before(Te,Tc).$$

Cs constraint Called the Start-Point constraint, the constraint should hold at the instant when the process starts.

$$satisfied(Cs(C,P)) \rightarrow holds(C,T1) \wedge start(P,T2) \wedge \neg(before(T1,T2) \wedge before(T2,T1))$$

Cv Constraint : Also called After-start constraint, the satisfaction of this constraint is dependent on the start of the process and the constraint should hold sometime after the process has started.

$$satisfied(Cv(C,P)) \rightarrow start(P,T1) \wedge holds(C,T2) \wedge before(T1,T2)$$

Chapter 3

Process Communication Model Description Language - PCMDL

3.1 Introduction

In this chapter we introduce a diagrammatic language, PCMDL to model business process interactions and to represent the constraints related to the model. We also define the generic properties applicable to process models. The emphasis of PCMDL will be on the inter-process interactions and not on the intra-process data flow. The language caters for the most general behaviour modeling requirements of the business process models. The language design is influenced by *Fundamental Business Process Modeling Language*(FBPML) and the *Unified Modeling Language*(UML) behaviour representational diagram - the Interaction diagram. The rationale behind the choice of the above languages as the basis is the need for the logical representational features of FBPML and the temporal expressibility of the UML diagrams. Before we describe the semantics of our PCMDL, we will briefly review the FBPML and the UML interaction diagrams from the perspective of the features that we wish to inherit from these languages.

3.2 Fundamental Business Process Modeling Language

Fundamental Business Process Modeling Language (FBPML)[7] is a visual business process modeling language that derives its design from IDEF[16], RAD[21] and PSL[28].[12]. It offers precise semantics for business processes in first order logic. FBPML is both executable and formal. FBPML is executable in that the use of FBPML process components can be interpreted using an execution mechanism.[7]. The formal approach allows for automatic/intelligent analysis, verification, validation and simulation. It has representations for timepoints, duration and lengths. It has a useful concept of 'Role' which is described as a set of activities which carry out a set of responsibilities. A role may be enacted by an individual, group of people, agents or software components[17]. However it does not make explicit the visual representation of information flow between the interacting processes and focuses on the definitions of activities.

3.3 Unified Modeling Language

UML has a rich collection of diagrams which help to express flexibly the structural and behavioural aspects of a model. The description of behaviour involves two aspects 1)the structural description of the participants and 2)The description of the communication patterns.[2] The communication pattern performed by instances playing the roles to accomplish a specific purpose is called an interaction. UML provides two forms of interaction diagrams namely *Collaboration diagrams* and *Sequence diagrams*. Though both of these diagrams represent the same information, they differ in their emphasis on the aspects of interaction. A sequence diagram shows the explicit sequence of communications while the collaboration diagram shows an interaction organized around the roles in the interaction and their relationships. As we are interested in the information flow sequence we adopt a few features of the sequence diagrams. A sequence diagram has two dimensions - time and role instance. It presents an interaction as a set of messages between the interacting roles. It allows for the representation of timing constraints using time expressions like *sendTime* on message names. It has the concept of lifeline which represents the existence of an instance at a particular time. The lifeline may be split into two or more concurrent lifelines to show condi-

tionals. Each separate split of the lifeline corresponds to a conditional branch in the communication.

3.4 PCMDL

PCMDL is a simple language focusing mainly on the representational aspects of process interactions. It incorporates the FBPML workflow notations and the sequence diagram message flow representation. With the use of roles, it mandates responsibility ownership for any activity carried out within a business process. Unassigned activities are not allowed in PCMDL. This restriction is advantageous in preventing flaws in the practical deployment of the BPM. The flow of messages is represented against a dimension of time, the other dimension being the process instances. Time is represented along the vertical axis and proceeds down the axis. Use of name or label in the diagram that begins with an uppercase letter indicates that it is a variable. There are 2 types of notational elements, the basic elements and the connector elements.

3.4.1 Notations and semantics for basic PCMDL elements

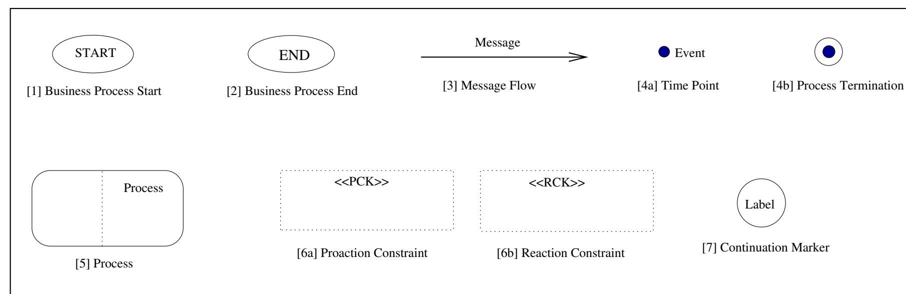


Figure 3.1: Basic PCMDL Elements

1:Business Process Start indicates the start of the business process being modeled. This is different from the notation used to explicitly mark the start of an individual process. Any model will have one business process start and several process start elements. This is the point where the user should start to read a particular flow.

2:Business Process End This notation is used mark the termination of the business process.

3:Message flow is represented by a solid arrow from the lifeline of one process to the lifeline of another process. The label on the arrow indicates the information that is sent from the originator of the message to the receiver. The message label M can have an optional element of iterator represented as [N]M where N represents that the message M needs to be sent N times. N could be a numeral or a variable which has been assigned a value earlier in the process.

4:Time point is the notation to mark particular point in time during the enactment of a process model. In our domain of modeling it is generally used to mark events and constraints. An event is a noteworthy occurrence which in our models will correspond to one of the following - send event, receive event, process start event, process end event and knowledge acquisition event. A send event and a receive event correspond to the two ends of a message interchange. The process termination event is notated as a circle around the timepoint. A knowledge acquisition event is linked diagrammatically to the *Knowledge* elements to indicate the information acquired by the process. Any event should always occur within a process.

5: Process Represented as a rectangular box with rounded corners, this is used to represent the process or a process role. The name of the process/process role is written at the top right corner. Optionally the name can also indicate the process instance. The dotted line at the center represents the lifeline of the process. Within a process the flow should be traced along the lifeline. A variation of this representation includes the graphical containment of a subprocess within the main process. The roles are represented as the processes. If an instance of a process changes roles during its lifetime, then the roles are represented graphically contained within the main process. Multiple instances are represented as a shadow to the process rectangle or using the stereotype ≪MI≫.

6: Knowledge is symbolized as a dotted rectangle with stereotypes ≪PCK≫ or ≪RCK≫. This is used to present the information that a process needs for the completion

of its task but the means of acquiring the information itself is outside the domain of modeling. The information may be obtained from an *external* entity or *self-processing*. An external entity is any process or enabler which is outside the boundary of the business process being modeled. Self processing is the set of activities that the process should perform internally to fulfill its responsibility. The stereotype ≪PCK≫ denotes that the information should be obtained before a message send event and is related to the message sent immediately after the knowledge acquisition. ≪RCK≫ indicates that the information should be acquired on receipt of a message and is associated with the message received immediately before the knowledge acquisition. ≪PCK≫ and ≪RCK≫ are called the *Proaction Constraint* and the *Reaction Constraint* respectively and will be explained in the context of agent simulation. Several knowledge acquisitions can be combined using the logical connectives 'and' and 'or'. The knowledge acquisition denotes a constraint on the progress of the process.

7:Continuation Marker When required to break the flow of the model, a marker is placed with unique label at point of discontinuation and an identical marker is placed also at the point where we would like the flow to continue. This aids in decomposed representation of complex models.

8:Constraint Any event can be explicitly marked to indicate a constraint on the process flow. Annotation *[C:Constraint_Type, ProcessP]* is used to represent any constraint. The *Constraint_Type* could be *Cp, Ci, Cf, Cs or Cv*. The *ProcessP* is the process on which *C* constitutes a constraint. This is by default the process/role in which the event takes place.

3.4.2 Notations and semantics for PCMDL connector elements

The connector elements are used to connect either processes or messages. The split and join connectors when used with processes are used to split and join the process lifelines. In the following section we give the informal semantics of the connectors.

C1 Sequence is the most common connector which signifies an ordering on the basic elements that it connects. For example, if the sequence connector is used to

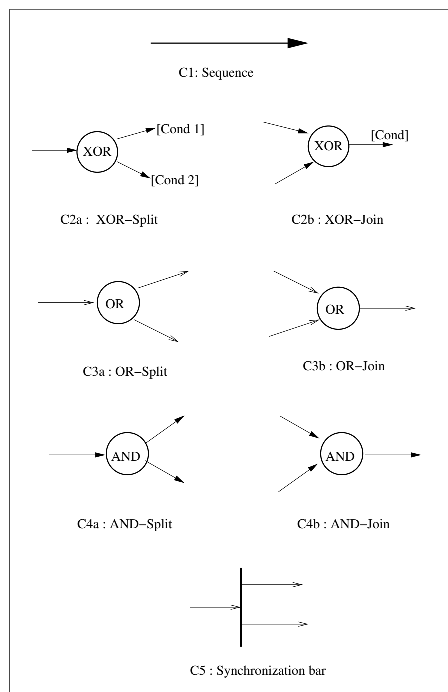


Figure 3.2: PCMDL Connector Elements

connect two processes then the process at the tail of the connector must have completed before the process at the head of the connector can begin.

C2a XOR-Split is represented as an XOR gate with single input and several outputs. This connector denotes the choice of only one of the outputs depending upon which of the conditions for the outputs was satisfied. This is used to model the usual if-then-else conditions and indicates an exclusive choice.

C2b XOR-Join indicates the selection of only one of the several input flows.

C3a OR-Split with one input and several outputs indicates multiple choice. One or more than one of the outputs can be selected.

C3b OR-Join combines multiple input flows into a single flow. If OR-Join is used to connect several input processes to an output process, completion of any one of the input processes implies the start of the output process. There may be multiple instantiations of the output process. This is used to represent the inclusive-or conditions.

C4a AND-Split splits a single input flow into multiple output flows without imposing any execution order on the output flows. It however indicates that all the output flows must be executed.

C4b AND-Join merges multiple flows into a single flow. It mandates the completion of all the input flows before allowing the output flow.

C5 Synchronization bar This is used to represent the split of a flow into several parallel flows or to combine several parallel flows into a single flow. When this is used to combine several parallel flows, it blocks the progress of tasks downstream until all the split flows merge.

The difference between the use of the synchronization bar and AND-split/join is that the parallelism that is enforced by the synchronization bar is absent in the AND connector. Junctions in FBPML correspond to PCMDL connectors. In addition to the FBML *'and', 'or', 'start'* and *'finish'* junctions, PCMDL has the XOR connector.

3.4.3 Formal representation of PCMDL

Our formal representation of the PCMDL revolves around roles and messages and is based on the formal approach to collaboration in UML by Gunnar *Ö*vergaard [8]. We adapt the approach to accomodate the notion of message sequences and connectors rather than activities. An illustration of the elements presented in this section is given in section 3.7 with respect to a loan scenario.

A PCMDL model M has roles and interactions denoted by *M.Roles* and *M.Interaction*. Each role is identified by a sequence of messages. An instance of a process conforms to a role if it has all the properties specified by the role i.e., if all the possible sequences of messages an instance can perform includes the sequences required by the role.

If 'R' denotes a role, the set of message sequences corresponding to the role R is given by *R.messageSequence*. The set of all possible message sequences for an instance I is given by *I.messageSequences*. Each message sequence is formally represented as

msq(SequenceId,MessageSet). The MessageSet is an ordered set consisting of messages and connectors. The representations for the messages and connectors are given later in this section.
An instance I is said to conform to a role R denoted as *conform(I,R)*

$$\textit{if}\ \forall\ ro \in R.messageSequence\ \exists\ io \in I.messageSequences$$
$$ro.SequenceId = io.SequenceId \wedge ro \sqsubseteq io$$

An interaction is defined by an ordered set of messages ($M, \leq$), M is set of messages and $\leq$ is the partial ordering on M. Each message of the set M is a tuple given by (sender,receiver,operation) and denoted by

$$message(name,sender,receiver,operation).$$

The messages specify the roles played by the sender and receiver processes. The operation in a message is formally represented as

$$op(opid,opname,arguments,eventList).$$

Each event in the *eventlist* is given by *event(eventId, eventType,eventDescription)*. The *eventType* could be one of *-process start,process end,send,receive or knowlegde acquisition*. Corresponding to the eventypes, the eventDescription could be start(process/role), end(process/role) or knowledge acquisition events which act as the constraints : proaction constraint *pck(C)* or a reaction constraint *rck(C)*. Though we have two other types of events namely the *send* and *receive*, these need not be explicitly stated in the operation eventlist as the message to which the operation belongs implicitly represents the send and receive event. There is a temporal precedence on these event types. For any message of a process P, the event precedence is given by,

$$start(P) < pck(C) < send < receive < rck(C) < end(P)$$

Similar to the representation of a role by a process instance during runtime, an instance of a message is recognised as a stimulus.
A stimulus S is said to conform to a message M (denoted as *conform(S,M)*)

$$\text{if } conform(S.sender,M.sender) \wedge$$
$$conform(S.receiver,M.receiver) \wedge$$
$$S.operation = M.operation$$

A sequence of stimuli SS made of $S_1, S_2, ...S_n$ is said to conform to a message sequence MS made up of messages $M_1, M_2, ...M_m$

$$\text{if } n{=}m \wedge \forall\, j \in \{1...m\}\ conform(S_j, M_j)$$

We now link the roles and interactions by means of the sequence of stimuli exchanged by the interacting role instances to that of the sequences which form a part of the interaction run. Let us consider *AS* to denote a message sequence. Let *stimuli(AS)* represent the set of stimuli exchanged by executing the message sequence *AS*. Let *IS* represent a set of interacting process instances and let *intr(IS)* represent the set of sequences of messages that can be exchanged by the interacting process instances IS. Let *linearizations(M)* be the set of all possible linearizations of a set of partially ordered messages. A linearization is a run of the interaction preserving the partial ordering defined on the set of messages which define an interaction. The set of process instances *IS* is said to conform to a model M if

$$\begin{gathered} conform(IS,M.Roles) \text{ and} \\ \forall int \in M.Interaction\ \forall ms \in linearizations(int.message) \\ \exists t \in intr(IS).\ \exists s \in stimuli(t).\ \exists ss \sqsubseteq s. \\ conforms(ss,ms) \end{gathered}$$

Stated in simple terms, for all the runs, the sequence of stimuli exchanged by the instances should conform to the role requirements and should be a part of the allowable sequences described by the interaction.
The connectors which act as the process operators for sequential composition are represented by

conn(Name, type, inputList, outputList)

where the name uniquely identifies each connector in the PCMDL and the type could be *AND, OR ,XOR, Sequence*. The *inputList* and *outputList* are constituted by the message list that are the inputs and outputs to the connectors. It is to be noted that the *conditions* associated with the XOR connector would be a part of the message event constraints. The sequence connector can also be used to connect roles. Finally the

Business Process Start and Business Process End elements are represented as *bpmstart(Process)* and *bpmend(Process)* where the *Process* in bpmstart indicates the process that should be taken up for the initiation of the model execution. bpmend(Process) indicates that the model execution terminates with the completion of the *Process*. A model can have more than one Business Process End elements in which case the *P* in *bpmend(P)* will be identifier composed of process and message identifiers.

The structural associations between the various elements introduced above is depicted in the diagram 3.3. Two types of relationships *Composition* and *Dependency* are used to express the associations. Composition, represented as a solid arrow, signifies that an element at the head of the arrow is made up of elements at the tail of the arrow. Dependency, represented as a dashed arrow indicates that an element at the tail of the arrow depends on the element at the head of the arrow. Dependency indicates the need for change in the dependent element, if the element on which it depends changes. A *Model* is made up of *Roles* and *Interactions*. A *Role* is a composition of several *MessageSequences*, each made up of several *Connector* and *Message* elements. An *interaction* is made up of an ordered collection of *Message* elements. The *Connector* element depends on the *Message* elements and the *Message* element in turn depends on the *Operation*. An *Operation* depends on the *event* element.

3.5 Formal definition of properties

Properties applicable to other standard models -*safety, liveness ,correctness, deadlock* and *termination* properties can be specified to each of the business process models that we develop. In this section we give the formal definition of the above mentioned properties. Let *satisfied(C,R)* represent the satisfaction of a condition C in run R. With the aid of this definition and several others that were introduced in section 3.4.3, we now define some properties that are suitable for verification of business process models. The system we develop verfies all but the deadlock property as we consider parallelization, where deadlock is more common, to be out of scope for our current work.

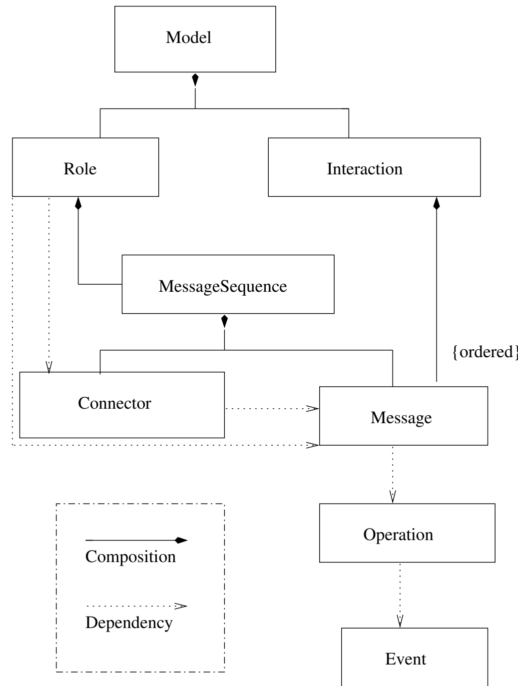


Figure 3.3: Structural relationship between the formal elements

Safety Property : If β represents some undesirable condition, a model is said to satisfy the safety property with respect to β, if in all the runs of the model, the condition β is never satisfied.

$$safety(\beta) \leftarrow \forall\ int \in M.Interaction.\ \forall\ run \in linearizations(int).\ \neg satisfied(\beta, run).$$

or

$$safety(\beta) \leftarrow \Box(\neg satisfied(\beta))$$

Liveness Property : If α represents some desirable condition, a model is said to possess the liveness property in any run, if at some point in the run the desirable condition is met.

$$Liveness(\alpha) \leftarrow \forall\ t \in M.Interaction.\ (run \in linearizations(t) \rightarrow satisfied(\alpha, run)).$$

or

$$Liveness(\alpha) \leftarrow \Diamond satisfied(\alpha)$$

Deadlock Property : Deadlock indicates a situation in which two or more processes wait for each other and are unable to proceed on their tasks as each is waiting for

the other. For a process model, each process waiting for a message from the other process before it can proceed with its part of the interaction exemplifies a deadlock. If *wait(P1,P2)* represents that process P1 waits for process P2, then

$$Deadlock \leftarrow wait(x,y).\ wait(y,x).\ x,y \in M.Roles$$

or

$$Deadlock \leftarrow \Box(wait(X,Y) \rightarrow wait(Y,X))$$

Correctness Property: Correctness property aims at meeting all the functional requirements expected of the given model. It aims to achieve the goal for which the model has been designed. By correctness property we consider the composition correctness described in [13], where the conjunction of all constraints is implied by the conjunction of all the property specifications and the conjunction of all constraints holds in the integrated behavioural model. Let us recall from section 3.4.1 that constraints are marked for a process or a role. If *M.Constraints* represents all the events that are marked as constraints for a model, the correctness property in any run will be satisfied if the constraints in *M.Constraints* for all the processes or roles invoked in the run are satisfied. Let *M.Constraints(r)* give the defined constraints for the role *r*.

$$correctness \leftarrow \forall\ t \in M.Interaction.\ \forall\ ms \in linearizations(t.messages).$$
$$\exists\ r \in M.Roles\ .\forall\ C \in M.Constraints(r).ms \in r.messageSequence \rightarrow satisfied(c)$$

or

$$correctness \leftarrow \forall R \in M.Roles(\ \forall c \in M.Constraints(R)\ (\Box(invoked(R) \rightarrow satisfied(c))))$$

Termination Property: Termination property requires the completion of an interaction. All the roles associated with the messages of an interaction should be closed. The closure of roles is explained in the context of agent simulation in section 4.2.2.

$$termination \leftarrow \forall\ t \in M.Interaction.\ \forall\ ms \in linearizations(t.messages).$$
$$\exists\ r \in M.Roles\ .\ ms \in r.messageSequence \rightarrow closed(r).$$

or

$$termination \leftarrow \forall R \in M.Roles\ (start(R) \rightarrow \diamond end(R))$$

3.6 PCMDL Model Execution

The execution of the model starts with the role indicated by the *bpmstart* element. Execution of a role implies the execution of a complete message sequence associated with the role. A role can be associated with more than one message sequences and one of them should be executed completely to mark the completion of the role. Each time a role is handed control or selected , the next unprocessed message or connector for the role from the message sequence should be executed. Execution of a connector chooses the next message to be processed. To execute a message would mean to execute the different events for the sender role complying with the precedence order given in section 3.4.3 and then to execute the different events for the receiver role. The execution of a message switches the selected role from the sender role to the receiver role. After execution of each message a check is made for completion of the selected role. The execution proceeds until the role associated with the *bpmend* element is completed. This marks the completion of the model execution. Each time a message sequence is chosen for execution, it should be checked against the sequences permitted by the interaction for further execution. The flowchart for the model execution is given in figure 3.4.

3.7 Illustration

We show the usage of PCMDL in modeling the following loan request scenario from Keith Mantell's article. [19].

"On receiving the loan request, the requested amount is compared to an amount (10000). If the requested amount is lower, then an Assessor service is called, otherwise the Approver service is used. If the Assessor deems the request to be high risk, it is also passed to the Approver. When either the Approver has completed or the Assessor has accepted, the approval information is returned"

The first step is to pick out the processes/roles. The two main interacting processes that need to be derived from the context of the above description are *requestor* and *bank*. The *assessor* and the *approver* services mentioned can be identified with the roles that the *bank* process might be required to play during the course of the scenario

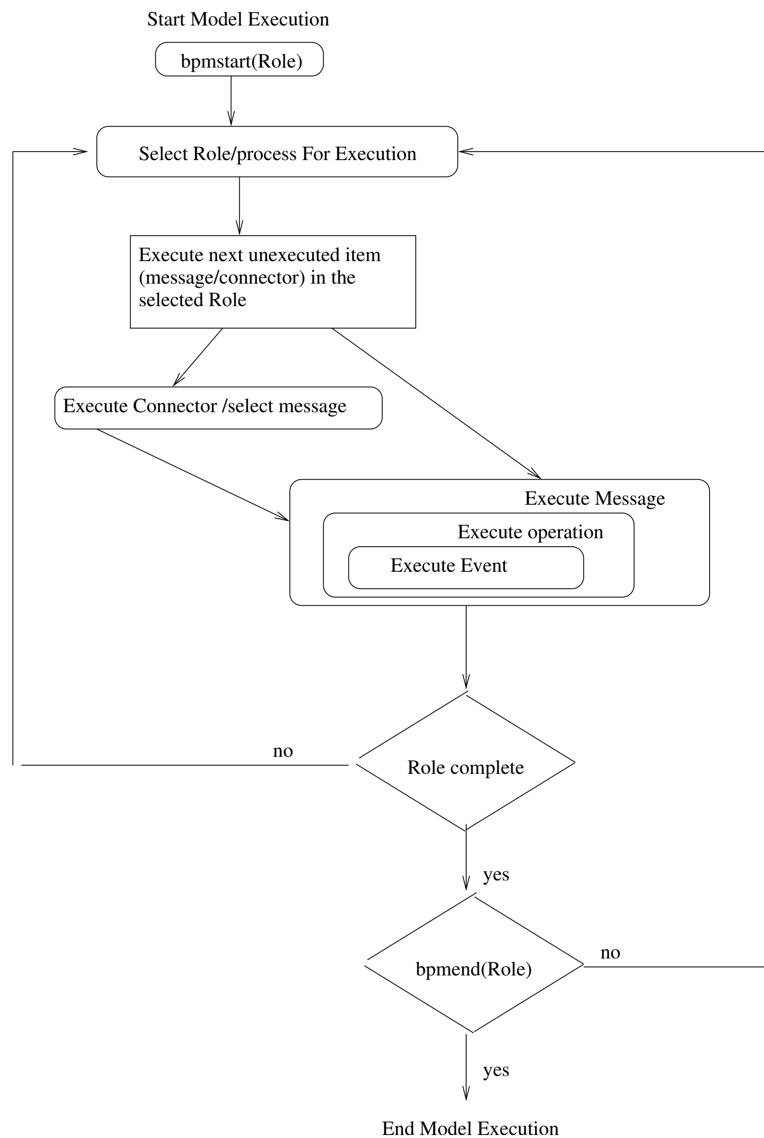


Figure 3.4: Flow chart for PCMDL model execution

execution.

Model = Loan Request

Model.Roles={*requestor, bank, assessor,approver*}

We now describe the scenario in terms of the messages exchanged by the roles that we have identified. It should be noted that though we give the semantics of element construction in the elaboration below, the user would be modeling using just the digrammatic notation and the information for the semantic representation is captured from the details supplied while constructing the diagram. The PCMDL for the scenario is given at the end of this section in figure 3.5.

The *requestor* sends a *loan_request* specifying the amount that he wishes to loan. The knowledge of the loan amount is modelled as the proaction constraint (≪PCK≫) on the loan_request message. On receipt of the loan_request, the *bank* processes the request in terms of determining the *request_level*. This constitutes an internal processing by the bank and is represented as a knowledge acquisition event (≪RCK≫). The elements of representation for this message exchange is given below.

message(m1,requestor,bank,opm1)

op(opm1,request_loan,[Amount,R],[pck(loan(Amount)),rck(request_level(Amount,10000,Level))])

The *Level* established by the *request_level* determines which of the two roles, namely *assessor* or *approver* that needs to be invoked. If the *Level* is *'l'* then assessor is called and if the *Level* is *'h'* then approver is called. These act as the constraints on the role transitions from bank to assessor or from bank to approver. The assessor role performs *assess_risk* to obtain the *Risk* factor for the requestor and the amount requested. Since no explicit operations are performed for role transitions the *opname* and *arguments* component of the operations in the representation below are set to '_'. The start of the subrole is indicated in the eventlist.

message(m3,bank,assessor,opm3)

op(opm3,_,_,[pck(Level=l),start(assessor),rck(assess_risk(R,Amount,Risk))])

message(m4,bank,approver,opm4)

op(opm4,_,_,[pck(Level=h),start(approver)])

The choice of message m3 or m4 after the execution of m1 is effected by an XOR connector.

conn(con1,XOR,[m1],[m3,m4])

If the *Risk* determined by the *assess_risk* is high-*'h'*, the assessor invokes the *Approver* else it calls on itself to process the request and set the *Result* to *'accept'*. The choice on the message flow is supported by an XOR connector.

message(m5,assessor,approver,opm5)
op(opm5,_,_,[pck(Risk=h),start(approver)])
message(m6,assessor,assessor,opm6)
op(opm6,_,_,[pck(Risk=l),rck(Result=accept)])
conn(con2,XOR,[m3],[m5,m6])

The *approver* is invoked either by the *bank* or by the *assessor*. An OR-Join is used to combine the two approver invocation message flows. Upon invocation, the approver calls on itself to process the request. This is indicated as a knowledge acquisition.

message(m7,approver,approver,opm7)
op(opm7,_,_,[rck(processed(R,Amount)),end(approver)])
conn(con3,OR,[m4,m5],m7)

On completion of the roles of *assessor* and/or *approver*, the bank sends the *loan_response* to the *requestor*. The flows from the assessor and the approver is combined by the use of an OR-Join.

conn(con4,OR,[m6,m7],[m2])
message(m2,bank,requestor,opm2)
op(opm2,loan_response,[Amount,Result],_)

As we now have all the messages that are needed to complete the model we describe the identified roles in terms of the message sequences.

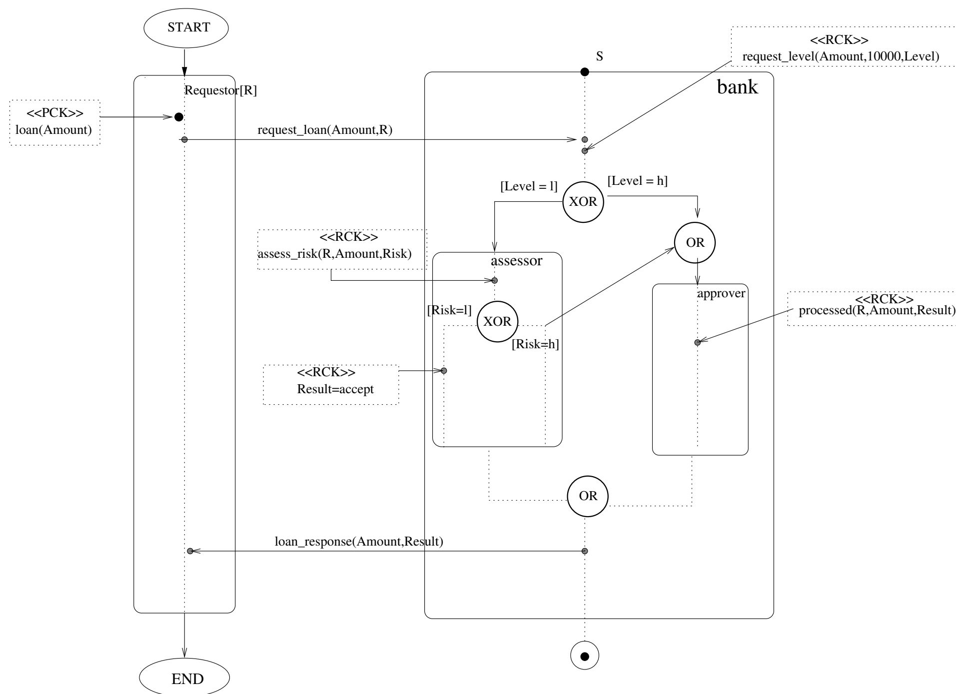


Figure 3.5: Scenario Loan Request

requestor = msq(seq1,{m1,m2})

bank=[msq(seq2,{m1,con1,assessor,con4,m2}),msq(seq3,{m1,con1,approver,con4,m2})]

assessor=[msq(seq4,{m3,con2,m5}),msq(seq5,{m3,con2,m6})]

approver=[msq(seq6,{m4,con3,m7}),msq(seq7,{m5,con3,m7})]

The Business Process Start and Business Process End for the scenario is associated with the *requestor* role.

BPMstart(requestor)

BPMend(requestor)

The interaction with the partially ordered messages is given by

Model.Interactions =

$\{m1,m2,m3,m4,m5,m6,m7 — (mi\{i=1..7\} < m2),(m3 < m5,m6),(m4,m5 < m7)\}$

The diagrammatic representation of the model is given in figure 3.5.

To summarize, the steps we followed to construct the model are listed below.
Identify the roles.
Model the messages in terms of operations and events.
Connect the messages with connectors to form the information flow.
Describe the roles in terms of the message sequences.
Identify the Business Process Start and Business Process End.
Describe the Interaction in terms of messages and their partial orders.

Chapter 4

Agent Simulation

4.1 Introduction

In this chapter we describe the Lightweight Coordination Calculus (LCC) protocol that forms the basis for the agent simulation that we use to simulate the interaction of business processes. A business process can be visualised as a community of negotiating, service providing agents. Each of the processes or roles will be enacted by an agent. In this chapter we make use of the term *agent* rather than referring to the business *processes* as it makes it more appropriate to describe the protocol designed for agents. We also succinctly describe the institution framework and give an illustration of describing a simple purchase scenario in LCC protocol. We then give an algorithm for the derivation of the agent simulation protocol from the PCMDL.

4.2 LCC protocol

Although efforts such as DARPA Agent Markup Language for Semantic web services (DAML-S), Agent Communication Language(ACL) and Knowledge Query and Manipulation Language(KQML) have aimed at standardising languages for description of interfaces between components, these are in themselves, insufficient to coordinate groups of disparate components in a way that allows substantial autonomy for individual agents while maintaining the basic rules of social interaction appropriate to partic-

ular coordinated tasks [24]. Though policy languages appear to provide a solution to coordination problems, they do not specify the interactions required between processes to the fine level of granularity as required for the specification of constraints on agent interaction. The electronic institution concept is identified as a possible solution.

Electronic institutions are based on the structure of human institutions and assist in enforcing the norms on the autonomous agents. Roles are assigned to the agents which act within the institution framework. The activities in an electronic institution are a composition of multiple, distinct, possibly concurrent, dialogic activities, each one involving different groups of agents playing different roles. For each activity, interactions between agents are articulated through agent group meetings, which are called *scenes*, that follow well-defined communication protocols [10]. Each scene is defined as a set of conversation states, roles that participate in the scene, and connections which specify the allowable transitions between scene states. Agent interactions are realised by message exchanges. Agents navigate from scene to scene constrained by the rules defining the relationships among scenes. Electronic institutions are characterised by *Performative Structures* and *Normative Rules*. The performative structure can be regarded as a network of scenes used to model the relationships between the scenes. It constrains the intra-scene and inter-scene behaviour of participating agents. It is specified as a set of scenes (with initial scene and final scene), transitions, connections. *Transitions* are special types of scenes which express the possible paths of the agents and *connections* link the scenes with the transitions. Normative rules are used to express the commitments, obligations and rights of participating agents.

LCC is a process calculus for specifying social norms. It is intended as a practical, executable specification language and can be supplied with a straightforward method for constraining the behaviour of an individual agent in a collaboration [24]. LCC not only allows for the specification of complex social norms but also overcomes the centralised control requirement of the state based institution systems. As described earlier, in a state based system, a set of state identifiers represent the stages in an interaction. Agent interactions are modelled in terms of agents entering and leaving the

states. LCC can also be used to address coordination problems where the behaviour of the agents are constrained in terms of the message sequences that they are allowed to send and receive.

4.2.1 LCC Syntax

The LCC interaction framework is shown in the figure below.

$$
\begin{aligned}
Framework &:= \{Clause, \ldots\} \\
Clause &:= Agent :: Def \\
Agent &:= a(Type, Id) \\
Def &:= Agent \mid Message \mid Def\ then\ Def \mid Def\ or\ Def \mid Def\ par\ Def \mid null \leftarrow C \\
Message &:= M \Rightarrow Agent \mid M \Rightarrow Agent \leftarrow C \mid M \Leftarrow Agent \mid M \Leftarrow Agent \leftarrow C \\
C &:= Term \mid C \wedge C \mid C \vee C \\
Type &:= Term \\
M &:= Term
\end{aligned}
$$

Where $null$ denotes an event which does not involve message passing; $Term$ is a structured term in Prolog syntax and Id is either a variable or a unique identifier for the agent. The operators $\leftarrow$, $\wedge$ and $\vee$ are the normal logical connectives for implication, conjunction and disjunction. $M \Rightarrow A$ denotes that a message, M, is sent out to agent A. $M \Leftarrow A$ denotes that a message, M, from agent A is received. The implication operator dominates the message operators, so for example $M \Rightarrow Agent \leftarrow C$ is scoped as $(M \Rightarrow Agent) \leftarrow C$

Figure 1. Syntax of LCC interaction framework

Figure from David Robertson's *A lightweight Coordination Calculus for Agent Systems* [24]

The interaction in LCC is achieved through message passing between the interacting agent roles. The framework permits role changes so as to meet the social norms. Constraints can be specified on the message send and receive events, and also on role changes. The constraints associated with message passing can be identified with the proaction and reaction constraints that we mentioned in PCMDL. In the case of constraint on message send event, the obligation in satisfying the constraint rests with the agent sending the message. In the case of the constraint associated with the message receive event, it is the receiver of the message that has to satisfy the constraint. The *Type* in *Agent* definition can be used to describe the agent's scene and role. This can also be used to specify information to a group of agents and thereby to check the constraints on groups of agents in terms of constraints on individual agents. The initiation

of the protocol is identified in terms of one of the clauses comprising the protocol. The termination of the protocol is determined by the use of *protocol closure* elaborated in the next section. The protocol progress is achieved by means of clause expansion by each agent and scene/role change.

4.2.2 Clause Expansion

The protocol expansion in LCC corresponds to the application of the *rewrite rules*(shown in figure 4.1) to unpack the protocol components received by an agent. With unpacking the agent gains the knowledge of its subsequent permitted moves. It then records the new state of dialogue. LCC does not prescribe the communication infrastructure for transmitting the message to the message exchange system. However it requires a message to be composed of an interaction identifier I, message receiving agent identifier A, role R of receiving agent, Message Content M as per LCC syntax in section 4.2.1 and Protocol P for continuing the social interaction. The protocol P consists of a set of clauses as per the LCC framework and a set of axioms K denoting the common knowledge assumed during the social interaction.

On receiving a message, it is added to the set of messages currently under consideration by the agent to give a message set M_i. Depending on its current role, the next step is determined by the selection of the appropriate clause C_i in the protocol P. Rewrite rules in figure 4.1 are then applied to expand C_i as C_n producing output message set O_n and remaining unprocessed messages M_n. The clause C_i in P is replaced with C_n to yield a modified protocol P_n. The agent sends a copy of P_n along with any message from message set O_n. As the clause store, which the agent consults to choose the next appropriate clause C_i, is carried with the protocol as the messages are sent, the interactions between the agents are restricted to be *linear*. An interaction is said to be *linear* if at any given time only one agent has the protocol or alters the state of the interaction. The rewrite rules require the satisfaction of constraints associated with the messages and role changes and, determination of clause closure. One of the methods to satisfy the constraints specified with the clauses C in protocol is by using the common knowledge K. A satisfiable instance of the clause C should be found in the common knowl-

edge K. Although the protocol supports the use of sophisticated constraint solvers, we have not used this in our simulations. A clause of the protocol is said to be closed if it has been covered or executed by the preceding message exchanges. The closure rules are also listed in the figure 4.1. When all the clauses in the protocol definition have been closed then the execution of the protocol terminates.

4.3 Simulator output

The simulation output gives the messages that were exchanged between the interacting agents in the order in which they were exchanged and also points of closure of the interacting agents and agent roles. The format of the output for the message M sent by an agent *Agent1* in role *Role1* to agent *Agent2* in role *Role2* is given as

$$con(Agent1,e(Role1,M \Rightarrow Role2))$$
$$or$$
$$con(Agent1,e(Role1, M \Rightarrow Role2 \leftarrow C))$$

The *C* above denotes the proaction constraint. The definition for roles is similar in syntax to the definition of Agent in section 4.2.1.
The simulation output format for the message received by an agent *Agent1* in role *Role1* from an agent *Agent2* in role *Role2* is as below.

$$con(Agent1,e(Role1, M \Leftarrow Role2))$$
$$or$$
$$con(Agent1,e(Role1,C \leftarrow M \Leftarrow Role2))$$

Whenever an agent changes roles, the transition is notated in the simulation output for a role change from *Role_F* to *Role_T* as

$$con(Agent,t(Role_F,Role_T))$$
$$or$$
$$con(Agent,t(Role_F,Role_T \leftarrow C))$$

When a constraint is specified on the agent without involving a message send, then the simulation output is

The following ten rules define a single expansion of a clause. Full expansion of a clause is achieved through exhaustive application of these rules. Rewrite 1 (below) expands a protocol clause with head A and body B by expanding B to give a new body, E. The other nine rewrites concern the operators in the clause body. A choice operator is expanded by expanding either side, provided the other is not already closed (rewrites 2 and 3). A sequence operator is expanded by expanding the first term of the sequence or, if that is closed, expanding the next term (rewrites 4 and 5). A parallel operator expands on both sides (rewrite 6). A message matching an element of the current set of received messages, M_i, expands to a closed message if the constraint, C, attached to that message is satisfied (rewrite 7). A message sent out expands similarly (rewrite 8). A null event can be closed if the constraint associated with it can be satisfied (rewrite 9). An agent role can be expanded by finding a clause in the protocol with a head matching that role and body B - the role being expanded with that body (rewrite 10).

$$
\begin{array}{ll}
A :: B \xrightarrow{M_i,M_o,\mathcal{P},O} A :: E & if \;\; B \xrightarrow{M_i,M_o,\mathcal{P},O} E \\
A_1 \; or \; A_2 \xrightarrow{M_i,M_o,\mathcal{P},O} E & if \;\; \neg closed(A_2) \wedge A_1 \xrightarrow{M_i,M_o,\mathcal{P},O} E \\
A_1 \; or \; A_2 \xrightarrow{M_i,M_o,\mathcal{P},O} E & if \;\; \neg closed(A_1) \wedge A_2 \xrightarrow{M_i,M_o,\mathcal{P},O} E \\
A_1 \; then \; A_2 \xrightarrow{M_i,M_o,\mathcal{P},O} E \; then \; A_2 & if \;\; A_1 \xrightarrow{M_i,M_o,\mathcal{P},O} E \\
A_1 \; then \; A_2 \xrightarrow{M_i,M_o,\mathcal{P},O} A_1 \; then \; E & if \;\; closed(A_1) \wedge A_2 \xrightarrow{M_i,M_o,\mathcal{P},O} E \\
A_1 \; par \; A_2 \xrightarrow{M_i,M_o,\mathcal{P},O_1 \cup O_2} E_1 \; par \; E_2 & if \;\; A_1 \xrightarrow{M_i,M_n,\mathcal{P},O_1} E_1 \wedge A_2 \xrightarrow{M_n,M_o,\mathcal{P},O_2} E_2 \\
C \leftarrow M \Leftarrow A \xrightarrow{M_i,M_i-\{M \Leftarrow A\},\mathcal{P},\emptyset} c(M \Leftarrow A) & if \;\; (M \Leftarrow A) \in M_i \wedge satisfy(C) \\
M \Rightarrow A \leftarrow C \xrightarrow{M_i,M_o,\mathcal{P},\{M \Rightarrow A\}} c(M \Rightarrow A) & if \;\; satisfied(C) \\
null \leftarrow C \xrightarrow{M_i,M_o,\mathcal{P},\emptyset} c(null) & if \;\; satisfied(C) \\
a(R,I) \leftarrow C \xrightarrow{M_i,M_o,\mathcal{P},\emptyset} a(R,I) :: B & if \;\; clause(\mathcal{P},a(R,I) :: B) \wedge satisfied(C)
\end{array}
$$

A protocol term is decided to be closed, meaning that it has been covered by the preceding interaction, as follows:

$$
\begin{array}{l}
closed(c(X)) \\
closed(A \; or \; B) \leftarrow closed(A) \vee closed(B) \\
closed(A \; then \; B) \leftarrow closed(A) \wedge closed(B) \\
closed(A \; par \; B) \leftarrow closed(A) \wedge closed(B) \\
closed(X :: D) \leftarrow closed(D)
\end{array}
$$

$satisfied(C)$ is true if C can be solved from the agent's current state of knowledge.
$satisfy(C)$ is true if the agent's state of knowledge can be made such that C is satisfied.
$clause(\mathcal{P},X)$ is true if clause X appears in the dialogue framework of protocol $\mathcal{P}$, as defined in Figure 4.2.1.

Figure 4.1: Rewrite rules for expansion of a protocol clause from [24]

con(Agent,e(Role,C))

The closure of role or dialogue of an agent is given as

closed(Agent)

4.4 Illustration

In this section we illustrate the specification of a purchase scenario in terms of LCC. The simple scenario is described below.

"A buyer wishes to buy a mercedes from a dealer called Drayton. He expresses his desire to buy the car and gets to know of the availability and its price from Drayton. Convinced of the price, he contracts Drayton to deliver the car to him."

4.4.1 Protocol Specification

The unique identifiers for the buyer and the seller agent is represented in the protocol as B and S respectively. The interaction initiation is by the agent *buyer* which first assumes the role of the *purchaser* and then *contractor*

a(buyer, B) ::=
    a(purchaser(S,X,Price), B) then a(contractor(S,X,Price), B).

In the role of the *purchaser*, the knowledge of the buyer that he needs to buy a mercedes from drayton dealer is represented as

known(_, obtain_from(mercedes,drayton))
The *purchaser* conveys this belief to the supplier in the role of the vendor and receives the vendor's intention of wanting to sell.

a(purchaser(S,X,Price), B) ::=
    want_to_buy(X) $\Rightarrow$ *a(vendor(B,X,Price), S)* $\leftarrow$ *obtain_from(X, S) then*
    want_to_sell(X,Price) $\Leftarrow$ *a(vendor(B,X,Price), S).*

The supplier takes on the roles of the *vendor* and the *contractee*.

a(supplier, S) ::=
a(vendor(B,X,Price), S) then a(contractee(B,X,Price), S).

The supplier in the role of the vendor, receives the message conveying the intention of the buyer in wanting to purchase a mercedes. The vendor needs to know the availability of mercedes and its price before he can respond with his intention to sell the mercedes. The knowledge of availability and the protocol specification for this part of interaction is as shown below.

known(_, available(mercedes,1999)).

a(vendor(B,X,Price), S) ::=
want_to_buy(X) $\Leftarrow$ *a(purchaser(S,X,Price), B) then*
want_to_sell(X,Price) $\Rightarrow$ *a(purchaser(S,X,Price), B)* $\leftarrow$ *available(X, Price).*

Having acquired the knowledge about the vendor's intention to sell mercedes and the price, the buyer takes on the role of a *contractor* to request for the delivery of the mercedes and subsequently receives the confirmation of the delivery from the supplier.

a(contractor(S,X,Price), B) ::=
deliver(X) $\Rightarrow$ *a(contractee(B,X,Price), S) then*
delivered(X) $\Leftarrow$ *a(contractee(B,X,Price), S).*

The part of the protocol specification for the supplier agent in the role of the contractee is given below.

a(contractee(B,X,Price), S) ::=
deliver(X) $\Leftarrow$ *a(contractor(S,X,Price), B) then*
delivered(X) $\Rightarrow$ *a(contractor(S,X,Price), B).*

4.4.2 Simulation Output

In this subsection, we describe the flow of messages constituting the interaction for the purchase scenario. Let *b* and *drayton* denote the instantiations corresponding to the identifiers B and S in the protocol definition. These roles are specified to the simulator as a Prolog list *[a(buyer,b),a(supplier,drayton)]*. The simulation begins with role transition of the buyer to that of the purchaser and the conditional message sent by the *purchaser*.

con(a(buyer,b),t(a(buyer,b),a(purchaser(drayton,mercedes,1999),b))).
con(a(buyer,b),
  e(a(purchaser(drayton,mercedes,1999),b),
    want_to_buy(mercedes)⇒a(vendor(b,mercedes,1999),drayton)
      ←obtain_from(mercedes,drayton))).

Subsequent to this the simulator focuses on the instance of the supplier which transitions to the role of the *vendor* to receive the message sent by the purchaser and sends its response. This completes role of the *vendor* and supplier changes role from *vendor* to *contractee*.

con(a(supplier,drayton),t(a(supplier,drayton),a(vendor(b,mercedes,1999),drayton))).
con(a(supplier,drayton),
  e(a(vendor(b,mercedes,1999),drayton),
    want_to_buy(mercedes)⇐a(purchaser(drayton,mercedes,1999),b))).
con(a(supplier,drayton),
  e(a(vendor(b,mercedes,1999),drayton),
    want_to_sell(mercedes,1999)⇒a(purchaser(drayton,mercedes,1999),b)
      ← available(mercedes,1999))).
closed(a(vendor(b,mercedes,1999),drayton)).
con(a(supplier,drayton),t(a(supplier,drayton),a(contractee(b,mercedes,1999),drayton))).

Receipt of the availability of mercedes with its price constitutes the end of *purchaser* role for the buyer which then changes to the role of the *contractor* and requests

for the delivery of mercedes to the supplier.

con(a(buyer,b),
e(a(purchaser(drayton,mercedes,1999),b),
want_to_sell(mercedes,1999)⇐a(vendor(b,mercedes,1999),drayton))).
closed(a(purchaser(drayton,mercedes,1999),b)).
con(a(buyer,b),t(a(buyer,b),a(contractor(drayton,mercedes,1999),b))).
con(a(buyer,b),
e(a(contractor(drayton,mercedes,1999),b),
deliver(mercedes)⇒a(contractee(b,mercedes,1999),drayton))).

Supplier Drayton in the role of the contractee acknowledges the delivery request with a delivered message. This completes the role of the contractor as well as the supplier.

con(a(supplier,drayton),
e(a(contractee(b,mercedes,1999),drayton),
deliver(mercedes)⇐a(contractor(drayton,mercedes,1999),b))).
con(a(supplier,drayton),
e(a(contractee(b,mercedes,1999),drayton),
delivered(mercedes)⇒a(contractor(drayton,mercedes,1999),b))).
closed(a(contractee(b,mercedes,1999),drayton)).
closed(a(supplier,drayton)).

As a consequence to the receipt of the delivered message, the contractor role ends and this marks the completion of the buyer dialogue.

con(a(buyer,b),
e(a(contractor(drayton,mercedes,1999),b),
delivered(mercedes)⇐a(contractee(b,mercedes,1999),drayton))).
closed(a(contractor(drayton,mercedes,1999),b)).
closed(a(buyer,b)).

4.5 Mapping PCMDL to protocol specification for the simulator

In this section we describe as to how we translate the PCMDL entities to the LCC framework. The agent protocol specification is in some respects similar to Prolog predicates with a predicate head and a body. So we will use these concepts when describing LCC below.

For each role in the PCMDL model, represent the process or rolename as the head of the agent dialogue definition *a(rolename,Instance)*. The *Instance* can be any identifier like a prolog variable. If the role under consideration occurs as a subrole in the PCMDL model, then this identifier should be the same as the parent or the enclosing role. Each of the main processes should have a unique identifier. Any process or role will have one starting lifeline which may later branch into several lifelines. To provide a definition to the agent dialogue, follow the flow along the role starting lifeline, represent each PCMDL model element with the corresponding LCC framework element until the process/role termination is encountered. The three possible elements are the messages, connectors and the subroles. The role under consideration could be a sender or receiver of the message and the message could be sent or received with or without preconditions. The diagram 4.2 gives the translation that needs to be done for each of the stated message-role associations. Each box in the diagram is a translation pattern in which the upper half is the PCMDL segment and the lower half its LCC translation. The translation for the PCMDL connectors is straight forward. The OR-Join and the OR-Split connectors map to the operator *'or'*. The XOR-Split and the XOR-Join also map to the *'or'* connector with the difference being that the condition in XOR should be added as the proaction or the reaction constraint to the message following the connector. The default PCMDL sequence connector maps onto the operator *then*. The AND-Join and the AND-Split are often translated to the *then* operator because during protocol specification the order of execution left unspecified by the PCMDL AND connector gets materialised to some implementor specified order. The synchronisation bar is represented using the *par* operator. We have not encountered the need

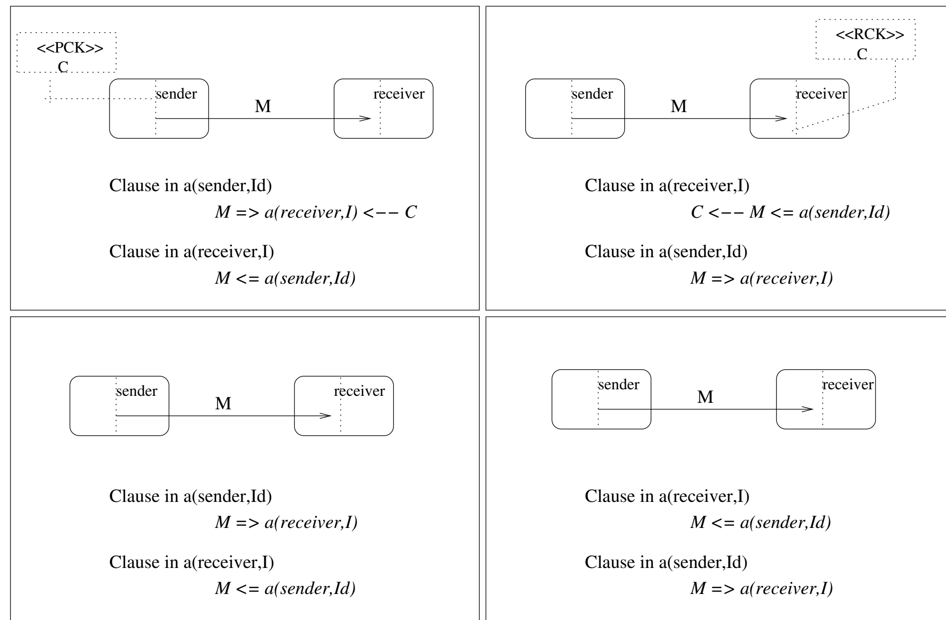


Figure 4.2: PCMDL to LCC Translation for message interchange

for use of AND connector in our simulation examples and the synchronisation bar was considered out of scope for this project. For the *or* and *par* operators, the two related operands must be enclosed in parentheses as in ((X)or(Y)), where X and Y may themselves be complex definitions composed of several connectors, roles and messages. If the PCMDL connector does not have a message or another connector or role following the connector under consideration, *null* is used as the second operand in the protocol specification.

The representation for the subrole is probably the simplest translation of the stated element types. The subrole name is specified as *a(subrole,Instance)* or *a(subrole,Instance)←C* where C represents the precondition on the role change. If the PCMDL has loop construct then, this gets translated to recursion in agent protocol definition.
The completed protocol specification for the role under consideration should appear as agent dialogue definition *a(rolename,Instance)::=Definition*. The above process is repeated for each of the roles and the subroles in the PCMDL model. The values for the knowledge elements (≪PCK≫ or ≪RCK≫) required for simulation are specified as *known(InstanceId,KnowledgeValue)* where the InstanceId is the *Instance* in agent

a(rolename,Instance) that requires the KnowledgeValue. The agents representing the main processes in the PCMDL (not as subroles) should be specified in the simulation initiation list.

The following gives the algorithm for PCMDL to LCC protocol translation. We refer to section 3.4.3 for the definitions of the PCMDL elements used in the algorithm. Translate (X) adds X to the cumulative agent definition at the appropriate location checking for duplicates and well-formedness as per LCC framework.

```
for each role r1 in PCMDL:
    Translate(a(r1,Instance))
    select a message sequence ms for r1
    for each element x of ms:
        if previous element of x is connector or x is a subrolename
           skip x
        else
             Generate(x,r1,Instance)
        if x is last element in ms
             Translate(end_dialogue)
        else
             Translate(then)
```

The algorithm for Generate(*x*,r1,I) is as follows

```
if x=message(Id,Sender,Receiver,Opid)
   get op(Opid,Opname,Arguments,Eventlist)
   if r1=Sender
      if Opname is not blank
         if no pck(C) in Eventlist
            Translate(Opname(Arguments) ⇒ Receiver)
```

```
        if pck(C) in Eventlist
           Translate(Opname(Arguments) ⇒ Receiver ← C)
     if Opname=blank
        if start(subrole sr) in Eventlist
           Generate(sr,r1,I)
        if pck(C) in Eventlist
           Translate(null ← C)
        else
             Translate(null)
     if r1=Receiver
        if Opname is not blank
           if no rck(C) in Eventlist
              Translate(Opname(Arguments) ⇐ Sender)
           if rck(C) in Eventlist
              Translate(C ← Opname(Arguments) ⇐ Sender)
     if Opname=blank
        if rck(C) in Eventlist
           Translate(null ← C)
        else
             Translate(null)


if x=conn(ConId,ConnType,IL,OL)
   if ConnType is Join
      Translate(Generate(m,r1,I))| m∈ OL)
   if ConnType=XOR
      Translate(((Generate(m_i,r1,I))or(Generate(m_j,r1,I)))|∀m_i,m_j∈ OL)
   if ConnType=OR
      Translate(((Generate(m_i,r1,I))or(Generate(m_j,r1,I)))|∀m_i,m_j∈ OL)
   if ConnType=AND
      Translate(Generate(m_i,r1,I) then Generate(m_j,r1,I)|m_i∈ IL,m_j∈ OL)
```

```
   else
        Translate(Generate(m_i,r1,I) then Generate(m_j,r1,I)|m_i∈ IL,m_j∈ OL)

if x=subrole sr
   get firstmessage m1 from a message sequence for sr where sender=r1
   get op(Opid_s,Opname_s,Arguments_s,Eventlist_s) for m1
   if pck(Cs) not in Eventlist_s
      Translate(a(sr,I))
   else if pck(Cs) in Eventlist_s
        Translate(a(sr,I)←Cs
```

Application of the above algorithm to the loan scenario in section 3.7 results in the following protocol specification.

a(requestor,R) ::=
    request_loan(Amount,R) ⇒ a(bank,B) ← loan(Amount) then
    loan_response(Amount,Result) ⇐ a(bank,B).

a(bank,B) ::=
    request_level(Amount,10000,Level) ← request_loan(Amount,R) ⇒ a(requestor,R) then
    ((a(assessor(Amount,R,Result),B) ← Level=l) or
    (a(approver(Amount,R,Result),B) ← Level=h)) then
    loan_response(Amount,Result) ⇒ a(requestor,R).

a(assessor(A,R,Result),B) ::=
    null ← assess_risk(A,R,Risk) then
    ((a(approver(A,R,Result),B) ← Risk=h) or
    (null ← Risk=l and Result=accept)).

a(approver(Amount,R,Result),B) ::=
    null ← processed(Amount,R,Result).

4.5.1 Problems in general automated translation

A few inadequacies in the PCMDL which offer resistance to automatic generation of LCC protocol specification from PCMDL and suggestions to address these deficiencies are discussed in this section. We also put forth a few observations to assist in relating the simulation entities to deployment entities.

The PCMDL does not accurately specify the parameters that are connected to the role name, that is the parameters that are needed to be passed on to a role upon invocation. This requires manual intervention to translate the PCMDL role to the LCC framework agent *Type*. This could be overcome by mandating the PCMDL modeller to specify the process names as a(Rolename(RoleParameters),Instance). Also PCMDL does not have accurate representation for the specifying how the value for the knowledge elements (≪PCK≫ or ≪RCK≫) should be computed. This does not pose a major problem to automating protocol generation as such because in most cases, the knowledge constraints represent the internal processing knowledge which for the simulation will be supplied by the user. However if an absolute need for accurate representation is felt, the use of Object Constraint Language (OCL) is recommended. The path chosen during a particular simulation run depends on the values specified for these knowledge elements. There should be an enhanced mechanism to capture such values as inputs to the model. In the case of PCMDL models with process self loops defined, the base condition to terminate the recursion implementing the loop in the simulator cannot be determined from the PCMDL model. Process self loops are the loops in which the process invokes itself.

The need for specifying timeouts on constraints is very helpful in many model design activities. As mentioned in [24] normal first order expressions should be used to construct the timeouts or temporal prohibitions. For simulations of PCMDL models with loop constructs, an agent with new identity is created for each iteration though in manual deployments these may correspond to the same individual or role. The time span of the activated role should be read as from the start of the agent role in the first iteration to the time of termination of the agent role in last iteration.

Chapter 5

Implementation for Constraint Verifier

5.1 Introduction

In this chapter we describe the implementation details of the constraint verifier. Due to the power and flexibility provided by the meta-interpreters, we opt to implement our verifier as a meta-interpreter. A *meta-interpreter for a language is an interpreter for the language written in the language itself* [30]. Our meta-interpreter is developed in Prolog. It depends on the model of time that acts as a foundation for superimposing the simulation output of the process model.

5.2 Concepts of time model

The primitive units of representation of time can be *instants* or *intervals*. It is possibile to construct intervals from instants and instants from intervals[9]. We build a meta-interpreter based on interval time representation and use the instant-interval interconversion to our advantage as and when required. We take time interval to be composed of several time instants. A time interval can hence be defined as an ordered pair (t1,t2) such that $t1<t2$ and is composed of instants t given by $\{t|t1<t<t2\}$. In the verifier that we build, when trying to reason about the time interval when a constraint holds, we need to relate two intervals. The following thirteen irreducible interval-interval relations ([Hamblin, 1969; Allen and Koomen,1983]) form the basis for our work [9].

1. (t,u) is **before** (v,w) if u<v
2. (t,u) **meets** (v,w) if u=v
3. (t,u) **overlaps** (v,w) if t<v<u<w
4. (t,u) **begins** (v,w) if t=v and u<w
5. (t,u) **falls within** (v,w) if v<t and u<w
6. (t,u) **finishes** (v,w) if v<t and u=w
7. (t,u) **equals** (v,w) if t=v and u=w

The other six relations of the thirteen are just the converses of the first six. The bounds constitute an important aspect in the definition of intervals. The output of the simulation which we superimpose on the time model establishes the bounds for our system which we call as the *system upper bound* and *system lower bound* in our ensuing elaboration.

5.3 Constraint language and Annotations

Our meta-interpreter uses annotations to represent the time intervals. The primary annotation is *interval*. A mundane example

lived('ShivaramKaranth') @ interval(1902,1997)

represents that Dr.K.Shivaram Karanth, a noted Indian writer lived during the period 1902-1997. A more relevant example relating to our process models would be *active(process) @ interval(T1,T2)* to denote the interval during which the process is active. Though several other models of time, permit the use of a finer granularity in the representation for times *T1* and *T2*, we restrict them to be integers as this suffices our purpose to annotate the events in the process model. As mentioned earlier we use the interval annotation to represent time instants as well. Using our earlier example of active process, the representation *active(process) @ interval(t,t)* indicates that a process is active at some time instant t. The first order definition of a formula annotated with *interval* is

$$X \text{ @ interval(t1,t2)} \leftrightarrow \forall t \ (t \in \text{interval(t1,t2)} \rightarrow X \text{ @ interval(t,t))}.$$

The other annotations that our interpreter supports are *after* and *before*. *after* is used to annotate atoms with semantics into the future from the current point of reference while *before* is used for annotating atoms with past semantics. Formula *X @ after(Num,I)* represents that *X* is true at an instant which is *Num* steps after the end of interval *I*. A variation of the *after* annotation, *X @ after(I)* gives the validity of X over an interval with the lower bound as the end of interval *I* and an interval upperbound determined by the system upper bound. Similarly, formula *X @ before(Num,I)* denotes that *X* is true at an instant, *Num* steps before the start of interval *I*. The variation *X @ before(I)* gives the validity of *X* over an interval with the system lower bound as its interval lower bound and the beginning of the interval I as the resultant interval upperbound.

The language we develop for the constraint verifier is given below. In each of the cases, the interval spans from the time instant the constraint holds to the system upper bound. This is based on the assumption in section 2.1.2 that a constraint once satisfied remains true thereafter.

ci(C,P) @ I : Constraint C holds within the process P.

cf(C,P) @ I : Constraint C holds after the end of the process P.

cp(C,P) @ I : Constraint holds before the start of the process P.

cs(C,P) @ I : Constraint holds at the start of the process P.

cv(C,P) @ I : Constraint holds sometime after the start of the process P.

the constraint *C* in each of the above cases can be represented using one of the structures below.

pck(C) : To denote the proaction constraint.
rck(C) : To denote the reaction constraint.
ks(I,M): To denote the message send event.
kr(I,M): To denote a message receive event.
C : Just as a constraint.

5.4 Preprocessor

The output of the simulator will be pre-processed to create a knowledgebase for use with the interpreter of the constraint verifier. The pre-processor gathers the information in the language of the interpreter described in section 5.3, along with annotations for the time instants. The following are the different structures for the elements of the knowledgebase.

start(A) @ I	start of a process or a role at time I
end(A) @ I	process or role is active over the interval I
ks(O,M) @ I	message M sent to a process/role instance O at time I
pck(C) @ I	proaction constraint C held at time I
kr(X,M) @ I	receipt of message M sent by a process/role instance X at time I
rck(C) @ I	reaction constraint C held at time I

The process or role notation will use the notation used in the simulator output while the time I will be in terms of *interval*.

5.5 Constraint Specification

The specification of the constraints follows a format very similar to that of the agent protocol specification. The basic format is shown in the figure 5.1. Property is an

```
cons(Property) ::= Property_Definition
Property = atom | Term
Property_Definition = W | W & W | W v W | W <--- D
W = Ci (C,P) @ I | Cp (C,P) @ I | Cf (C,P) @ I | Cs (C,P) @ I | Cv (C,P) @ I | role(Role,Instance) @ I | null | cons(Z)
C = ks (Y,M) | kr (Y,M) | C
I = after (I) | before (I) | I
D = Term | D & D | D v D
```

Figure 5.1: Constraint specification format

identifier of the constraint that is specified to the verifier. Property_Definition is a composition of several constraints of the form indicated above. The operator *::=* is

overloaded to be used in LCC specification as well as constraint specification. Each C in Cx(C,P) can be of the form ks(Y,M) or kr(Y,M) or just C. The ks(Y,M) format is used to verify if a message M was sent to instance Y while kr(Y,M) is from the receiver's perspective to check if a message M sent by instance Y was received. The proaction and the reaction constraints in the protocol specification are specified just as constraints. role(IRole,Ins) @ I is used to check if the instance Ins was in role IRole in the interval I. The constraint specification also allows dynamic specification with the format dyn(X) and dyn((X⟵Def)). The dynamic specification adds support for loop constructs, conditional constraint specification and means to supply the verifier with values for determining the satisfaction of constraints. Any verified property is satisfied only if all the elements in its Property_Definition are satisfied. It is possible to verify the order of constraint satisfaction by the use of annotations in section 5.3 as
Ci(C_1,P_1) @ I & Ci(C_2,P_2) @ after(I) to check if the constraint C_1 on process P_1 was satisfied before satisfying the constraint C_2 on process P_2 or Ca @ I & Cb @ I to check if both constraints Ca and Cb were satisfied within the interval I. If a constraint is satisfied, then the interval of validity returned by the verifier will have its upper bound as the system upper bound in accordance with our assumption A4 in section 2.1.2 that a constraint once satisfied will continue to hold thereafter.

5.6 The Interpreter

The proof strategy used in the interpreter is from [26]. This is informally described below

A given statement, P, follows from an argument A if any of rules 1 to 5 below apply, attempting in the order given.

1. P is fact which is asserted in Argument A.

2. P is of the form P1 $\wedge$ P2 and :

- *i P1 can be established from A and*
- *ii P2 can be established from A.*

3. P is of the form P1 $\vee$ P2 and P1 can be established from A.

4. P is of the form P1 $\vee$ P2 and P2 can be established from A.

5. Argument A contains an expression P $\leftarrow$ C and its condition, C, can be established from A

The formal proof strategy for the above listed informal strategy is given below.

A $\vdash$ P denotes that a given statement ,P, follows from an argument A.

1. solve(A $\vdash$ P) $\leftarrow$ P $\in$ A.

2. solve(A $\vdash$ (P1 $\wedge$ P2)) $\leftarrow$ solve(A $\vdash$ P1) $\wedge$ solve(A $\vdash$ P2).

3. solve(A $\vdash$ (P1 $\vee$ P2)) $\leftarrow$ solve(A $\vdash$ P1).

4. solve(A $\vdash$ (P1 $\vee$ P2)) $\leftarrow$ solve(A $\vdash$ P2).

5. solve(A $\vdash$ P) $\leftarrow$ (P $\leftarrow$ C) $\in$ A $\wedge$ solve(A $\vdash$ C).

Prolog implementation of the above rules supports the use of the annotations that we defined earlier. The knowledgebase created from simulation output is carried as one of the parameters in each of the *solve* clauses of our meta-interpreter along with the constraint specification. So the *solve* predicate has the structure

solve(knowledgebase,ConstraintDefinition, InputClause, OutputClause)

A goal of the form X @ I can be solved if there exists in the knowledgebase a fact X @ I or X @ I2 and, I and I2 can be combined using the interval-interval relation described in section 5.2 to yield an interval common to I and I2 over which X holds.

```
solve(KB,_,X @ I, X @ Ic) :-
    member(X @ I2,KB),
    combine(I,I2,Ic),!.
```

A goal of the form A & B can be solved if both A and B hold, or if the goal is of the form A v B and either A or B holds.

```
solve(KB,ConDef,A & B,A1 & B1):- !,
```

```
     solve(KB,ConDef,A,A1),
     solve(KB,ConDef,B,B1).

solve(KB,ConDef,A v B,R):-
     (solve(KB,ConDef,A,R);
     solve(KB,ConDef,B,R)).
```

The other means of satisfying the goal X @ I would be with the use of a rule with a the pre-condition in knowledgebase for a same or different interval and combining the intervals to yield a common interval.

```
solve(KB,ConDef,X @ I1,X @ I):-
     X @ I2 ⟸ Precondition,
     solve(KB,ConDef,Precondition , Ip),
     combine(I1,I2,I).
```

The interpreter also supports the use of rules which are valid irrespective of time though this is not of much use in our case as we preprocess the simulator output to be annotated with interval definition.

```
solve(KB,ConDef,X,X1):-
     \+(X = _ @ I),
     member(X ⟸ Precondition,KB),
     solve(KB,ConDef,Precondition,X1).
```

At times it may be required to verify the negation as goal, hence we have

```
solve(KB,ConDef,not(X),not(X)):- \+ solve(KB,ConDef,X,_).
```

Two intervals can be combined if either or both of them are uninstantiated and can be unified and normalised. If both the intervals are initialised, they are normalised and

then a unifying interval is found.

```
combine(E1,E2,I):-
     var(E1),
     var(E2),
     E1=E2,
     I=E1.

combine(interval(T1,T2),E2,I):-
     var(T1),var(T2),
     normalise(E2,I2),
     interval(T1,T2)=I2,
     I=I2.

combine(E1,interval(T1,T2),I):-
     var(T1),var(T2),
     normalise(E1,I1),
     interval(T1,T2)=I1,
     I=I1.

combine(E1,E2,I):-
     ((var(E1),nonvar(E2));
     (var(E2),nonvar(E1))),
     E1=E2,
     normalise(E1,I).

combine(E1,E2,I):-
     nonvar(E1),nonvar(E2),normalise(E1,I1),normalise(E2,I2),
     unify_intervals(I1,I2,I).
```

The normalise process smoothens the time definitions used with non-primitive an-

notations like *after* and *before* to the primitive *interval(t1,t2)* structure.

```
normalise(N, interval(N,N)):- integer(N).
normalise(interval(T1,T2),interval(T1,T2)).
normalise(after(N,E),interval(F,F)):-
     normalise(E,interval(_,T1)),
     future(T1,N,F).
normalise(after(E),interval(T,N)):-
     timelength(L),N=L,
     normalise(E,interval(_,T)).
normalise(before(N,E),interval(G,G)):-
     normalise(E,interval(T,_)),
     past(T,N,G).
normalise(before(E),interval(1,T)):-
     normalise(E,interval(T,_)).
```

In determining the time instant N units in the future we use the system upperbound identified below as *timelength(L)*.

```
future(X,N,X):-var(X),timelength(L),X=L.

future(T,N,F):-
     integer(T),
     F is T+N,
     timelength(L),
     F=<L.
```

Similarly for the determination of the past time interval, the system lower bound which is always 1 in our case is used.

```
past(1,N,1).
past(T,N,G):-
     integer(T),
     G is T-N,
     G>0.
```

The interval unification in accordance with the interval-relations in section 5.2 is implemented by the following functions taking into account the system bounds.

```
unify_intervals(interval(T1,T2),interval(T3,T4),interval(T5,T6)):-
     largest_lb(T1,T3,T5),
     smallest_ub(T2,T4,T6),
     t_consistent(T5,T6).

largest_lb(1,1,1).
largest_lb(1,T,T):-integer(T).
largest_lb(T,1,T):-integer(T).
largest_lb(T1,T2,T1):-integer(T1),integer(T2),T1>=T2.
largest_lb(T1,T2,T2):-integer(T1),integer(T2),T1<T2.

smallest_ub(L,L,L):-timelength(N),N=L.
smallest_ub(L,T,T):-integer(T),timelength(N),N=L.
smallest_ub(T,L,T):-integer(T),timelength(N),N=L.
smallest_ub(T1,T2,T1):-integer(T1),integer(T2),T1=<T2.
smallest_ub(T1,T2,T2):-integer(T1),integer(T2),T1>T2.
```

The requirement for a definition to be that of an interval is checked using t_consistent.

```
t_consistent(1,1).
t_consistent(1,T):-integer(T).
t_consistent(T,L):-integer(T),timelength(L).
t_consistent(T1,T2):-integer(T1),integer(T2),T1=<T2.
```

The following predicates are added to implement the constructs of constraint specification listed in section 5.5 to support looping and dynamic specification.

```
solve(KB,ConDef,cons(A),R):-
     expand_condef(cons(A),ConDef,Exp),!,
     solve(KB,ConDef,Exp,R).

solve(KB,ConDef,X←—P,X1):-
     solve(KB,ConDef,P,P1),
     solve(KB,ConDef,X,X1).

solve(KB,ConDef,A,_):-
     call_direct(A),
     A.

solve(KB,ConDef,null,null).
solve(KB,ConDef,X,C1):-
     protocol_member(dynamic_spec, ConDef, dyn(X ←— C)),
     solve(KB,ConDef,C,C1).

solve(KB,ConDef,X,X):-
     protocol_member(dynamic_spec, ConDef, dyn(X)).
```

The expand_condef expands the constraint rule definition. while the protocol_member checks for the dynamic definiton of a clause which is loaded as a part of the constraint specification.

As our meta-interpreter incorporates a process of normalisation, reasoning in terms of temporal intervals is limited in applicability as it assumes the normalisation to be able to bind intervals to particular time points which cannot be guaranteed always [25].

However since the simulation output establishes the required bounds in our system, this method of reasoning suffices our purpose.

5.7 Verifier output

The output of the verifier simply responds 'holds' or 'no' depending on whether the specified constraint was satisfied or not. If satisfied, the various variables used in the constraint specification are instantited to appropriate values.

5.8 Illustration of constraint specification and verification

In this section, we continue with the example of the loan scenario in 3.7 to show how the constraint specification for the properties can be done. For brevity, we limit our discussion to just the safety and liveness properties.

Safety Property :A probable safety property for the scenario could be to avoid contradictory responses from being sent to the loan requester. The loan request should either be accepted by the assessor or processed by the approver. It should never be the case that the loan request has been processed separately by both assessor and approver. From the PCMDL in section 3.7 for the scenario it can be gathered that the constraint specification should check if the risk has been assessed low and the result has been set to accept by the bank in the role of acceptor or the request is processed by the approver.

```
cons(safety) :: =
    (ci(Risk=l and Result=accept, a(assessor(A,R,Result),B)) @ I
        v
    ci(processed(Amount,R,Result),a(approver(Amount,R,Result),B)) @ I )
   &
   (not(ci(Risk=l and Result=accept,a(assessor(A,R,Result),B)) @ I
            &
       ci(processed(Amount,R,Result),a(approver(Amount,R,Result),B)) @ I)).
```

Liveness Property :Satisfaction of the liveness property requires that eventually a response be sent to the requester. From the PCMDL, it is the *loan_response* message that

should be sent by the bank to the requestor. We specify the constraint specification from the perspective of the requestor.

```
cons(liveness) ::=
    ci(kr(B,loan_response(Amount,Result)),a(requestor,R)) @ I
        &
    role(bank,B) @  I.
```

The output of the constraint verifier gives the safety property to hold while the liveness property fails. The failure of the liveness property is because the check for the role of the sender of loan_response message (that is the bank) does not exist when the message was received by the requestor. This shows that the bank process has no means of addressing the queries of the loan requestor once an initial response has been sent. This indication could be taken as an input to modify the loan scenario model to cater to queries arising from the dispatch of loan response.

Chapter 6

Evaluation and Discussion

6.1 Introduction

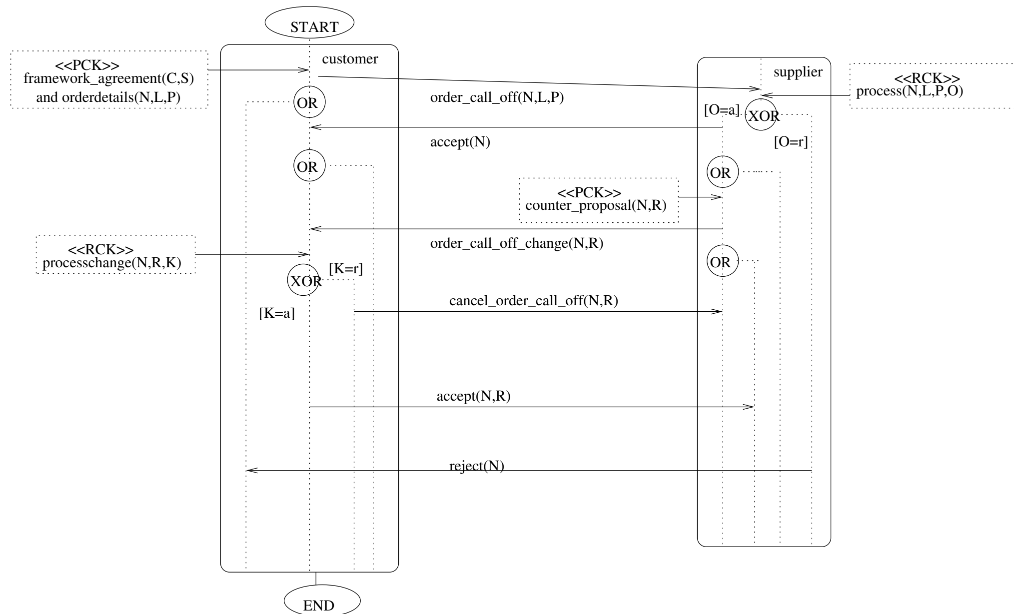
In this chapter we describe a few of the simulation processes that were used to evaluate the verifier. The domains used were supply chain management, software quality management, tender process evaluation, university student enrolment process, university student assignment submission process and telecom inventory procurement and commissioning process. We describe only the first three and for each domain we give the process description, PCMDL, protocol specification, constraint specification and an evaluation of the verifier.

6.2 Supply chain management (SCM) - Order call off

6.2.1 Process Description

"When a customer recognises a need for a product the supply of which is covered by a framework agreement with a particular supplier, the customer raises an order-calloff. The supplier either confirms his acceptance or rejects the order. If the supplier cannot meet all the order conditions, he sends a counter proposal with modified order conditions. The customer either accepts the revised order or cancels the order."

6.2.2 PCMDL



SCM process model

6.2.3 Protocol Specification

The interaction in the scenario described requires two roles identified as *customer* and *supplier*. The initiation of the interaction is by the agent in the role of the *customer*. The customer agent is supplied an order number on its instantiation. The customer knows the order details and having known the existence of a framework agreement with a supplier, sends an *order_call_off* message to the supplier. The supplier processes the request and determines whether the order can be fulfilled or not and this processing is indicated in the PCMDL as the reaction constraint *process*. The possibilities of the supplier response to the request are indicated by the *accept* and *reject* messages which is determined by the outcome of the *process* being *a* or *r* respectively. Further the *accept* response may have a variation in which the supplier indicates the fulfilment of the order request with a counter proposal wherein the order can be met with a slight change in the requested order items. This is indicated by the *order_call_off_change* message with a proaction constraint *counter_proposal*. The counter proposal is processed by the customer which is indicated by the constraint *processchange*. This processing indicates

to the customer whether to accept or cancel the order. Depending on the outcome of *processchange* the customer indicates its acceptance or rejection with the *accept* or *cancel_order_call_off* messages. The protocol specification from the perspective of the customer agent is shown in the figure 6.1.

```
a(customer(N),C) ::=
        order_call_off(N,L,P) => a(supplier,S))  <−−framework_agreement(C,S) and orderdetails(N,L,P)
        then
        ((accept(N) <= a(supplier,S)
              then
              (processchange(N,R,K) <−− order_call_off_change((N.R) <= a(supplier,S)
                    then
                    (accept(N,R) => a(supplier,S) <−− K=a
                                      or
                    cancel_order_call_off(N,R) => a(supplier,S) <−− K=r))
                 or
                 null)
           or
           reject(N) <= a(supplier,S)).
```

Figure 6.1: Protocol definition for Customer agent

The protocol specification from the perspective of the supplier agent is given in the figure 6.2. As per the PCMDL, the processing of the *order_call_off* message from the customer is indicated as a reaction constraint *process* and the determination of the counter proposal by the proaction constraint *counter_proposal*. Apart from the above dialogue specification, the knowledge possessed by the agents also forms a part of the protocol specification. This will be given in section 6.2.6.

6.2.4 Properties for verification

In this section we describe the generic safety, liveness, termination and correctness properties in terms of the supply chain management domain. Here *satisfied (X)* indicates the satisfaction of a property X, *send(X,R)* indicates X was sent to R, *receive(X,S)* indicates X was received from S.

```
a(supplier,S) ::=
    process(N,L,P,Outcome) <-- order_call_off(N,L,P) <= a(customer(N),C)
    then
    ((accept(N) => a(customer(N),C) <-- Outcome=a
      then
      order_call_off_change(N,R) => a(customer(N),C) <-- counter_proposal(N,R)
        then
        (accept(N,R)  <= a(customer(N),C)
                    or
         cancel_order_call_off(N,R)  <= a(customer(N),C)))
       or
       null)
     or
     reject(N) => a(customer(N),C) <-- Outcome=r).
```

Figure 6.2: Protocol definition for Supplier agent

Termination : Both the customer and the supplier roles will eventually end.

$$\textit{satisfied(termination)} \longleftarrow \diamond \textit{end(supplier)} \wedge \diamond \textit{end(customer)}$$

Safety : The customer never receives any contradictory responses from the vendor to his order request.

$$\textit{satisfied(safety(customer))} \longleftarrow \textit{(send(order,supplier)} \\ \wedge \neg\, \textit{(receive(response(accept),supplier)} \wedge \textit{receive(response(reject),supplier)))}$$

Safety property stated in terms of the supplier would require the supplier not to send contradictory responses to an order request.

$$\textit{satisfied(safety(supplier))} \longleftarrow \textit{(receive(order,customer)} \\ \wedge \neg \textit{(send(response(accept),customer)} \wedge \textit{send(response(reject),customer)))}$$

Liveness : Having placed an order before a supplier under a framework agreement, sometime in the future the customer gets an accept or reject response from the vendor.

$$\textit{satisfied(liveness)} \longleftarrow \textit{(send(order,supplier)} \rightarrow \diamond \textit{(receive(response(accept),supplier)} \\ \vee\, \textit{receive(response(reject),supplier))}$$

Correctness : The correctness property for the customer would require it to check the presence of a framework agreement with the supplier and place the order to the supplier, evaluate the counter proposal (if any) and accept or reject the counter proposal.

$$satisfied(correctness(customer)) \longleftarrow exists(frameworkagreement(S)) \wedge supplier(S) \wedge send(Order,S) \wedge (receive(response(accept),S) \wedge (((receive(changedorder(R)),S) \wedge processchange(R,Result) \wedge send(response(Result),S)) \vee null) \vee receive(response(reject),S))$$

The correctness property for the supplier would require it to check if it could meet the order placed by a customer either directly or with a counter proposal and send his response accordingly.

$$satisfied(correctness(supplier)) \longleftarrow receive(Order,C) \wedge customer(C) \wedge process(Order,R) \wedge send(response(R),C) \wedge ((counter_proposal(Order,P) \wedge send(changedorder(P),C) \wedge receive(response(N),S)) \vee null)$$

6.2.5 Constraint Specification

The constraint specification for the properties stated in section 6.2.4 is explained below. To satisfy the termination property, the verifier checks for the completion of the roles of the supplier and the customer agents.

cons(termination) ::= end(a(customer(N),C) @ _ & end(a(Supplier,S)) @ _.

The constraint specification for the safety property checks for satisfaction of safety properties from the supplier as well as the customer perspectives. The supplier should in no case send both *accept* and *reject* messages in response to the *order_call_off* message. Likewise the constraint verification on the customer checks for the reception of the response messages. Verifying from both perspectives can be avoided if we assume the channel to be lossless in the sense that a message sent will always be received by the intended recipient. In the specification below, we also check if the sequences of the messages sent are as per the requirements. For example, the *accept* or the *reject* messages should be sent only after the *order_call_off* message has been received by the supplier. The use of *after* annotation indicates this. We have used the after-start Cv

constraint so that we are able to detect the satisfaction of the safety property even in cases where the termination property is violated which cannot be done if In-Process Ci constraint is used.

```
cons(safety) ::= cons(safety(sup)) & cons(safety(cus)).

cons(safety(sup)) ::=
    cv(kr(C,order_call_off(N,L,P),a(supplier,s)) @ I
      &
    not(cv(ks(C,accept(N)),a(supplier,S)) @ after(I)
          &
       cv(ks(C,reject(N)),a(supplier,S)) @ after(I))

cons(safety(cus)) ::=
    cv(ks(S,order_call_off(N,L,P)),a(customer(N),C)) @ I
      &
    not(cv(kr(S,accept(N)),a(customer(N),C)) @ after(I)
          &
       cv(kr(S,reject(N)),a(customer(N),C)) @ after(I))
  .
```

The constraint specification for the liveness property checks for an eventual response from the supplier.

```
cons(liveness) ::=
    (cv(kr(S,accept(N)),a(customer(N),C)) @ I
              v
       cv(kr(S,reject(N)),a(customer(N),C)) @ I
    &
    role(supplier,S) @ before(I) <---(agreements(A) & member(S,A)).
```

In the specification above we also introduce a check for the supplier to be one among the several suppliers with which the customer has agreements. This demonstrates the use of dynamic constraint specification wherein we supply the verifier with the knowledge of the list of suppliers with which the customer has agreements as *dyn(agreements([s,s1,s2,s3]))*.

The correctness property just reiterates through each step of the protocol to ensure that the protocol was correctly followed. The constraint specification uses the stricter In-Process Ci constraint as we want to check the satisfaction of the constraint within a role. The specification given below lists the checks separately for the roles of the customer and the supplier though these could be combined.

```
cons(correctness) ::= cons(correctness(supplier)) & cons(correctness(customer)).

cons(correctness(supplier)) ::=
    ci(kr(C,order_call_off(N,L,P)),a(supplier,S)) @ I
      &
    role(customer(N),C) @ I
      &
    ci(process(N,L,P,G),a(supplier,S)) @ I2
      &
      (ci(ks(C,accept(N)),a(supplier,S)) @ after(I2) <--- G=a
          &
          ((ci(counter_proposal(N,R),a(supplier,S)) @ I3
                &
            ci(ks(C,order_call_off_change(N,R)),a(supplier,S)) @ after(I3)
                &
             (ci(kr(C,accept(N,R)),a(supplier,S)) @ after(I3)
                       v
              ci(kr(C,cancel_order_call_off(N,R)),a(supplier,S)) @ after(I3)))
         v
          null)
       v
       ci(ks(C,reject(N)),a(supplier,S)) @ after(I2) <---G=r.

cons(correctness(customer)) ::=
       ci(framework_agreement(C,S) and orderdetails(N,L,P),a(customer(N),C)) @ _
          &
        ci(ks(S,order_call_off(N,L,P)),a(customer(N),C)) @ I
          &
           ((ci(kr(S,accept(N)),a(customer(N),C)) @ after(I)
                 &
                (ci(kr(S,order_call_off_change(N,R)),a(customer(N),C)) @ I3
                         &
                   ci(processchange(N,R,K),a(customer(N),C)) @ I4
                         &
                      (ci(ks(S,accept(N,R)),a(customer(N),C)) @ after(I4) <---K=a
                                   v
                       ci(ks(S,reject(N)),a(customer(N),C)) @ after(I4) <---K=r)
                  v
                  null)
                v
              ci(kr(S,reject(N)),a(customer(N),C)) @ after(I)).
```

6.2.6 Evaluation and Discussion

The verifier is evaluated by subjecting it to several scenarios and checked to confirm if it was able to successfully determine the satisfaction of the various constraints identified for the supply chain management scenario in the previous section. Different scenarios can be simulated by just changing the knowledge possessed by the agents to effect the choice of the path for the interaction. The knowledge possessed by the agents for each scenario discussed is listed in figure 6.3. Each scenario is identified in terms of an order number that is supplied to the customer agent which is responsible

for initiating an interaction. The verification tests were done exhaustively for different combinations of incorrect deployments for different scenario alternatives. In the following discussion we give only a few of them for brevity and to demonstrate the knowledge specification and faulty protocol specification for simulations. To begin the simulation we need to instantiate the role of the customer and the supplier agents. This is done by supplying the simulator a list *[a(customer(sc4),c),a(supplier,s)]* where sc4 is the order number supplied to the customer agent. All our scenarios are related to a domain of supply chain management for a automobile spare parts and motor service dealer. The first of our cases is a simple scenario where the customer places an order

```
knowledge common to all scenarios
known(c,(orderdetails(N,L,P) <--- lineItem(N,L) and prodref(N,P))).
known(c,(processchange(N,R,K) <--- processedchange(N,R,K))).
known(s,(process(N,L,P,Outcome) <---processed(N,Outcome))).
known(_,framework_agreement(c,s)).

Scenario 1    order number sc3
known(c,lineItem(sc3,[(mirrorcoat,100)])).
known(c,prodref(sc3,refp3)).
known(s,processed(sc3,a)).

Scenario 2    order number sc1
known(s,counter_proposal(sc1,[(alphadec1011,1),(acfit,1),(steeringcover,2),(jack,10)]))
known(c,lineItem(sc1,[(sonydec1098,1),(acfit,1),(steeringcover,2),(jack,10)])).
known(c,prodref(sc1,refp1)).
known(s,processed(sc1,a)).
known(c,processedchange(sc1,R,a)).

Scenario 3    order number sc4
known(s,counter_proposal(sc4,[(pinkwheelguard,4)])).
known(c,lineItem(sc4,[(redwheelguard,4)])).
known(c,prodref(sc4,refp4)).
known(s,processed(sc4,a)).
known(c,processedchange(sc4,r)).

Scenario 4    order number sc2
known(c,lineItem(sc2,[(footrug,50),(rearmirror,2000),(bumper,190),(jack,100)])).
known(c,prodref(sc2,refp2)).
known(s,processed(sc2,r)).
```

Figure 6.3: Agent Knowledge for supply chain management scenario

for a service of hundred scratch resistant mirror coats. The dealer is able to meet the

demand and accepts the order. This corresponds to the order number sc3 in figure 6.3. We have an implementation that is correct in the sense the implementation is as per the design with the customer checking the framework agreement and sending the order to the appropriate supplier. In response, the supplier sends correctly the accept response without any contradictory responses. The output of the verifier correctly detected the satisfaction of safety, termination, liveness and correctness property.

We then simulate a couple of faulty deployments of the model with the supplier ignoring the order request or responding with both the accept and the reject responses. The faults are introduced in the agent protocol specification as in figure 6.4 for the case where the supplier sends contradictory responses. For the later case, the knowledge possessed by the supplier agent about the *process* outcome is changed to anything but *a* or *r* which causes it not to respond to the order request. In the first of the two faulty deployments, we would expect the verifier to detect the violation of liveness, termination and correctness property and also the satisfaction of the safety property. The output of the verifier was in conformance to our expectations. As for the second case too the verifier correctly detected the violation of the safety property and the satisfaction of all other properties.

```
a(supplier,S) ::=
    process(N,L,P,Outcome) <-- order_call_off(N,L,P) <= a(customer(N),C)
    then
    ((accept(N) => a(customer(N),C)
       then
       order_call_off_change(N,R) => a(customer(N),C) <-- counter_proposal(N,R)
          then
          (accept(N,R)  <= a(customer(N),C)
                           or
           cancel_order_call_off(N,R)  <= a(customer(N),C)))
        or
        null)
     then
     reject(N) => a(customer(N),C)).
```

Figure 6.4: Protocol definition for erroneous supplier agent

The next test case scenario requires the supplier to make a counter proposal which the customer accepts. This corresponds to order number sc1 in figure 6.3. The customer

order comprises of a sony stereo system among other things. The supplier does not have the requested sony stereo system in stock. Apart from the requested sony stereo system, the supplier is able to supply all the other items the customer has requested. Hence the supplier makes a counter proposal offering to supply an alpha dec music system instead of sony which the customer accepts. The correct deployment of the scenario caused the verifier to correctly detect the satisfaction of the properties. As for the faulty deployment of the scenario we consider the case where the customer does not check for the existence of a framework agreement and tries to send the order to some other supplier of its choice. For the simulation we have an instance for the chosen supplier who interacts normally as a supplier with an agreement would. For fault induction, the precondition or the proaction constraint *framework_agreement(C,S)* is removed from the dialogue corresponding to the customer agent in the protocol specification. The verifier was successful in correctly detecting the violation of the correctness property and satisfaction of the others.

Next test scenario is similar to the previous one with the difference being that the counter proposal made by the supplier is rejected by the customer. For better illustration we have the case where the customer requests for four red coloured wheel guards. The supplier attempts a counter proposal to supply with pink coloured wheel guards which the customer rejects. This corresponds to order number sc4. For the faulty implementation case, a customer receiving a counter proposal refrains from sending a response. This is effected by changing the *processchange* constraint output to anything but *a* or *r*. Both the correct and faulty implementations of the scenario showed the correct satisfaction and violation of the properties. The violation of the liveness property for the faulty implementation is because of the inclusion of the role determination of the message sender in the liveness property which cannot be resolved due to the non-termination of the supplier and the customer roles.

The scenario corresponding to the order number sc2 is the case where the supplier rejects the customer order outright as he cannot meet the requirements specified by the order even with a counter proposal. The order in the example has a large number of

foot rugs, rear view mirrors, bumpers and jacks. This too was verified correctly.

6.2.7 Conclusion

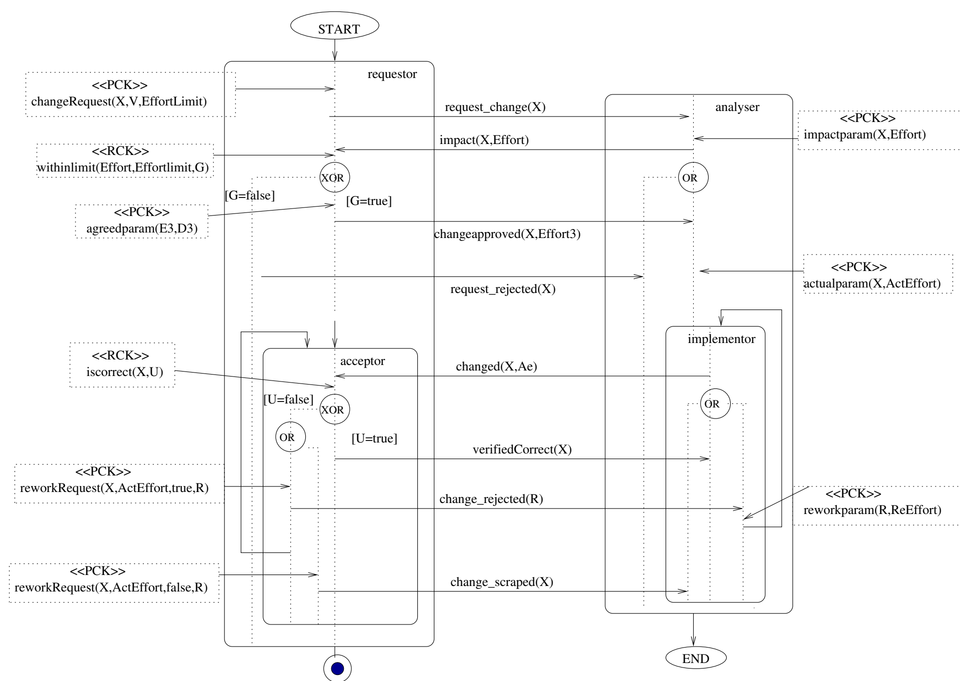
For this scenario simulation, the algorithm for automatic protocol generation in 4.5 was used. The faulty protocol simulation reflects the very common corrupt practices which might go undetected if not verified. With the verifier as a tool, it is possible to easily detect the errors. Though the verifier supports the determination of constraint satisfaction or violation for all the possible interactions, such an attempt on even a slightly complex scenario takes a very long time.

6.3 Software Quality Management - Change management process

6.3.1 Process Description

"A functional change in a released software has to be carried out within stipulated effort limits. An impact analysis of the requested change is performed by the supplier to estimate the effort required. If effort suggested by impact analysis is within limits, the requester approves the change for a negotiated effort and evaluates the changed software for requirement compliance. Any deviation would require a rework with additional effort and the process of change-evaluation iterates until compliance is established or the effort limit is exceeded. "

6.3.2 PCMDL



SQM change management process model

6.3.3 Protocol Specification

As indicated in the PCMDL, the change management process has two main interacting processes, each with a subprocess. The scenario simulation requires two agents and four roles. The identified roles are *requestor, analyser, acceptor* and *implementor*. The roles of requestor and acceptor is performed by one agent and, the roles of analyser and implementor is enacted by the other. It is the requestor agent which triggers the interaction. It knows the details of the change that is needed and the budget in terms of effort that is allocated for this particular change. For the purpose of simplicity, we restrict the known details about a change to be just the change request number X in the knowledge constraint *changeRequest(X,V,EffortLimit)* where V corresponds to the supplier/analyser who should be approached with the request for the required change. Without revealing the limits on the effort, the requestor communicates the change request to the analyser agent. In turn, the analyser performs the impact analysis, indicated as a constraint *impactparam* and responds to the request with the *impact* message indicating the effort estimate for the requested change. Equipped with this knowledge about the estimated effort and the previously known effort limit, the requestor determines whether the go ahead for implementation of the change in terms of approval can be granted. This is shown as the reaction constraint *withinlimit* and the possible decision outcomes, by the messages *changeapproved* and *request_rejected* messages. The *request_rejected* marks one of the possible business process termination messages. The *changeapproved* message is based on the assumption that the negotiations required for agreeing upon an effort by both the parties has been carried out and the agreed effort is acquired by the requestor agent as indicated by the knowledge acquisition *agreedparam*. In live deployments of this business process, the actual implementation process after change approval might involve several interactions between the two interacting parties. We opt to adhere to the process description in 6.3.1 and consider the details of implementation as external to the modelling boundaries. Hence the role of *implementor* in our case is summoned to take care of communicating to the requestor of the completed change and the actual effort that was consumed in the process of implementing the requested change. Thus on receipt of the *changeapproved* message, the analyser invokes the implementor with the constraint *actualparam*, that

supplies the actual effort to the implementor. The implementor then notifies the completion of change implementation and the actual effort in the *changed* message to the requestor, which has meanwhile changed its role to that of an acceptor after having sent the *changeapproved* message. Acceptor on its part validates the change for compliance (modelled as constraint *iscorrect*) and if found satisfactory, it sends the *verifiedCorrect* message to the implementor. This constitutes another possible business process termination. However if the change implementation is not satisfactory, the acceptor uses the actual effort sent to explore the possibility of a rework. A rework can be called for if there is still some effort left in the set budget. This decision is indicated by the *reworkRequest* which also creates a new request number corresponding to the change only if rework can be afforded. A negative rework decision causes the acceptor to notify the implementor to abandon the implementation with the *change_scraped* message. If the rework decision created a rework request number, this is sent to the implementor in the *change_rejected* message. This requires the implementation cycle to be repeated and as before we consider the actual implementation to be completed before the implementor is called upon with the actual effort supplied in this case by *reworkparam*. The acceptor also calls upon itself to await the details of rework. The subsequent interactions with the *changed* and *change_rejected* message is repeated until either a *verifiedCorrect* or *change_scraped* causes a termination of the business process.

The specification of the protocol for the two agents is given in figures 6.5 and 6.6. It may be noted that the model may have many implementor and acceptor agents instantiated as the roles call upon themselves in the *changed* - *change_rejected* message loop. The association of each implementor and agent instances is one-one. That is for each instantiation of an implementor role for a rework request a corresponding acceptor agent is instantiated.

6.3.4 Assumptions

Summarising the assumptions a few of which have already been stated explicitly in the protocol specification we have the following :
- The functional elaboration of the individual subprocesses is outside the modelling scope.

```
a(requestor(V,X),C) ::=
    request_change(X)  => a(analyser(C,X),V) <-- changeRequest(X,V,EffortLimit)
    then
     withinlimit(Effort,EffortLimit,G) <-- impact(X,Effort) <= a(analyser(C,X),V)
        then
        (request_rejected(X) => a(analyser(C,X),V) <--- G=false
        or
        (changeapproved(X,Effort3) => a(analyser(C,X),V) <-- G=true and agreedparam(X,Effort3)
            then
            a(acceptor(V,X,ActEffort),C))).
```

protocol specification for requestor agent

```
a(acceptor(V,X,ActEffort),C) ::=
     iscorrect(X,U) <-- changed(X,ActEffort) <= a(implementor(C,X,ActEffort),V)
    then
        (verifiedCorrect(X) => a(implementor(C,X,ActEffort),V) <-- U=true
        or
        (change_rejected(R) => a(implementor(C,X,ActEffort),V) <-- U=false and reworkRequest(X,ActEffort,true,R)
            then
            a(acceptor(V,R,ReEffort),C))
        or
        change_scraped(X) => a(implementor(C,X,ActEffort),V) <-- U=false and reworkRequest(X,ActEffort,false,_)).
```

protocol specification for acceptor agent

Figure 6.5: Requestor Agent Protocol

- The negotiated effort for an approved change is available with the requestor.
- The role of the implementor is called upon after the completion of the change implementation. The details of actual implementation is considered to be out of scope.
- Whenever the implementor agent reports the actual effort, it reports the total effort for the change request and not just the rework effort.
- Any message sent by an agent will eventually be received by the receiver agent.

6.3.5 Properties for verification

As in the earlier SCM example, *satisfied(X)* indicates the satisfaction of a property X, *send(M,R)* indicates M was sent to R, *receive(M,S)* indicates M was received from S.
Termination : The change process initiated should always complete. This requires all the initiated roles to eventually terminate. The roles of *requestor* and *analyser* will always be initiated while the *implementor* and the *acceptor* are initiated conditionally

```
a(analyser(C,X),V) ::=
    request_change(X) <= a(requestor(V,X),C)
    then
    impact(X,Effort) => a(requestor(V,X),C) <-- impactparam(X,Effort)
    then
      (request_rejected(X) <= a(requestor(V,X),C)
       or
      (changeapproved(X,Effort3) <= a(requestor(V,X),C)
          then
          a(implementor(C,X,ActEffort),V) <--  actualparam(X,ActEffort)))
```

protocol specification for analyser agent

```
a(implementor(C,X,ActEffort),V) ::=
       changed(X,ActEffort) => a(acceptor(V,X,ActEffort),C)
       then
           (verifiedCorrect(X) <= a(acceptor(V,X,ActEffort),C)
           or
           (change_rejected(R) <= a(acceptor(V,X,ActEffort),C)
               then
                a(implementor(C,R,ReEffort),V) <-- reworkparam(R,ReEffort)
           or
           change_scraped(X) <= a(acceptor(V,X,ActEffort),C)).
```

protocol specification for the implementor agent

Figure 6.6: Analyser Agent Protocol

depending upon the effort estimate, approval and rework decisions.

$$\begin{aligned} \mathit{satisfied(termination)} \longleftarrow\ & \diamond \mathit{end(requestor)} \wedge \diamond \mathit{end(analyser)} \\ & \wedge (\mathit{start(implementor)} \rightarrow \diamond \mathit{end(implementor)}) \\ & \wedge (\mathit{start(acceptor)} \rightarrow \diamond \mathit{end(acceptor)}) \end{aligned}$$

Safety : The effort limit should never be exceeded.

$$\mathit{satisfied(safety)} \longleftarrow \mathit{effort(actual)} < \mathit{effort(limit)}$$

Liveness : Any change-request should either be accepted or rejected or scraped.

$$\mathit{satisfied(liveness)} \longleftarrow \mathit{result(change,accept)} \vee \mathit{result(change,reject)} \vee \mathit{result(change,scraped)}.$$

Correctness : The onus of ensuring the correctness of the flow is borne mainly by the

requestor agent. The requestor should respect the limits on effort and approve the implementation of a change only if the subsequent impact analysis confirms the estimated effort to be less than the effort limit. The acceptor should accept the change if found to be correct, else should request for a rework if the actual effort does not exceed the limits and scrap the request if the actual effort exceeds limits. On its part, the analyser agent should respond to the requestor agent with impact and actual effort.

```
satisfied( correctness(customer)) ⟵ request(R) ∧ analyser(V) ∧ effort(limit,L) ∧
        send(request(R),V) ∧ receive(effort(impact,I),V) ∧
        ((withinlimit(I,L,true) ∧ send(approve,V)
        ∧ receive(response(change,effort(actual,A)),V) ∧
            (correct(change) ∧ send(accept,V)
            ∨ incorrect(change)
                ∧ ((exceedlimit(L,true) ∧ send(scrap,V))
                ∨ exceedlimit(L,false) ∧ send(rework,V)))
        ∨ (withinlimit(I,L,false) ∧ send(reject,V)))).
satisfied( correctness(analyser)) ⟵ receive(request(R),C) ∧ customer(C)
        ∧ analyse(R,effort(impact,I)) ∧ send(effort(impact,I),C)
        ∧ ((receive(approve,V) → send(response(change,effort(Actual,A))))
        ∨ (receive(rework,V) → send(response(change,effort(Actual,N)))))
```

6.3.6 Constraint Specification

The constraint specification for the termination property is given below.

```
cons(termination) ::=
    end(a(requestor(V,X),C) @ I & end(a(analyser(C,X),V) @  after(I2)
     &
        ((start(a(acceptor(V,_,_),C)) @ I3 & end(a(acceptor(V,_,_).,C)) @ before(after(0,I)))
          &
        ((start(a(implementor(C,_,_),V)) @ I4 & end(a(implementor(C,_,_),V) @ before(after(0,I2)))
        v
        (not(start(a(acceptor(V,_,_),C)) @ I3) & not(sart(a(implementor(C,_,_),V)) @ I4)))
```

The specification checks for the completion of the analyser and the requestor roles which are instantiated whenever the business process is triggered. The acceptor and

implementor roles may not be called upon in some cases as in the case when the requestor determines that the impact analysis has revealed an effort which cannot be accommodated within budget. Hence the part of the specification checking for the termination of the acceptor and implementor roles, test for the start of these roles. Further as we stated earlier the association between acceptor and implementor is one-one and hence we check for both of these roles to have started or not started.
The specification of the safety property checks for the actual effort to be less than the effort limit. The specification is shown below. The conditions *requests* in the specification provides the verifier with all the request numbers associated with a change request. It may be recalled from the protocol specification that whenever a rework is requested a new request number is generated. Hence a single change request can have more than one request numbers. The safety property is checked for each of the cases where the actual effort is sent to the requestor.

```
cons(safety) ::=
        ci(changeRequest(X,V,El) , a(requestor(V,X),C)) @ _ <--- (requests(Rs) & member(X,Rs))
            &
        cons(chksafety(Rs,V,El)).


cons(chksafety(Rs,V,El)) ::=
        ((null <--- Rs=[X1 | T]
            &
            (((ci(kr(V,changed(X1,ActEffort)),a(requestor(V,X1),C)) @ _
                        v
               ci(kr(V,changed(X1,ActEffort)),a(acceptor(V,X1,ActEffort),C)) @ _ )
               &
               ActEffort =<El)
             v
            not(ci(kr(V,changed(X1,ActEffort)),a(Role,C)) @ _))
         v
        null <--- Rs=[]).
```

The liveness property requires that the *request_rejected, verifiedCorrect* or the *change_scraped* message be eventually sent to the implementor. Since the process allows for the rework to be done with a different request number, the constraint specification needs to give the details of all the change and rework request numbers related to a particular change request. This is done as in safety specification by using the *requests(L)* dynamic con-

straint specification. As an additional check we add the constraint that the instance sending the message should be in the role of the acceptor.

```
cons(liveness) ::=
   ((ci(kr(C,request_rejected(X),a(analyser(C,X),V)) @  I & role(requestor(V,X),C)) @ before(I))
        v
  (ci(kr(C,verifiedCorrect(A)),a(implementor(C,X,_),V)) @ I <---(requests(L) & member(A,L))
           v
       ci(kr(C,change_scraped(X)),a(implemementor(C,X,_),V)) @ I)
        &
       role(acceptor(V,X,_),C) @before(I).
```

The correctness property checks if every interaction is as per model requirement. As we have detailed the interactions in the protocol specification, we give the constraint specification with just an overview. To maintain legibility in specification, the correctness constraint is specified in three parts, *cons(correctness)* is the main specification which uses the two other parts *cons(correctness(acceptor(N)))* and *cons(correctness(implementor))* corresponding to the roles of acceptor and the implementor. The correctness is checked for each of the acceptor instances created during the rework cycle. We assume that a message sent by an agent will eventually be received by the receiver. Hence we do not check for the reception of message by each implementor instance.

```
cons(correctness) ::=
  ci(changeRequest(X,V,El),a(requestor(V,X),C)) @ I1
  &
   ci(kr(V,impact(X,E)),a(requestor(V,X),C) @ after(I1)
  &
  role(analyser(C,X),V) @ I1
  &
  ci(withinlimit(E,El,G),a(requestor(V,X),C)) @ I2
  &
     ((ci(G=false,a(requestor(V,X),C)) @ I3
          &
          ci(ks(V,request_rejected(X)),a(requestor(V,X),C)) @ after(I3))
       v
     (ci(G=true and agreedparam(X,E3),a(requestor(V,X),C) @ I2
          &
        ci(ks(V,changeapproved(X,E3)),a(requestor(V,X),C)) @ after(I2)
          &
          cons(correctness(acceptor(L))) <--- requests(L)))
  &
  ci(kr(C,request_change(X)),a(analyser(C,X),V) @ I6
  &
  ci(impactparam(X,_),a(analyser(C,X),V)) @ after(I6)
  &
  (cons(correctness(implementor))
     v
   null).
```

```
cons(correctness(acceptor(N))) ::=
        (ci(kr(V,changed(X,Ae)),a(acceptor(V,X,Ae),C)) @ I4 <---N=[X|tail]
                &
                role(implementor(C,X,_),V) @ I4
                &
                ci(iscorrect(X,U),a(acceptor(V,X,Ae),C)) @ I5
                &
                   ((ci(U=true,a(acceptor(V,X,Ae),C)) @ after(I5)
                        &
                     ci(ks(V,verifiedCorrect(X)),a(acceptor(V,X,Ae),C)) @ after(I5))
                     v
                     (ci(U=false and reworkRequest(X,_,true,R),a(acceptor(V,X,Ae),C)) @ I6

                        &
                        ci(ks(V,change_scraped(X)),a(acceptor(V,X,Ae),C)) @ after(I6)))
                        &
                        cons(correctness(acceptor(Tail))))
        v
        null <--- N=[].


cons(correctness(implementor)) ::=
          cp(actualparam(X,_),a(implementor(C,X,_),V)) @ I7
          v
          cp(reworkparam(J,_),a(implementor(C,J,_),V)) @ I7 <---(requests(L) & member(J,L)).
```

6.3.7 Evaluation and Discussion

Below we list four of the possible alternatives with a possible test case for each of the alternatives. Efforts are stated in units of person days(pd). The knowledge content of the agents is given after the scenario alternative description.

Alternative 1 : The impact analysis is within the effort limit. The implementation does not comply to the requirement in the first iteration but meets the requirement in the second iteration with the total effort well within the limit.
TEST CASE: The requester has an effort limit of 300pd, the impact analysis suggests an effort of 200pd. Actual effort utilised in the first iteration is 250pd. The correct change is delivered in the second iteration with an additional effort of 25pd.

Alternative 2 : The impact analysis is within the established effort limit and the change is verified to be correct on first delivery of the implementation.
TEST CASE: A rather too ideal test case with the budgeted, estimated and agreed effort of 200 pd is considered.

Alternative 3 : The impact analysis exceeds budgeted effort suggesting that the change cannot be effected within the set effort limit.
TEST CASE: The test case uses a budgeted effort of 200pd while the estimated effort after impact analysis is stated at 250pd.

Alternative 4 : The impact analysis is within the effort limit. However compliance to the requirement is not met even after the allocated effort is exhausted, necessitating the change implementation to be abandoned.
TEST CASE: The effort limit is set at 400pd while the impact analysis reveals an effort of 300pd, the work is started with a negotiated effort of 280 pd but the actual effort consumed in the first iteration is at 300pd. A rework raises the actual effort spent to 450 pd causing the project to be scraped.

The knowledge specifications for scenarios described above is given below

knowledge common to all the scenarios

```
known(c,(reworkRequest(Cr,Ae,true,Rcr) ← changeRequest(Cr,_,El) and Ae=<El and name(Cr,K) and name(r,D) and append(D,K,L)
and name(Rcr,L))).
known(c,(reworkRequest(Cr,Ae,false,_) ← changeRequest(Cr,_,El) and Ae>El)).
known(c,(withinlimit(Effort,EffortLimit,true) ← Effort=<EffortLimit)).
known(c,(withinlimit(Effort,EffortLimit,false) ← Effort>EffortLimit)).
```

knowledge for **scenario 1** - changerequest cr1

```
known(c,changeRequest(cr1,v,300)).
known(c,iscorrect(cr1,false)).
known(c,iscorrect(rcr1,true)).
known(c,agreedparam(cr1,180)).
known(v,impactparam(cr1,200)).
known(v,actualparam(cr1,250)).
known(v,reworkparam(rcr1,275)).
```

knowledge for **scenario 2** - changerequest cr2

known(c,changeRequest(cr2,v,200)).

known(c,iscorrect(cr2,true)).

known(v,impactparam(cr2,200)).

known(c,agreedparam(cr2,200)).

known(v,actualparam(cr2,200)).

knowledge for **scenario 3** - changerequest cr3

known(c,changeRequest(cr3,v,200)).

known(v,impactparam(cr3,250)).

knowledge for **scenario 4** - changerequest cr4

known(c,changeRequest(cr4,v,400)).

known(c,iscorrect(cr4,false)).

known(c,iscorrect(rcr4,false)).

known(v,impactparam(cr4,300)).

known(c,agreedparam(cr4,280)).

known(v,actualparam(cr4,300)).

known(v,reworkparam(rcr4,450)).

The verifier correctly identified the satisfaction of properties. All the above correct scenarios show the satisfaction of the termination, safety, liveness and correctness properties except for the scenario in Alternative 4 where the safety property is correctly determined as violated.

The scenarios for the faulty protocol requirements are listed below
Alternative 1 : The change is approved even though the effort estimate after the impact analysis exceeds the budgeted effort limit.
TEST CASE : An effort limit of 200pd is set while the estimated effort turns out to be 250pd. The other data items required for the test are set such that the change is verified correct and the process terminates in the first iteration with an actual effort of 300pd.

The change in the protocol specification is brought about by removing the constraint checking the output of the *withinlimit* knowledge acquisition on the *changeapproved* message.
The verifier correctly identifies the violation of the safety and the correctness properties while correctly detecting the satisfaction of the liveness and termination property. The violation of the safety property is because of the actual effort of 300pd exceeding the effort limit of 200pd. The correctness property is violated because of the absence of the constraint on *changeapproved* message send event.

Alternative 2 : Acceptor fails to respond after receiving the *changed* response from the implementor.
TEST CASE : To simulate this case we use the scenarios with change request number *cr2* from the correct protocol implementation and change the knowledge possessed by the acceptor agent about the correctness of the implementation that is the *iscorrect* constraint, such that it is neither is true nor false. This prevents any further message from the acceptor and ends the role of the acceptor while the implementor is left waiting for a response to its *changed* message.
All the properties are indicated as being violated. This is because the termination of the implementor cannot be detected, all other properties which use ci specification for the implementor fail.

Alternative 3 : requestor sends *changeapproved* as well as the *request_rejected* messages.
TEST CASE : In order to bring about the above stated effect, the protocol specification of the requestor is changed as below such that it ignores the outcome of the *withinlimit* processing and sends both the *request_rejected* and *changeapproved* messages.

```
a(requestor(V,X),C) ::=
    request_change(X)  => a(analyser(C,X),V) <-- changeRequest(X,V,EffortLimit)
    then
    withinlimit(Effort,EffortLimit,G) <-- impact(X,Effort) <= a(analyser(C,X),V)
       then
       (request_rejected(X) => a(analyser(C,X),V)
       then
       (changeapproved(X,Effort3) => a(analyser(C,X),V)
           then
           a(acceptor(V,X,ActEffort),C))).
```

Faulty protocol specification for requestor agent

Since the analyser receives the *request_rejected* message first, it terminates. However the acceptor is kept alive waiting for a response to the *changeapproved* message from the analyser which it will never receive because the analyser has already terminated. This causes a failure of the termination property. Since all other property specifications are in terms of ci constraints, they too are not satisfied.

Alternative 4 : The acceptor does not verify the correctness of the received changes and responds *verfiedCorrect*.
TEST CASE : The reaction constraint *iscorrect* is eliminated from the protocol specification for the acceptor role. The verifier correctly detects the violation of the correctness property while the safety, liveness, and termination properties are satisfied.

Alternative 5: The analyser agent fails to perform the impact analysis but sends the *impact* message.
TEST CASE: Any of the correct protocol knowledge specifications can be used for this case. The protocol is changed such that the analyser reports an impact without performing the impact analysis. The part of the protocol for the role of the analyser change bringing in this effect is given below where the analyser sends an impact effort of 215pd without performing impact analysis.

$$impact(X,Effort) \Rightarrow a(requestor(V,X),C) \leftarrow impactparam(X,Effort) \text{ is changed to}$$
$$impact(X,215) \Rightarrow a(requestor(V,X),C)$$

The verifier correctly identified the violation of the correctness property while the safety, liveness and termination properties were satisfied.

6.3.8 Conclusion

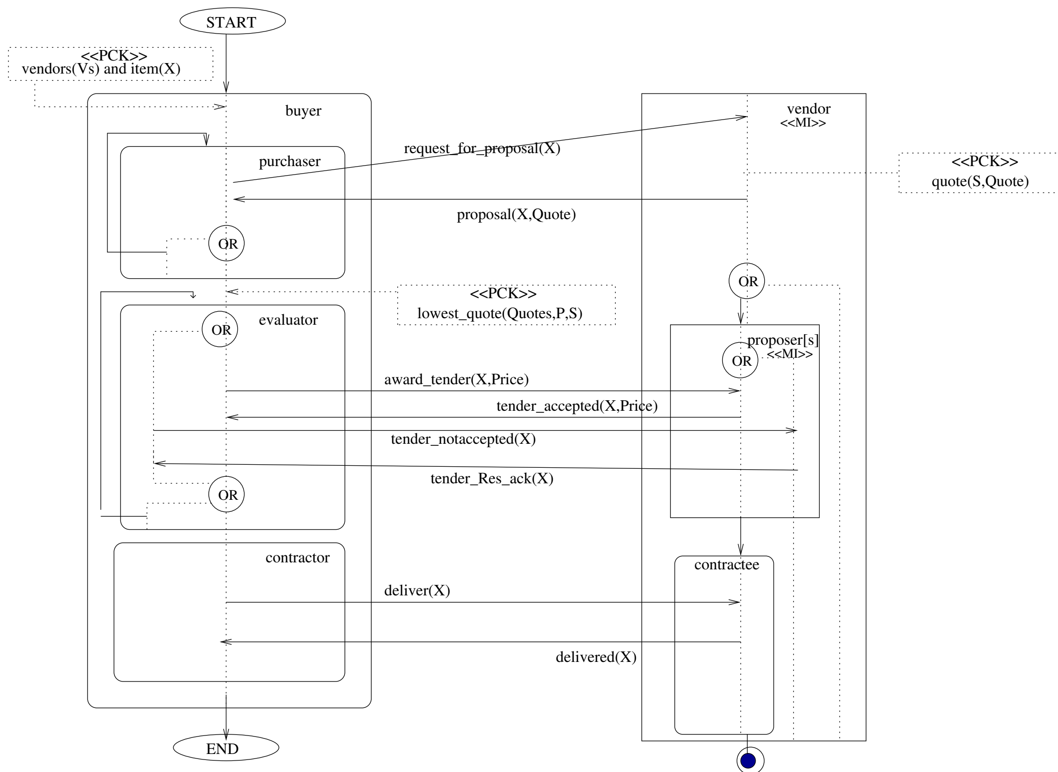
With this business process we have demonstrated the use of process loops or repeated invocation of a part of the business process. Such invocations are prone to live-lock conditions. An example of live-lock could be case with the *changed-change_rejected* loop in our case. Though while stating the properties we have not explicitly mentioned about live-lock, the liveness property takes care of detecting the live-lock conditions. The simulation creates a new instance of an agent in acceptor and implementor roles for each interaction. If we have to correspond an individual in actual deployment of this business process with an entity in the BPM, we need to associate role with the individual. The time span during which the role remains activated will be from the instantiation of the first instance of the role to the termination of the last instance of the role. As the complexity of the constraint definition in terms of alternatives or disjunctions increases the time taken for the verification increases tremendously. Hence in some cases the constraint specification had to be split up to verify the satisfaction part by part.

6.4 Tender invitation and award

6.4.1 Process Description

"Company ABC wishes to outsource its software migration project. It has already shortlisted five consultancy firms for the purpose. The request for proposal (RFP) is sent out to all the five firms and their proposals are evaluated. The company is convinced with the technical competence of the shortlisted firms and the sole criteria for the award of tender would be the cost factor. Hence the lowest bidder bags the contract."

6.4.2 PCMDL



Tender invitation and award process model

In the emanating discussion we use the term *vendor* to refer to the consultancy firms.

6.4.3 Protocol Specification

As identified in the PCMDL, the scenario requires two main roles, that of a *buyer* representing the company ABC in the process description and *vendor* representing the various shortlisted companies. This model requires a one to many relationship between the buyer and several vendors. In particular the one-many relationship is used with the purchaser-vendor and the evaluator-proposer role interactions. The simulation is initiated by an agent performing the role of a *buyer*. During the course of the simulation the buyer transforms its role from *purchaser* to *evaluator* and then to a *contractor*.

a(buyer,B) ::=
a(purchaser(X,Vs,Qs),B) ← vendors(Vs) and item(X) then
a(evaluator(Vs,S,X,Price), B) ← lowest_quote(Qs,Price,S) then
a(contractor(S,X,Price), B).

In the role of the buyer, the agent should have the knowledge of the list of vendors(Vs) to which it should send the RFP and also the details of the it_migration service represented as X for which the tenders are to be invited. This knowledge is a constraint on the change of role from *buyer* to *purchaser*. The purchaser role iterates over the list of vendors sending the *request_for_proposal* message to each of the vendors and collecting the *proposal* with quote sent by each of the vendors.

a(purchaser(X,Vs,Qs), B) ::=
(request_for_proposal(X) ⇒ a(vendor(B,X),S) ← Vs=[S | R] then
proposal(X,Quote) ⇐ a(vendor(B,X),S) then
a(purchaser(X,R,Qp),B) ← Qs=[q(S,Quote) | Qp]) or
null ← Vs=[] and Qs=[].

The quotes gathered in the role of the purchaser is used by the buyer to determine the lowest quote and the identity of the vendor who proposed the lowest quote. These are passed to the evaluator and constitute the knowledge constraint *lowest_quote* imposed on the role change from *purchaser* to *evaluator*. In the role of the evaluator the mes-

sage *award_tender* is sent to the proposer with the lowest quote and *tender_notaccepted* message is sent to the remaining proposers. The role of the evaluator is completed after it has received acknowledgements from all the proposers. The proposer who received the award_tender message acknowledges with the *tender_accepted* message while the other proposers respond with the tender result acknowledged (*tender_res_ack*) message. Role evaluator helps in iterating through the list of vendors and sending appropriate messages to each of the vendors.

a(evaluator(Vs,S,X,Price), B) ::=
(((award_tender(X,Price) $\Rightarrow$ *a(proposer(B,X),S1)* $\leftarrow$ *Vs=[S1|R] and S1=S then*
tender_accepted(X,Price) $\Leftarrow$ *a(proposer(B,X),S1)) or*
(tender_notaccepted(X) $\Rightarrow$ *a(proposer(B,X),S1)* $\leftarrow$ *Vs=[S1|R] and S1\=S then*
tender_res_ack(X) $\Leftarrow$ *a(proposer(B,X),S1))) then*
a(evaluator(R,S,X,Price),B)) or
null $\leftarrow$ *Vs=[].*

In the role of the contractor the buyer requests for the delivery of the system and the role completes on receipt of the delivery from the chosen vendor.

a(contractor(S,X,Price), B) ::=
deliver(X,Price) $\Rightarrow$ *a(contractee(B,X,Price), S) then*
delivered(X,Price) $\Leftarrow$ *a(contractee(B,X,Price), S).*

We now give the specification for the vendor role. There would be two different interactions depending on whether the vendor is successful or not. All the vendors will take on the role of the *proposer* after responding to the RFP and only the successful vendor will further take on the role of the *contractee*. Each of the vendors should have the knowledge of the quote that it would propose and this acts as a constraint on the sending of the *proposal* message. Further, if the vendor does not receive a RFP, there is no transition in the role from the vendor to the proposer. This is indicated by the use of null in the specification.

a(vendor(B,X),S) ::=
((request_for_proposal(X) ⇐ a(purchaser(X,Vs,Qs), B) then
proposal(X,Quote) ⇒ a(purchaser(X,Vs,Qs), B) ← quote(S,Quote) then
a(proposer(B,X),S)) v null).

In the role of the proposer, the vendor waits for the receipt of the tender evaluation result which could be either *award_tender* or *tender_notaccepted* and will respond accordingly. The role change for the successful vendor from proposer to contractee will be initiated in this part of the specification.

a(proposer(B,X),S1) ::=
(tender_notaccepted(X) ⇐ a(evaluator(S1,S,X,Price),B) then
tender_res_ack(X) ⇒ a(evaluator(S1,S,X,Price),B))
or
(award_tender(X,Price) ⇐ a(evaluator(S1,S,X,Price),B) then
tender_accepted(X,Price) ⇒ a(evaluator(S1,S,X,Price),B) then
a(contractee(B,X,Price), S)).

As a contractee, the vendor needs to complete the delivery of the item before completion of its role.

a(contractee(B,X,Price),S) ::=
deliver(X,Price) ⇐ a(contractor(S,X,Price), B) then
delivered(X,Price) ⇒ a(contractor(S,X,Price), B).

6.4.4 Assumptions

- All the shortlisted vendors respond to the RFP.
- There is no concept of deadline and the company ABC waits until it has received the proposals from all the vendors before it can evaluate the proposals.
- Instead of using broadcast of the RFP message, a sequential iteration through the

vendor list is used.

- The communication channel is loss less. Therefore the messages sent are always received by the intended recipients.

6.4.5 Properties for verification

Termination : Requires all the roles for the buyer and the vendor to terminate implying the termination of the purchaser, evaluator, contractor, proposer and contractee roles to terminate.

$$\textit{satisfied(termination)} \longleftarrow \diamond(\textit{end(buyer)} \wedge \textit{end(purchaser)} \wedge \textit{end(evaluator)} \wedge \textit{end(contractor)} \wedge \textit{end(proposer(X))} \wedge \textit{end(contractee)} \wedge \textit{end(vendor(X)))} \wedge X \in \textit{vendors(Vs)}.$$

Safety : For this scenario we have two safety properties.
Safety1 Award of tender should be to only one of the shortlisted candidate firms.

$$\textit{satisfied(safety1)} \longleftarrow \textit{send(award(it_migration),X)} \wedge \textit{proposer(X)} \wedge \neg(\textit{send(award(it_migration),Y)} \wedge \textit{proposer(Y)} \wedge X \backslash = Y)$$

Safety2 If a firm is awarded the contract then tender-not-accepted notification should not be sent to the same firm.

$$\textit{satisfied(safety2)} \longleftarrow \textit{send(award_tender(it_migration),X)} \wedge X \in \textit{vendors(Vs)} \longrightarrow \neg \textit{send(tender_notaccepted(it_migration),X)}$$

Liveness : The tender should eventually be awarded to some proposer.

$$\textit{satisfied(liveness)} \longleftarrow \diamond\ \textit{send(award(it_migration),X)} \wedge \textit{proposer(X)}.$$

Correctness : The requirements comprising the migration activities and the short list of the consultancy firms should be available, request for proposals should be sent to all the shortlisted vendors. On receiving the proposals from the shortlisted vendors, evaluation of the proposals should be done. The contract should be awarded to the lowest bidder and notifications sent to all the other contenders.

$$\begin{aligned}
&\textit{satisfied(correctness)} \longleftarrow \textit{cp(vendors(Vs) and item(M),purchaser)} \\
&\quad \wedge (\forall Z \in \textit{vendors(Vs)} \longrightarrow (\textit{send(request_for_proposal(M),Z)} \wedge \textit{receive(proposal(M,Quote),Z))} \\
&\quad \wedge \exists Y \in \textit{vendors(Vs)} \wedge \textit{send(award_tender(X,Price),Y)} \wedge \textit{lowest_quote(Quotes,Price,Y)} \\
&\quad \wedge (\forall K \in \textit{vendors(Vs)} \wedge K \neq Y \longrightarrow \textit{send(tender_notaccepted(M),K))).}
\end{aligned}$$

6.4.6 Constraint Specification

Having described the properties that we are interested in verifying in earlier section 6.4.5 we now give the constraint specification of the termination, safety, liveness and correctness properties. As in previous simulations, satisfaction of termination property expects all roles to terminate. We might provide a lenient version of the specification which checks for the closure of the main processes i.e, the buyer and the five vendor roles. The specification listed in figure 6.7 is that of a stricter version which ensures the closure of even the purchaser and evaluator iteration roles along with all the roles named in the protocol specification. Part of the specification *cons(termbuyerroles(Vs,X))* iterates over the buyer subroles while *cons(termvendor(Vs))* iterates over the list of vendors in the role of the proposers. The *award_tender* message should be sent to no more than one vendor.

The specification for the safety1 property in the figure 6.8 checks that the number of vendors to which the award_tender message is sent is exactly one. To do this it uses the *cons(chkaward(Vs,As))* specification which iterates through the list of vendors collecting the identities of the vendors which receive the *award_tender message*. The safety property2 ensures that the *award_tender* and *tender_notaccepted* messages are not sent to the same vendor in the same run of the simulation. It additionally checks that either of the two messages is sent.

The liveness property requiring the tender to be awarded to one of the listed vendors is shown in figure 6.9. The specification checks that the recipient of the *award_tender* message is in the role of a vendor. This property may seem rather ideal because not always will a tendering process come to a perfect termination. There might be cases where none of the proposals met the internal constraint of functional correctness or cost. However the treatment of such cases would require elaboration of the model or

```
cons(termination) ::=
        end(a(buyer,B)) @ _   &
        cons(termbuyerroles(Vs,X)) <--- vendors(Vs)  & item(X)&
        end(a(contractor(S,X,_),B)) @ _   &
        end(a(contractee(B,X,_),S)) @ _    &
        cons(termvendor(Vs)).

cons(termbuyerroles(Vs,X)) ::=
      (Vs  = [_ | R]   &
         end(a(purchaser(X,Vs,_),B)) @ _
           &
         end(a(evaluator(Vs,S,X,Price),B)) @ _
           &
         cons(termbuyerroles(R,X)))
       v
      null <--- Vs=[].

 cons(termvendor(Vs)) ::=
       (end(a(vendor(B,X),A)) @ _ <--- Vs=[A|T]
            &
          end(a(proposer(B,X),A)) @ _
            &
          cons(termvendor(T)))
        v
       null <--- Vs=[]
```

Figure 6.7: Constraint specification for termination property

separate model to be created to handle every alternate scenario.

The correctness property encompasses the correctness of every aspect of whole scenario interaction. Since we assume the channel to be loss-less we check the correctness from the perspective of the buyer agent. In the specification given in figure 6.10, *cons(correctness)* is the main specification which calls upon *cons(purchaser(Vs,X,Qus)* and *cons(eval(Vs,Qs,S))* constraint rules. As stated earlier, the purchaser and evaluator roles are involved in one to many association with the vendor roles. Hence the specification cons(purchaser(Vs,X,Qus) is used to check the satisfaction of the responsibilities of the buyer in the role of the purchaser. It iterates through the list of vendors supplied as input to check if all the vendors were sent the RFP and if the proposals were received from all the vendors. Similarly, the part of the specification in cons(eval(Vs,Qs,S)) iterates to check the appropriate result notification to each of the vendors. Finally the contractor-contractee interaction is checked. In addition we can also check the plausibility of the bidder with lowest quote receiving the award of tender. Such checks are very important while modelling business processes. Dynamic

```
cons(safety1) ::=
     cons(chkaward(Vs,As)) <--- vendors(Vs).
     & As=[H]

cons(chkaward(Vs,As)) ::=
      (((ci(ks(P,award_tender(X,Price)),a(evaluator(Vs,S,X,Price), B)) @ _<--- Vs=[P | R]
                &
           cons(chkaward(R,Ap) <--- As=[P|Ap])
         v
         (not(ci(ks(P,award_tender(X,Price)),a(evaluator(Vs,S,X,Price),B)) @ _) <--- Vs=[P | R]
                &
           cons(chkaward(R,As))))
      v
      null <--- Vs=[] & As=[]).

cons(safety2) ::=
   (not(ci(ks(S,award_tender(X,Price)),a(buyer,B)) @ _ &
         &
       ci(ks(S,tender_notaccepted(X)),a(buyer,B)) @ _ ))
   &
   ((ci(ks(S,award_tender(X,Price)),a(buyer,B))  @ _
         v
       ci(ks(S,tender_notaccepted(X)),a(buyer,B)) @ _ )).
```

Figure 6.8: Constraint specification for the safety1 and safety2 properties

specification can be used to supply the verifier with information on ways to perform the calculation of the lowest quote. As an example, suppose the company ABC has a deal with one of the listed vendors to acquire services for a subsidised rate. This can form a part of the lowest quote calculation in the dynamic constraint specification. Such a constraint specification would allow to cross check the calculations.

6.4.7 Evaluation and Discussion

The instantiations of the agents is done by specifying to the simulator the list *[a(buyer,b),(5,vendor(_,_))]* where (5,vendor(_,_)) indicates 5 instances in the role of the vendor needs to be instantiated. For the input where the protocol is correct and no errors have been introduced into the protocol specification, we have an output wherein all the properties have been satisfied. With very few conditional constraints in the protocol, we give a single scenario for the correct version of the protocol with the knowledge content of the agents as specified below where *b* and *ax*, x an identifying

```
cons(liveness) ::=
        ci(ks(S,award_tender(X,Price)),a(buyer,B)) @ _
            &
        role(vendor(B,X),S) @ _
            &
        ci(vendors(Vs) and item(X),a(buyer,B)) @ _
            &
        member(S,Vs).
```

Figure 6.9: Constraint specification for the liveness property

number, correspond to the buyer and the vendor instances. The knowledge specification also includes the method for calculation of the lowest quote.

```
known(b,item(it_migration)).
known(b,vendors([a1,a2,a3,a4,a5])).
known(b,(lowest_quote(Qs,Q,S) ← busort(Qs,Lk) and Lk=[q(S,Q)|R])).
known(b,(busort(L,Ls) ← append(X,[q(A,Q1),q(K,Q2)|R],L)
and Q2<Q1 and append(X,[q(K,Q2),q(A,Q1)|R],M) and busort(M,Ls))).
known(b,(busort(L,L))).
known(a1,quote(a1,500)).
known(a2,quote(a2,200)).
known(a3,quote(a3,100)).
known(a4,quote(a4,300)).
known(a5,quote(a5,400)).
```

The verifier correctly identifies the satisfaction of the safety, liveness, termination and correctness properties of the model simulation. The scenarios for the faulty run of the protocol and the corresponding verifier output are described below. The simulation is performed with vendors $\{a1,a2,a3,a4,a5\}$.

Scenario1 : We simulate a faulty protocol implementation where the RFP is not sent to one of the listed vendors a3. The protocol specifications for the purchaser and the evaluator roles are changed in such a way that these roles interact with all vendors except vendor instance a3. The verifier detects the satisfaction of the safety(1 and 2) and liveness properties and, the violation of termination and correctness properties. The violation of the termination property is because the verifier checks for the termination

```
cons(correctness) ::=
        cp(vendors(Vs) and item(X), a(purchaser(X,_,Qs),B)) @ _
              &
        cons(purchaser(Vs,X,Qus))
              &
        cons(eval(Vs,Qs,S)) <---(lowest_quote(Qus,Q,S))
              &
        ci(ks(S,deliver(X,Price)),a(contractor(S,X,Price),B)) @ _
              &
        ci(kr(S,delivered(X,Price)),a(contractor(S,X,Price),B)) @ _.

cons(purchaser(Vs,X,Qs)) ::=
      (ci(ks(S,request_for_proposal(X)),a(purchaser(X,Vs,Qs),B))  @ _ <--- Vs=[S |R]
            &
          ci(kr(S,proposal(X,Quote)),a(purchaser(X,Vs,Qs),B)) @ _
            &
          cons(purchaser(R,X,Qus)))
        v
      null <--- Vs=[].

cons(eval(Vs,Qs,S)) ::=
       (Vs=[S1|R]
        &
           ((ci(ks(S,award_tender(X,Price)),a(evaluator(Vs,S,X,Price),B)) @ _
                     &
                 ci(kr(S,tender_accepted(X,Price)),a(evaluator(Vs,S,X,Price),B)) @ _)
             v
            (ci(ks(S1,tender_notaccepted(X)),a(evaluator(Vs,S,X,Price),B)) @ _
                     &
                 ci(kr(S1,tender_res_ack(X)),a(evaluator(Vs,S,X,Price),B)) @ _))
       &
       cons(eval(R,Qs,S)))
     v
      null<---Vs=[].
```

Figure 6.10: Constraint specification for the Correctness property

of vendor a3 in the role of the proposer. Since the vendor a3 has not received any RFP, it remains in the role of the vendor throughout the interaction. The correctness property is detected as violated as the RFP message and the tender result notification messages are not sent to vendor a3 and there was no proposal received from a3.

Scenario2 : As the next test case scenario we consider the lowest quote calculation to be flawed where the tender is awarded to the vendor with the highest bid. To simulate this effect the knowledge for lowest_quote in the protocol specification is changed to give the highest quote and the vendor with the highest bid which in our case is a1.

The verifier correctly detected the violation of the correctness property and showed the satisfaction of all the other properties.

Scenario3 : The vendor with the lowest quote identified in the process is ignored and the tender is awarded to a different vendor who is also one among the listed vendors. The protocol specification is changed so that the evaluator and the contractor roles are invoked with a2 as the successful bidder without taking into consideration the outcome of the lowest_quote constraint. However the constraint specification had the a3, the bidder with the lowest bid as being successful. As for the verifier output the safety(1 and 2), liveness and termination were detected as satisfied while the correctness property was not as it is only the correctness property that has the constraint check for the lowest quote.

Scenario 4: The business process terminates without the tender being awarded to any vendor. The desired fault for test is induced by invoking the evaluator with a non-existent vendor a10 as the selected vendor and conditionally invoking the role of the contractor with the condition that a10 is in the vendors set {a1,a2,a3,a4,a5}. This would cause the protocol to terminate without the transition of the buyer to the role of the contractor. The safety1, termination, liveness,correctness properties are violated. Only the safety2 property is satisfied. The termination property does not detect the closure of the contractor and the contractee roles and hence is detected as violated. All other property violations are because of the check for the *award_tender* message in their constraint specification.

Scenario 5: The notification of unsuccessful bid is not sent to any of the other vendors. The protocol of the evaluator is changed as below to introduce the above faulty test scenario.

a(evaluator(Vs,S,X,Price), B) ::=
    (((award_tender(X,Price) $\Rightarrow$ *a(proposer(B,X),S1)* $\leftarrow$ *Vs=[S1|R] and S1=S then*
      tender_accepted(X,Price) $\Leftarrow$ *a(proposer(B,X),S1)) then*
      a(evaluator(R,S,X,Price),B)) or
    (a(evaluator(R,S,X,Price),B) $\leftarrow$ *Vs=[S1|R] and S1\=S))*
    or

null ← *Vs=[ ].*

The Safety(1 and 2) and liveness properties are satisfied. Termination and correctness properties are identified as violated. The vendors in the roles of proposers wait for a response form the evaluator after having sent the bid and do not terminate. The absence of the notification message in the interaction causes the violation of the correctness property.

Scenario 6: All the proposers incorrectly receive the award of tender whether successful in their bid or otherwise. The desired effect is induced by changing the proaction constraint on the *award_tender* message from '*Vs=[S1|R] and S1=S*' to '*Vs=[S1|R]*'. The safety1,termination and correctness properties are not satisfied while safety2,liveness properties are satisfied. Since the number of vendors receiving the *award_tender* message is more than one, the safety1 property is violated. All the proposers change their roles to that of a contractee expecting to receive the deliver message but the contractor role of the buyer does not loop and sends the deliver message to only one contractee. This leaves all the proposer roles without terminating. The correctness property is violated due because it checks for the notification of the vendors other than the vendor indicated by the lowest calculation.

6.4.8 Conclusion

Owing to the non-discrete representation of time, we had to make some hypothetical assumptions that all vendors will respond to the proposals and had to implement the protocol without considering the absolute time. It might have been more realistic to have set deadline for each milestone in the tendering process like, last date for the receipt of the proposals, last date for completion of evaluation and award of tender etc.,In this business process we demonstrate the use of one-many relationship between the interacting processes in the purchaser-vendor and the evaluator-proposer interactions. However we restrict the broadcast of RFP by the purchaser to be sequential than being parallel because of our scope considerations. In this scenario, the PCMDL for the purchaser and the evaluator roles show the loops that require the iteration of these

roles. This may seem more biased towards the obvious notions of an LCC educated modeler. For a business modeler the purchaser and evaluator role iteration might not be that obvious. These might be represented as a single process which is more close to the real life deployment of the model. In such cases the translation can be enhanced by use of inference of iteration for one-many associations.

Chapter 7

Conclusion and future directions

The verifier built is intended as a management tool with simulation and verification to aid the diagnosis of a business process design in terms of satisfiability of constraints. With this project we have demonstrated the use of light weight temporal logic for determining the satisfaction of properties of business process models expressed at user level as constraints on message flow. Allowing the end-user to define properties puts the system to its best use without binding the end-user to design-time properties. As shown in the sample simulation scenarios, faulty deployments can be identified for some examples of bad practices. Ability to verify conformance to temporal ordering requirements via interaction constraint satisfaction is an effective feature in minimising inter-department communication issues in practical BPM deployments. Using an agent-based simulator with the LCC interaction framework is advantageous as it can be related very closely to the manual, automated or the future agent oriented deployments. In our system we assume monotonic constraints. We may need to use non-monotonic constraints in the world of agent deployment for BPM processes, as the agents are allowed to revise their beliefs. This however, is less of a problem than it might seem because, although the agents themselves are non-monotonic, the protocols used by the agents are monotonic. Owing to the arduous manual intervention required in the usage of the verifier to create the model, deriving the protocol and constraint specification limits its acceptability and makes it error-prone. Having an integration of the components would be a good idea.

7.1 Future Directions

In this project we have restricted to linear interactions between business processes. Evaluating and extending the verifier to cater to parallelisation techniques would unleash a wide range of interesting properties concerning efficiency of the BPM to be determined. Though the verifier supports the determination of constraint satisfaction or violation for all the possible interactions, such an attempt on even a slightly complex scenario takes a very long time. This is because the constraint solver used for this project is simply Prolog which is not optimised for constraint solving. Prolog is a general programming language. It would be a useful extension to this project to replace our Prolog constraint solver with a more sophisticated satisfiability solver or model checker. The design-simulate-verify cycle can be automated making the system user-friendly if the few issues identified in automatic translation are resolved. One of the first steps in this direction would be to have a method to translate the identified constraints in the PCMDL to a corresponding constraint specification.

Appendix A

Appendix A - Business Term Glossary

Business Process Management technology is a framework of applications that effectively tracks and orchestrates business process. Business Process Management helps to automate tasks allowing manual intervention where required. The trend favouring automation is refered to as *Straight Through Processing*.[3]. Not all business processes are simple enough for automation. To accommodate those processes which are too complex for automation, the alternative trend in Business process management that favours data-driven workflow with authorizations to skip or undo activities is called *Case Handling*. The automation is controlled by the rules which define the sequence in which the tasks are performed.

A *workflow management system* is defined as "A system that defines, creates and manages the execution of workflows through the use of software, running on one or more workflow engines, which is able to interpret the process definition, interact with workflow participants and, where required, invoke the use of IT tools and applicatiions." [3]

As per Geoffrey Sparks, "A *Business Process* is a collection of activities designed to produce a specific output for a particular customer or market ". [29]. A *Process* is an ordered sequence of activities across time and place with a beginning, end, inputs and outputs. A business process can be composed of other business processes while itself being a part of other business process. A single business process may affect more

than one organisational unit. The customer to a business process could be internal to the organisation or an external customer. The inputs to the business process could be either *Resources* or *Information*, the difference being that the resources are consumed during the processing while the information is not. Further elaborations of the Busienss Process definition by Chris Menzel is “An objective real world event, described totally as a sequence of events(activities,sub-processes) occuring over time containing certain objects having certain properties standing in certain relations” [28][6]

Jeffery Herrmann : “ A process can be decomposed into other processes. A process begins and ends at points in time. One can view a process from different perspectives that include different things. Objectives or drivers may be part of one perspective but not another: if included, they could be seen as instructions”[28][6].

Bibliography

[1] Integration definition for function modelling (idef0). http://www.idef.com/Downloads/pdf/idef0.pdf.

[2] *UML 1.5 Specification, UML Notation Guide*. Object Management Group, Inc, 250 First Ave. Suite 100, Needham, MA 02494, U.S.A.

[3] Business process management: A survey. In *Business Process Management, international conference, BPM 2003, Eindhoven, the Netherlands, June 26-27, 2003: proceeedings*. Springer, 2003. Lecture Notes in computer science, 2678.

[4] Colin Allen and Michael Hand. *Logic Primer*. Addison-Wessley, 1992.

[5] Maria Bergholtz, Prasad Jayaweera, Paul Johannesson, and Petia Wohed. Process models and business models - a unified framework. Department of Computer and System Sciences, Stockholm University and Royal Institute of Technology Forum 100, SE-164 40 Kista, Sweden.

[6] Yun-Heh Chen-Burger and David Robertson. *Automating Business Modelling, A use of logic to represent enterprise models and support reasoning*. Springer, 2004.

[7] Yun-Heh Chen-Burger, Austin Tate, and Dave Robertson. *Enterprise Modelling: A Declarative Approach for FBMPL.*

[8] Gunnar Övergaard. A formal approach to collaborations in unified modelling language. In *≪UML≫'99-The unified Modelling Language, Beyond the*

Standard,Second International Conference Fort Collins, CO, USA,October 28-30,1999 proceedings, Royal Institute of Technology, Stokholm,Sweden, 1999. Springer. Lecture Notes in Computer Science 1723.

[9] D.M.Gabbay, C.J.Hogger, and editors J.A.Robinson. *Handbook of Logic in Artificial Intelligence and Logic Programming, volume 4, Epistemic and Temporal Reasoning*. Clarendon Press, Oxford, 1995.

[10] Marc Esteva, Julian Padget, and Carles Sierra. Formalizing a language for institutions and norms. *Intelligent Agents VIII, Lecture Notes in Artificial Intelligence*, 2333:348–366, 2002.

[11] Thom Frühwirth. Temporal annotated constraint logic programming. *Journal of Symbolic Computation*, 22:555–583, 1995.

[12] Li Guo, Yun-Heh Chen-Burger, and Dave Robertson. Mapping a business process model to a semantic web services model. In *In Proceedings, The IEEE International Conference on Web Services, San Diego, California, USA*, 2004.

[13] Xudong He, Huiqun Yu, Tianjun Shi, Junhua Ding, and Yi Deng. Formally analyzing software architectural specifications using sam. *Journal of Systems and Software*, 71:11–29, 2002.

[14] David Hollingsworth. *Workflow Management Coalition, The Workflow Reference Model*. Workflow Management Coalition, Avenue Marcel Thirty 204,1200 Brussels,Belgium, 1994.

[15] Michael Jackson. *Business Process Implementation, Building workflow systems*. Pearson Education Print, 1999.

[16] Richard J.Mayer, Christopher P.Menzel, Michael K.Painter, Paula S. deWitte, Thomas Blinn, and Benjamin Perakath. Information integration for concurrent engineering (iice) idef3 process decription capture method report. http://www.idef.com/idef3.html.

[17] Hsiang-Ling Kuo. *Knowledge Management using Business Process Modelling and Workflow Techniques*. M.Sc Thesis, School of Artificial Intelligence, Division of Informatics,University of Edinburgh,Edinburgh, UK, 2002.

[18] Jintae Lee, Michael Gruninger, Yan Jin, Thomas Malone, Austin Tate, Gregg Yost, and other members of the PIF working group. *The pif interchange format and framework. The knowledge engineering review,13(1).* March 1998.

[19] Keith Mantell. From uml to bpel : Model driven architecture in a web services world. http://www-106.ibm.com/developerworks/webservices/library/ws-uml2bpel/.

[20] Richard J. Meyer, Michael K.Painter, and Madhavi Lingineni. *Information integration for concurrent engineering(IICE) toward a method for business constraint discovery(IDEF9).* Knowledge Based Systems,Inc. One KBSI Place, 1500 University Drive East, College Station TX 77840-2335, April 1995.

[21] Martyn A. Ould. *Business Processes: Modelling and analysis for Reengineering and Improvement*. John Wiley and Sons, 1995.

[22] P.D.O'Brien and W.E.Wiegand. Agent based process management : applying intelligent agents to workflow. In *The Knowledge Engineering Review Vol 13:2,1998*, pages 1–14(Draft), 1998.

[23] Peter Rittgen. Paving the road to business process automation. In *European Conference on INformation Systems (ECIS) 2000, Vienna, Austria*, pages 313–319, July 2000.

[24] David Robertson. *A Lightweight Coordination Calculus for Agent Systems*. Informatics, University of Edinburgh, 2004.

[25] David Robertson. A lightweight method for coordination of agent oriented web service. In *In Proceedings of AAAI Spring Symposium on Semantic Web Services, California, USA*, 2004.

[26] David Robertson and Jaume Augusti. *Software Blueprints, Lightweight Uses of Logic in Conceptual Modelling*. Addison-Wesley, 2003.

[27] James Rumbaugh, Ivar Jaconson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison Wesley Longman, Inc.,, 1999.

[28] Craig Schlenoff, Amy Knutila, and Steve Ray. Proceeding of the process specification language(psl) roundtable. Gaithersburg,MD, 1997. National Institute of Standards and Technology. NIS-TIR 6081.

[29] Geoffrey Sparks. *An Introduction to UML, The Business Process Model*. Enterprise Architect,, 2000.

[30] Leon Sterling and Ehud Shapiro. *The Art of Prolog*. The MIT Press, Cambridge, Massachusetts London, England, 1987.

[31] Thomas W.Malone, John Quimby, Kevin Crowston, Abraham Bernstein, Jintae Lee, George A Herman, Brian T. Pentland, Mark Klein, Chrysanthos Delarocas, Charles S.Osborn, George M.Wyner, and Elisa O'Donnell. *Tools for inventing organizations : Toward a handbook of organizational Processes*. MIT Process Handbook, 2003.

[32] W.Reisig. *Petri nets, an introduction*. EATCS, Monographs on Theoritical Computer Science, 1985.

[33] Christian Zirpins and Giacomo Piccinelli. Interaction-driven definition of e-business processes. 2002.