Design, Development and Verification of a Compensable Workflow Modeling Language

By

Fazle Rabbi

Submitted in partial fulfillment of the requirements for the degree of Masters of Science in Computer Science at Saint Francis Xavier University Antigonish, Nova Scotia January 2011

© Copyright by Fazle Rabbi, 2011

Saint Francis Xavier University
Department of
Mathematics, Statistics and Computer Science

The undersigned hereby certify that they have read a thesis entitled "**Design, Development and Verification of a Compensable Workflow Modeling Language**" by **Fazle Rabbi** in partial fulfillment of the requirements for the degree of **Masters of Science**.

Dated: ______________________________

Supervisor: ______________________________
Dr. Wendy MacCaull

Dedicated to my parents

Abstract

In recent years, Workflow Management Systems (WfMSs) have been studied and developed to provide automated support for defining and controlling various activities associated with business processes. The automated support reduces costs and overall execution time for business processes, by improving the robustness of the processes and increasing productivity and quality of service. As business organizations continue to become more dependent on computerized systems, the demand for reliability has increased. Most WfMSs provide little or no verification facilities; this causes the resulting implementation of large and complex workflow models to be at risk of undesirable runtime executions. Design validation, ensuring the correctness of the design at the earliest stage possible, is a major challenge. Model checking is a promising and powerful approach to automatic verification of systems, but model checking frequently suffers from the state explosion problem and modeling with the input language of a model checker is time consuming.

To address these issues, a compensable workflow modeling language called *CWML* is designed and developed to provide both flexibility in the design, and also reliability in the execution of a workflow system. In this research an automated translator is developed and studied which can translate a graphical workflow model and an abstract task specification (written in Java) to the modeling language of the model checker DiVinE.

To handle the state explosion problem a workflow reduction algorithm is developed and integrated into the translator. A Service Oriented Architecture (SOA) based workflow engine is designed and developed as part of the work. The effectiveness of the system has been studied by developing a workflow based on the National Principles and Norms of Practice of Canadian hospice palliative care. Finally, a sophisticated user friendly browser is discussed with which one can see records in hierarchical fashion, travel to a past record and can generate charts by selecing parameters. We show that the browser can be used as a cause and effect analysis tool, which will aid the user for root cause analysis and decision making.

Acknowledgements

I am grateful to my supervisor Professor Dr. Wendy MacCaull for her help to solidify my ideas in countless discussions. Her support and patience has been priceless during the process of writing this thesis. The Centre for Logic and Information at StFX provided a great environment to research with help from Keith Miller, Dr. Cristian Cocos, Dr. Ji Ruan, Ahmed Mashiyat, Nazia Leyla, Maxwell Graham, Igor Vecei, Mary Heather Jewers and many others. I owe special thanks to Dr. Hao Wang, who contributed substantially to the ideas in this thesis and shared his knowledge of model checking and high performance computing with me.

Additionally, I want to thank Dr. Man Lin and Dr. Laurence Yang for reading this thesis and their feedback.

Contents

1 Introduction **1**

2 Workflow systems overview **4**

2.1 What is a workflow? 4

2.2 Workflow modeling languages 5

2.3 Workflow enactment services 8

3 Compensable transactions **11**

3.1 The t-Calculus 15

3.2 The t-Calculus operators and their behavioral dependencies 17

3.2.1 Sequential composition (;) 17

3.2.2 Parallel composition ($||$) 18

3.2.3 Internal choice ($\sqcap$) 18

3.2.4 Speculative choice ($\otimes$) 19

3.2.5 Alternative forwarding ($\rightsquigarrow$) 19

3.2.6 Backward handling ($\unrhd$) 20

3.2.7 Forward handling ($\rhd$) 20

3.2.8 Programmable compensation ($\circledast$) 21

3.2.9 Associativity 22

4 The compensable workflow modeling language **28**

4.1 Compensable workflow nets 28

4.2 The compensable workflow modeling language and its Petri net representation 34

4.3 Analysis 44

5 Model checking and automated translation **48**

5.1 Model checking 48

5.1.1 The DiVinE model checker and its modeling language 50

5.2 Workflow translation to a model checker 52

5.2.1 Petri net to DVE translation 55

5.2.2 Proof of correctness 59

6 Workflow model reduction **64**

6.1 Related work 64

6.1.1 Partial order reduction 64

6.1.2 Other work 70

6.2 Workflow model reduction 70

6.3 Proof of stuttering equivalence 75

6.4 Effectiveness 87

7 Tool overview **90**

7.1 NOVA workflow 91

7.1.1 The NOVA editor 91

7.1.2 The NOVA engine 93

7.1.3 The NOVA translator 96

7.1.4 The NOVA browser 98

8 Case study **104**

8.1 Hospice palliative care 104

8.2 Verification of the palliative care process 113

9 Conclusion and future work **120**

Bibliography **123**

List of Figures

2.1 Workflow system characteristics 5
2.2 An example of a Petri net 7
2.3 Workflow reference model - components & interfaces 9

3.1 State transition diagram of a compensable transaction 12

4.1 Petri net representation of an atomic uncompensable task 29
4.2 Petri net representation of an atomic compensable task 30
4.3 Graphical representation of CWML 34
4.4 n-fold split and join tasks 36
4.5 Petri net representation of and composition 36
4.6 Petri net representation of xor composition 37
4.7 Petri net representation of or composition 37
4.8 Petri net representation of sequential composition 38
4.9 Petri net representation of internal choice composition 39
4.10 Petri net representation of alternative forward composition 40
4.11 Petri net representation of parallel composition 41
4.12 Petri net representation of speculative choice composition 43
4.13 CWF-net with one atomic task 45

4.14 CWF-net with one compensable task 46
4.15 CWF-net with more compensable tasks 47

5.1 A Petri net 56

6.1 Execution of independent transitions 67
6.2 If AP' = {p} then α is invisible 67
6.3 Two stuttering equivalent paths 68
6.4 Example of a task syntax tree 71
6.5 The workflow M_{ex} 74
6.6 The task syntax tree for M_{ex} 75
6.7 The reduced workflow M'_{ex} 76
6.8 Forming a syntax tree of size $k+1$ from one of size k 79
6.9 Sequential composition ($\bullet$) of uncompensable atomic tasks 80
6.10 Reduced syntax tree τ'_{k+1} 81
6.11 Reduced syntax tree τ'_{k+1} 83
6.12 Workflow with and composition 88

7.1 SOA based architecture of NOVA workflow 92
7.2 NOVA editor in eclipse IDE 93
7.3 NOVA engine guides the service flow 94
7.4 An example of a service class extension 94
7.5 Syntax for assigning non-deterministic data 97
7.6 DVE code for non-deterministic data 98
7.7 Hierarchical data representation in the NOVA browser 100
7.8 Example of a chart view 103

8.1 Overview of CHPCA model 105
8.2 Palliative care workflow: Overall 107
8.3 GASHA Form: Adult pain meter 107
8.4 Registration 111
8.5 Palliative care workflow: Intake 111
8.6 Palliative care workflow: Regular Assessment 112
8.7 Palliative care workflow: Team Building 114

Chapter 1

Introduction

Workflow management systems (WfMS) provide an important technology for the design of computer systems which can improve process, communication and information system development in dynamic and distributed organizations. Current Workflow Management Systems (WfMSs) facilitate the enactment of workflows with some degree of fault-tolerance, e.g., exception handling, but often provide limited formal verification capacity which is especially important in safety critical systems. For example, YAWL (Yet Another Workflow Language) [40] can verify the soundness property of workflow nets (a sub class of Petri nets) which guarantees the absence of live-locks, deadlocks, and other anomalies without domain knowledge [44]. There are several other graphical tools for modeling workflow systems (e.g., Petri nets [33], ADEPT2 [37]) but they do not provide formal verification. WSEngineer [8] and BPEL2PN [6] have recently been developed for the verification of BPEL (Business Process Execution Language) [20], but the built-in support for compensation in BPEL does not provide rich semantics of compensation compared to the *t*-Calculus [29]. Moreover, these tools lack advanced workflow reduction techniques.

In this thesis we present our new graphical workflow modeling language, the Compensable Workflow Modeling Language (CWML), with which one can model a workflow with compensation. The foundation of the CWML is based on Petri nets [33]. We incorporated rich semantics of compensation into the CWML with the help of t-Calculus [29] operators. We developed a tool named NOVA Workflow [7] to design, develop, verify and analyze compensable workflows. We detail our algorithm to translate a CWML workflow model to DVE, the input language of a model checker DiVinE [1], and give its proof of correctness. DiVinE is a parallel distributed model checker which can verify large systems. In addition to that we give our algorithm for a workflow reduction technique which pre-processes a workflow model and reduces the model in such a way that the reduced workflow model is stuttering equivalent to the original model with respect to an LTL property. The pre-processing significantly reduces the size of the state space while verifying the workflow in the DiVinE model checker. The tool was used to model and verify properties of the national model of CHPCA [16] which shows its applicability. In chapter 2, we give a brief description of existing workflow management systems along with their modeling languages. We are especially interested in graphical workflow modeling languages. Chapter 3 provides a detailed description of compensable transactions and t-Calculus operators. The internal constraints and behavioral dependencies of compensable transactions described in this chapter help clarify the concept of compensable transaction. In chapter 4 we define a new Compensable Workflow Modeling Language (CWML) and present the graphical representation of compensable tasks. Compensable tasks are based on the idea of compensable transactions and t-Calculus operators. We use Petri nets to describe the semantics of compensable tasks. In chapter 5 we give an algorithm to translate a CWML workflow to the model checker DiVinE. The proof of

correctness of the algorithm is shown in this chapter. In order to verify a large workflow by a model checker, we provide a workflow reduction algorithm in chapter 6. The proof of stuttering equivalence of the original and reduced model and its effectiveness are shown in this chapter. Chapter 7 provides a tool overview of the workflow suite, called NOVA Workflow, that we developed. NOVA Workflow consists of four components, i) the NOVA Editor, ii) the NOVA Translator, iii) the NOVA Engine and iv) the NOVA Browser. This chapter gives a description of each component. With this tool we can input a workflow modeled with CWML using the graphical editor, and an LTL_{-X} formula. The reduction, translation and model checking then all proceed automatically giving either a counter-model if the specification fails, or a statement that the model satisfies the specification. We provide a case study in chapter 8. A workflow was developed for a community based palliative care program using NOVA Workflow and a number of properties were verified. Chapter 9 summarizes our specific contributions and discusses some of our future work.

Chapter 2

Workflow systems overview

2.1 What is a workflow?

Workflow is concerned with the automation of a process, in whole or part, during which documents, information or tasks are passed from one participant to another for action (activities), according to a set of procedural rules. A participant may be a person or an automated process (computer system). Workflow can be a sequential progression of work activities or a complex set of processes each taking place concurrently, eventually impacting each other according to a set of rules, routes, and roles. A Workflow Management System is a system that completely defines, manages and executes "workflows" through the execution of software whose order of execution is driven by a computer representation of the workflow logic. At the highest level, all WfM systems may be characterised as providing support in three functional areas [19]:

- Build-time functions, concerned with defining, and possibly modelling, the workflow process and its constituent activities;

- Run-time control functions concerned with managing the workflow processes in an operational environment and sequencing the various activities to be handled as part of each process;
- Run-time interactions with human users and IT application tools for processing the various activity steps.

Fig. 2.1 illustrates the basic characteristics of WfM systems and the relationships between these main functions.

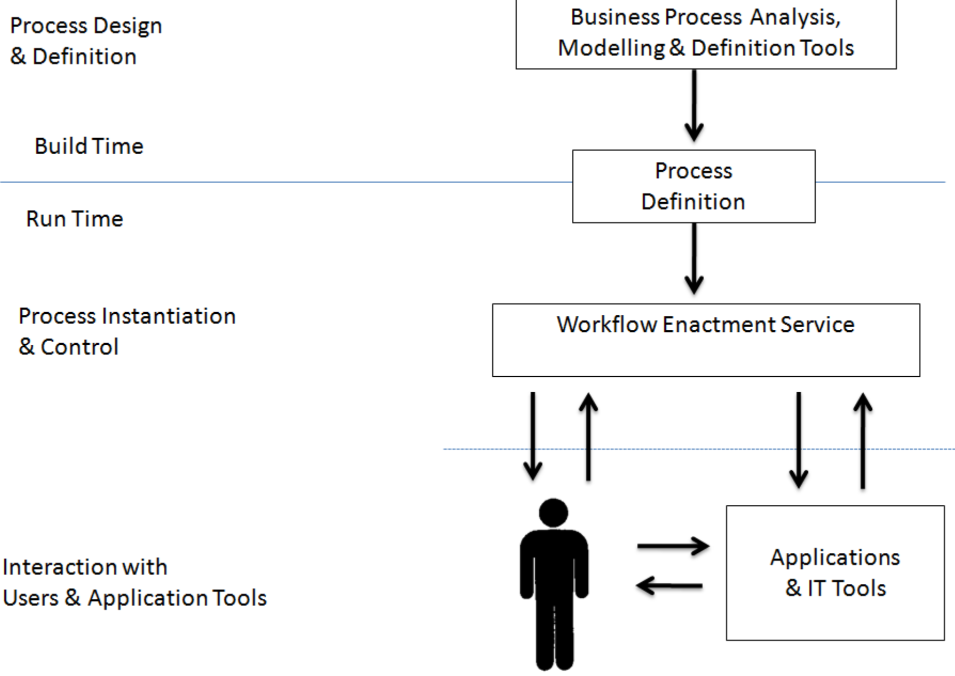

Figure 2.1: Workflow system characteristics

2.2 Workflow modeling languages

A number of graphical process-modeling languages are available to define the detailed routing and processing requirements of a typical workflow [46]. For the purpose of our

work we are interested in languages with a sound mathematical foundation such as Petri nets and Workflow nets.

Petri nets

Historically speaking, Petri nets originate from the early work of Carl Adam Petri [33]. Since then the use and study of Petri nets have increased considerably. For a review of the history of Petri nets and an extensive bibliography the reader is referred to [32].

The classical Petri net is a directed bipartite graph with two node types called places and transitions. The nodes are connected via directed arcs. Connections between two nodes of the same type are not allowed. Places are usually represented by circles and transitions are usually represented by rectangles. The mathematical definition of a Petri net is given below:

Definition 2.1. *A Petri net is a 5-tuple,* $PN = (P, T, F, W, M_0)$ *where:*

- $P = \{p_1, p_2, ..., p_m\}$ *is a finite set of places,*
- $T = \{t_1, t_2, ..., t_n\}$ *is a finite set of transitions,*
- $F \subseteq (P \times T) \cup (T \times P)$ *is a set of arcs (flow relation),*
- W: $F \rightarrow \{1,2,3,...\}$ *is a weight function,*
- M_0: $P \rightarrow \{0,1,2,3,...\}$ *is the initial marking,*
- $P \cap T = \phi$ *and* $P \cup T \neq \phi$.

A marking of a Petri net is a multiset of its places, i.e., a mapping $M : P \rightarrow \mathbb{N}$. *We say the marking assigns to each place a number of tokens.*

A 4-tuple $N = (P, T, F, W)$ is called a Petri net structure (no specific initial marking)

Places may contain tokens and the distribution of tokens among the places of a Petri net determine its *state* (or *marking*). Fig. 2.2 shows an example of a Petri net where $P1, P2, P3, P4$ are places, $t1$ and $t2$ are transitions and the dots represent the tokens of places $P1, P3$ and $P4$.

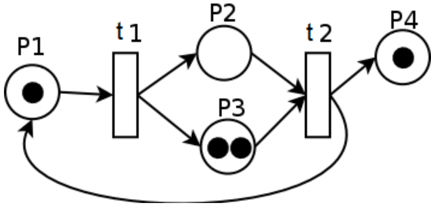

Figure 2.2: An example of a Petri net

Workflow nets

Workflow nets, based on the characteristics of Petri nets, is a powerful and flexible language to model control flows [41].

Definition 2.2. *A Petri net structure $N = (P, T, F, W)$ is called a workflow net (WF-net) if and only if:*

- *N has one source place i, called the initial place.*
- *N has one sink place f, called the final place.*
- *for every node $n \in P \cup T$, there exists a path from i to n and a path from n to f.*

Places in the set P correspond to conditions, transitions in the set T correspond to tasks. Tokens in a WF-net represent the *workflow state* of a single instance of a

workflow execution. One of the advantages of using Petri nets for workflow modeling is the availability of many Petri net based analysis techniques [42].

Other modeling languages

The Business Process Modeling Language (BPML) [9] is an XML based markup language designed to model business processes deployed over the Internet. The BPML specification provides an abstract model and XML syntax for expressing executable business processes and supporting entities. BPML specifies transactions, data flow, messages and scheduled events, business rules, security roles, and exceptions. It supports both synchronous and asynchronous distributed transactions.

2.3 Workflow enactment services

A workflow enactment service provides the run-time environment in which process instantiation and activation occurs. Fig. 2.3 illustrates the major components and interfaces within the workflow reference model [19].

Definition 2.3. *Workflow Enactment Service is a software service that may consist of one or more workflow engines in order to create, manage and execute workflow instances. Applications may interface with this service via a workflow application programming interface (API).*

Definition 2.4. *A Workflow Engine is a software service or "engine" that provides the run time execution environment for a workflow instance.*

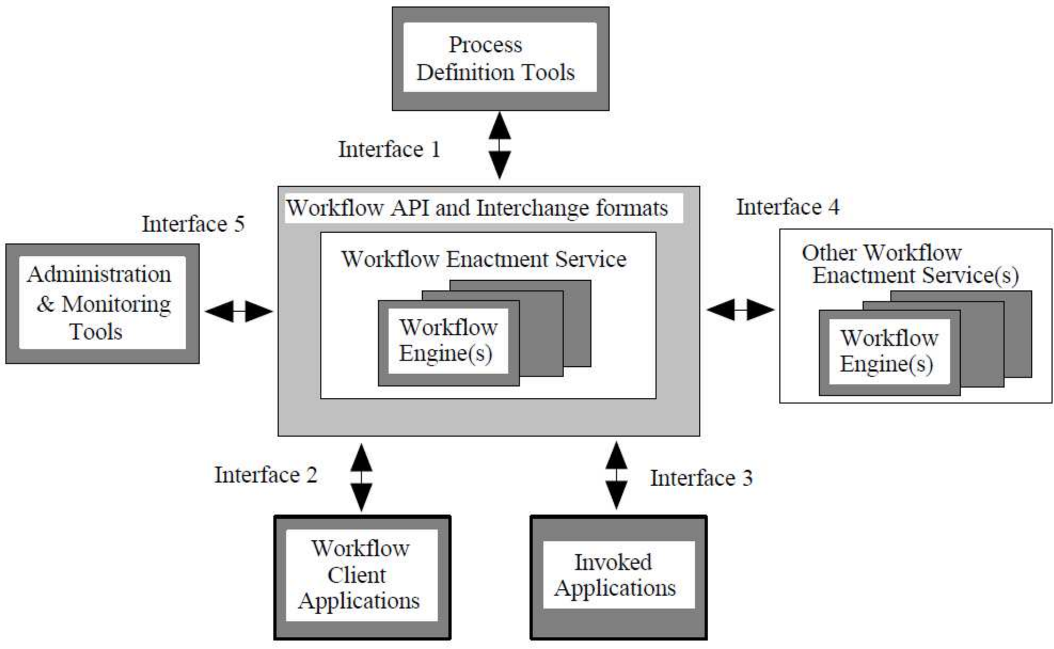

Figure 2.3: Workflow reference model - components & interfaces

Interaction with external resources accessible to the particular enactment service occurs via one of two interfaces:

- The client application interface, through which a workflow engine interacts with a worklist handler, responsible for organising work on behalf of a user resource. It is the responsibility of the worklist handler to select and progress individual work items from the work list. Activation of application tools may be under the control of the worklist handler or the end-user.

- The invoked application interface, which enables the workflow engine to directly activate a specific tool to undertake a particular activity. This would typically be a server-based application with no user interface; where a particular activity uses a tool which requires end-user interaction. The server based application would normally be invoked via the worklist interface to provide more flexibility for user task

scheduling. By using a standard interface for tool invocation, future application tools may be workflow enabled in a standardised manner.

Chapter 3

Compensable transactions

A traditional system which consists of ACID (Atomic, Consistent, Isolated, Durable) transactions cannot handle *long lived transactions* as it has only a flow in one direction. A long lived transaction system is composed of sub-transactions and therefore has a greater chance of partial effects remaining in the system in the presence of some failure. These partial effects make traditional rollback operations infeasible or undesirable. A transaction is called a *compensable transaction*, when its effects can be semantically removed by some compensating actions [26]. A compensable transaction has two flows: a forward flow and a compensation flow. The forward flow executes the normal business logic according to the system requirements, while the compensation flow removes all partial effects by acting as a backward recovery mechanism in the presence of some failure.

The concept of a compensable transaction was first proposed by Garcia-Molina and Salem [17], who called this type of long-lived transactions, a *saga*. A saga can be broken into a collection of sub-transactions that can be interleaved in some way with other sub-transactions. This allows sub-transactions to *commit* prior to the completion

of the whole saga. Here commit refers to the idea of making a set of tentative changes permanent. To make sure that the system is consistent while performing any transaction, it needs to lock system resources (e.g., database tables, files, etc.). If a system resource is locked for a long time, it might increase the chance of deadlock. Dividing a long-lived transactions into sub-transactions (possibly short-lived transactions) releases resources earlier and reduces the possiblity of deadlock. If the system needs to rollback in case of some failure, each sub-transaction executes an associated compensation to semantically undo the committed effects of its own committed transaction.

A compensable transaction may be described by its external state. In [27] we find there is a finite set of eight independent states, called *transactional states*, which can be used to describe the external state of a transaction at any time. These transactional states are *idle* (*idl*), *active* (*act*), *aborted* (*abt*), *failed* (*fal*), *successful* (*suc*), *undoing* (*und*), *compensated* (*cmp*), and *half-compensated* (*hap*), where *idl*, *act*, etc. are the abbreviated forms. Among the eight states, *suc*, *abt*, *fal*, *cmp*, *hap* are the terminal states. The transition relations among the states are illustrated in Fig. 3.1 [27].

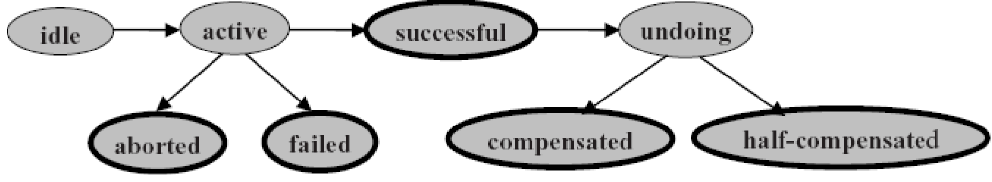

Figure 3.1: State transition diagram of a compensable transaction

$\sum$ is used to represent the finite set of transactional states. Δ is used to represent the set of terminal states, which is a subset of $\sum$, i.e.,

$$\sum = \{ \text{idl, act, suc, abt, fal, und, cmp, hap} \}$$

$$\Delta = \{ \text{suc, abt, fal, cmp, hap} \}$$

$$\Delta \subseteq \sum.$$

Before activation, a compensable transaction is in the *idle* state. Once activated, the transaction eventually moves to one of five terminal states. A *successful* transaction has the option of moving to the *undoing* state. If the transaction can successfully undo all its partial effects it goes to the *compensated* state, otherwise it goes to the *half-compensated* state.

An ordered pair consisting of a compensable transaction and its state is called a *transactional action* (called an action in [27]). Transactional actions are used to describe the behavioural dependencies of compensable transactions. In [27], five binary relations were proposed to define the constraints applied to transactional actions on compensable transactions. Informally the relations are described in Table 3.1, where both a and b are transactional actions:

1. $a < b$	only a can fire b
2. $a \prec b$	b can be fired by a
3. $a \ll b$	a is the precondition of b
4. $a \leftrightarrow b$	a and b both occur or both not
5. $a \nleftrightarrow b$	the occurance of one transactional action inhibits the other

Table 3.1: Behavioural dependencies of compensable transactions

The first three relations specify the order of execution, whereas the last two do not. $a < b$ indicates that a must precede b when the two transactional actions both occur, and that either the two transactional actions both occur or neither occurs. $a \leftrightarrow b$ indicates that either both transactional actions occur or neither occurs but $a \leftrightarrow b$ does

not impose any temporal constraint on transactional actions. $a \prec b$ tells us that if a occurs b must follow, but b can occur without a previous occurance of a. $a \ll b$ tells us that whenever b occurs, a must occur earlier. However, the occurrence of a does not guarantee a following occurrence of b. Finally, $a \nleftrightarrow b$ denotes that the two transactional actions must be mutially exclusive. These relations can be mathematically expressed by the following formulae, where s is a sequence of transactional actions and $s[i]$ denotes the ith element in the sequence:

$(R1)$ s satisfies a < b iff $\exists i, j$ such that $(i{<}j \wedge s[i] = a \wedge s[j] = b\ ) \vee \forall i.(s[i] \neq a \wedge s[i] \neq b)$

$(R2)$ s satisfies $a \prec b$ iff $\forall i\ (s[i] = a \Rightarrow \exists j.\ (j > i \wedge s[j] = b))$

$(R3)$ s satisfies $a \ll b$ iff $\forall i\ (s[i] = b \Rightarrow \exists j.\ (j{<}i \wedge s[j] = a))$

$(R4)$ s satisfies $a \leftrightarrow b$ iff $\exists i, j$ such that$(s[i] = a \wedge s[j] = b) \vee \forall i.(s[i] \neq a\ \wedge s[i] \neq b)$

$(R5)$ s satisfies $a \nleftrightarrow b$ iff $\exists i$ such that $(s[i] = a \Rightarrow \forall j.s[j] \neq b)$

The relations $<$, $\prec$, $\ll$ are anti-symmetric and transitive. The relation $\leftrightarrow$ is reflexive, symmetric and transitive, while $\nleftrightarrow$ is irreflexive, symmetric and intransitive. In addition, these relations exhibit the following useful properties [27]:

- **Law 1.** If $a < b$ and $a \leftrightarrow b$ then $b \leftrightarrow c$
- **Law 2.** If $a < b$ and $b \leftrightarrow c$ then $a \leftrightarrow c$
- **Law 3.** If $a < b$ and $b \circ c$ $(\circ \in \{\prec, \ll\})$ then $a \circ c$
- **Law 4.** If $a \circ b$ $(\circ \in \{\prec, \ll\})$ and $b < c$ then $a \circ c$
- **Law 5.** If $a \circ b$ $(\circ \in \{<, \prec, \leftrightarrow\})$ and $b \nleftrightarrow c$ then $a \nleftrightarrow c$
- **Law 6.** If $a \circ b$ $(\circ \in \{<, \ll, \leftrightarrow\})$ and $a \nleftrightarrow c$ then $b \nleftrightarrow c$

For an arbitrary compensable transaction T , all the transactional actions occurring during its execution must satisfy some constraints which are shown in Table 3.2.

$(T,idl) \ll (T,act)$	$(T,act) \ll (T,suc)$	$(T,act) \ll (T,abt)$	$(T,act) \ll (T,fal)$
$(T,suc) \nleftrightarrow (T,abt)$	$(T,suc) \nleftrightarrow (T,fal)$	$(T,abt) \nleftrightarrow (T,fal)$	$(T,cmp) \nleftrightarrow (T,hap)$
$(T,suc) \ll (T,und)$	$(T,und) \ll (T,cmp)$	$(T,und) \ll (T,hap)$	

Table 3.2: Intra-constraints of a compensable transaction

The transactional language t-Calculus was introduced by Li et al [29] to model business flow in terms of compensable transactions. t-Calculus provides a framework to combine compensable transactions allowing one to setup a long running business transaction which has compensation as its main error recovery technique.

3.1 The *t*-Calculus

The transactional language t-Calculus is intended to describe the behavior of top-level transactions [26]. Transactions are modeled in terms of atomic activities and a number of operators are introduced to support compensable transactions. An atomic activity is an activity for which no errors can take place during the execution. We use an infinite set of names to represent atomic activities ranged over by $A, B, ...$. Moreover, we consider two other special activities: the empty activity 0 always completes but has no effect; the error activity $\diamondsuit$ always leads to a fail state.

A compensable transaction consists of two parts: a forward flow and a compensation flow. In case of failure, compensation will be activated to compensate its forward flow. The basic way to construct a compensable transaction is through a transactional pair $A \div B$, where A is

the forward flow and B is its compensation. The compensation B is responsible for undoing the effect of A. Especially, $A \div 0$ denotes that the forward flow A is associated with an empty compensation. In other words, the effect made by A does not need to be removed when error occurs. Besides, not every activity can be semantically undone, so sometimes the application designer cannot find a suitable compensation. In this case, we use $A \div \Diamond$ to denote that the forward flow A is associated with an unacceptable compensation which always encounters a failure.

There are three variations for basic transactions. *Skip* stands for a successfully completed transaction without anything really done. *Abort* means a certain error has taken place and all composed compensations should be activated to recover from this failure. *Fail* indicates an error too; however, it has no mechanism to enable compensations and causes an exception instead.

The syntax of t-Calculus is made up of several operators which perform compositions of compensable transactions. Table 3.3 shows eight binary operators, where S and T represent arbitrary compensable transactions. These operators specify how compensable transactions are coupled and how the behaviour of a certain compensable transaction influences that of the other. The operators are discussed in detail in [27] [21] [18] and are described in the following section.

Sequential Composition	$S \; ; \; T$	Parallel Composition	$S \; \vert\vert \; T$
Internal Choice	$S \sqcap T$	Speculative Choice	$S \otimes T$
Alternative Forwarding	$S \rightsquigarrow T$	Backward Handling	$S \trianglerighteq T$
Forward Handling	$S \rhd T$	Programmable Composition	$S \divideontimes T$

Table 3.3: t-calculus operators

The syntax of this transactional calculus is summarized as follows.

$$BT ::= A \div B \mid A \div 0 \mid A \div \Diamond \mid Skip \mid Abort \mid Fail$$

$$S,T ::= BT \mid S;T \mid S||T \mid S \sqcap T \mid S \otimes T \mid S \rightsquigarrow T \mid S \trianglerighteq T \mid S \rhd T \mid S \varoast T$$

Where BT denotes a basic transaction, S,T denote arbitrary transaction.

3.2 The *t*-Calculus operators and their behavioral dependencies

t-Calculus operators can be semantically defined by behavioural dependencies, expressed using the five relations $<$, $\prec$, $\ll$, $\leftrightarrow$, $\nleftrightarrow$ (see Table 3.1). The functionality and behavioural dependencies of these operators are discussed in this section.

3.2.1 Sequential composition (;)

The sequential composition $S\ ;\ T$, denotes a sequential ordering of transactions. In the composite transaction $S;T$ the transaction S would begin execution, and the transaction T starts its execution once S has completed successfully. However, whenever T is aborted or compensated, the completed transaction S would be compensated so as to remove all the partial effects. The above description is reflected in the following behavioral dependencies [27]:

$(S,suc) < (T,act)$;

$(T,abt) \prec (S,und)$;

$(T,cmp) \prec (S,und)$.

Additional behavioural dependencies can be derived from the behavioural dependencies of sequential composition using the laws governing the internal constraints (found in Table 3.3) of compensable transactions [27]. Some of them are listed below:

$(S,suc) \ll (T,\alpha)$, $\alpha \in \Delta$;

$(S,\alpha) \nleftrightarrow (T,\beta)$, $\alpha \in \{abt, fal\}$ and $\beta \neq idl$;

(S,α) $\nleftrightarrow$ *(T,β)*, $\alpha \in \{cmp, hap\}$ and $\beta \in \{fal, hap\}$.

3.2.2 Parallel composition (||)

The parallel composition, $S \mid\mid T$, describes the composition of two compensable transactions running in parallel. Their compensations are activated concurrently when semantic rollback is needed. If one transaction aborts or fails, the other transaction tries to abort. This is achieved by an internal mechanism called forceful abort, which forcefully halts a transaction and leaves no partial effects. In other words, the two transactions reach the *successful* state or reach either the *abort* or *fail* state. Below is a listing of the behavioural dependencies of parallel composition [27]:

(S,act) $\leftrightarrow$ *(T,act)* ;

(S,und) $\leftrightarrow$ *(T,und)* ;

(S,suc) $\leftrightarrow$ *(T,suc)* .

Other dependencies can be deduced. For example:

(S,α) $\nleftrightarrow$ (T,β) $\alpha \in \{abt, fal\}$, $\beta \in \{suc, cmp, hap\}$;

(T,α) $\nleftrightarrow$ (S,β) $\alpha \in \{abt, fal\}$, $\beta \in \{suc, cmp, hap\}$.

3.2.3 Internal choice (⊓)

The internal choice composition, $S \sqcap T$, denotes the selection and execution of one, and only one, of the composed sub-transactions. Therefore, either S or T must be activated but not both. The transaction is selected on the basis of some internal data of the system. There is one basic behavioural dependency for $\sqcap$ [27]:

(S,act) $\nleftrightarrow$ *(T,act)* .

An additional behavioural dependency can be derived:

(S,α) $\nleftrightarrow$ *(T,β)*, $\{\alpha, \beta\} \subseteq \Delta$.

3.2.4 Speculative choice ($\otimes$)

The speculative choice composition, $S \otimes T$, provides a way for designers to permit two or more threads to finish one task. If one thread is aborted, the other thread is still active trying to achieve the same task. In $S \otimes T$, S and T are two sub-transactions with similar goals. The two sub-transactions have the same priority and they are arranged to be executed concurrently. The choice is delayed until one sub-transaction has succeeded. That is, only one sub-transaction will be selected to achieve its business goal. When one sub-transaction terminates successfully, the other one cannot succeed but aborts either internally or forcibly. Also, as with parallel composition, if either sub-transaction results in a failed state then the entire speculatively composed transaction will result in the failed state. This is due to the unavoidable remaining partial effect of that failed sub-transaction. The related behavioral dependencies are formalized below [27]:

(S,act) $\leftrightarrow$ *(T,act)* ;

(S,suc) $\not\leftrightarrow$ *(T,suc)* ;

(S,suc) $\not\leftrightarrow$ *(T,fal)* ;

(T,suc) $\not\leftrightarrow$ *(S,fal)* .

The above dependencies imply that only one sub-transaction can be compensated if compensation is needed, i.e.:

(S,α) $\not\leftrightarrow$ *(T,β)*, $\{\alpha, \beta\} \subseteq \{cmp, hap\}$.

3.2.5 Alternative forwarding ($\rightsquigarrow$)

As with speculative choice, the alternative forwarding composition, $S \rightsquigarrow T$, provides two functionally equivalent sub-transactions to achieve one business goal. The difference is that with $\rightsquigarrow$, these two sub-transactions have distinct priorities. In addition, the one with the higher priority is executed first and the other is activated only when the first one has been

aborted. Thus, in $S \rightsquigarrow T$, S runs first and T is the backup of S. The related dependency is given below [27]:

$(S,abt) < (T,act)$.

and the following dependencies can be derived:

$(S,abt) \ll (T,\alpha)\ \alpha \in \Delta$;

$(S,\alpha) \nleftrightarrow (T,\beta)\ \alpha \in \Delta - \{abt\}, \beta \in \Delta$.

3.2.6 Backward handling ($\trianglerighteq$)

The backward handling composition, $S \trianglerighteq T$, is one of two error handling operators which exists in the t-Calculus. Backward handling focuses on handling any partial effects or data inconsistencies resulting from an error. If an error occurs in S then T will attempt to remove any partial effects and if this is completed successfully then the compensating flow of the system is triggered. This behaviour is represented by the following basic behavioural dependency [27]:

$(S,fal) < (T,act)$.

From this dependency the following behavioural dependencies can be derived:

$(S,fal) \ll (T,\alpha),\ \alpha \in \Delta$;

$(S,\alpha) \nleftrightarrow (T,\beta),\ \alpha \in \Delta - \{fal\}, \beta \in \Delta$.

3.2.7 Forward handling ($\triangleright$)

The forward handling composition, $S \triangleright T$, is the second of the two error handling composition operators of the t-Calculus. Forward handling serves the same purpose as backward handling, in that its purpose is to remove any remaining partial effects from S if an error occurs in S. The main difference between the two operators is that the forward handling operator does not start the compensating flow once an error is detected. Instead, the forward handling operator continues with the forward flow of the system, in an effort to complete the goal of the system.

The forward handling operator has the following basic behavioural dependency [27]:

$(S,fal) < (T,act)$.

These following behaviour dependencies can be derived from the basic behavioural dependency listed above:

$(S,fal) \ll (T,\alpha)$, $\alpha \in \Delta$;

$(S,\alpha) \nleftrightarrow (T,\beta)$, $\alpha \in \{suc, abt, cmp, hap\}$ and $\beta \in \Delta$.

3.2.8 Programmable compensation ($\circledast$)

The programmable compensation composition, $S \circledast T$, focuses on providing better readability to the language of t-Calculus. Its purpose is to replace the left transaction's (S) compensation with the right transaction (T). Therefore, once the left transaction has successfully completed, the right transaction is stored as its compensation. Traditionally, the compensation of a transaction is performed by its attached compensating flow sub-transaction. For example, if S were a basic transaction, the transactional pair $s \div s'$ would ordinarily install or store the sub-transactions upon the successful completion of s. The sub-transaction s' of S performs the transaction's compensation. Given the programmable compensation composition described above, the sub-transaction s' would be replaced by the transaction T. This will make the transaction T the compensation for the transaction S. This behaviour can be expressed in the form of the following relation [27]:

$(S,suc) \ll (T,act)$.

Using the laws in the previous section, the intra-constraints of a compensable transaction (found in Table 3.2), and the behavioural dependency above, the following behavioural dependency can be deduced:

$(S,suc) \ll (T,\alpha)$, $\alpha \in \Delta$.

3.2.9 Associativity

From the definitions of t-Calculus operators, we can to show that the t-Calculus operators $||, \sqcap, \rightsquigarrow$ and $\otimes$ are associative. First we show the associativity for parallel composition operator $(||)$. We will show that $(A||B)||C \equiv A||(B||C)$, by using the behavioral characteristics of the operator $(||)$;

$(A||B)||C$ has occurred *iff* the following 3 conditions hold:

1. $((A||B),act) \leftrightarrow (C,act)$;

   iff $(A||B)$ and C both are in *act* or neither is in *act*;

   iff A, B and C all are in *act* or none of them is in *act*.

2. $((A||B),und) \leftrightarrow (C,und)$;

   iff $(A||B)$ and C both are in *und* or neither is in *und*;

   iff A, B and C all are in *und* or none of them is in *und*.

3. $((A||B),suc) \leftrightarrow (C,suc)$;

   iff $(A||B)$ and C both are in *suc* or neither is in *suc*;

   iff A, B and C all are in *suc* or none of them is in *suc*.

Therefore, $(A||B)||C$ has occurred *iff* 1. A, B and C all are in *act* or none of them is in *act*, 2. A, B and C all are in *und* or none of them is in *und* and 3. A, B and C all are in *suc* or none of them is in *suc*.

$A||(B||C)$ has occurred *iff* the following 3 conditions hold:

1. $(A,act) \leftrightarrow ((B||C),act)$;

   iff A and $(B||C)$ both are in *act* or neither is in *act*;

   iff A, B and C all are in *act* or none of them is in *act*.

2. $(A,und) \leftrightarrow ((B||C),und)$;

   iff A and $(B||C)$ both are in *und* or neither is in *und*;

   iff A, B and C all are in *und* or none of them is in *und*.

3. *(A,suc)* $\leftrightarrow$ *((B||C),suc)* ;

iff A and $(B||C)$ both are in *suc* or neither is in *suc*;

iff A, B and C all are in *suc* or none of them is in *suc*.

Therefore, $A||(B||C)$ has occurred *iff* 1. A, B and C all are in *act* or none of them is in *act*, 2. A, B and C all are in *und* or none of them is in *und* and 3. A, B and C all are in *suc* or none of them is in *suc*.

So, $(A||B)||C \equiv A||(B||C)$, as they have the same behavioral characteristics.

Now we will show that $(A \sqcap B) \sqcap C \equiv A \sqcap (B \sqcap C)$ by showing that they have same basic behavioral characteristics:

$(A \sqcap B) \sqcap C$ has occurred *iff* the following condition holds:

1. *((A ⊓ B),act)* $\nleftrightarrow$ *(C,act)* ;

iff if $(A \sqcap B)$ is in *act* then C can't be in act or vice versa;

iff if any one from A, B or C is in *act*, the other two can't be in *act*.

$A \sqcap (B \sqcap C)$ has occurred *iff* the following condition hold:

1. *(A,act)* $\nleftrightarrow$ *((B ⊓ C),act)* ;

iff if A is in *act* then $(B \sqcap C)$ can't be in *act* or vice versa;

iff if any one from A, B or C is in *act*, the other two can't be in *act*.

Therefore, it is easy to see that the operator, $\sqcap$ is associative.

Now we will show that the alternative choice $\rightsquigarrow$ is associative; i.e., $((A \rightsquigarrow B) \rightsquigarrow C \equiv A \rightsquigarrow (B \rightsquigarrow C))$:

1. $(A \rightsquigarrow B) \rightsquigarrow C$ has occurred *iff* the following condition holds:

(a). *((A ⇝ B),abt)* $<$ *(C,act)*;

iff if C is in *act* then B is in *abt* and A is in *abt*;

2. $A \rightsquigarrow (B \rightsquigarrow C)$ has occurred *iff* the following condition holds:

(b). *(A,abt)* $<$ *((B ⇝ C),act)*;

iff if $(B \rightsquigarrow C)$ is in *act* then A is in *abt*;

We will prove:

I. (a) is equivalent to a statement holding for 1, which we will show holds for 2, and

II. (b) is equivalent to a statement holding for 2, which we will show holds for 1.

From 2, $A \rightsquigarrow (B \rightsquigarrow C)$ it is obvious that in order to get C in *act*, A and B must be in *abt*. Thus (a) holds for 2.

If $B \rightsquigarrow C$ is in *act* we have two possibilities:

i) B is in *act* and C is in *idl*;

ii) B is in *abt* and C is in *act*.

<u>case i</u> Show: if B is in *act* and C is in *idl* then A is in *abt*.

But in 1. $(A \rightsquigarrow B) \rightsquigarrow C$, if B is in *act* then A is in *abt*. Hence if B is in *act* and C is in *idl* then A is in *abt*.

<u>case ii</u> Show if B is in *abt* and C is in *act* then A is in *abt*.

But in 1. $(A \rightsquigarrow B) \rightsquigarrow C$, if C is in *act* then $A \rightsquigarrow B$ is in *abt*.

Therefore, if C is in *act* and B is in *abt* then $A \rightsquigarrow B$ is in *abt*.

Hence if C is in *act* and B is in *abt* then A is in *abt*.

In both cases if B is in *abt* and C is in *act* then A is in *abt*. i.e., (b) holds for 1.

Therefore we can conclude, the operator, $\rightsquigarrow$ is associative.

Now we will show that $(A \otimes B) \otimes C \equiv A \otimes (B \otimes C)$, by using the behavioral characteristics of the operator $\otimes$:

$(A \otimes B) \otimes C$ has occurred *iff* the following 3 conditions hold:

1. *((A ⊗ B),act)* $\leftrightarrow$ *(C,act)* ;

   iff $(A \otimes B)$ and C both are in *act* or neither is in *act*;

   iff A, B and C all are in *act* or none of them is in *act*.

2. *((A ⊗ B),suc)* $\nleftrightarrow$ *(C,suc)* ;

   iff if $(A \otimes B)$ is in *suc* then C can't be in *suc* or vice versa;

iff any one from A, B or C is in *suc*, the other two can't be in *suc*.

3. *((A $\otimes$ B),suc)* $\nleftrightarrow$ *(C,fal)* *iff* the following hold:

   i) if A is in *suc*, then C can't be in *fal*;

   ii) if B is in *suc*, then C can't be in *fal*;

   iii) if C is in *fal*, then A can't be in *suc*;

   iv) if C is in *fal*, then B can't be in *suc*;

4. *((A $\otimes$ B),fal)* $\nleftrightarrow$ *(C,suc)* ; *iff* the following hold:

   v) if A is in *fal*, then C can't be in *suc*;

   vi) if B is in *fal*, then C can't be in *suc*;

   vii) if C is in *suc*, then A can't be in *fal*;

   viii) if C is in *suc*, then B can't be in *fal*;

Two basic characteristics for $A \otimes B$ are:

(A,suc) $\nleftrightarrow$ *(B,fal)* which mean

   ix) if A is in *suc*, then B can't be in *fal*;

   x) if B is in *fal*, then A can't be in *suc*;

and *(A,fal)* $\nleftrightarrow$ *(B,suc)* which mean

   xi) A is in *fal*, then B can't be in *suc*;

   xii) B is in *suc*, then A can't be in *fal*;

From (i-xii) we find if any one from A, B and C is in *suc*, then neither of the others can be in *suc* and neither can be in *fal*.

Therefore, $A \otimes (B \otimes C)$ has occurred *iff* 1. A, B and C all are in *act* or none of them is in *act*, 2. any one from A, B or C is in *suc*, the other two can't be in *suc* and 3. any one from A, B and C is in *suc*, then neither of the others can be in *suc* and neither can be in *fal*.

$A \otimes (B \otimes C)$ has occurred *iff* the following 3 conditions hold:

1. *(A,act)* $\leftrightarrow$ *((B $\otimes$ C),act)* ;

   iff A and $(B \otimes C)$ both are in *act* or neither is in *act*;

iff A, B and C all are in *act* or none of them is in *act*.

2. *(A,suc)* $\nleftrightarrow$ *((B $\otimes$ C),suc)* ;

   iff A is in *suc* then $(B \otimes C)$ can't be in *suc* or vice versa;

   iff any one from A, B or C is in *suc*, the other two can't be in *suc*.

3. *(A,suc)* $\nleftrightarrow$ *((B $\otimes$ C),fal)* *iff* the following hold:

   i) if A is in *suc*, then B can't be in *fal*;

   ii) if A is in *suc*, then C can't be in *fal*;

   iii) if B is in *fal*, then A can't be in *suc*;

   iv) if C is in *fal*, then A can't be in *suc*;

   4. *(A,fal)* $\nleftrightarrow$ *((B $\otimes$ C),suc)* *iff* the following hold:

   v) if A is in *fal*, then B can't be in *suc*;

   vi) if A is in *fal*, then C can't be in *suc*;

   vii) if B is in *suc*, then A can't be in *fal*;

   viii) if C is in *suc*, then A can't be in *fal*;

Two basic characteristics for $B \otimes C$ are:

(B,suc) $\nleftrightarrow$ *(C,fal)* which mean

   ix) if B is in *suc*, then C can't be in *fal*;

   x) if C is in *fal*, then B can't be in *suc*;

and *(B,fal)* $\nleftrightarrow$ *(C,suc)* which mean

   xi) if B is in *fal*, then C can't be in *suc*;

   xii) if C is in *suc*, then B can't be in *fal*;

From (i-xii) we find if any one from A, B and C is in *suc*, then neither of the others can be in *suc* and neither can be in *fal*.

Therefore, $(A \otimes B) \otimes C$ has occurred *iff* 1. A, B and C all are in *act* or none of them is in *act*, 2. any one from A, B or C is in *suc*, the other two can't be in *suc* and 3. any one from A, B and C is in *suc*, then neither of the others can be in *suc* and neither can be in *fal*.

Therefore, the operator, $\otimes$ is associative.

Altogether we have provided full details to prove the following theorem:

Theorem 3.1. The t-Calculus operators $||, \sqcap, \rightsquigarrow$ and $\otimes$ are associative.

Chapter 4

The compensable workflow modeling language

A workflow consists of steps or tasks that represent a work process. Currently, most workflow languages support the basic constructs of sequence, splits ("and" and "xor") and joins ("and" and "xor"). In this chapter we define a new workflow modeling language, called the Compensable Workflow Modeling Language (CWML), which allows us to extend the notion of task element with the concept of compensation, using ideas borrowed from the t-Calculus. In particular we define the notion of compensable task, and compose tasks using t-Calculus operators. It is worth mentioning that we are the first who are using t-Calculus in a workflow modeling language. We will use Petri nets to give our definitions and a sound mathematical foundation.

4.1 Compensable workflow nets

An atomic task is an indivisible unit of work. Atomic tasks can be either compensable or uncompensable.

Definition 4.1. *An atomic uncompensable task t is a tuple (s, P) such that:*

- *P is a Petri net, as shown in Fig. 4.1;*
- *s is a set of unit states {idle, active, successful}; the unit state idle indicates that transition pt_1 is disabled and there is no token in place p_suc; active indicates that transition pt_1 is enabled and successful indicates that there is a token in the place p_suc;*

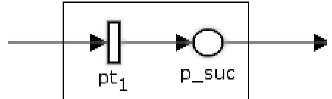

Figure 4.1: Petri net representation of an atomic uncompensable task

Remark that the unit states of a task are different from the state (marking) of a Petri net. The state of a Petri net is determined by the marking of its places, but in P the task is in the *idle* state if there is no token its input place. Fig. 4.1 is the Petri net representation of an atomic uncompensable task.

Definition 4.2. *An atomic compensable task t_c is a tuple (s_c, P_c) such that:*

- *P_c is a Petri net as shown in Fig. 4.2;*
- *s_c is a set of unit states {idle, active, successful, undoing, aborted}, where*
  - *idle indicates that transitions pt_1, pt_2, pt_3 are disabled and there is no token in place p_suc and p_abt;*
  - *active indicates that the transition pt_1 is enabled;*
  - *successful indicates that there is a token in place p_suc;*
  - *undoing indicates that the transition pt_3 is enabled and*

– *aborted indicates that there is a token in place p_abt.*

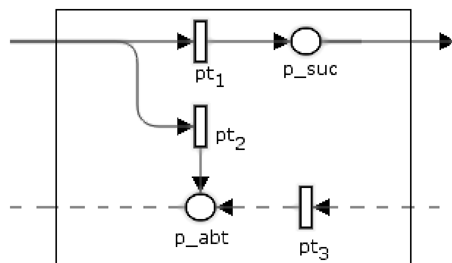

Figure 4.2: Petri net representation of an atomic compensable task

Fig. 4.2 gives the Petri net representation of an atomic compensable task. Solid lines represent the forward flow and broken lines represent the compensation flow. The task t_c transits to the unit state *active* after getting a token in the input place of transition pt_1. The token can move to either `p_suc` or `p_abt` representing unit states *successful* or *aborted*, respectively. The unit state *aborted* indicates an error occurred performing the task and the effects was successfully removed. The backward (compensation) flow is started from this point. Note that t_c can transit to the unit state *aborted* either before or after the unit state *successful*. We made a simplification for compensable tasks by excluding $fail$ (failed), *hap* (half-compensated) and *cmp* (compensated) states. In order to design a structurally sound workflow (discussed in section 4.3) without explicit error handlers we have removed $failed$ and *hap* states from t_c. On the other hand the state *cmp* overlapped with *abort* in t_c.

A compensable task can be composed with other compensable tasks using the t-Calculus operators. When a task executes, it performs some actions, and the execution of a task may depend on some conditions. The formal definition of pre-condition and action are given below:

Definition 4.3. *A term, σ, is defined using BNF as follows:*

$$\sigma ::= c \mid \chi \mid \sigma \oplus \sigma, \text{ where } \oplus \in \{+, -, \times, \div\},$$

c is a real number and χ is a real variable.

A pre-condition is a formula, ψ_{pre}, is defined as ψ_{pre} ::= $\sigma \diamond \sigma$ | $(\psi_{pre} \uplus \psi_{pre})$, where $\diamond \in \{<, \leq, >, \geq, ==\}$, $\uplus \in \{\&\&, ||\}$ and σ is a term.

An action, ψ_{act}, of a task is an assignment defined as ψ_{act} ::= $v = \sigma$; v is called a mapsTo variable and σ is a term

Definition 4.4. *A compensable task, T_c, is recursively defined by the following well formed formula:*

$$T_c = t_c(\{\psi_{act}\}, \{\psi'_{act}\}) \mid (\{\psi_{pre}\}T_c \odot \{\psi_{pre}\}T_c)$$

where t_c is an atomic compensable task, $\{\psi_{act}\}$ and $\{\psi'_{act}\}$ are the set of actions (forward and compensation, respectively) of t_c, $\{\psi_{pre}\}$ is a set of pre-conditions of T_c and $\odot \in \{;, ||, \sqcap, \otimes, \rightsquigarrow\}$ is a t-Calculus operator defined for tasks in a manner identical to that for compensable transactions in section 3.2.1 - 3.2.5.

Note that in our definition of atomic compensable task t_c, we assume if activated, t_c is either completes successfully or fully compensate and as a result the backward handling operator ($\trianglerighteq$), forward handling operator ($\triangleright$) and programmable compensation operator ($\circledast$) are not needed here.

Any task can be composed with uncompensable and/or compensable tasks to create a new task. As above, a task may be considered as a formula and we use BNF to represent the set of "well formed" tasks or formulas.

Definition 4.5. *A task, T, is recursively defined by the following BNF formula:*

$$T ::= t\{\psi_{act}\} \mid T_c \mid (\{\psi_{pre}\}T \ominus \{\psi_{pre}\}T)$$

where t is an uncompensable atomic task, $\{\psi_{act}\}$ is the set of actions of t, $\{\psi_{pre}\}$ is the set of

pre-conditions of T, T_c *is a compensable task and* $\ominus \in \{\wedge, \vee, \times, \bullet\}$ *is a control flow operator defined as follows:*

- $T_1 \wedge T_2$*:* T_1 *and* T_2 *will be executed in parallel,*
- $T_1 \vee T_2$*:* T_1 *or* T_2 *or both will be executed in parallel,*
- $T_1 \times T_2$*: exclusively one of the task (either* T_1 *or* T_2*) will be executed,*
- $T_1 \bullet T_2$: T_1 will be executed first then T_2 will be executed.

A subformula of a well-formed formulae is also called a *subtask*. Any task which is built up from the operators $\{\wedge, \vee, \times, \bullet\}$ is deemed as uncompensable. Thus if T_1 and T_2 are compensable tasks, then T_1;T_2 denotes another compensable task while $T_1 \bullet T_2$ denotes a task consisting of two distinct compensable subtasks. We remark that the operators $\wedge, \vee, \times$ and $\bullet$ as well as the t-Calculus operators $||, \sqcap, \rightsquigarrow$ and $\otimes$ are all associative.

In order for the underlying Petri net construction to be complete, we add a pair of *split* and *join* routing tasks for operators $\wedge$, $\vee$, $\times$, $||$, $\sqcap$, $\otimes$, and $\rightsquigarrow$ and we give their graphical representation in the following section (Fig. 4.3). Each of these routing tasks has a corresponding Petri net representation, e.g., for the speculative choice operator $T_{c_1} \otimes T_{c_2}$, the *split* routing task will direct the forward flow to T_{c_1} and T_{c_2}; the task that performs its operation first will be accepted and the other one will be aborted.

We are now ready to make the formal definition of Compensable WorkFlow nets.

Definition 4.6. *A Compensable Workflow net (CWF-net)* C_N *is a tuple (*$\mathtt{i}$, $\mathtt{o}$, $\mathbb{T}$, $\mathbb{T}_c$, F*) such that:*

- $\mathtt{i}$ *is the input condition,*
- $\mathtt{o}$ *is the output condition,*

- $\mathbb{T}$ *is a set of atomic tasks, split and join tasks*
- $\mathbb{T}_c \subseteq \mathbb{T}$ *is a set consists of the compensable tasks, and* $\mathbb{T} \backslash \mathbb{T}_c$ *is the set of uncompensable tasks,*
- $F \subseteq (\{\boldsymbol{i}\} \times \mathbb{T}) \cup (\mathbb{T} \times \mathbb{T}) \cup (\mathbb{T} \times \{\boldsymbol{o}\})$ *is the flow relation (for the net),*
- *The first compensable subtask of a compensable task is called the initial subtask; the backward flow from the initial subtask is directed to the uncompensable task or the output condition followed by the compensable task, and every task in a workflow is on a directed path from* $\boldsymbol{i}$ *to* $\boldsymbol{o}$*.*

  The elements of a workflow (i.e., tasks, input condition, output condition and flow relations) are called workflow components.

If a compensable task T_c in a CWF-net aborts, the system starts to compensate. After the full compensation, the backward flow reaches the initial subtask of T_c and the flow terminates, as the backward flow of an initial task of T_c is connected with an uncompensable task or the output condition followed by T_c. The reader must distinguish between the flow relation (F) of the net, as above and the internal flows of the atomic (uncompensable and compensable) tasks. A CWF-net such that $\mathbb{T}_c = \mathbb{T}$ is called a *fully Compensable workflow net* (CWF$_f$-net). To organize a large CWF-net, it is convenient to divide a large CWF-net into small CWF-net's. Each small CWF-net representing a subformula is known as a *subnet.* A placeholder for the *subnet* is used in the large CWF-net instead of a subformula. The placeholder is known as a *composite task.* Example of a *subnet* may be found in chapter 8.

4.2 The compensable workflow modeling language and its Petri net representation

We first present a graphical representation of tasks, then present the contruction principles for modeling a compensable workflow. Our notation is inspired by YAWL [40], ADEPT2 [37] and t-Calculus operators [29]. Fig. 4.3 gives a graphical representation of tasks, where t stands for an uncompensable task and t_c stands for a compensable task.

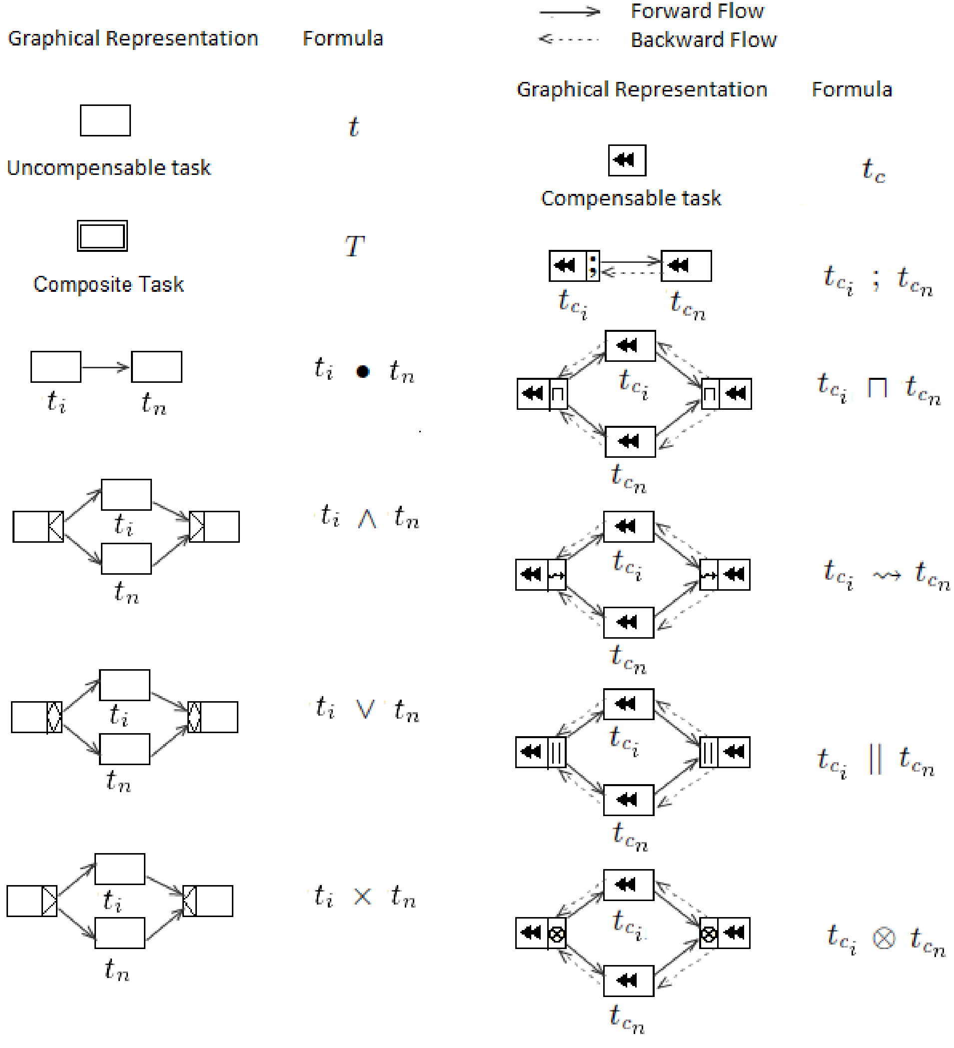

Figure 4.3: Graphical representation of CWML

Construction Principle: *Construction principles for the graphical representation of tasks are as follows:*

- *The operators* [$\bullet$, ;] *are used to compose the operand tasks sequentially. Atomic uncompensable tasks and atomic compensable tasks are connected by a single forward flow. Atomic compensable tasks are connected by a forward flow if they are composed using* ($\bullet$) *and by both a forward flow and a backward flow if they are composed using the sequential operator* (;)*;*

- *(The convention of ADEPT2 [37]) A pair of split and join routing tasks are used for tasks composed by* $\{\wedge, \vee, \times, ||, \sqcap, \otimes, \rightsquigarrow\}$*. Atomic uncompensable tasks are connected with split and join tasks by a single forward flow. Atomic compensable tasks are connected with split and join tasks by two flows (forward and backward). The operators and their corresponding split and join tasks are shown in Table 4.1;*

- *For those operators that are associative, an n-fold composition is represented using the appropriate n-fold split and join. For example* $(t_1 \wedge t_2) \wedge t_3$ *which is the same as* $t_1 \wedge (t_2 \wedge t_3)$ *is represented by* $t_1 \wedge t_2 \wedge t_3$*, see Fig. 4.4.*

If these principles are followed, the resulting graph is said to be "correct by construction" (Terminologies borrowed from [37]).

Tasks composed with <u>and composition</u> are executed in parallel. In Fig. 4.3, we can see the tasks t_i and t_n are composed with an "and" ($\wedge$) operator. It represents the formula $t_i \wedge t_n$. Fig. 4.5 shows the Petri net representation of $t_i \wedge t_n$. In this figure t_s ("and" split) and t_j ("and" join) are two routing tasks. During the execution, both tasks t_i and t_n run in parallel.

Tasks composed with <u>xor composition</u> will be selected and activated depending on some internal decisions. During execution, only one branch will be activated. In Fig. 4.3, we

Operator	Split task	Join task
$\wedge$	and-split	and-join
$\vee$	or-split	or-join
$\times$	xor-split	xor-join
$\|\|$	parallel-split	parallel-join
$\sqcap$	internal-split	internal-join
$\otimes$	speculative-split	speculative-join
$\rightsquigarrow$	alternative-split	alternative-join

Table 4.1 : Operators and their associated split and join tasks

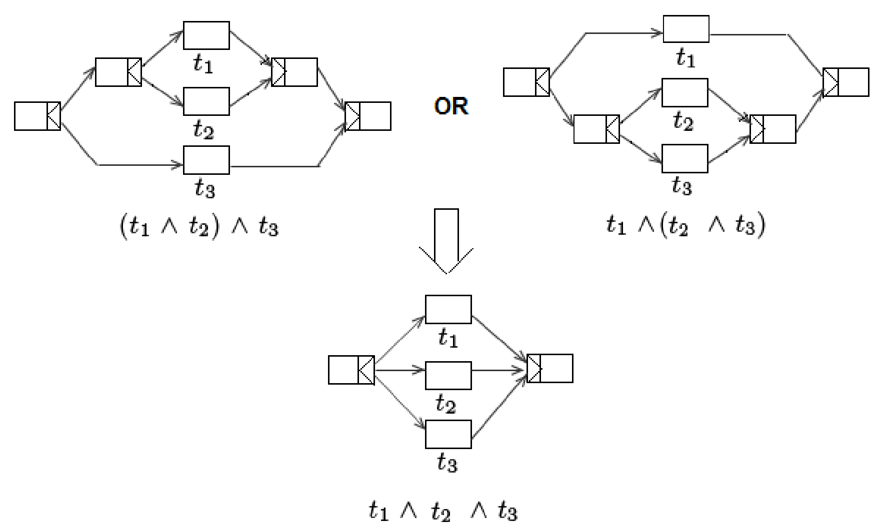

Figure 4.4: n-fold split and join tasks

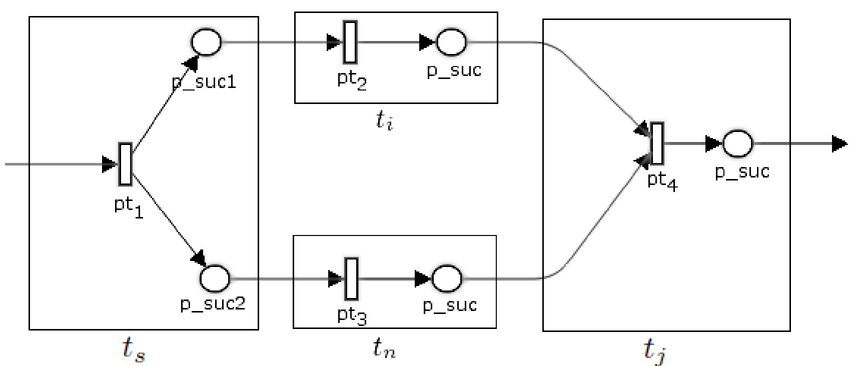

Figure 4.5: Petri net representation of and composition

can see tasks t_i and i_n composed with an *xor* ($\times$) operator, representing the formula $t_i \times t_n$. Fig. 4.6 shows the Petri net representation of $t_i \times t_n$.

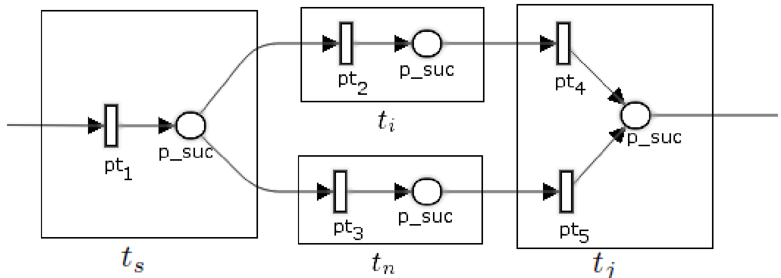

Figure 4.6: Petri net representation of xor composition

The or composition is used to decide between two or more tasks. Two tasks t_i and t_n composed with an *or choice* ($\vee$) are shown in Fig. 4.3. Fig. 4.7 shows the Petri net representation of the t_s (*or split*) and t_j (*or join*) tasks. During execution, either t_i and t_n both, or only t_i will execute.

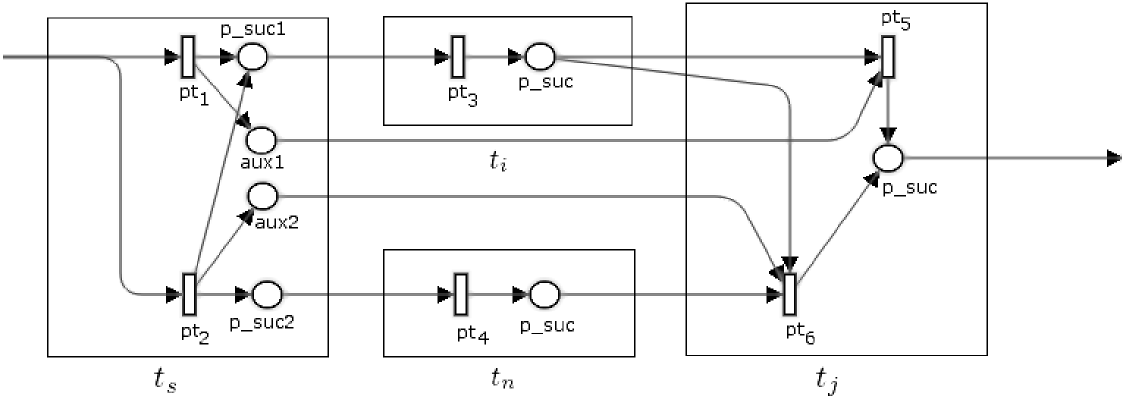

Figure 4.7: Petri net representation of or composition

Now we give the Petri net representation of compensable tasks and their compositions. The behavioral dependencies described in chapter 3 for t-Calculus operators also

hold for the compensable task compositions.

Two compensable tasks t_{c_i} and t_{c_n} can be composed with sequential composition as shown in Fig. 4.3, which represents the formula t_{c_i} ; t_{c_n}. Task t_{c_n} will be activated only when task t_{c_i} finishes its operations successfully. For the compensation flow, when t_{c_n} is aborted, t_{c_i} will be activated for compensation, i.e., to remove its partial effects. One of the behavioral dependency for the composition $t_{c_i};t_{c_n}$ is $(t_{c_i}, suc) < (t_{c_n}, act)$, meaning t_{c_n} will be activated *iff* t_{c_i} was *successful*. It is obvious by inspection from the Petri net representation of $t_{c_i};t_{c_n}$ from Fig. 4.8. The transition pt_4 of t_{c_n} is connected with the place *p_suc* of t_{c_i} by an incoming arc. In order to activate the transition pt_4, there has to be a token in place *p_suc* of t_{c_i}. Other behavioral dependencies for the sequential composition can be found from the Petri net representation. Note that dependencies which include state *fal, hap, cmp* does not hold here as we simplified the representation of atomic task by removing those states.

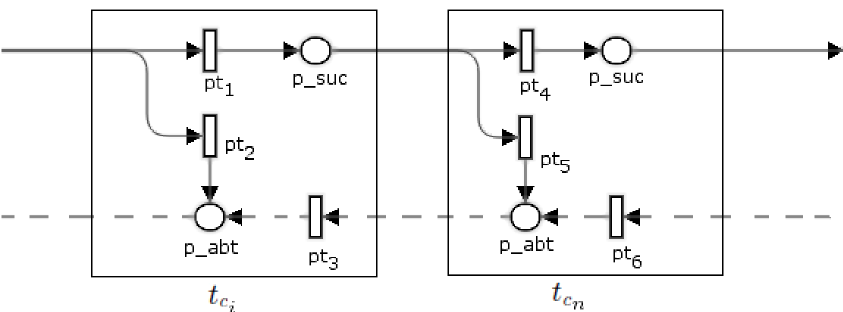

Figure 4.8: Petri net representation of sequential composition

Tasks composed using internal choice will be selected and activated depending on some internal decisions. During execution, only one branch will be activated and upon abort the compensable flow will be executed. In Fig. 4.3, we can see tasks t_{c_i} and t_{c_n} composed with the internal choice composition, representing the formula $t_{c_i} \sqcap t_{c_n}$. The basic

behavioural dependency indicates that only one of the tasks, t_{c_i} or t_{c_n}, will activate: $(t_{c_i}, act) \nleftrightarrow (t_{c_n}, act)$. The Petri net representation of the internal choice *split* and *join* tasks are shown in Fig. 4.9.

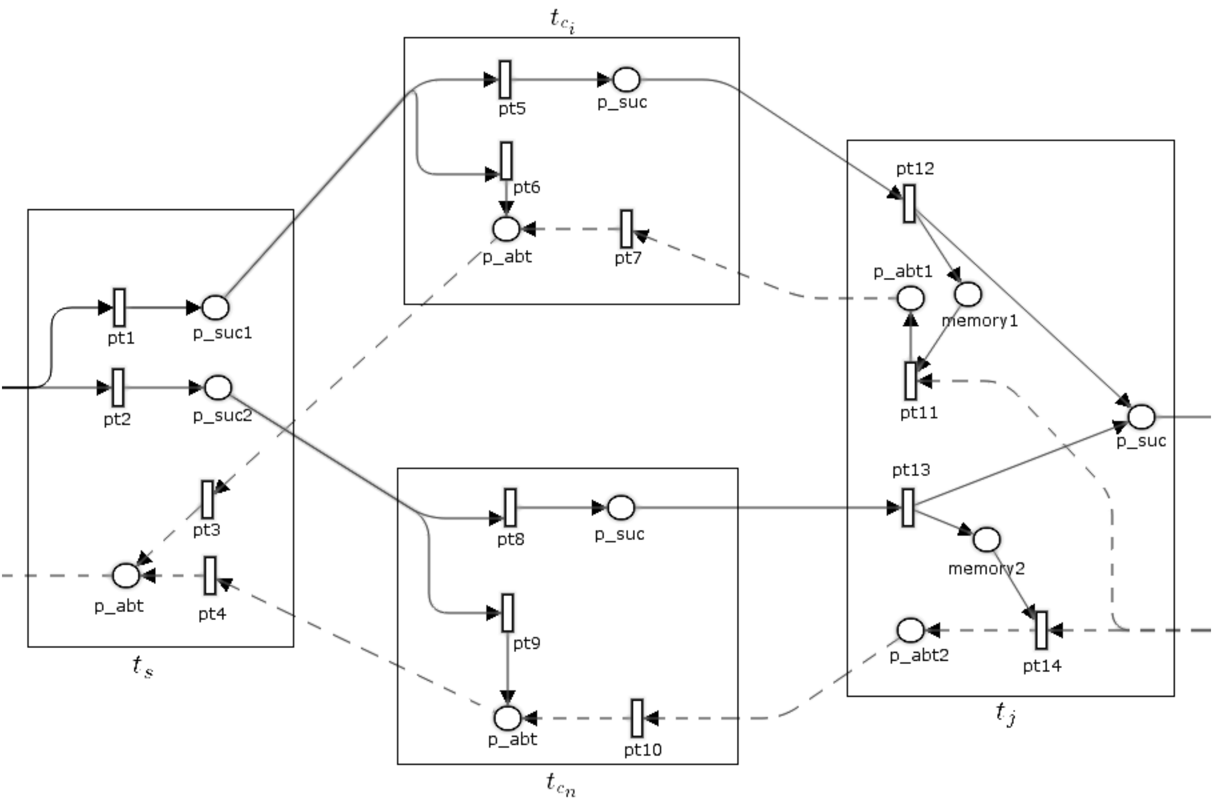

Figure 4.9: Petri net representation of internal choice composition

The alternative forwarding composition is used to decide between two or more equivalent tasks with the same goal. Alternative forwarding implies a preference between the tasks, and it does not execute all branches in parallel. For example, if the alternative forwarding composition is used to buy air tickets, one airline may be preferred to the other and an order is first placed to the preferred airline. The other airline will be used to place an order only if the first order aborts. Fig. 4.10 gives a Petri net representation of $t_{c_i} \rightsquigarrow t_{c_n}$. Compensable tasks that are composed using parallel composition are executed in par-

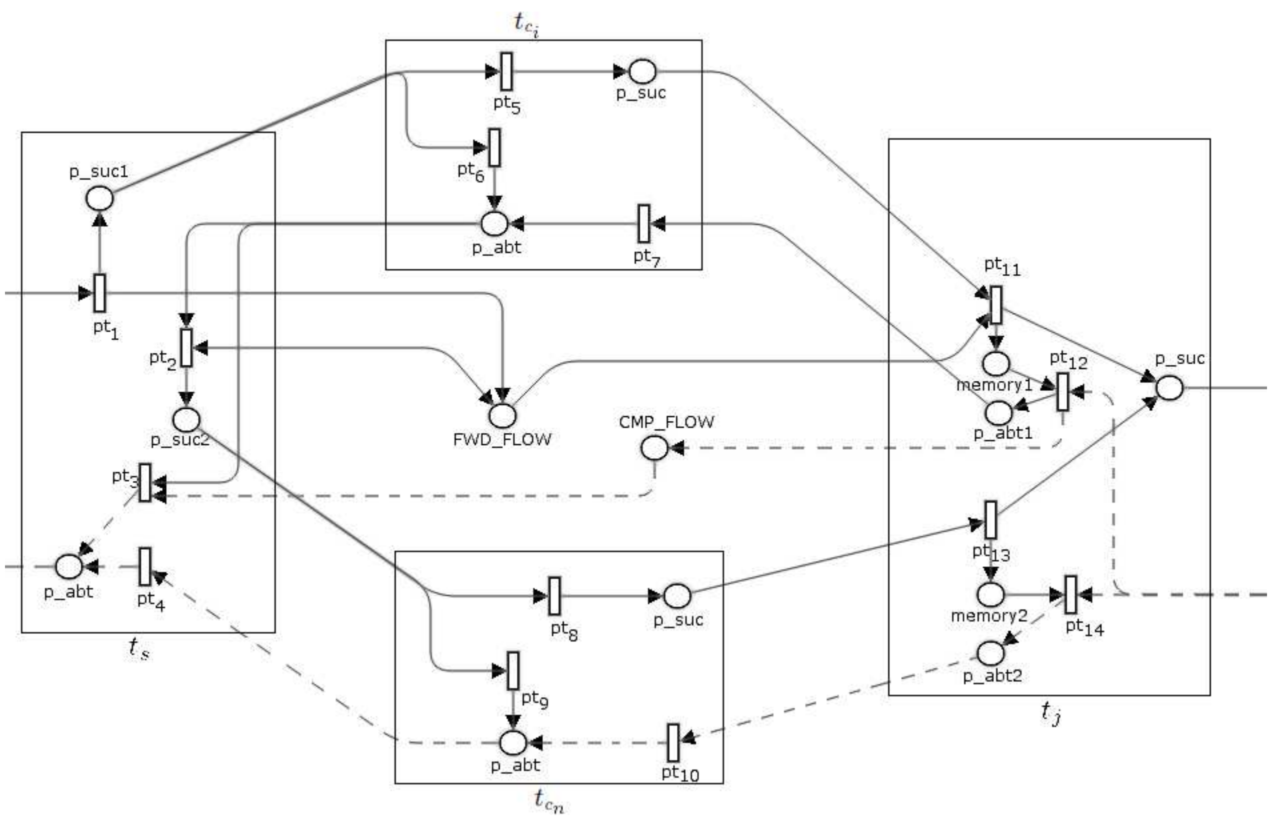

Figure 4.10: Petri net representation of alternative forward composition

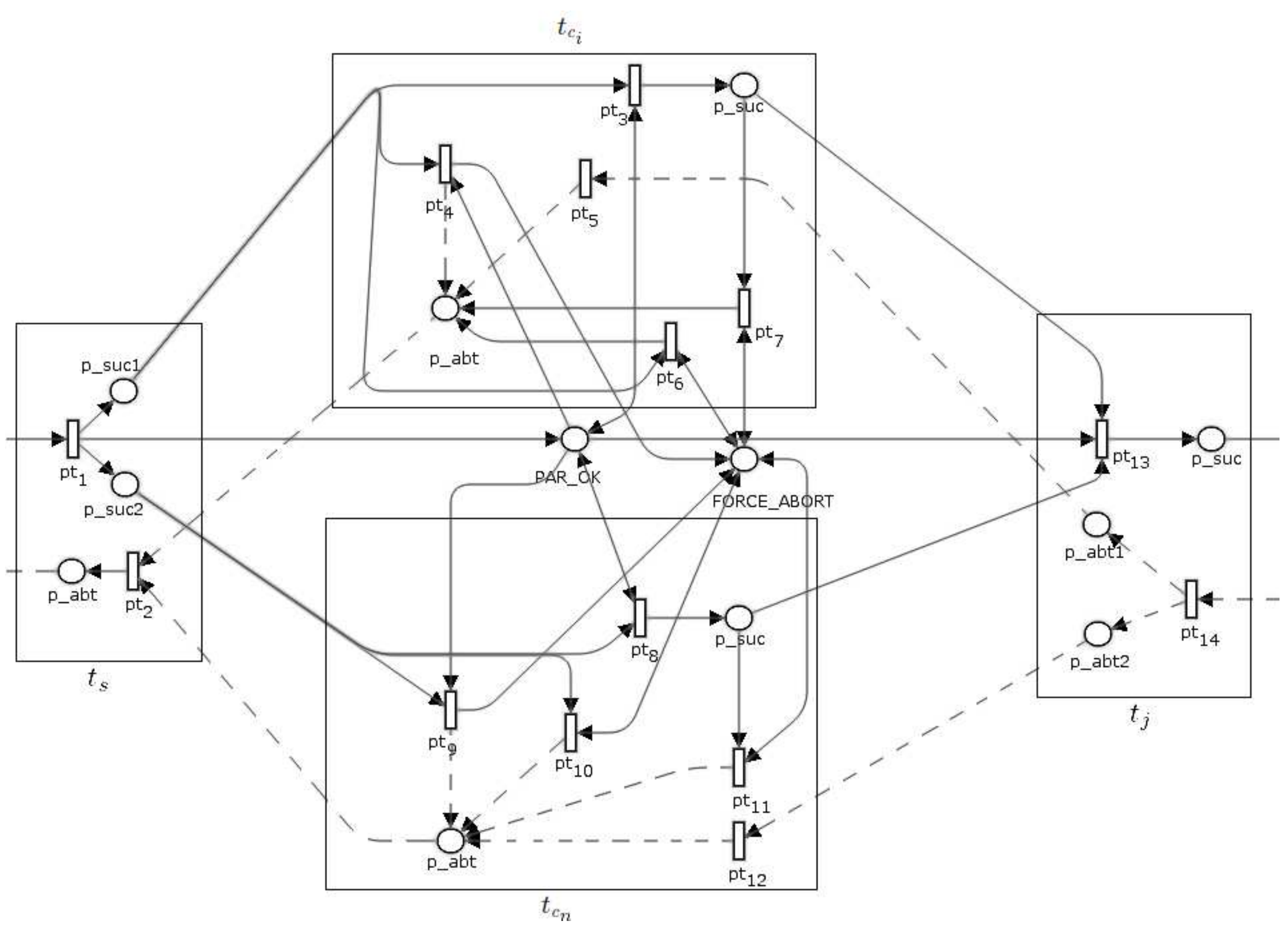

Figure 4.11: Petri net representation of parallel composition

allel. In Fig. 4.3, we can see the tasks t_{c_i} and t_{c_n} which are composed in parallel. It represents the formula $t_{c_i} \mid\mid t_{c_n}$. Compensable tasks t_{c_i} and t_{c_n} will run in parallel but if either of the tasks aborts, the other task will be aborted forcefully. The Petri net representation of the parallel composition is shown in Fig. 4.11. Here we see two new places PAR_OK and $FORCE_ABORT$, and two extra transition in each of t_{c_i} and t_{c_n}. In the Petri net representation, the *split* task t_s activates both t_{c_i} and t_{c_n} and produces a token in place PAR_OK. Note that parallel composition requires that if one branch aborts then the other branch should be stopped to save time and resources. This is achieved by these two extra places PAR_OK and $FORCE_ABORT$. In order to transit to the *successful* state t_{c_i} and t_{c_n} requires a token in place PAR_OK. If any of the task from t_{c_i} or t_{c_n} is aborted, it consumes the token from the place PAR_OK, and produces a token in place $FORCE_ABORT$. A token in place $FORCE_ABORT$ ensures that other tasks (if activated) transit to the *abort* state.

The speculative choice composition is used to decide between two or more equivalent tasks which have the same or similar goals. Speculative choice will execute two independent tasks in parallel and will select the task which completes first. It is designed to reduce the time complexity of a system by executing two tasks simultaneously which could satisfy a requirement, but there is no preference between either tasks. The process of buying air tickets can be modeled with speculative choice tasks. For example a system orders tickets from two different airlines in parallel, then takes the one that is confirmed first and cancels the other booking. In Fig. 4.3, we can see the tasks t_{c_i} and t_{c_n} which are composed by speculative Choice. It represents the formula $t_{c_i} \otimes t_{c_n}$. Fig. 4.12 shows the Petri net representation of the speculative choice composition. Here we see two new places $SPEC_OK$ and $SPEC_ABORT$, and one extra transition in each of t_{c_i} and t_{c_n}.

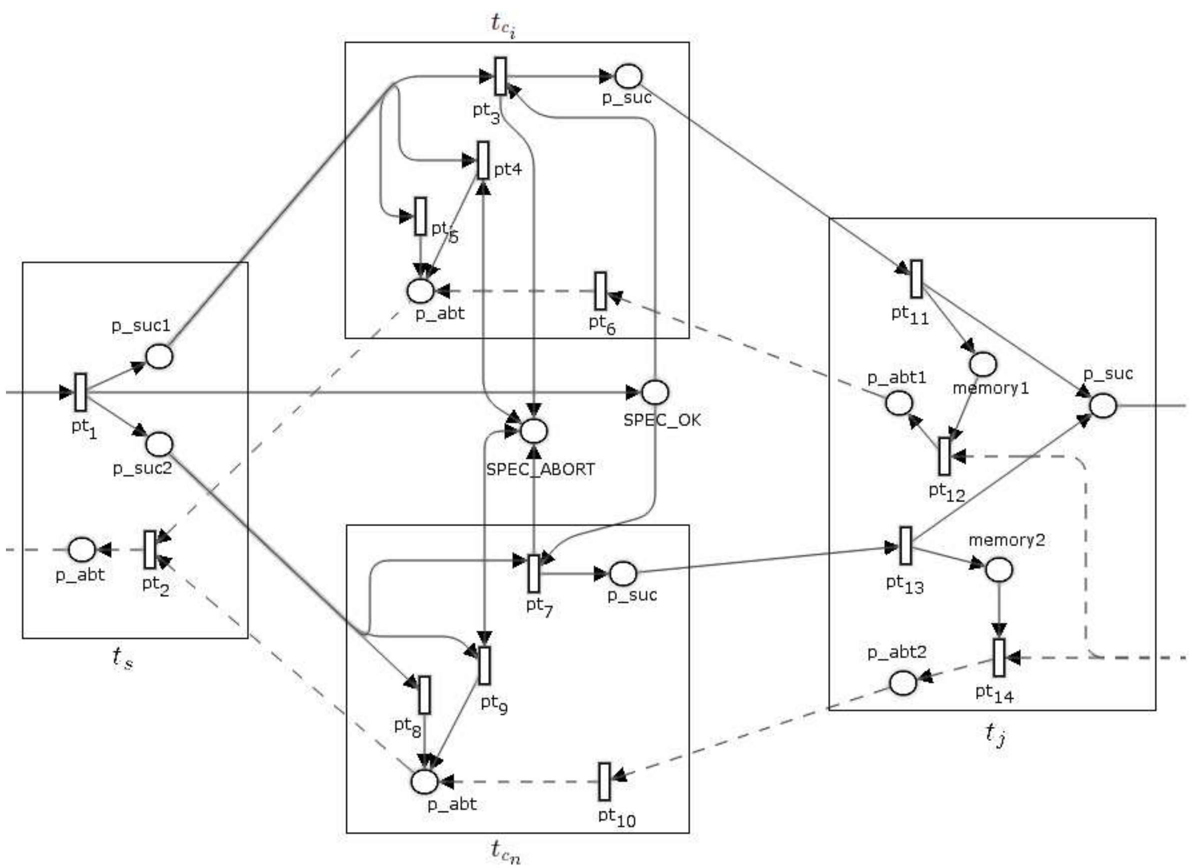

Figure 4.12: Petri net representation of speculative choice composition

Note that, if one task entered the *aborted* state before either task has completed then the other task will continue to operate.

It is important to note that the speculative choice is a unique operator with respect to the structural soundness of the whole Petri net. Let $T_c = t_{c_1} \otimes t_{c_2} ... \otimes t_{c_n}$ ($n \geq 2$ is a finite integer); T_c can be deemed as successful if t_{c_i} ($1 \leq i \leq n$) succeeds and all other tasks are compensated. However, only when T_c is aborted can the compensation flow proceed to the task immediate preceding T_c. Therefore, the tokens in all of the compensated subtasks will remain in their `p_abt` places. As this situation will not affect the success of the overall workflow, we consider these tokens as *invisible* and will ignore them in the discussion of structural soundness (see the next section).

4.3 Analysis

The definition of soundness for CWF-nets is adapted from [43]. Informally, the soundness of a CWF-net requires that for any case, the underlying Petri net will terminate eventually, and at the moment it terminates, there is a token in the output condition and all other places are empty. Formally, the soundness of CWF-nets is defined as follows:

Definition 4.7. *A CWF-net $C_N = (i, o, \mathbb{T}, \mathbb{T}_c, F)$ is sound (or structurally sound) iff, considering the underlying Petri net:*

1. *For every state (marking) M reachable from the initial state M_i, there exists a firing sequence leading from M to the final state M_f, where M_i indicates that there is a token in the input condition and all other places are empty and M_f indicates*

that there is a token in the output condition and all other places are empty;

2. M_f *is the only state reachable from* M_i *with at least one token in the output condition;*

3. *There are no dead transitions in* C_N. Formally: $\forall_{t \in T}, \exists_{M,M'} M_i \rightarrow^* M \rightarrow^t M'$ (where $\rightarrow^*$ denotes 0 or more transitions).

Theorem 4.1. *A CWF-net is sound.*

Proof: Let C_N be a CWF-net which consists of some uncompensable and compensable tasks.

- *Case 1,* C_N consists of only one uncompensable atomic task (t): as t is connected to the input condition and the output condition, t will be activated by the input condition and will continue the forward flow to the output condition. Hence the flow terminates. This is obvious by inspection from Fig. 4.13, which shows the Petri net representation of a CWF-net with an atomic task. This satisfies the three conditions of soundness.

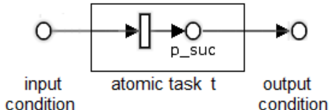

Figure 4.13: CWF-net with one atomic task

- *Case 2,* C_N consists of only atomic uncompensable tasks composed by operators $\{\wedge, \vee, \times, \bullet\}$: according to the construction principle, every type of *split* task must have the corresponding type of *join* task. This pair of *split* and *join* tasks provides

a safe routing for the forward flow; all the tasks of the workflow are on a path from the input condition to the output condition, which ensures that there is no dead transition in the workflow and the flow always terminates. This satisfies the three condition of soundness. Therefore C_N is sound.

- *Case 3,* C_N includes some atomic uncompensable tasks and atomic compensable tasks. First let us consider that C_N has one atomic compensable task (t_c). t_c is activated by some uncompensable atomic task or the input condition. If t_c is successful during the execution, it will activate the next task (or the output condition) by continuing the forward flow. If t_c is aborted, it will start the compensation flow. As this is the only compensable task (by definition it is the initial task, see Definition 4.7), the compensation flow is connected to the next uncompensable task or to the output condition. It is easy to see from Fig. 4.14 that if t_c is aborted, the flow also terminates. An analogous argument holds if C_N has one (nonatomic) compensable task.

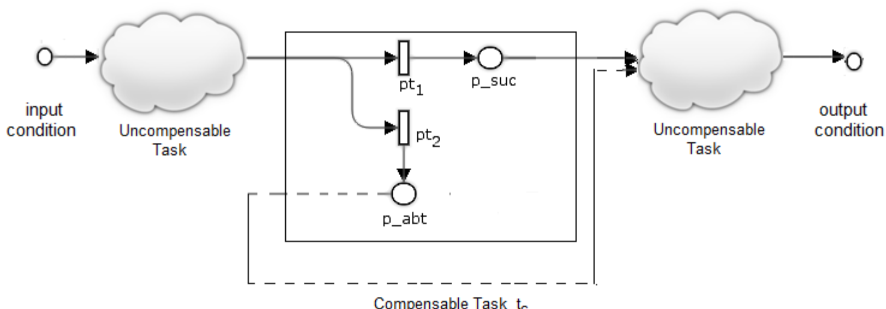

Figure 4.14: CWF-net with one compensable task

Now let us consider there is more than one compensable task in C_N. For every compensable task there is an initial subtask and the compensation flow of the initial subtask is connected to the next uncompensable task or the output condition

(Fig. 4.15). If the compensable tasks do not abort, they will continue the forward flow until the output condition is reached. If the composition of compensable tasks is aborted, the compensation flow will reach the initial subtask, which will direct the compensation flow to the next uncompensable task or the output condition. Therefore it satisfies the conditions of the soundness. Thus C_N is sound.

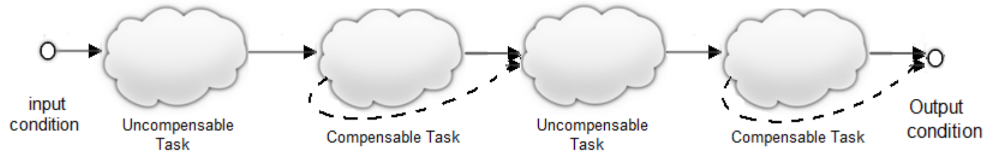

Figure 4.15: CWF-net with more compensable tasks

Chapter 5

Model checking and automated translation

5.1 Model checking

Model checking is an automatic technique for verifying finite-state reactive systems. The overall behaviour of a reactive system is modeled as a transition system. It can be checked whether such a transition system is a model of a temporal logic formula, by a technique originally developed by Clarke and Allen Emerson [14, 15]. Quielle and Sifakis [35] independenty and shortly thereafter discovered a similar verification technique. This technique, known as 'model checking', has several important advantages over mechanical theorem provers or proof checkers for verification of circuits and protocols [13]. The most important is that the procedure is highly automatic. Typically, the user provides a high level representation of the model and the specification to be checked, written in a suitable temporal logic. The model checker will either terminate with the answer true, indicating that the model satisfies the specification, or give a counterexample execution

that shows one execution in which the formula is not satisfied. Such counterexamples are particularly important in finding subtle errors in complex reactive systems.

Kripke structure

A Kripke structure is a type of nondeterministic finite state machine proposed by Saul Kripke in 1963 [23], which is used in model checking to represent the behaviour of a system. It is a graph whose nodes represent the reachable states of the system and whose edges represent state transitions.

Definition 5.1. *Let AP be a non-empty set of atomic propositions. A Kripke structure is a four tuple $M = (S, s_0, R, L)$, where*

- *S is a finite set of states,*
- *s_0 is an initial state,*
- *$R \subseteq S \times S$ is a transition relation, for which it holds that $\forall s \in S : \exists s' \in S : (s, s' \in R)$,*
- *$L : S \to 2^{AP}$ is a function, called the labeling function, which labels each state with the atomic propositions which hold in that state.*

Linear temporal logic

Temporal logic is a particular kind of modal logic. It was introduced by Pnueli [34] in connection with applications to the specification, development and verification of possibly parallel or non-deterministic processes, and uses modal operators to express notions of relative time, such as, “next”, “eventually”, “until”, etc.

LTL is a type of temporal logic which, in addition to classical logical operators, uses the temporal operators such as: always (G), eventually (F), until (U), and next time (X) [22]. A well formed LTL formula, ϕ, is recursively defined by the BNF formula:

$$\phi ::= p \mid \neg\phi \mid \phi \rightarrow \phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid X\ \phi \mid F\ \phi \mid G\ \phi \mid \phi\ U\ \phi$$

where p is a propositional variable. The subset of LTL formula not containing the X operator is denoted as $LTL_{\text{-}X}$. The semantics of LTL are defined with respect to a Kripke model. Let M be a Kripke model, let $\pi = s_0, s_1, ..$ be a path in the model M, let ϕ_1 and ϕ_2 be LTL formulas, and let p be a propositional variable. The notation $M, \pi \models \phi_1$ will be used to mean that formula ϕ_1 holds or is satisfied along the path π in the model M. We say a model M satisfies the formula ϕ, denoted as $M \models \phi$, iff all of its runs, emanating from the initial state s_0, satisfy ϕ. The satisfaction relation, $\models$, is formally defined as follows, where π^i denotes the suffix of the path π starting at s_i:

$$
\begin{array}{lcl}
M, \pi \models p & \Longleftrightarrow & p \in L(s_0) \\
M, \pi \models \neg\phi & \Longleftrightarrow & M, \pi \not\models \phi \\
M, \pi \models \phi_1 \vee \phi_2 & \Longleftrightarrow & M, \pi \models \phi_1 \text{ or } M, \pi \models \phi_2 \\
M, \pi \models X\phi & \Longleftrightarrow & M, \pi^1 \models \phi \\
M, \pi \models G\phi & \Longleftrightarrow & \forall i \geq 0\ M, \pi^i \models \phi \\
M, \pi \models F\phi & \Longleftrightarrow & \exists i \geq 0\ M, \pi^i \models \phi \\
M, \pi \models \phi_1 U \phi_2 & \Longleftrightarrow & \exists k \geq 0\ M, \pi^k \models \phi_2 \text{ and } \forall j,\ 0 \leq j < k,\ M, \pi^j \models \phi_1
\end{array}
$$

5.1.1 The DiVinE model checker and its modeling language

DiVinE is a parallel, distributed-memory explicit-state model checking tool for verification of concurrent systems. The tool employs the aggregate power of network-interconnected clusters to verify systems whose verification is beyond the capability of

sequential tools [1]. The property to be specified is described by an LTL formula. Both the system model and the LTL formula are represented by automata. Then the model checking problem is reduced to detecting in the combined automaton graph whether there is an accepting cycle, i.e., a cycle in which one of the vertices is marked 'accepting' with distributed algorithms assigning different portions of the state space to be explored by different machines. DiVinE can (1) verify much larger system models; (2) finish the verification in significantly less time for larger models (both in comparison with the well-known explicit state LTL model checker SPIN [11]).

DVE is the modeling language of DiVinE. DVE is rich enough to describe systems made of synchronous and asynchronous processes communicating via shared memory. As with Promela (the modeling language of SPIN) a model described in DVE consists of processes, message channels and variables. Each process, identified by a unique name, consists of a list of local variable declarations, process state declarations, an initial state declaration and a list of transitions, each of which starts using the keyword *trans*. Variables can be *global* (declared at the beginning of the DVE source code) or *local* (declared at the beginning of a process), they can be of byte or int type. A transition transfers the process from one state to another. The transition may contain a guard (which decides whether the transition can be executed), a synchronization (which communicates data with another process) and effects (which assign new values to local or global variables). A guard contains the keyword *guard* followed by a Boolean expression and an *effect* contains the keyword *effect* followed by a list of assignments.

5.2 Workflow translation to a model checker

Once a workflow is designed with compensable tasks, its properties can be verified by model checkers such as SPIN, SMV or DiVinE. Modeling a workflow with the input language of a model checker is tedious and error-prone. Leyla et al. [25] translated a number of established workflow patterns into DVE for verifying properties of workflow models. The translation process shown in [25] was a manual translation. Rabbi et al. proposed an automatic translator in [36] which translates a graphical workflow model constructed using the YAWL editor to DVE. Here we give the translation from compensable workflow nets modeled in CWML to DVE. It was shown in chapter 4 that each workflow task of CWML has a Petri net structure. If each workflow component of a workflow model is represented by a Petri net model, the whole workflow is represented by a Petri net model.

The NOVA Translator automatically translates a workflow from CWML to DVE, the input language of the DiVinE model checker. In order to show that the translation is correct, it is sufficient to show that a Petri net model (i.e., a compensable workflow net modeled as a Petri net) can be correctly translated to a DiVinE model. Let us go through some basic definitions first.

Definition 5.2. *Let N be a Petri net structure. For each $t \in T$:*

1. $^{\bullet}t = \{p \mid p\ F\ t\}$ *is called the preset of t,*
2. $t^{\bullet} = \{p \mid t\ F\ p\}$ *is called the postset of t*

Rule 1. *The firing rules of a Petri net are as follows:*

1. *A transition t is said to be* **`ready`** *if each input place p of t is marked with at least w(p,t) tokens, where w(p,t) is the weight of the arc from p to t,*

$$t \text{ is ready iff, } \forall_{p \in {}^{\bullet}t}\ M(p) \geq w(p,t)\ .$$

2. *A* **`ready`** *transition may or may not fire (depending on whether or not the event actually takes place).*

3. *A firing of a* **`ready`** *transition t removes w(p,t) tokens from each input place p of t, and adds w(t,p) tokens to each output place p of t, where w(t,p) is the weight of the arc from t to p.*

The set of all `ready` transitions for a marking M is denoted by $T_{ready(M)}$. If a transition t is `ready` with marking M, $ready(t, M)$ is true, otherwise it is false.

Definition 5.3. *Let N be a Petri net structure, and M a marking of N. The marking M′ to N, obtained from M by firing transaction t, where* $t \in T_{ready(M)}$*, written* $M\ [\rangle_t\ M'$*, is defined as:*

$$\forall_{p \in P_N} M'(p) = \begin{cases} M(p) + w(t,p) & \text{if p} \in t^{\bullet} \\ M(p) - w(p,t) & \text{if p} \in {}^{\bullet}t \\ M(p) & \text{otherwise} \end{cases}$$

The pre-condition (ψ_{pre}) of a task T is the condition for its execution. The pre-condition ψ_{pre} is encoded into the Petri net model as $w(p,t)$. On the other hand, the action ψ_{act} is encoded into the Petri net as $w(t,p)$. let $\pi = M_0[\rangle_{t_j} M'_1[\rangle_{t_k}....$ be a path in a Petri net model $PM = (N, M_0)$, let ϕ_1 and ϕ_2 be LTL formulas, and let p be a propositional variable. The notation $PM, \pi \vDash \phi_1$ will be used to mean that formula ϕ_1

holds or is satisfied along the path π in the model PM. We say a model PM satisfies the formula ϕ, denoted as $PM \vDash \phi$, iff all of its runs, emanating from the initial marking M_0, satisfy ϕ. The satisfaction relation, $\vDash$ for Petri net, is defined in a manner similar to that for the satisfaction relation of a Kripke model in section 5.1. $PM, \pi \vDash p$ *iff* $PM, M_0 \vDash p$ which means that there is a token in place p. Now we define the DiVinE model and provide the translation principle:

Definition 5.4. *A DVE Petri net model is an 8-tuple,* $DM = (V, Proc, T, G, E, F, W, S_0)$ *where:*

- $V = \{var_1, var_2, ..\}$ *is a finite set of variables,*
- $Proc = \{Proc_1, Proc_2, ...\}$ *is a finite set of processes,*
- $T = \{t_1, t_2, ...\}$ *is a finite set of transitions,*
- $G \subseteq (V \times T)$ *is a set of guards,*
- $E \subseteq (T \times V)$ *is a set of effects,*
- $F = (G \cup E)$ *is a set of flow relations,*
- $W : F \rightarrow \{1, 2, 3...\}$ *is a weight function,*
- $S_0 : V \rightarrow \{1, 2, 3, ...\}$ *is the initial marking.*

A 7-tuple $D = (V, Proc, T, G, E, F, W)$ *is called a DVE Petri net structure (no specific initial marking).*

Remark: Generally a DVE model can have other features (i.e., channels, arrays, etc.) [1] but we do not require these features here.

Definition 5.5. *The state S_i of a DVE Petri net model is determined by:*

$S_i = \{(var_{p_0},\ S_i(var_{p_0})),\ (var_{p_1},\ S_i(var_{p_1})),\ .\ .\ (var_{p_n},\ S_i(var_{p_n}))\}$, *where* var_{p_v} *is a DVE variable, and* $S_i(var_{p_v})$ *is the value of the variable* var_{p_v}.

5.2.1 Petri net to DVE translation

Translation Principle 1. *A Petri net model PM* (N, M_0) *is translated to a DVE Petri net model DM* (D, S_0) *by the following rules:*

- *for each place* $p_i \in P_N$, *there corresponds a variable* var_i *in DM; the initial value of the variables are set with the initial marking* M_0 *of the Petri net,*

$$S_0(var_i) = M_0(p_i).$$

- *for each transition* $t_i \in T_N$, *there corresponds a process* $Proc_i$ *in DM;* $Proc_i$ *has a transition* t'_i*; the guard and effect of* t'_i *are determined by the weight function of* t_i*;*

- *a transition* t' *in a DVE Petri net model is* ***ready*** *if it satisfies the following guard condition:*

$$\forall_{v \in {}^\bullet t'} S(var_v) \geq w(v, t'),\ \text{where } {}^\bullet t' = \{v \mid v\ G\ t'\} \text{ and } w(v, t') = w(p, t);$$

  if t' *is ready at state S,* $ready(t', S)$ *is true, otherwise it is false;*

- *the firing of a ready transition t' changes the state of a DVE Petri net model. The new state S' is obtained from S by the firing of t'; the path $S\ [\rangle_{t'}\ S'$ is defined formally as:*

$$\forall_{v \in V} S'(v) = \begin{cases} S(v) + w(t', v) & \textit{if } v \in t'^{\bullet} \\ S(v) - w(v, t') & \textit{if } p \in {}^{\bullet}t' \\ S(v) & \textit{otherwise} \end{cases}$$

  Note that the translated weight function preserves the source Petri net's weight information; hence $\forall_{p \in P_N} w(p, t) = w(var_p, t')$ and $\forall_{p \in P_N} w(t, p) = w(t', var_p)$;

Note that the satisfaction relation, $\models$ for a DVE model $DM = (D, S_0)$ is identical to the satisfaction relation of a Petri net model. Algorithm 1 is the algorithm to translate a Petri net model to DVE.

We will describe the translation using a simple example. Fig. 5.1 shows a Petri net with four places *P1, P2, P3, P4,* two transitions *t1, t2,* six arcs *(p1, t1), (p2, t1), (p2, t2), (t1, p3), (t1, p4), (t2, p4).* Initially *p1 and p2* have *4* and *3* tokens respectively. Each arc has a weight that is specified above the arc.

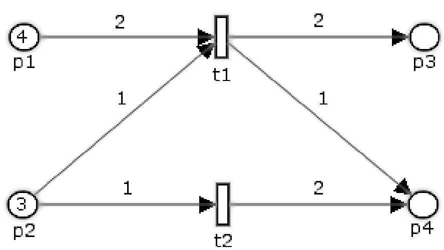

Figure 5.1: A Petri net

The translated DVE model will have four variables (i.e., *var_p1, var_p2, var_p3, var_p4*). *var_p1* and *var_p2* will be assigned with *4* and *3* as initial values. The tran-

Algorithm 1: Translation of a Petri net model to a DVE model

```
Input: Petri net model (P_N)
Result: DVE model (D_N)
dveCode = initialize();
for p ∈ P do
    dveCode += GetVariableStatement( p, M_0(p));
for t ∈ T do
    processStr = GetProcessStatement( t );
    guardStmt = initialize();
    effectStmt = initialize();
    for p ∈ •t do
        guardStmt.append(var(p), "≥", w(p,t));
        effectStmt.append(decrStmt(var(p), w(p,t)));
    for p ∈ t• do
        effectStmt.append(incrStmt(var(p), w(t,p)));
    processStr += guardStatement + "; " + effectStatement + "; }; }";
    dveCode += processStr;
```

sitions *t1* and *t2* will be translated as processes *Proc_t1* and *Proc_t2*, respectively. For our example in Fig. 5.1, the guard condition of process *Proc_t1* will be ($var_p1 \geq 2$ *and* $var_p2 \geq 1$), as transition *t1* has two incoming arcs connected with *p1* and *p2* where $w(p1, t1) = 2$ and $w(p2, t1) = 1$ (see Fig. 5.1). On the other hand, *Proc_t1* will increase the value of *var_p3* and *var_p4* by *2* and *1* respectively as $w(t1, p3) = 2$ and $w(t1, p4) = 1$. The DVE code for the DVE Petri net model shown in Fig. 5.1 is provided here:

```
int var_p1 = 4;
int var_p2 = 3;
int var_p3 = 0;
int var_p4 = 0;


process Proc_t1{
state tr;
init  tr;
trans
tr -> tr{ guard var_p1 >= 2 & var_p2 >= 1 ;
  effect var_p1 = var_p1 - 2, var_p2 = var_p2 - 1,
         var_p3 = var_p3 + 2, var_p4 = var_p4 + 1;
};
}
process Proc_t2{
state tr;
init  tr;
trans
tr -> tr{ guard var_p2 >= 1 ;
```

```
  effect var_p2 = var_p2 - 1,
         var_p4 = var_p4 + 1;
};
}

system async;
```

5.2.2 Proof of correctness

Definition 5.6. *Let PM (N, M_0) be a Petri net model and DM (D, S_0) be a DVE Petri net model. A Petri net state M_i of PM, and a DVE state S_i of DM are equivalent, denoted $M_i \cong S_i$ iff:*

$$\forall_{p \in P_N} M_i(p) = S_i(var_p), \text{ where } var_p \text{ is the variable corresponding to place } p$$

Definition 5.7. *A path $\pi = M_0[\rangle_{t_j} M_1[\rangle_{t_k} ...$ in a Petri net model PM and path $\pi' = S_0[\rangle_{t'_j} S_1[\rangle_{t'_k} ...$ in a DVE Petri net model DM correspond written $(\pi \cong \pi')$ iff $\forall i \geq 1$ $M_i \cong S_i$.*

Remark: If two paths π and π' correspond then for all i, π^i and π'^i correspond.

Definition 5.8. *A Petri net model PM (N, M_0) and a DVE Petri net model DM (D, S_0) are equivalent $(PM \cong DM)$ iff:*

- $M_0 \cong S_0$,

- *for every path starting from* M_0 $(\pi = M_0[\rangle_{t_i} M_1[\rangle_{t_j} ...)$ *there is a corresponding path starting from* S_0*,* $(\pi' = S_0[\rangle_{t'_i} S_1[\rangle_{t'_j} ...)$ *and for every path starting from* S_0 *there is a corresponding path starting from* M_0*.*

Theorem 5.1. *If* DM *is the DVE translation of a Petri net* PM*, then* $PM \cong DM$*.*

Proof: Let $PM = (N, M_0)$ be a Petri net model and $DM = (D, S_0)$ be the DVE model that we get after the translation of PM.

$S_0 = \{(var_{p_0}, a_1), (var_{p_1}, a_2), ...(var_{p_n}, a_n)\}$, where

$\forall_{p_i, 0 \leq i \leq n} S_0(var_{p_i}) = M_0\ (p_i)$

Let π be a path in PM, we will show by induction on the number of transitions in π that π', the translation of π, corresponds to π.

Base Case: Show: $M_0 \cong S_0$.

The initial marking M_0 of PM and S_0 of DM are equivalent as:

$$\forall_{p \in P_N} M_0(p) = S_0(var_p)$$

hence $M_0 \cong S_0$; this proves our base case.

Induction step: Show that if for all paths π of length k in PM there is a corresponding path π' of length k in DM, then for all path π of length $k+1$ in PM, there is a corresponding path π' of length $k+1$ in DM.

Let us assume that for any path π of length k, $M_k \cong S_k$ (induction hypothesis). So we have

$$\forall_{t \in T_{ready(M_k)}} ready(Proc_t, S_k) = true.$$

If any of the transition $t \in T_{ready(M_k)}$ fires, we will get the following changes to the marking:

$\forall_{p \in {}^{\bullet}t} M_{k+1}(p) = M_k(p) - w(p,t)$, and $\forall_{p \in t^{\bullet}} M_{k+1}(p) = M_k(p) + w(t,p)$.

Similarly the DVE process $Proc_t$ for the transition t' will change the values of the variables as follows:

$\forall_{v \in {}^{\bullet}t'} S_{k+1}(v) = S_k(v) - w(v,t')$, and $\forall_{v \in t'^{\bullet}} S_{k+1}(v) = S_k(v) + w(t',v)$.

Again, by Translation Principle 1 (see section 5.2.1), we may conclude:

$$\forall_{p \in P_N} M_{k+1}(p) = S_{k+1}(var_p).$$

hence $M_{k+1} \cong S_{k+1}$; this proves our induction step. Therefore it is established that for every path $\pi \in PM$ starting from M_0, there is a corresponding path $\pi' \in DM$ starting from S_0.

Similarly we can show that if DM is the translation of PM, where S_0 corresponds to M_0, then for every path π' in DM starting from S_0, there is a corresponding path π in PM starting from M_0. Let π' be a path in DM starting from S_0; we will show that π' corresponds to π by doing induction on the number of transitions in π'

Base Case: Show $S_0 \cong M_0$.

By the Translation Principle 1, the initial state S_0 of DM and the initial marking M_0 of PM are equivalent. Which proves our base case.

Assuming for all path π' of length k in DM there is a corresponding path π of length k in PM (induction hypothesis), we get $S_k \cong M_k$. For each ready transition $t' \in T_{ready(S_k)}$ there is a transition t in PM which is ready for marking M_k. If any transition $t' \in T_{ready(S_k)}$ fires, it will make the following changes in DM:

$\forall_{v \in {}^{\bullet}t'} S_{k+1}(v) = S_k(v) - w(v,t')$ and $\forall_{v \in t'^{\bullet}} S_{k+1}(v) = S_k(v) + w(t',v)$.

The firing of t''s corresponding transition t in PM will make the following changes: $\forall_{p \in {}^\bullet t} M_{k+1}(p) = M_k(p) - w(p,t)$ and $\forall_{p \in t^\bullet} M_{k+1}(p) = M_k(p) + w(t,p)$.

As $\forall_{v \in V} S_{k+1}(v) = M_{k+1}(p_v)$, $S_{k+1} \cong M_{k+1}$, which proves our induction step.

Therefore we can conclude that for any path $\pi' = S_0[\rangle_{t'_i} S_1[\rangle_{t'_j}...$ in DM starting from S_0, there is a corresponding path $\pi = M_0[\rangle_{t_i} M_1[\rangle_{t_j}...$ starting from M_0 in PM. As both of the models have the same state space, so $PM \cong DM$.

Proposition 1. *Let π be a path in PM corresponding to a path π' in DM. Then for any LTL formula ϕ, $\pi \vDash \phi$ iff $\pi' \vDash \phi$.*

Proof: By structural induction on the formula ϕ.

The p be a propositional variable and let σ and τ be propositional formulas.

Let $\pi = M_0[\rangle_{t_i} M_1[\rangle_{t_j} M_2...$

$\pi \vDash p$ iff $M_0 \vDash p$ iff $S_0 \vDash p$ (since $M_0 \cong S_0$) iff $\pi' \vDash p$

$\pi \models \neg\sigma$ iff $M_0 \vDash \neg\sigma$ iff $M_0 \nvDash \sigma$ iff $S_0 \nvDash \sigma$ (induction) iff $S_0 \vDash \neg\sigma$ iff $\pi' \vDash \neg\sigma$

The case $\phi = \sigma \vee \tau$ is analogous to the above.

Now consider formulas with temporal operators:

$\pi \vDash X\sigma$ iff $\pi^1 \vDash \sigma$ iff $\pi^{1'} \vDash \sigma$ (since π corresponds to π' so π^1 corresponds to $\pi^{1'}$, by the induction hypothesis) iff $\pi' \vDash X\sigma$

$\pi \vDash \sigma U \tau$ iff $\exists k$ such that $\forall_{0 \leq i < k} \pi^i \vDash \sigma$ and $\pi^k \vDash \tau$. Since π corresponds to π', π^i corresponds to $\pi^{i'}$ and π^k corresponds to $\pi^{k'}$. So by induction, $\forall_{0 \leq i < k} \pi^{i'} \vDash \sigma$ and $\pi^{k'} \vDash \tau$. Hence $\pi' \vDash \sigma U \tau$.

The cases $\pi = F\ \sigma$ and $\pi = G\ \sigma$ are established in a similar manner.

The case $\pi' \vDash \phi'$ implies $\pi \vDash \phi$ is established similarly.

Theorem 5.2. *Let ϕ be an LTL formula, and let $PM \cong DM$. Then $PM \vDash \phi$ iff $DM \vDash \phi$.*

Proof: Suppose $PM \vDash \phi$ for some LTL formula ϕ. Then for all paths π in PM starting at M_0, $\pi \vDash \phi$. We wish to show that for all paths π' in DM starting at S_0, $\pi' \vDash \phi$.

Let π' be a path in DM starting at S_0. Since $PM \cong DM$, there is a path π in PM starting at M_0 and corresponding to π'. By assumption, $\pi \vDash \phi$. The previous proposition allows us to conclude that $\pi' \vDash \phi$.

Analogously we may show that if $DM \vDash \phi$ then $PM \vDash \phi$.

Chapter 6

Workflow model reduction

Model checking [22] or other such verification techniques are required to ensure that the process model exhibits the desired behavior. While current model checkers are much more powerful than their predecessors, they still suffer from the state explosion problem [22]. This chapter describes a workflow reduction algorithm. The workflow reduction is based on the dependency relation that exists among task variables and the order of execution of the tasks. This reduction method specifies which tasks or variables should be included in the reduced model and which should not. Work related to handling the state explosion are presented in the following section.

6.1 Related work

6.1.1 Partial order reduction

The partial order [22] reduction reduces the size of the state space that needs to be searched by model checking algorithms. It constructs a reduced state graph where the behaviors (i.e., paths) of the reduced graph are a subset of the behaviors of the full state

graph. It ensures that if a behavior is not present in the reduced state graph, then a behavior equivalent with respect to a property being verified is included. This reduction technique is best suited for asynchronous systems. We will discuss this method in detail as it served as inspiration for our results.

A common observation about *concurrent asynchronous systems* is that the interleaving model imposes an arbitrary ordering between concurrent events. To avoid discriminating against any particular ordering, the events are interleaved in all possible ways. The ordering between independent transitions is often largely meaningless. However, common specification languages, including many temporal logics, can distinguish between behaviors that only differs in this manner. Partial order reduction aims to take advantage of the cases where the specifications do not distinguish between such behaviors. In these cases, the partial order reduction only checks a subset of the behaviors. However, it checks sufficiently many of them to guarantee the soundness of the verification [22].

Definition 6.1. *A state transition system is a quadruple* (S,T,S_0,L) *where*

- S *is the set of states,*
- S_0 *is the set of initial states,*
- $L : S \rightarrow 2^{AP}$ *is the labeling function that labels each state with a set of atomic propositions,*
- T *is a set of transitions such that for each* $\alpha \in T$, $\alpha \subseteq S \times S$.

A Kripke structure $M = (S, R, S_0, L)$ may be obtained by defining R so that $R(s, s')$ holds when there exists a transition $\alpha \in T$ such that $\alpha(s, s')$. For a transition $\alpha \in T$, we

say that α is enabled in a state s if there is a state s' such that $\alpha(s, s')$ holds. Otherwise α is disabled in s. The set of transitions enabled in s is $enabled(s)$.

Depth First Search (DFS) with partial order reduction

The reduction is performed by the DFS used to construct the state graph, as in Algorithm 2. The search starts with an initial state s_0 and proceeds recursively. For each state s it selects only a subset $ample(s)$ of the enabled transitions $enabled(s)$, rather than the full set of enabled transitions, as in the full state space construction. The DFS explores only successors generated by these transitions.

Algorithm 2: Depth-first search with partial order reduction [22]

```
hash(s0);
set on_stack(s0);
expand_state(s0);
procedure expand_state(s)
work_set(s) := ample(s);
while work_set(s) is not empty do
    let α ∈ work_set(s);
    work_set(s) := work_set(s) \ {α};
    s' := α(s);
    if new(s') then
        hash(s');
        set on_stack(s');
        expand_state(s');
    create_edge(s, α, s');
set completed(s);
end procedure
```

Definition 6.2. *An Independence relation $I \subseteq T \times T$ is a symmetric, antireflexive relation, satisfying the following two conditions for each state $s \in S$ and for each $(\alpha,\beta) \in I$:*

- *Enabledness: If α , $\beta \in$ enabled(s) then $\alpha \in$ enabled(β(s)).*
- *Commutativity: α , $\beta \in$ enabled(s) then $\alpha(\beta(s)) = \beta(\alpha(s))$.*

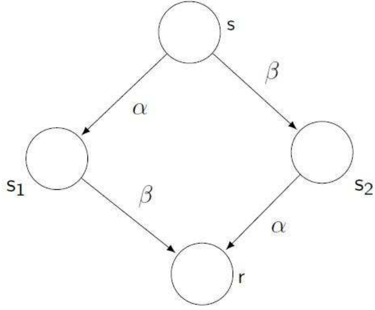

Figure 6.1: Execution of independent transitions

The Dependency relation D is the complement of I, namely $D = (T \times T) \setminus I$.

A transition $\alpha \in T$ is invisible with respect to a set of propositions $AP' \subseteq AP$ if for each pair of states $s, s' \in S$ such that $s' = \alpha(s), L(s) \cap AP' = L(s') \cap AP'$. In other words, a transition is invisible when its execution from any state does not change the value of the propositional variable in AP'. A transition is visible if it is not invisible.

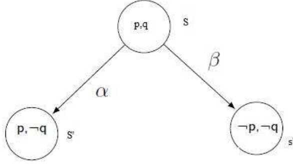

Figure 6.2: If AP' = {p} then α is invisible

Stuttering equivalent paths

Two infinite paths $\sigma = s_0 \stackrel{\alpha_0}{\rightarrow} s_1 \stackrel{\alpha_1}{\rightarrow} ...$ and $\rho = r_0 \stackrel{\beta_0}{\rightarrow} r_0 \stackrel{\beta_0}{\rightarrow} ...$ are stuttering equivalent ($\sigma \sim_{st} \rho$) if there are two infinite sequences of positive integers $0 = \mathrm{i}_0 < i_1 < i_2 < ...$ and $0 = j_0 < j_1 < j_2 ...$ such that for every $\mathrm{k} \geq 0$,

$\mathrm{L}(\mathrm{s}_{i_k}) = \mathrm{L}(\mathrm{s}_{i_k+1}) = ... = \mathrm{L}(\mathrm{s}_{i_{k+1}-1}) = \mathrm{L}(\mathrm{r}_{j_k}) = \mathrm{L}(\mathrm{r}_{j_k+1}) = ... = \mathrm{L}(\mathrm{r}_{j_{k+1}-1})$

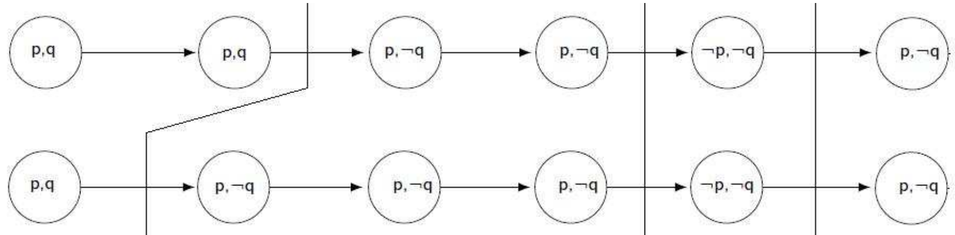

Figure 6.3: Two stuttering equivalent paths

In other words, the paths can be partitioned into infinitely many blocks, such that the states in the kth block of one are labeled the same as the states in the kth block of the other. The blocks can have different lengths. An LTL formula f is invariant under stuttering if and only if for each pair of paths π and π' such that $\pi \sim_{st} \pi'$,

$\pi| = f$ if and only if $\pi'| = f$

We denote the subset of the logic LTL without the next time operator by LTL_{-X}.

Theorem 6.1. *Any LTL_{-X} property is invariant under stuttering.*

The theorem is proved using a simple induction on the size of the LTL formula [22].

Partial order reduction for LTL_{-X}

When the specification is invariant under stuttering (i.e., formula in LTL_{-X}), partial order reduction can use commutativity and invisibility which avoids generating some

states. In [13] author suggests a systematic way of selecting an ample set of transitions for any given state. The *ample* sets will be used by the DFS algorithm in Algorithm 2 to construct a reduced state graph so that for every path not considered by the DFS algorithm there is a stuttering equivalent path that is considered. This guarantees that the reduced state graph is stuttering equivalent to the full state graph. There are four conditions for selecting $ample(s) \subseteq enabled(s)$ by which we can make sure that the satisfaction of the LTL_{-X} specification is preserved [22].

Condition C0: If a state has at least one successor, then the reduced state graph also contains a successor for this state; i.e.:

C0 $ample(s) = \phi$ if and only if $enabled(s) = \phi$

Condition C1: Along every path in the full state graph that starts at s, the following condition holds:

a transition that is dependent on a transition in $ample(s)$ cannot be executed without a transition in $ample(s)$ occurring first.

Note: C1 refers to paths in the full state graph

Condition C2 [Invisibility]: If s is not fully expanded, then every $\alpha \in ample(s)$ is invisible.

Condition C3 [Cycle condition]: A cycle is not allowed if it contains a state in which some transition α is enabled, but is never included in $ample(s)$ for any state s on the cycle.

The complexity of checking the conditions

Condition C0 for a particular state can be checked in constant time. Condition C2 is also simple to check, by examining the transitions in the set. Condition C1 is a constraint

that is not immediately checkable by examining the current state of the search, in that it refers to future states (some of which need not even be in the reduced state graph). The next theorem shows that, in general, checking C1 is at least as hard as searching the full state space.

Theorem 6.2. *Checking Condition C1 for a state s and a set of transitions $T \subseteq$ enabled(s) is at least as hard as checking* **`Reachability`** *for the full state space.*

Proof: see [13].

6.1.2 Other work

In [38] [39] the authors provided reduction rules as a stand alone approach to reduce the complexity of the process model. In [45] [31] the authors applied similar reduction techniques as an engineering approach. A reduction procedure for BPMN graphs was provided in [10] but the authors did not provide any reduction algorithm for workflows with variables. In none of these approaches is there a proof of stuttering equivalence.

6.2 Workflow model reduction

In this section we will discuss a workflow reduction algorithm for the CWML. The reduction algorithm is based on the property subject to verification and the workflow model. The workflow reduction method reads both the workflow file and the file containing an LTL specification (called the LTL-property file) and based on the specification that we want to verify, reduces the workflow. The reduced workflow is then translated into DVE, the input language of DiVinE. The algorithm is provided in this section. In chapter 4

we gave the definition of tasks, and compensatable tasks. A task can be represented by a syntax tree where a non-leaf node represents an operator and a leaf node represents an atomic task.

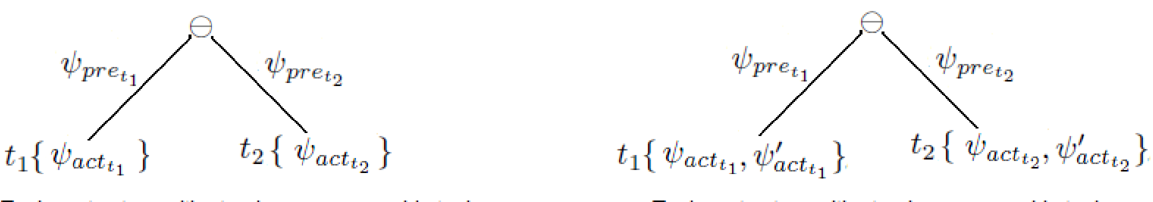

Figure 6.4: Example of a task syntax tree

In chapter 4 we defined a workflow net which consists of an input condition, tasks and an output condition; indeed, excluding the input and output conditions, a workflow can be viewed as one complex task. Thus a workflow can be represented by a task syntax tree. For a given workflow model M, we build the syntax tree putting pre-conditions on the edges of the tree and then reduce the tree based on the LTL-property that we want to verify. In workflow reduction process, we mark the variables specified in the LTL-property. We define a set *pElmnts*, to store tasks, pre-conditions, variables, actions, etc., that will be preserved in the reduced workflow. *pElmnts* is calculated in Algorithm 3.

Definition 6.3. *Let ϕ be an LTL property and let V' be the set of propositional variables occurring in ϕ; the Visible Label function (L_ϕ) is defined as follows:*

$L_\phi(s) = L(s) \cap V'$, *where s is a state.*

The definition of *visible pre-condition*, *action* and *task* are given here:

Definition 6.4. ***(visible pre-condition, action and task)*** *(a) A pre-condition (ψ_{pre}) is visible iff there exists a variable in ψ_{pre} which is in pElmnts. If a pre-condition ψ_{pre} is visible, we say $visible(\psi_{pre}) = true$.*

(b) An action (ψ_{act}) is visible iff the mapsTo variable of ψ_{act} is in pElmnts. If an action ψ_{act} is visible, we say $visible(\psi_{act}) = true$.

(c) A task t is visible iff there exists an action of t, ψ_{act_t}, such that $visible(\psi_{act_t}) = true$. If a task t is visible, we say $visible(t) = true$.

If a pre-condition, task or action is not visible, we say that it is *invisible*. The reduction algorithm is provided in Algorithm 3. For a given workflow model M and LTL-property ϕ, the algorithm constructs a syntax tree τ from M. The set *pElmnts* is initialized with the empty set and the variables occurring in the specification ϕ are inserted into it. Then the algorithm recursively inserts those tasks, actions, and pre-conditions into *pElmnts* which are visible in τ and continues until there are no visible elements left in τ to be added. The algorithm constructs the reduced syntax tree τ' and the reduced model M' using the elements in *pElmnts*. The construction of M' from τ' is straightforward as a task syntax tree represents formula and a workflow can be generated from the formula.

Example

Fig. 6.5 gives a workflow model M_{ex} containing 11 atomic tasks. The property we want to verify for this workflow is: $G((v_1 == 1) \rightarrow F(v_2 == 1))$, meaning if v_1 is set with value '1', v_2 will eventually be set with value '2'. Task pre-conditions are shown above the edges and task actions are shown below the tasks. We will reduce this workflow according to the reduction algorithm.

The following formulas represent the workflow M_{ex}.

Algorithm 3: Reduction Algorithm

Input: Workflow Model M, LTL specification ϕ

Result: Reduced Workflow Model M'

construct syntax tree τ; **set** $pElmnts = empty$;

for $v \in \phi$ **do**
- $pElmnts.add(v)$;

$size = 0$;

while $size\ != pElmnts.size$ **do**
- **for** $\psi_{pre} \in \tau$ **do**
  - **if** $visible(\psi_{pre}) = true$ **then**
    - $pElmnts.add(\psi_{pre})$;
- **for** *task* $t \in \tau$ **do**
  - **if** $visible(\psi_{act_t}) = true$ **then**
    - $pElmnts.add(t)$; $pElmnts.add(\psi_{act_t})$;
    - $pElmnts.add(\psi_{pre_t})$; $visitingTask := t.parentNode$;
    - **while** $visitingTask\ != rootNode$ **do**
      - $pElmnts.add(visitingTask)$; $pElmnts.add(\psi_{pre_{visitingTask}})$;
      - **if** *visitingTask is an XOR or OR task* **then**
        - $leftNode := visitingTask.leftChild$;
        - $rightNode := visitingTask.rightChild$;
        - $pElmnts.add(\psi_{pre_{leftNode}})$; $pElmnts.add(\psi_{pre_{rightNode}})$;
      - $visitingTask = visitingTask.parentNode$;
- **if** $size = pElmnts.size$ **then**
  - set tree $\tau' := \tau$;
  - **for** *task* $t \in \tau'$ **do**
    - **if** $t \notin pElmnts$ **then**
      - $\psi_{act_t} := NIL$; $t := NIL$; // take out stuff marked as NIL
  - **for** $\psi_{pre} \in \tau'$ **do**
    - **if** $\psi_{pre} \notin pElmnts$ **then**
      - $\psi_{pre} := NIL$;
  - generate M' by visiting τ';
  - return;
- $size = pElmnts.size$;

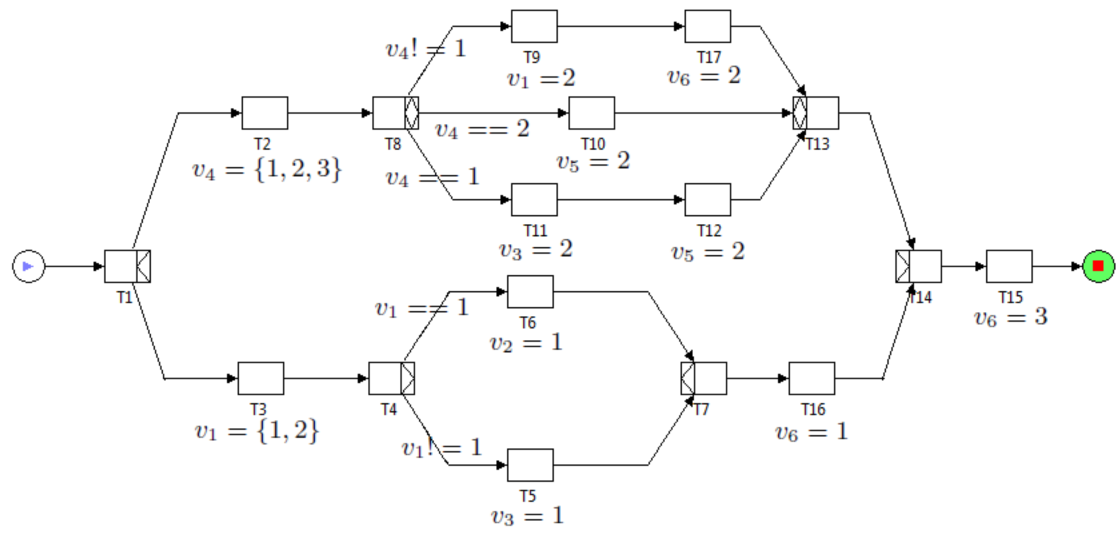

Figure 6.5: The workflow M_{ex}

$T4 = T5 \times T6$

$T8 = ((T11 \bullet T12) \vee T10) \vee (T9 \bullet T17)$

$T1 = (T2 \bullet T8) \wedge (T3 \bullet T4 \bullet T16)$

$T = (T1 \bullet T15)$

Fig. 6.6 shows the task syntax tree for M_{ex}. The pre-conditions of tasks T5 ($v_1! = 1$) and T6 ($v_1 == 1$) are inserted into $pElmnts$ as the variable v_1 occurs in the given LTL specification; this is indicated by thick lines in Fig. 6.6. Tasks T3, T6 and T9 are visible as they have visible actions. Paths from T3, T6 and T9 to the root node become visible and all the pre-conditions along the way become visible (represented by thick lines). As v_4 becomes a visible variable, the pre-conditions $(v_4 == 1), (v_4 == 2), (v_4! = 1)$ become visible. Task T2 becomes a visible task as it has an action ($v_4 = \{1, 2, 3\}$) which changes the value of a visible variable. Nothing more is visible in the tree, hence we stop inserting elements into $pElmnts$.

The reduced workflow M'_{ex} is shown in Fig. 6.7. M'_{ex} has fewer concurrent tasks but

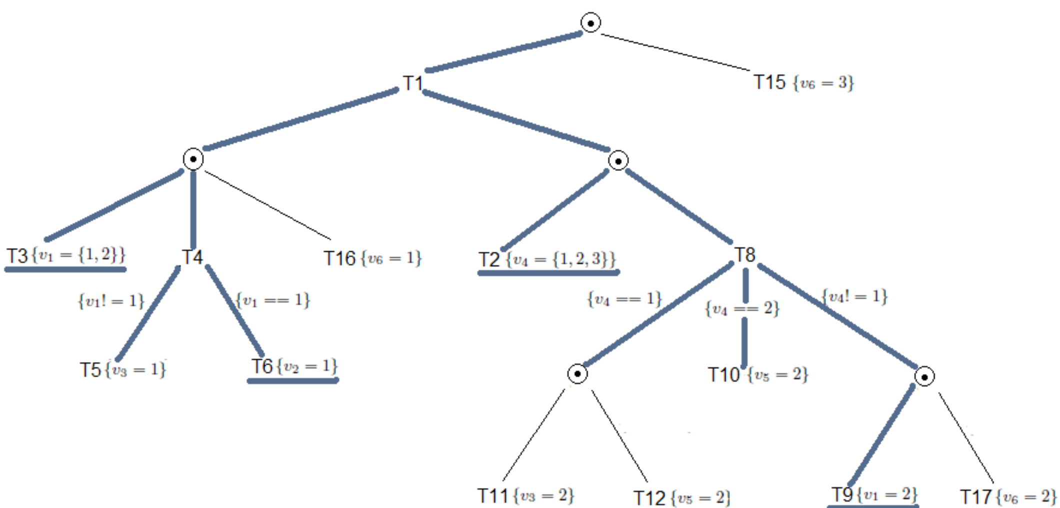

Figure 6.6: The task syntax tree for M_{ex}

will provide the same verification result for the given LTL specification (see the proof of stuttering equivalence in the following section). Experimental result shows that the property does not hold and one of the counter examples is given below:

$s_0 \overset{T1}{\rightarrow} s_1 \overset{T3}{\rightarrow} s_2 \overset{T2}{\rightarrow} s_3 \overset{T8}{\rightarrow} s_4 \overset{T9}{\rightarrow} s_5 \overset{T4}{\rightarrow} s_6 \overset{T7}{\rightarrow} s_7 \ldots$

The counter example shows that the variable v_1 is set with the value '1' in task T3, and it is reset with another value (i.e., $v_1 = 2$) in task T9; for this execution the property $G((v_1 == 1) \rightarrow F(v_2 == 1))$ does not hold in M'_{ex} and hence not in M_{ex}.

6.3 Proof of stuttering equivalence

If each workflow component of a workflow model is represented by a Petri net model, the whole workflow is represented by a Petri net model. A CWF$_{net}$ has one initial state, which is the initial marking of the Petri net.

Definition 6.5. *(adapted from [22]) Two infinite paths* $\sigma = s_0 \overset{\alpha_0}{\rightarrow} s_1 \overset{\alpha_1}{\rightarrow} \ldots$ *and*

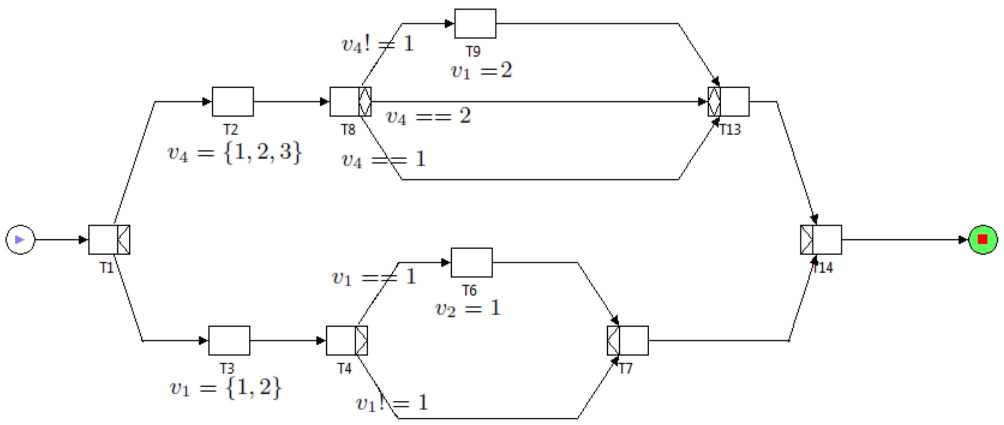

Figure 6.7: The reduced workflow M'_{ex}

$\rho = r_0 \xrightarrow{\beta_0} r_0 \xrightarrow{\beta_0} \ldots$ *are stuttering equivalent* ($\sigma \sim_{st} \rho$) *with respect to the LTL-formula* ϕ *if there are two infinite sequences of positive integers* $0 = i_0 < i_1 < i_2 < \ldots$ *and* $0 = j_0 < j_1 < j_2 \ldots$ *such that for every* $k \geq 0$,

$$L_\phi(s_{i_k}) = L_\phi(s_{i_k+1}) = \ldots = L_\phi(s_{i_{k+1}-1}) = L_\phi(r_{j_k}) = L_\phi(r_{j_k+1}) = \ldots = L_\phi(r_{j_{k+1}-1})$$

(where L_ϕ *is as defined in Definition 6.3).*

Thus $\sigma \sim_{st} \rho$ w.r.t ϕ iff the paths can be partitioned into infinitely many blocks, such that the states in the kth block of σ are lableled (w.r.t ϕ) the same as the states in the kth block of ρ.

Definition 6.6. *(adapted from [13]) Two workflow models* M *and* M' *are stuttering equivalent* ($M \sim_{st} M'$) *with respect to an LTL formula* ϕ*, iff:*

- $L_\phi(s_0) = L_\phi(r_0)$*, where* s_0, r_0 *are the initial states of* M *and* M' *respectively, i.e.,* M *and* M' *have the same set of inital states (one each);*
- *For each path* π *of* M *that starts from the initial state* s_0 *of* M *there exists a path*

σ of M' from the initial state r_0 w.r.t ϕ such that $\pi \sim_{st} \sigma$, and;

- *for each path σ of M' that starts from the initial state r_0 of M' there exists a path π of M from the initial state s_0 w.r.t ϕ such that $\sigma \sim_{st} \pi$.*

Thus, in order to show that the reduction algorithm is correct it is sufficient to show that M and M' are stuttering equivalent. We begin with some Lemmas.

Lemma 1. *Any task below an invisible pre-condition of a task syntax tree is invisible.*

Proof: We will prove this by contradiction. Let t_v be a visible task below an invisible pre-condition ψ_{pre_i}. As t_v is a visible task, then according to Algorithm 3, all pre-conditions from t_v to the root node are recursively added to *pElmnts*. From the definition of visible pre-condition, we know that a pre-condition ψ_{pre} which is in *pElmnts* is a visible pre-condition. Thus if ψ_{pre_i} belongs to the path from t_v to the root node, it cannot be invisible.

Lemma 2. *An invisible task cannot change the flow of a visible task.*

Proof: By definition, an invisible task contains only invisible actions. An invisible action ψ_{act_i} cannot change the value of a visible property. So an invisible action can only change invisible pre-conditions. From Lemma 1 we know that an invisible pre-condition can only change the flow of an invisible task. Thus an invisible action cannot change the control flow of a visible task, which proves that an invisible task cannot change the control flow to a visible task.

Theorem 6.3. *A Workflow model M and its reduced model M' are stuttering equivalent.*

Proof: Let ϕ be an LTL specification and AP' be the set of propositional variables in ϕ. The Workflow model M is reduced to M' according to the Workflow Reduction Algorithm with respect to the specification ϕ. τ and τ' are two syntax trees representing workflows M and M' respectively. We will prove this theorem by doing structural induction on the number of atomic tasks of τ.

Base Case: Let M_1 be a model such that the syntax tree τ_1 has one atomic task $t_1\{\psi_{act_{t_1}}\}$. We have to show that $M_1 \sim_{st} M_1'$ There are two possibilities:

1. $t_1\{\psi_{act_{t_1}}\}$ *is visible:* Since τ_1 has one vertex, and it is visible, there is no reduction. Here, τ' and τ are same; thus M and M' are identical.

2. $t_1\{\psi_{act_{t_1}}\}$ *is invisible:* In this case τ_1' is empty. The only possible path in M_1 is $(\pi = s_0 \xrightarrow{t_1} s_1)$ which is stuttering equivalent to the initial state r_0 of M_1' as $L_\phi(s_0) = L_\phi(s_1) = L_\phi(r_0)$; thus $M_1 \sim_{st} M_1'$.

In either situation, $M_1 \sim_{st} M_1'$.

Induction: Assume that for all models M_k with k atomic tasks in τ_k, $M_k \sim_{st} M_k'$; we have to show that, for all models M_{k+1} with $k+1$ atomic tasks in τ_{k+1}, $M_{k+1} \sim_{st} M_{k+1}'$.

To get a model with $k+1$ atomic tasks we can build one from a model with k atomic tasks, by replacing an atomic task by a task in the form $t_1\{\psi_{act_{t_1}}\} \ominus t_2\{\psi_{act_{t_2}}\}$ where $\ominus \in \{\bullet, \wedge, \times, \vee, ;, ||, \sqcap, \otimes, \rightsquigarrow\}$. Equivalently we can take a syntax tree with k leaf nodes and replace a leaf t_1 with the subtree with two leaf nodes t_1 and t_2. Let a leaf node $t_i\{\psi_{act_{t_i}}\}(1 \leq i \leq k)$ be replaced by a control flow operator $(\ominus_n)$ in τ_k to get a tree τ_{k+1} with $k+1$ atomic tasks. Let $t_n\{\psi_{act_{t_n}}\}$ be a new task added to the right branch of $\ominus_n$ while $t_i\{\psi_{act_{t_i}}\}$ is added to the left branch of $\ominus_n$. This can be visualized in Fig. 6.8.

Here τ_{k+1} is the tree representation of the workflow model M_{k+1} and τ_{k+1}' is the

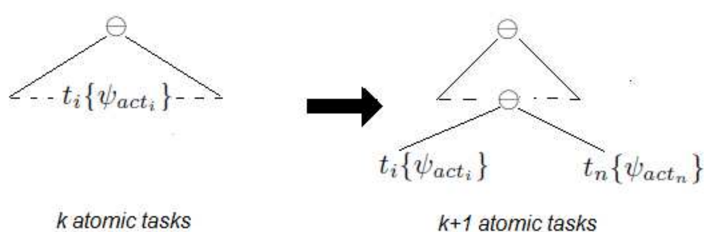

Figure 6.8: Forming a syntax tree of size $k+1$ from one of size k

reduced tree for the reduced workflow model M'_{k+1}. It is obvious that paths $\{\pi_{s_0 \stackrel{t_i}{\not\to}}\}$ (starting from s_0) which do not go through $t_i\{\psi_{act_{t_i}}\}$ in M_k, cannot go through the newly added control flow operator $\ominus_n$ in M_{k+1}. For each path of $\pi_{s_0 \stackrel{t_i}{\not\to}}$ in M_k, there is a stuttering equivalent path $\sigma_{r_0 \stackrel{t_i}{\not\to}}$ (Induction hypothesis) in M'_k starting from the state r_0. Now it is easy to see that those paths $\{\sigma_{r_0 \stackrel{t_i}{\not\to}}\}$ which do not go through $\ominus_n$ are also present in M'_{k+1}. So each path starting from s_0 that does not go through $\ominus_n$ in M_{k+1} has a stuttering equivalent path in M'_{k+1} starting from state r_0.

On the other hand, paths $\{\pi_{s_0 \stackrel{t_i}{\to}}\}$ which go through t_i in M_k, must go through $\ominus_n$ in M_{k+1}. Let $\pi_{s_0 \stackrel{\ominus_n}{\to}}$ be a path which starts from s_0 in M_{k+1} and goes through $\ominus_n$. Decompose $\pi_{s_0 \stackrel{\ominus_n}{\to} s_{end}}$ into two sub paths, $\pi_{s_0 \stackrel{\ominus_n}{\to} s_{\ominus_n}}$ and $\pi_{s_{\ominus_n} \to s_{end}}$, the first goes from state s_0 to the execution of $\ominus_n$ and the second consists of the rest of the path. We must show that $\pi_{s_0 \stackrel{\ominus_n}{\to} s_{end}} \sim_{st} \sigma_{r_0 \stackrel{\ominus_n}{\to} r_{end}}$ where $\sigma_{r_0 \stackrel{\ominus_n}{\to} r_{end}}$ is a path starting from r_0 and goes through the operator $\ominus_n$ in M'_{k+1}. We first consider the operator $\ominus_n = (\bullet)$. The following chart details all four cases, giving the reduced tree τ'_{k+1} and the proof that $\pi_{s_0 \to s_{end}} \sim_{st} \sigma_{r_0 \to r_{end}}$.

Fig. 6.9 shows the Petri net representation of task t_i and t_n composed with operator $(\bullet)$. This Petri net representation helps the reader understand the paths.

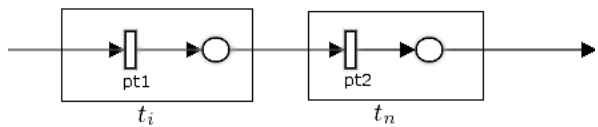

Figure 6.9: Sequential composition (•) of uncompensable atomic tasks

Case	Reduced tree τ'_{k+1}	Paths
$visible(t_i) = true$ $visible(t_n) = false$	$\odot$ $t_i\{\psi_{act_{t_i}}\}$ NIL	$\pi_{\to\ominus} = s_0 \to ...s_\ominus \mid\overset{t_i(pt_1)}{\to} s_{t_i} \overset{t_n(pt_2)}{\to} s_{t_n} \mid\to ...s_{end1}$ $\sigma_{\to\ominus} = r_0 \to ...r_\ominus \mid\overset{t_i(pt_1)}{\to} r_{t_i} \mid\to ...r_{end1}$ Here pt_1, pt_2 are Petri net transitions of t_i and t_n (see Fig. 6.9) $\pi_{s_0\to s_\ominus} \sim_{st} \sigma_{r_0\to r_\ominus}$ (from Induction hypothesis) $\pi_{s_\ominus\to s_{t_n}} \sim_{st} \sigma_{r_\ominus\to r_{t_i}}$ as $L_\phi(s_\ominus) = L_\phi(r_\ominus)$ $L_\phi(s_{t_i}) = L_\phi(s_{t_n}) = L_\phi(r_{t_i})$ $\pi_{s_{t_n}\to s_{end1}} \sim_{st} \sigma_{r_{t_i}\to r_{end1}}$ (Lemma 2) $\pi_{s_0\to s_{end1}} \sim_{st} \sigma_{r_0\to r_{end1}}$ (concatenating)
$visible(t_i) = false$ $visible(t_n) = true$	$\odot$ NIL $t_n\{\psi_{act_{t_n}}\}$	$\pi_{\to\ominus} = s_0 \to ...s_\ominus \overset{t_i(pt_1)}{\to} s_{t_i} \mid\overset{t_n(pt_2)}{\to} s_{t_n} \mid\to ...s_{end2}$ $\sigma_{\to\ominus} = r_0 \to ...r_\ominus \mid\overset{t_n(pt_2)}{\to} r_{t_n} \mid\to ...r_{end2}$ $\pi_{s_0\to s_\ominus} \sim_{st} \sigma_{r_0\to r_\ominus}$ (from Induction hypothesis) $\pi_{s_\ominus\to s_{t_n}} \sim_{st} \sigma_{r_\ominus\to r_{t_n}}$ as $L_\phi(s_\ominus) = L_\phi(r_\ominus) = L_\phi(s_{t_i})$; $L_\phi(s_{t_n}) = L_\phi(r_{t_n})$ $\pi_{s_{t_n}\to s_{end2}} \sim_{st} \sigma_{r_{t_n}\to r_{end2}}$ (Lemma 2) $\pi_{s_0\to s_{end2}} \sim_{st} \sigma_{r_0\to r_{end2}}$ (concatenating)

Case	Reduced tree τ'_{k+1}	Paths
$visible(t_i) = false$ $visible(t_n) = false$	$\odot$ (NIL, NIL)	$\pi_{\rightarrow\ominus} = s_0 \rightarrow ... \mid s_\ominus \overset{t_i(pt_1)}{\rightarrow} s_{t_i} \overset{t_n(pt_2)}{\rightarrow} s_{t_n} \mid\rightarrow ...s_{end3}$ $\sigma_{\rightarrow\ominus} = r_0 \rightarrow ... \mid r_\ominus \mid\rightarrow ...r_{end3}$ $\pi_{s_0 \rightarrow s_\ominus} \sim_{st} \sigma_{r_0 \rightarrow r_\ominus}$ (from Induction hypothesis) $\pi_{s_\ominus \rightarrow s_{t_n}} \sim_{st} \sigma_{r_\ominus}$ as $L_\phi(s_\ominus) = L_\phi(s_{t_i}) = L_\phi(s_{t_n})$ $= L_\phi(r_\ominus)$; $\pi_{s_{t_n} \rightarrow s_{end3}} \sim_{st} \sigma_{r_\ominus \rightarrow r_{end3}}$ (Lemma 2) $\pi_{s_0 \rightarrow s_{end3}} \sim_{st} \sigma_{r_0 \rightarrow r_{end3}}$ (concatenating)
$visible(t_i) = true$ $visible(t_n) = true$	$\odot$ ($t_i\{\psi_{act_{t_i}}\}$, $t_n\{\psi_{act_{t_n}}\}$)	$\pi_{\rightarrow\ominus} = s_0 \rightarrow ...s_\ominus \mid\overset{t_i(pt_1)}{\rightarrow} s_{t_i} \overset{t_n(pt_2)}{\rightarrow} s_{t_n} \mid\rightarrow ...s_{end4}$ $\sigma_{\rightarrow\ominus} = r_0 \rightarrow ...r_\ominus \mid\overset{t_i(pt_1)}{\rightarrow} r_{t_i} \overset{t_n(pt_2)}{\rightarrow} r_{t_n} \mid\rightarrow ...r_{end4}$ $\pi_{s_0 \rightarrow s_{end4}} \sim_{st} \sigma_{r_0 \rightarrow r_{end4}}$ (as no reduction)

For the operator $\ominus_n = (\wedge)$, tasks $t_i\{\psi_{act_{t_i}}\}$ and $t_n\{\psi_{act_{t_n}}\}$ both will execute but they may interleave. If either of them is invisible, it will be removed from the workflow, resulting in a shorter path but stuttering equivalent to a path of the original workflow, as an invisible task cannot change the flow of a visible task. Fig. 4.5 shows the Petri net representation of two tasks t_i and t_n composed with "and" operator $(\wedge)$.

For the operator $\ominus_n = (\times)$, either $t_i\{\psi_{act_{t_i}}\}$ or $t_n\{\psi_{act_{t_n}}\}$ will execute. In the case both t_i and t_n are visible, there is no reduction. We present the proof of only one case; the other two cases are similar. Fig. 4.6 shows the Petri net representation of two tasks t_i and t_n composed with "xor" operator $(\times)$. Fig. 6.10 shows the reduced syntax tree τ'_{k+1} for the case $visible(t_i) = true$ and $visible(t_n) = false$. There are two paths to consider:

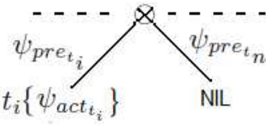

Figure 6.10: Reduced syntax tree τ'_{k+1}

$\pi_{\to\ominus} = s_0 \to ...s_\ominus \overset{t_s(pt_1)}{\to} s_{t_s} |\overset{t_i(pt_2)}{\to} s_{t_i} \overset{t_j(pt_4)}{\to} s_{t_j} |\!\to ...s_{end1}$

$\sigma_{\to\ominus} = r_0 \to ...r_\ominus \overset{t_s(pt_1)}{\to} r_{t_s} |\overset{t_i(pt_2)}{\to} r_{t_i} \overset{t_j(pt_4)}{\to} r_{t_j} |\!\to ...r_{end1}$

$\pi_{s_0 \to s_{end1}} \sim_{st} \sigma_{r_0 \to r_{end1}}$ (as t_i executes in both)

(2) $\pi_{\to\ominus} = s_0 \to ... \mid s_\ominus \overset{t_s(pt_1)}{\to} s_{t_s} \overset{t_n(pt_3)}{\to} s_{t_n} \overset{t_j(pt_5)}{\to} s_{t_j} |\!\to ...s_{end2}$

$\sigma_{\to\ominus} = r_0 \to ... \mid r_\ominus \overset{t_s(pt_1)}{\to} r_{t_s} \overset{t_j(pt_5)}{\to} r_{t_j} |\!\to ...r_{end2}$

$\pi_{s_0 \to s_\ominus} \sim_{st} \sigma_{r_0 \to r_\ominus}$ (from Induction hypothesis)

$\pi_{s_\ominus \to s_{t_j}} \sim_{st} \sigma_{r_\ominus \to r_{t_j}}$ as

$L_\phi(s_\ominus) = L_\phi(s_{t_s}) = L_\phi(s_{t_n}) = L_\phi(s_{t_j}) = L_\phi(r_\ominus) = L_\phi(r_{t_s}) = L_\phi(r_{t_j})$

$\pi_{s_{t_j} \to s_{end2}} \sim_{st} \sigma_{r_{t_j} \to r_{end2}}$ (Lemma 2)

$\pi_{s_0 \to s_{end2}} \sim_{st} \sigma_{r_0 \to r_{end2}}$ (concatenating)

Similarly we can show that for operator ($\vee$), for each path starting from s_0 in M_{k+1} and going through the operator ($\vee$) there exists a stuttering equivalent path starting from r_0 in M'_{k+1}. Fig. 4.7 shows the Petri net representation of two tasks t_i and t_n composed with "or" operator ($\vee$). From the Petri net we can see either t_i and t_n both or only t_i will execute.

Proof for the compensation operators ($;, ||, \sqcap, \otimes, \rightsquigarrow$) are more complex as the forward and backward flows give us more paths to consider. First we demonstrate the proof for one case for the compensation operator $\ominus_n = (;)$. Proof for the other three cases are similar. The syntax tree τ_{k+1} is obtained by replacing one atomic task t_i by two compensable tasks t_{c_i} and t_{c_n}. The reduced syntax tree τ'_{k+1} for the case $visible(t_{c_i}) = true$ and $visible(t_{c_n}) = false$ is shown in Fig. 6.11. The Petri net representation of two tasks t_{c_i} and t_{c_n} composed with the sequential operator $(;)$ is shown in Fig. 4.8.

There are three paths to consider:

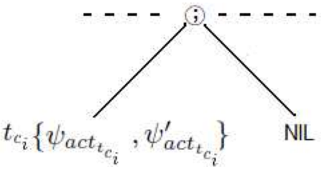

Figure 6.11: Reduced syntax tree τ'_{k+1}

(1) forward flow: $\pi_{\rightarrow\ominus} = s_0 \rightarrow ...s_\ominus \mid\stackrel{t_{c_i}(pt_1)}{\rightarrow} s_{t_{c_i}} \stackrel{t_{cn}(pt_4)}{\rightarrow} s_{t_{cn}} \mid\rightarrow ...s_{end1}$;

$\sigma_{\rightarrow\ominus} = r_0 \rightarrow ...r_\ominus \mid\stackrel{t_{c_i}(pt_1)}{\rightarrow} r_{t_{c_i}} \mid\rightarrow ...r_{end1}$. Here, $\pi_{s_0\rightarrow s_{end}} \sim_{st} \sigma_{r_0\rightarrow r_{end}}$ (same as the operator $\bullet$).

(2) compensation flow:

$\pi_{\rightarrow\ominus} = s_0 \rightarrow ...s_\ominus \mid\stackrel{t_{c_i}(pt_1)}{\rightarrow} s_{t_{c_i}} \stackrel{t'_{cn}(pt_5)}{\rightarrow} s_{t'_{cn}} \mid\stackrel{t'_{c_i}(pt_3)}{\rightarrow} s_{t'_{c_i}} \mid\rightarrow ...s_{end2}$;

$\sigma_{\rightarrow\ominus} = r_0 \rightarrow ...r_\ominus \mid\stackrel{t_{c_i}(pt_1)}{\rightarrow} r_{t_{c_i}} \mid\stackrel{t'_{c_i}(pt_3)}{\rightarrow} r_{t'_{c_i}} \mid\rightarrow ...r_{end2}$. Here t'_{c_i}, t'_{cn} represent the execution of compensation actions and $s_{t'_{cn}}$, $r_{t'_{c_i}}$, etc. represent compensated states. In this case, $\pi_{s_0\rightarrow s_\ominus} \sim_{st} \sigma_{r_0\rightarrow r_\ominus}$ (from Induction hypothesis).

$\pi_{s_\ominus\rightarrow s_{t'_{c_i}}} \sim_{st} \sigma_{r_\ominus\rightarrow r_{t'_{c_i}}}$ as $L_\phi(s_\ominus) = L_\phi(r_\ominus)$; $L_\phi(s_{t_{c_i}}) = L_\phi(s_{t'_{cn}}) = L_\phi(r_{t_{c_i}})$; $L_\phi(s_{t'_{c_i}}) = L_\phi(r_{t'_{c_i}})$.

$\pi_{s_{t'_{c_i}}\rightarrow s_{end2}} \sim_{st} \sigma_{r_{t'_{c_i}}\rightarrow r_{end2}}$ (Lemma 2).

Therefore, $\pi_{s_0\rightarrow s_{end2}} \sim_{st} \sigma_{r_0\rightarrow r_{end2}}$ (concatenating).

(3) compensation flow:

$\pi_{\rightarrow\ominus} = s_0 \rightarrow ...s_\ominus \mid\stackrel{t'_{c_i}(pt_2)}{\rightarrow} s_{t'_{c_i}} \mid\rightarrow ...s_{end3}$;

$\sigma_{\rightarrow\ominus} = r_0 \rightarrow ...r_\ominus \mid\stackrel{t'_{c_i}(pt_2)}{\rightarrow} r_{t'_{c_i}} \mid\rightarrow ...r_{end3}$.

Here $\pi_{s_0\rightarrow s_{end3}} \sim_{st} \sigma_{r_0\rightarrow r_{end3}}$ (obvious)

Now we show the induction step of the proof for one case of the compensation operator $\ominus_n = (||)$ (see Fig. 4.11), the other 3 cases are similar. For case $visible(t_{c_i}) = true$

and $visible(t_{c_n}) = false$ there are six paths to consider. Stuttering equivalent blocks are shown in the paths by the vertical lines (|). Proof of stuttering equivalence follows as above using the induction hypothesis and lemmas:

(1) forward flow:

$\pi_{\to\ominus} = s_0 \to ... \mid s_\ominus \overset{t_s(pt_1)}{\to} s_{t_s} \mid\overset{t_{c_i}(pt_3)}{\to} s_{t_{c_i}} \overset{t_{c_n}(pt_8)}{\to} s_{t_{c_n}} \overset{t_j(pt_{13})}{\to} s_{t_j} \mid\to ...s_{end1}$

$\sigma_{\to\ominus} = r_0 \to ... \mid r_\ominus \overset{t_s(pt_1)}{\to} r_{t_s} \mid\overset{t_{c_i}(pt_3)}{\to} s_{t_{c_i}} \overset{t_j(pt_{13})}{\to} r_{t_j} \mid\to ...r_{end1}$

(2) forward flow:

$\pi_{\to\ominus} = s_0 \to ... \mid s_\ominus \overset{t_s(pt_1)}{\to} s_{t_s} \overset{t_{c_n}(pt_8)}{\to} s_{t_{c_n}} \mid\overset{t_{c_i}(pt_3)}{\to} s_{t_{c_i}} \overset{t_j(pt_{13})}{\to} s_{t_j} \mid\to ...s_{end2}$

$\sigma_{\to\ominus} = r_0 \to ... \mid r_\ominus \overset{t_s(pt_1)}{\to} r_{t_s} \mid\overset{t_{c_i}(pt_3)}{\to} r_{t_{c_i}} \overset{t_j(pt_{13})}{\to} r_{t_j} \mid\to ...r_{end2}$

(3) compensation flow:

$\pi_{\to\ominus} = s_0 \to ... \mid s_\ominus \overset{t_s(pt_1)}{\to} s_{t_s} \mid\overset{t_{c_i}(pt_3)}{\to} s_{t_{c_i}} \overset{t'_j(pt_9)}{\to} s_{t'_j} \mid\overset{t'_{c_i}(pt_7)}{\to} s_{t'_{c_i}} \overset{t'_s(pt_2)}{\to} s_{t'_s} \mid\to ...s_{end3}$

$\sigma_{\to\ominus} = r_0 \to ... \mid r_\ominus \overset{t_s(pt_1)}{\to} r_{t_s} \mid\overset{t_{c_i}(pt_3)}{\to} r_{t_{c_i}} \mid\overset{t'_{c_i}(pt_7)}{\to} r_{t'_{c_i}} \overset{t'_s(pt_2)}{\to} r_{t'_s} \mid\to ...r_{end3}$

(4) compensation flow:

$\pi_{\to\ominus} = s_0 \to ... \mid s_\ominus \overset{t_s(pt_1)}{\to} s_{t_s} \overset{t_j(pt_8)}{\to} s_{t_j} \mid\overset{t'_{c_i}(pt_4)}{\to} s_{t'_{c_i}} \overset{t'_j(pt_{11})}{\to} s_{t'_j} \overset{t'_s(pt_2)}{\to} s_{t'_s} \mid\to ...s_{end4}$

$\sigma_{\to\ominus} = r_0 \to ... \mid r_\ominus \overset{t_s(pt_1)}{\to} r_{t_s} \mid\overset{t'_{c_i}(pt_4)}{\to} r_{t'_{c_i}} \overset{t'_s(pt_2)}{\to} r_{t'_s} \mid\to ...r_{end4}$

(5) compensation flow:

$\pi_{\to\ominus} = s_0 \to ... \mid s_\ominus \overset{t_s(pt_1)}{\to} s_{t_s} \mid\overset{t'_{c_i}(pt_4)}{\to} s_{t'_{c_i}} \overset{t'_j(pt_9)}{\to} s_{t'_j} \overset{t'_s(pt_2)}{\to} s_{t'_s} \mid\to ...s_{end5}$

$\sigma_{\to\ominus} = r_0 \to ... \mid r_\ominus \overset{t_s(pt_1)}{\to} r_{t_s} \mid\overset{t'_{c_i}(pt_4)}{\to} r_{t'_{c_i}} \overset{t'_s(pt_2)}{\to} r_{t'_s} \mid\to ...r_{end5}$

(6) compensation flow:

$\pi_{\to\ominus} = s_0 \to ... \mid s_\ominus \overset{t_s(pt_1)}{\to} s_{t_s} \overset{t'_j(pt_9)}{\to} s_{t'_j} \mid\overset{t'_{c_i}(pt_4)}{\to} s_{t'_{c_i}} \overset{t'_s(pt_2)}{\to} s_{t'_s} \mid\to ...s_{end6}$

$\sigma_{\to\ominus} = r_0 \to ... \mid r_\ominus \overset{t_s(pt_1)}{\to} r_{t_s} \mid\overset{t'_{c_i}(pt_4)}{\to} r_{t'_{c_i}} \overset{t'_s(pt_2)}{\to} r_{t'_s} \mid\to ...r_{end6}$

Now we show the induction step of the proof for one case of the compensation opera-

tor $\ominus_n = (\otimes)$, the other 3 cases are similar. Fig. 4.12 shows the Petri net representation of $t_{c_i} \otimes t_{c_n}$. For case $visible(t_{c_i}) = true$ and $visible(t_{c_n}) = false$ there are eight paths to consider:

(1) forward flow:

$\pi_{\to\ominus} = s_0 \to ... \mid s_\ominus \overset{t_s(pt_1)}{\to} s_{t_s} \mid\overset{t_{c_i}(pt_3)}{\to} s_{t_{c_i}} \overset{t'_{cn}(pt_9)}{\to} s'_{t_{cn}} \overset{t_j(pt_{11})}{\to} s_{t_j} \mid\to ...s_{end1}$

$\sigma_{\to\ominus} = r_0 \to ... \mid r_\ominus \overset{t_s(pt_1)}{\to} r_{t_s} \mid\overset{t_{c_i}(pt_3)}{\to} s_{t_{c_i}} \overset{t_j(pt_{11})}{\to} r_{t_j} \mid\to ...r_{end1}$

(2) forward flow:

$\pi_{\to\ominus} = s_0 \to ... \mid s_\ominus \overset{t_s(pt_1)}{\to} s_{t_s} \mid\overset{t_{c_i}(pt_3)}{\to} s_{t_{c_i}} \overset{t'_{cn}(pt_8)}{\to} s'_{t_{cn}} \overset{t_j(pt_{11})}{\to} s_{t_j} \mid\to ...s_{end2}$

$\sigma_{\to\ominus} = r_0 \to ... \mid r_\ominus \overset{t_s(pt_1)}{\to} r_{t_s} \mid\overset{t_{c_i}(pt_3)}{\to} s_{t_{c_i}} \overset{t_j(pt_{11})}{\to} r_{t_j} \mid\to ...r_{end2}$

(3) forward flow:

$\pi_{\to\ominus} = s_0 \to ... \mid s_\ominus \overset{t_s(pt_1)}{\to} s_{t_s} \overset{t_{cn}(pt_7)}{\to} s_{t_{cn}} \mid\overset{t'_{c_i}(pt_4)}{\to} s'_{t_{c_i}} \overset{t_j(pt_{13})}{\to} s_{t_j} \mid\to ...s_{end3}$

$\sigma_{\to\ominus} = r_0 \to ... \mid r_\ominus \overset{t_s(pt_1)}{\to} r_{t_s} \mid\overset{t'_{c_i}(pt_4)}{\to} s'_{t_{c_i}} \overset{t_j(pt_{13})}{\to} r_{t_j} \mid\to ...r_{end3}$

(4) forward flow:

$\pi_{\to\ominus} = s_0 \to ... \mid s_\ominus \overset{t_s(pt_1)}{\to} s_{t_s} \overset{t_{cn}(pt_7)}{\to} s_{t_{cn}} \mid\overset{t'_{c_i}(pt_5)}{\to} s'_{t_{c_i}} \overset{t_j(pt_{13})}{\to} s_{t_j} \mid\to ...s_{end4}$

$\sigma_{\to\ominus} = r_0 \to ... \mid r_\ominus \overset{t_s(pt_1)}{\to} r_{t_s} \mid\overset{t'_{c_i}(pt_5)}{\to} s'_{t_{c_i}} \overset{t_j(pt_{13})}{\to} r_{t_j} \mid\to ...r_{end4}$

(5) forward flow:

$\pi_{\to\ominus} = s_0 \to ... \mid s_\ominus \overset{t_s(pt_1)}{\to} s_{t_s} \overset{t'_{cn}(pt_8)}{\to} s'_{t_{cn}} \mid\overset{t_{c_i}(pt_3)}{\to} s_{t_{c_i}} \overset{t_j(pt_{11})}{\to} s_{t_j} \mid\to ...s_{end5}$

$\sigma_{\to\ominus} = r_0 \to ... \mid r_\ominus \overset{t_s(pt_1)}{\to} r_{t_s} \mid\overset{t_{c_i}(pt_3)}{\to} s_{t_{c_i}} \overset{t_j(pt_{11})}{\to} r_{t_j} \mid\to ...r_{end5}$

(6) forward flow:

$\pi_{\to\ominus} = s_0 \to ... \mid s_\ominus \overset{t_s(pt_1)}{\to} s_{t_s} \mid\overset{t'_{c_i}(pt_5)}{\to} s'_{t_{c_i}} \overset{t_{cn}(pt_7)}{\to} s_{t_{cn}} \overset{t_j(pt_{13})}{\to} s_{t_j} \mid\to ...s_{end6}$

$\sigma_{\to\ominus} = r_0 \to ... \mid r_\ominus \overset{t_s(pt_1)}{\to} r_{t_s} \mid\overset{t'_{c_i}(pt_5)}{\to} s'_{t_{c_i}} \overset{t_j(pt_{13})}{\to} r_{t_j} \mid\to ...r_{end6}$

(7) compensation flow:

$\pi_{\to\ominus} = s_0 \to ... \mid s_\ominus \overset{t_s(pt_1)}{\to} s_{t_s} \mid\overset{t'_{c_i}(pt_5)}{\to} s'_{t_{c_i}} \overset{t'_{cn}(pt_8)}{\to} s'_{t_{cn}} \overset{t'_s(pt_2)}{\to} s'_{t_s} \mid\to ...s_{end7}$

$\sigma_{\to\ominus} = r_0 \to ... \mid r_\ominus \overset{t_s(pt_1)}{\to} r_{t_s} \mid\overset{t'_{c_i}(pt_5)}{\to} s'_{t_{c_i}} \overset{t'_s(pt_2)}{\to} r'_{t_s} \mid\to ...r_{end7}$

(8) compensation flow:

$\pi_{\rightarrow\ominus} = s_0 \rightarrow ... \mid s_{\ominus} \stackrel{t_s(pt_1)}{\rightarrow} s_{t_s} \stackrel{t'_{c_n}(pt_8)}{\rightarrow} s'_{t_{c_n}} \mid\stackrel{t'_{c_i}(pt_5)}{\rightarrow} s'_{t_{c_i}} \stackrel{t'_s(pt_2)}{\rightarrow} s'_{t_s} \mid\rightarrow ...s_{end8}$

$\sigma_{\rightarrow\ominus} = r_0 \rightarrow ... \mid r_{\ominus} \stackrel{t_s(pt_1)}{\rightarrow} r_{t_s} \mid\stackrel{t'_{c_i}(pt_5)}{\rightarrow} s'_{t_{c_i}} \stackrel{t'_s(pt_2)}{\rightarrow} r'_{t_s} \mid\rightarrow ...r_{end8}$

Now we demonstrate the proof for the operator $\ominus_n = (\sqcap)$. For the case $visible(t_{c_i}) = true$ and $visible(t_{c_n}) = false$ there are four paths to consider (see Fig. 4.9):

(1) forward flow:

$\pi_{\rightarrow\ominus} = s_0 \rightarrow ... \mid s_{\ominus} \stackrel{t_s(pt_1)}{\rightarrow} s_{t_s} \mid\stackrel{t_{c_i}(pt_5)}{\rightarrow} s_{t_{c_i}} \stackrel{t_j(pt_{12})}{\rightarrow} s_{t_j} \mid\rightarrow ...s_{end1}$

$\sigma_{\rightarrow\ominus} = r_0 \rightarrow ... \mid r_{\ominus} \stackrel{t_s(pt_1)}{\rightarrow} r_{t_s} \mid\stackrel{t_{c_i}(pt_5)}{\rightarrow} s_{t_{c_i}} \stackrel{t_j(pt_{12})}{\rightarrow} r_{t_j} \mid\rightarrow ...r_{end1}$

(2) forward flow:

$\pi_{\rightarrow\ominus} = s_0 \rightarrow ... \mid s_{\ominus} \stackrel{t_s(pt_2)}{\rightarrow} s_{t_s} \mid\stackrel{t_{c_n}(pt_8)}{\rightarrow} s_{t_{c_n}} \stackrel{t_j(pt_{13})}{\rightarrow} s_{t_j} \mid\rightarrow ...s_{end2}$

$\sigma_{\rightarrow\ominus} = r_0 \rightarrow ... \mid r_{\ominus} \stackrel{t_s(pt_2)}{\rightarrow} r_{t_s} \mid\stackrel{t_j(pt_{13})}{\rightarrow} r_{t_j} \mid\rightarrow ...r_{end2}$

(3) compensation flow:

$\pi_{\rightarrow\ominus} = s_0 \rightarrow ... \mid s_{\ominus} \stackrel{t_s(pt_1)}{\rightarrow} s_{t_s} \mid\stackrel{t'_{c_i}(pt_6)}{\rightarrow} s'_{t_{c_i}} \stackrel{t'_s(pt_3)}{\rightarrow} s'_{t_s} \mid\rightarrow ...s_{end3}$

$\sigma_{\rightarrow\ominus} = r_0 \rightarrow ... \mid r_{\ominus} \stackrel{t_s(pt_1)}{\rightarrow} r_{t_s} \mid\stackrel{t'_{c_i}(pt_6)}{\rightarrow} r'_{t_{c_i}} \stackrel{t'_s(pt_3)}{\rightarrow} r'_{t_s} \mid\rightarrow ...r_{end3}$

(4) compensation flow:

$\pi_{\rightarrow\ominus} = s_0 \rightarrow ... \mid s_{\ominus} \stackrel{t_s(pt_2)}{\rightarrow} s_{t_s} \mid\stackrel{t'_{c_n}(pt_9)}{\rightarrow} s'_{t_{c_n}} \stackrel{t'_s(pt_4)}{\rightarrow} s'_{t_s} \mid\rightarrow ...s_{end4}$

$\sigma_{\rightarrow\ominus} = r_0 \rightarrow ... \mid r_{\ominus} \stackrel{t_s(pt_2)}{\rightarrow} r_{t_s} \mid\stackrel{t'_s(pt_4)}{\rightarrow} r'_{t_s} \mid\rightarrow ...r_{end4}$

The other 3 cases are similar. Now we demonstrate the proof for the operator $\ominus_n = (\rightsquigarrow)$. Fig. 4.10 shows the Petri net representation of $t_{c_i} \rightsquigarrow t_{c_n}$. For case $visible(t_{c_i}) = true$ and $visible(t_{c_n}) = false$ there are three paths to consider:

(1) forward flow:

$\pi_{\rightarrow\ominus} = s_0 \rightarrow ... \mid s_{\ominus} \stackrel{t_{s_1}(pt_1)}{\rightarrow} s_{t_{s_1}} \mid\stackrel{t_{c_i}(pt_5)}{\rightarrow} s_{t_{c_i}} \stackrel{t_j(pt_{11})}{\rightarrow} s_{t_j} \mid\rightarrow ...s_{end1}$

$\sigma_{\to\ominus} = r_0 \to ... \mid r_\ominus \overset{t_{s_1}(pt_1)}{\to} r_{t_{s_1}} \mid\overset{t_{c_i}(pt_5)}{\to} s_{t_{c_i}} \overset{t_j(pt_{11})}{\to} r_{t_j} \mid\to ...r_{end1}$

(2) forward flow:

$\pi_{\to\ominus} = s_0 \to ... \mid s_\ominus \overset{t_{s_1}(pt_1)}{\to} s_{t_{s_1}} \mid\overset{t'_{c_i}(pt_6)}{\to} s'_{t_{c_i}} \overset{t_{s_2}(pt_2)}{\to} s_{t_{s_2}} \overset{t_{cn}(pt_8)}{\to} s_{t_{cn}} \overset{t_j(pt_{13})}{\to} s_{t_j} \mid\to ...s_{end2}$

$\sigma_{\to\ominus} = r_0 \to ... \mid r_\ominus \overset{t_{s_1}(pt_1)}{\to} r_{t_{s_1}} \mid\overset{t'_{c_i}(pt_6)}{\to} r'_{t_{c_i}} \overset{t_{s_2}(pt_2)}{\to} r_{t_{s_2}} \overset{t_j(pt_{13})}{\to} r_{t_j} \mid\to ...r_{end2}$

(3) compensation flow:

$\pi_{\to\ominus} = s_0 \to ... \mid s_\ominus \overset{t_{s_1}(pt_1)}{\to} s_{t_{s_1}} \mid\overset{t'_{c_i}(pt_6)}{\to} s'_{t_{c_i}} \overset{t_{s_2}(pt_2)}{\to} s_{t_{s_2}} \overset{t'_{cn}(pt_9)}{\to} s'_{t_{cn}} \overset{t'_s(pt_4)}{\to} s'_{t_s} \mid\to ...s_{end3}$

$\sigma_{\to\ominus} = r_0 \to ... \mid r_\ominus \overset{t_{s_1}(pt_1)}{\to} r_{t_{s_1}} \mid\overset{t'_{c_i}(pt_6)}{\to} r'_{t_{c_i}} \overset{t_{s_2}(pt_2)}{\to} r_{t_{s_2}} \overset{t'_s(pt_4)}{\to} r'_{t_s} \mid\to ...r_{end3}$

The other 3 cases are similar. We have shown for all operators that any path starting from s_0 in M_{k+1} there is a stuttering equivalent path in M'_{k+1} starting from r_0. This proves our induction step.

We can provide similar arguments and an induction proof to show that for any path starting from r_0 in M'_{k+1} there is a stuttering equivalent path in M_{k+1} starting from s_0.

6.4 Effectiveness

In this section, we present a performance comparison of the workflow reduction algorithm with partial order reduction for different types of workflows. The relation of the Workflow Reduction (WR) with Partial order reduction (POR) is complementary; indeed the results show how effective the Workflow Reduction algorithm is when it is performed before model checking. These experiments were done using DiVinE 2.4 on a single CPU with 3GB of Memory. Workflow with an "and" split task and an "and" join task is taken with a different number of concurrent branches. Fig. 6.12 shows the workflow with concurrent tasks *Task1*, *Task2*, $\cdots$ *Task15*. The LTL-property we verified is whether

Task_4 and *Task_5* can occur concurrently.

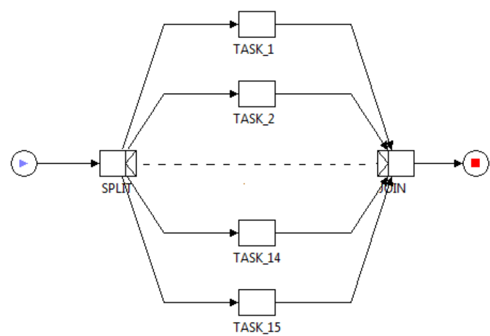

Figure 6.12: Workflow with and composition

The LTL-property is given below:

#define task4_working (_AndTest_TASK_4_SUC == 1)

#define task5_working (_AndTest_TASK_5_SUC == 1)

#property G (task4_working && ! task5_working)

Table 6.1 shows the number of transitions for the verification with Workflow Reduction (WR) and Partial Order Reduction (POR) for various numbers of concurrent tasks. Note that we have proved this property by contradiction, so 'Accepting Cycle' 'YES' means task4 and task5 execute concurrently.

Tasks	Acc Cycle	POR		WR + POR	
		States	Time (s)	States	Time (s)
5	YES	107	< 1	15	< 1
10	YES	4396	< 1	15	< 1
15	YES	48784	3	15	< 1

Table 6.1: Comparison for and composition

Workflows with a "xor" composition were tested using various number of branches, and it was determined whether Task_4 and Task_5 were mutually exclusive.

#define task4_working (_XorTest_TASK_4_SUC == 1)

#define task5_working (_XorTest_TASK_5_SUC == 1)

#property G(task4_working → F ! task5_working) || G (task5_working → F ! task4_working)

Table 6.2 shows the comparison.

Tasks	Acc Cycle	POR		WR + POR	
		States	Time (s)	States	Time (s)
5	NO	27	< 1	24	< 1
10	NO	37	< 1	29	< 1
15	NO	47	< 1	34	< 1

Table 6.2: Comparison for xor composition

Workflows with an “or” composition were tested using various number of branches, and it was determined whether the join task is eventually reachable.

#define join_task_working (_OrTest_JOIN_6_SUC == 1)

#property G F join_task_working

Table 6.3 shows the comparison.

Tasks	Acc Cycle	POR		WR + POR	
		States	Time (s)	States	Time (s)
5	NO	99	< 1	1025	< 1
10	NO	20197	3	1025	< 1

Table 6.3: Comparison for or composition

From these experimental results we can see the effectiveness of the reduction, it becomes especially significant in situations where there are “and” composition or “or” composition with many branches.

Chapter 7

Tool overview

Object oriented programming has been successfully applied to numerous software systems. The quest for formal verification of object oriented systems motivates a great deal of research; however, there are many challenges to be dealt with [24]. Real life software applications usually consists of many components; each component has many business logics. On the other hand, client applications require an enormous programming effort to provide sophisticated Graphical User Interfaces (GUI). To verify such complex software systems without abstraction is challenging. We developed a workflow management system named NOVA WorkFlow [7] which allows the user to graphically input a compensable workflow and specification in LTL and automatically verify if the specification holds. NOVA Workflow deals with the problem of verifying a complex and safety critical software system by abstraction and reduction.

The NOVA Editor was developed as an Eclipse Plugin to make the development of a system easier by seamlessly integrating modeling into the overall development process. The NOVA Translator translates the workflow model and Java specification in DVE, the input language of the parallel distributed model checking program, DiVinE [1]. Such

parallel and distributed model checkers can verify large models as they can more effectively handle the state explosion problem using high performance computing facilities. The NOVA Engine is a workflow engine developed based on Service Oriented Architecture (SOA). Fig. 7.1 shows the architecture of NOVA WorkFlow. The engine was developed on the Spring [5] and Hibernate [4] platforms both of which can be deployed to various application servers. Spring is a widely used open source framework that helps developers build high quality applications faster. Spring provides a consistent programming and configuration model that is well understood and used by developers worldwide. On the other hand, Hibernate is an object-relational mapping (ORM) library for the Java language, providing a framework for mapping an object-oriented domain model to a traditional relational database. The NOVA Browser is a user friendly browser based on the mind map paradigm [12] which provides better user experience by flexibility and brainstorming. The browser incorporates a time travel view and a chart view that helps to analyze large amount of data for real life applications.

7.1 NOVA workflow

7.1.1 The NOVA editor

The NOVA Editor is a visual modeling tool for the Compensable Workflow Modeling Language, CWML. We recall that a compensable workflow model consists of compensable tasks and uncompensable tasks. To represent the control flows, CWML uses the t-Calculus [28] operators and traditional control flow operators (i.e., and, xor, or). Compensable transactions along with the traditional control flows can be easily edited and displayed graphically in the editor. The editor produces workflow models which are *cor-*

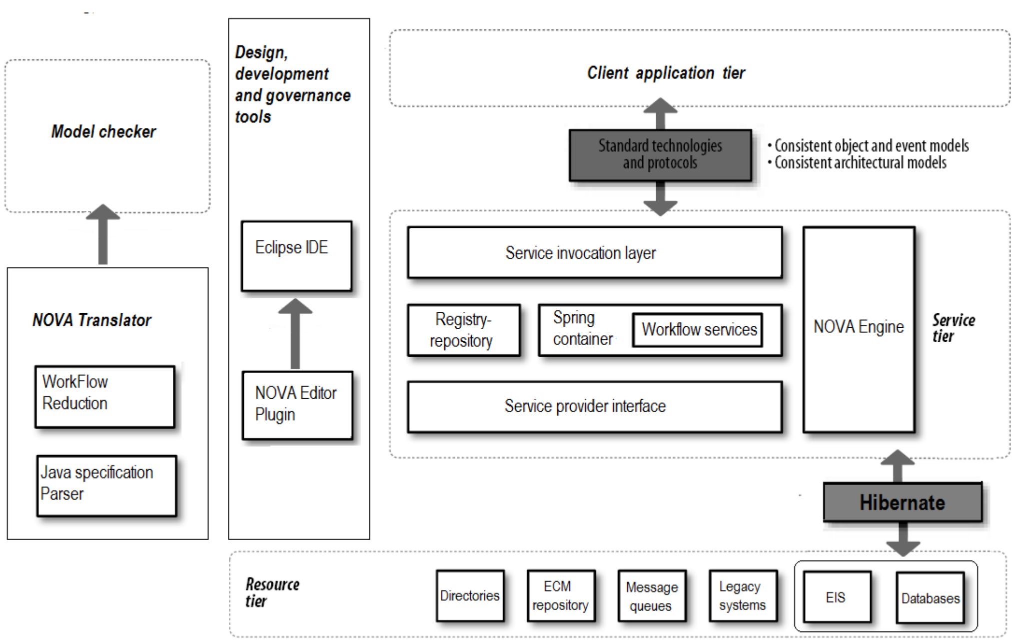

Figure 7.1: SOA based architecture of NOVA workflow

rect by construction (see chapter 4). In other word, the data flow correctness prevents incorrect composition of workflow activities. In chapter 4 we showed that each workflow component has an underlying Petri net structure.

The editor is built as an Eclipse Plugin [2] using the Eclipse Graphical Editing Framework (GEF) [3]. Because of this architecture, the NOVA Editor is available in the development platform. Application developers can create models in a Java project, and generate workflow service classes from it (see section 7.1.2). As a whole, modeling, development and verification can be done in the same Eclipse Platform. Fig. 7.2 shows the workbench and workflow components view of the NOVA Editor in the Eclipse IDE.

Process description

Workflow process descriptions are stored in XML files. Each workflow is stored in its own file rather than in a single process description file. The NOVA Editor stores the

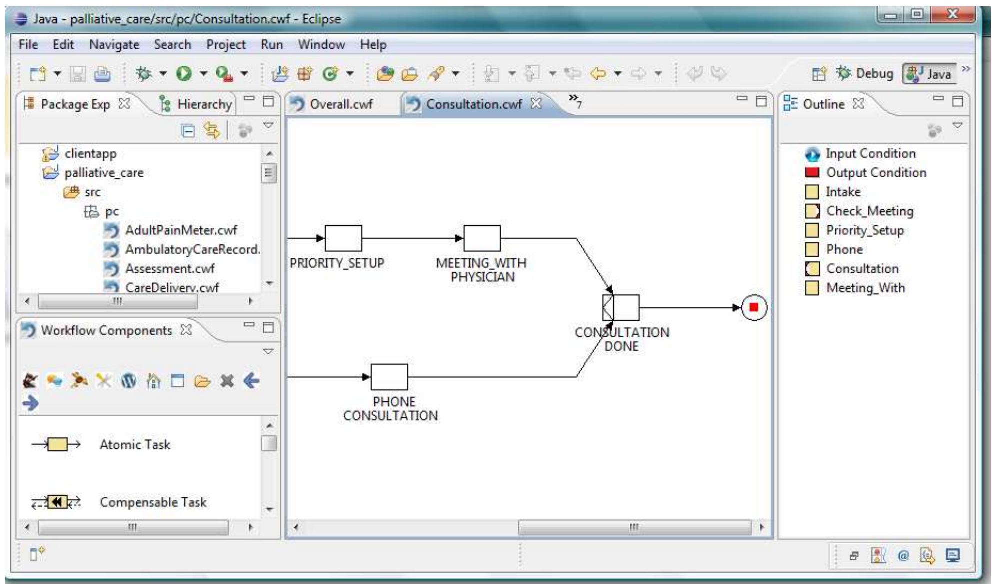

Figure 7.2: NOVA editor in eclipse IDE

task type and control flows in the process description file; other information (i.e., task properties or specifications) is stored in the task's property file which is a Java file. Typically a workflow project has many subnet workflows and different people develop them based on their expertise and knowledge. As the NOVA Editor stores each subnet in a different file, many resources (i.e., system architects or developers) can work in parallel on different subnets at a time.

7.1.2 The NOVA engine

The NOVA Workflow is developed using an SOA architecture. From the NOVA Editor, Workflow service classes are automatically generated to be deployed in the Spring container. These services are exposed to the outside world by service provider interfaces. The Workflow service bean's (class that contains the business logic) life cycle is managed by the spring container and at the time of instantiation, service beans regis-

ter themselves to the workflow engine. For a particular workflow instance, the service bean's execution flow is guided by the workflow engine (Fig. 7.3). Fig. 7.4 details how a WorkFlow service class is extended by an application service bean. Note that here the `Overall_Appointment` class was generated automatically by the NOVA workflow service generator tool.

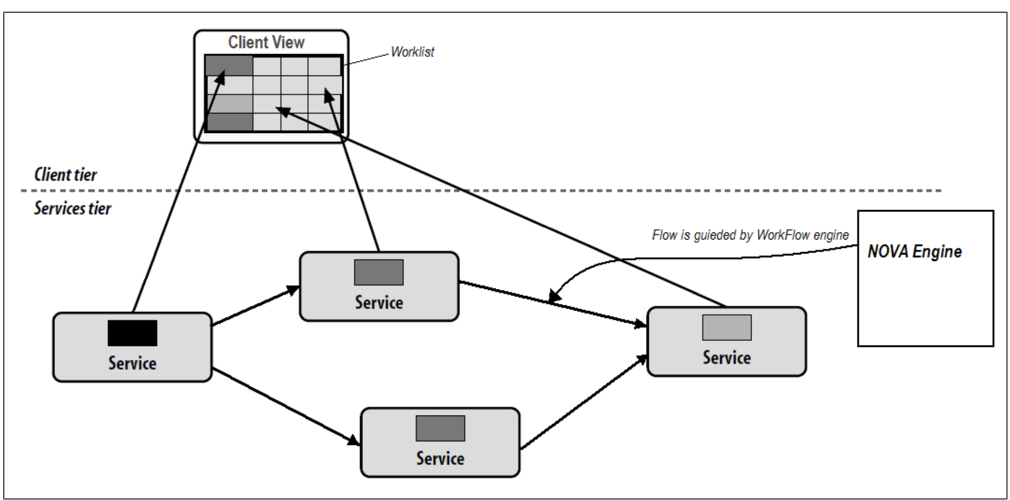

Figure 7.3: NOVA engine guides the service flow

```
public class AppointmentServiceImpl extends Overall_Appointment implements IAppointmentService{

    private IAppointmentDao appointmentDao;
    private IPhysicianService physicianService;

    public AppointmentDTO saveAppointment(AppointmentDTO appointmentInfo) throws IlligalOperationException {
        // validate physician
        if( physicianService.findPhysicianInfoById(appointmentInfo.getPhysicianId()) == null)
            throw new IlligalOperationException("Physician id : " + appointmentInfo.getPhysicianId() + " does not exists.");

        appointmentDao.saveEntity(appointmentInfo);
        action(appointmentInfo.getInstanceId());
        return appointmentInfo;
    }
```

Extending NOVA WorkFlow Service Class

Invoking super.action() after actual work

Figure 7.4: An example of a service class extension

The NOVA WorkFlow engine provides an interface `IWorkFlowEngineService`. Through this service interface, a client application can display work list items and can perform workflow related operations. The methods are described in the appendix (see Table

A.3).

```
public interface IWorkflowEngineService {


public WfInstance createNewWorkflowInstance(WfInstance newInstance);
public WfInstance getInstance(long id);
public List<WfInstance> getAllActiveInstances();
public List<InstanceInfo> getAvailableMethods(WfInstance theInstance);
public List<InstanceInfo> getAvailableMethods(Long instanceId, String taskId);
}
```

Attributes of class ***InstanceInfo*** is shown here:

```
private Long id;
private Long instanceId;
private String taskId;
private String workflowName;
private String varName;
private Integer value;
private String availableMethod;
private String actor;
```

The NOVA Engine provides two flavours for workflow engine integration: i) loosely coupled integration, ii) tightly coupled integration. Which one is to be selected depends on the system architecture. When the application services are not deployed in the spring framework, loosely coupled integration should be selected; by this integration, workflow services are invoked by application services after performing their activities. The NOVA

Engine updates the task status when a particular service bean is invoked. On the other hand, if the application services are deployed in the spring container, tightly coupled integration is recommended; here application services are extended by workflow service classes.

7.1.3 The NOVA translator

In [36] we proposed an automatic translator which translates a graphical model constructed using the YAWL editor to DVE. The approach was different and it was based on channels, signals, etc rather than Petri net based. The NOVA Workflow incorporates an automated translator from CWML to DVE the modeling language of the DiVinE model checker using methods described in chapter 5. To verify a workflow, a task's specifications need to be written in its property file. A task's property file is a Java class which extends abstract classes of the NOVA Translator API (see Table A.1 in appendix). A parser reads a task's specifications and the translator translates them to DVE. The NOVA translator has two additional features with which it can handle the state explosion problem for a complex workflow system: i) Data abstraction, ii) Workflow Reduction.

Data abstraction

It is important to note that not all the attributes of Java entities are important for model checking; for example, a patient's name generally does not carry any important information that can direct a health care workflow. When writing the specifications for tasks, one should concentrate on the attributes that can directly or indirectly affect the flow of execution. NOVA WorkFlow provides a Util class which has a method named ***getNonDeterministicData()***. Using this method the system designer can specify

```
public class Receive_Referral extends UncompensableTaskMCImpl
{
ReferralInfoDTO referralA;
int age;

@Override
public void initialize() {
age = (Integer) Util.getNonDeterministicData(new Integer[]{15,30,45,60});
}
......
}
```

Figure 7.5: Syntax for assigning non-deterministic data

the data set for a particular variable. For model checking it is expected to use Fuzzy values which covers sample values from every category of the domain. For example the property age of a person can have many discrete values. If this is an integer, the domain is -32768 to +32767; however, if we consider a Banking system, it runs its business based on 3 or 4 categories of age values, making a different policy for junior, adult and senior. If a model checking program is allowed to have all possible values from the domain, there will be huge state explosion which can be drastically reduced by taking only abstract values for a variable. Data abstraction is an essential and important issue for model checking. The code snippet in Fig. 7.5 shows an example usage of data abstraction. The NOVA translator will generate four non-deterministic choices (15,30,45,60) for the age variable in the DVE translation. Fig. 7.6 shows the translated DVE code of these non-deterministic values.

```
}
process Overall_Receive_Referral{
int age = 0;
state tr;
init tr;
trans
tr -> tr { guard _Overall_Start_SUC > 0 ; effect age = 15, referralA_age = age, _Overall_Start_SUC = _Overall_Start_SUC - 1  ,
tr -> tr { guard _Overall_Start_SUC > 0 ; effect age = 30, referralA_age = age, _Overall_Start_SUC = _Overall_Start_SUC - 1  ,
tr -> tr { guard _Overall_Start_SUC > 0 ; effect age = 45, referralA_age = age, _Overall_Start_SUC = _Overall_Start_SUC - 1  ,
tr -> tr { guard _Overall_Start_SUC > 0 ; effect age = 60, referralA_age = age, _Overall_Start_SUC = _Overall_Start_SUC - 1  ,
}
```

Figure 7.6: DVE code for non-deterministic data

NOVA Workflow allows you to write abstract task specification using limited number of Java syntax in a task property file. As this file will be translated to the input language of a model checker, not all java syntax is supported. The syntax of writing abstract task specification is provided in the appendix.

Workflow reduction

The NOVA translator incorporates the Reduction algorithm described in chapter 6 which reads a property to be verified, written in Linear Temporal Logic, the workflow model and the task's property (i.e., pre-conditions, actions) and reduces the workflow model by removing the tasks and properties which do not directly or indirectly affect the execution flow, before verifying the property. If no LTL-property file is specified it translates the entire workflow without performing any reduction.

7.1.4 The NOVA browser

The NOVA Browser hierarchically represents database records using a mind map. A mind map [12] is a graphical way to represent ideas and concepts. A mind map (as opposed to traditional notes or text) structures information in a way that resembles much more closely how the brain actually works. Since it is an activity that is both

analytical and artistic, it engages the human brain in a much richer way, helping in all its cognitive functions. Mind maps are used to generate, visualize, structure, and classify ideas, and as an aid to studying and organizing information, solving problems, making decisions, and writing. Pictorial methods for recording knowledge and modelling systems have been used for centuries in learning, brainstorming, memory, visual thinking, and problem solving by educators, engineers, psychologists, and others. Although these methods have been used for a long time by analysts and individuals, its use in software systems to provide a means to involve the user more with the system is rare. The browser displays all the necessary information for a user in one page. Information (nodes) are presented into the map radially around the centre node. For example for a HealthCare application, a physician can select a patient's case and view all the information related to the patient in a hierarchical fashion allowing the user to concentrate more on the patient's condition and health. Fig. 7.7 shows a patient's information in hierarchical fashion in the NOVA Browser.

Time travel view

The browser incorporates a time travel view with which the user can go back in time and can check previous information. In order to enable the time travel view, the database tables need some extra columns to preserve the historical information and time. Let us define the Timed Table here:

Definition 7.1. *A Timed Table has the following 5 columns in addition to other custom columns:*

- *id: is a primary key,*

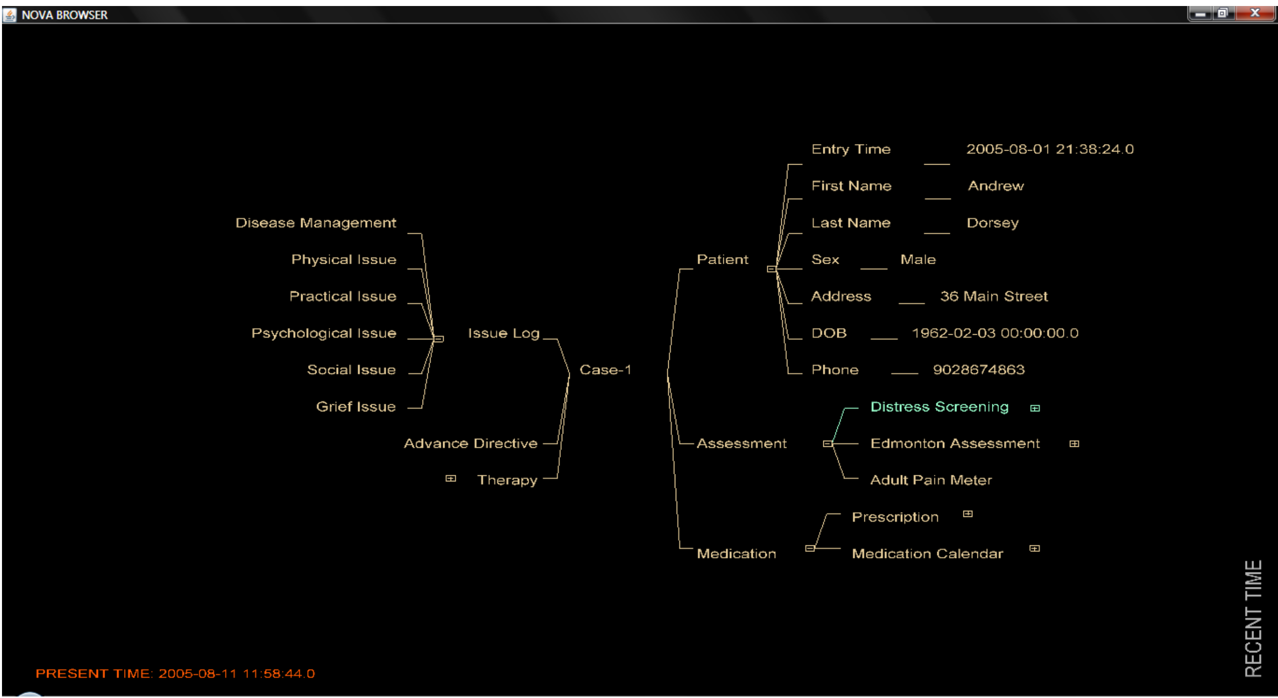

Figure 7.7: Hierarchical data representation in the NOVA browser

- *status: can be either NEW or UPDATE or DELETE,*
- *entryTime: the time when the record was inserted,*
- *caseId: the instance id or case number for a particular case,*
- *refId: is the parent record's id.*

Remark: In a timed table, records are not deleted or updated by replacing the original record; instead of an update operation to a row, a new entry with status UPDATE is inserted into the table and the refId column is used to indicate the parent record, whose information is being updated.

When displaying the records in the browser, only the latest records are shown; the user does not see the historical information. When the user travels back, the browser fetches historical records and displays them in the map. The time travel view provides

an easy way for the user to go back to when a certain record was inserted or updated and can check its effect by travelling forward from that time. The NOVA Browser provides four types of time travel:

- Travel back to a past time when a selected record was inserted, updated or deleted; in this case, the user needs to select a node.
- Travel forward to a future time when a selected record was inserted, updated or deleted; here the user also needs to select a node.
- Travel one step back to the previous time when any record was inserted or updated for the selected case; in this case the user does not select a node and the search operation is performed globally on all tables for the case.
- Travel one step forward to the next time when any record was inserted or updated for the selected case; here the user does not select a node and thus the search operation is performed globally on all tables.

If the user selects the 'Assessment' node and travels back, the browser will jump to the time when an Assessment record was inserted or updated. Note that, Assessment is an abstract base class with four concrete subclasses. NOVA Browser displays the transition from one mind map to another by doing animation about the Z-axis, giving the impression of travelling backward and forward.

The Chart view

Charts and graphs play an important role for analyzing information. The visual representation of complex information can help researchers process large amounts of data

to detect and observe patterns. However it is very difficult to pre-configure all types of charts with the different combination of parameters that are needed by many applications. The NOVA Browser incorporates a chart view where the user can select his chart parameters. The user selects some nodes from the browser and adds them to the parameter list. The user selects a time range and the chart viewer generates the chart using those selected parameters by plotting time on the X-axis and the parameters on the Y-axis. As the user can select any node from the browser, the browser will either plot the exact value of the parameter or present them symbolically. Table 7.1 shows the parameter data types and their graphical representation. Fig. 7.8 shows an example of NOVA Chart where Radiology, Administered Medicine are string parameters, Pain and Nausea are integer parameters.

Parameter Data Type	Rresentation
String	String
boolean	Symbolic (✓,×)
int,float,double	Bar chart
date	Symbolic (✓)
Class	Symbolic (✓,×)

Table 7.1: Parameter data types and their graphical representations

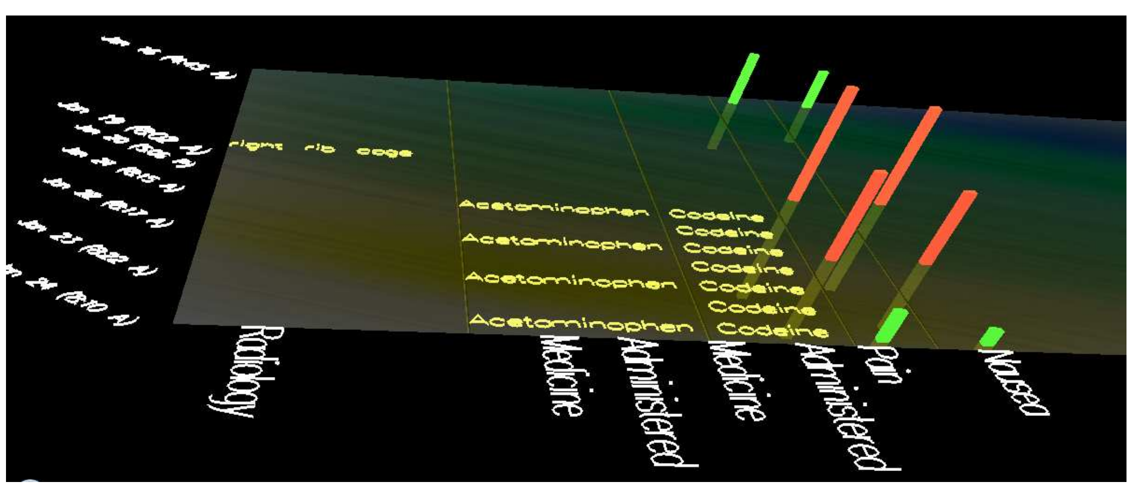

Figure 7.8: Example of a chart view

Chapter 8

Case study

In this chapter we will see how NOVA Workflow can be useful in the verification of properties of a real life workflow. NOVA Workflow was used for modeling and verifying of a Hospice Palliative Care workflow developed in collaboration with a local health authority, the Guysborough Antigonish Strait Health Authority (GASHA).

8.1 Hospice palliative care

Palliative Care refers to the medical or comfort care that reduces the severity of a disease or slows its progress, rather than providing a cure. For incurable diseases, in cases where the cure is not recommended due to other health concerns, and when the patient does not wish to pursue a cure, palliative care becomes the focus of treatment. For example, if surgery cannot be performed to remove a tumor, radiation treatment might be tried to reduce its rate of growth, and pain management could help the patient manage physical symptoms. Hospice Palliative Care (HPC) in Canada is guided by the Canadian Hospice Palliative Care Association (CHPCA) National Model (2002) [16]

which espouses a collaborative, patient/family centred approach to care.

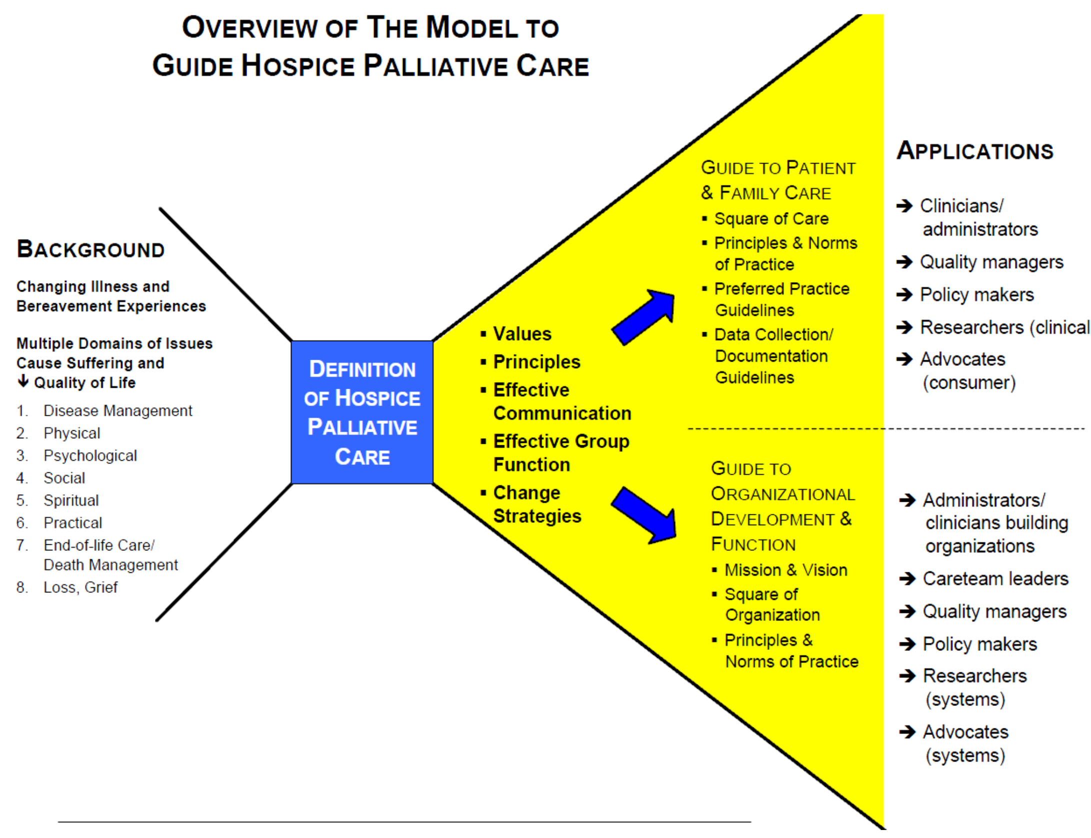

Figure 8.1: Overview of CHPCA model

The CHPCA National Model was built on an understanding of health, the illness and bereavement experiences, and the role hospice palliative care plays in relieving suffering and improving quality of life. This national model is a tool to guide all activities related to hospice palliative care. It was developed in consultation with experts across the country, and based on patient and family issues/needs (as opposed to existing funding and service delivery models), and created a shared vision and set the stage for a consistent, standardized approach to patient and family care, organizational development, education and advocacy across the country. It was developed to guide both:

- the process of providing care to patients and families through both the illness and

bereavement experiences

- the development and function of hospice palliative care organizations.

The palliative care model has principles to guide data collection and the model is based on "norms of practice" that support the development of local standards while supporting the goal of quality care.

Modeling of palliative care process

The process for providing care involves the following six essential and several basic steps that guide the interaction between caregivers, and the patient and family:

1. Assessment
2. Information Sharing
3. Decision-making
4. Care Planning
5. Care Delivery
6. Confirmation

A palliative care workflow model was designed from CHPCA norms; GASHA forms were mapped into processes (i.e., using form attributes in task pre-conditions and actions). Fig. 8.2 shows the high level model of the workflow consisting of composite tasks. Each of the composite task has a subnet workflow. Some of them are described in subsequent sections. Fig. 8.3 shows a GASHA form to record pain level.

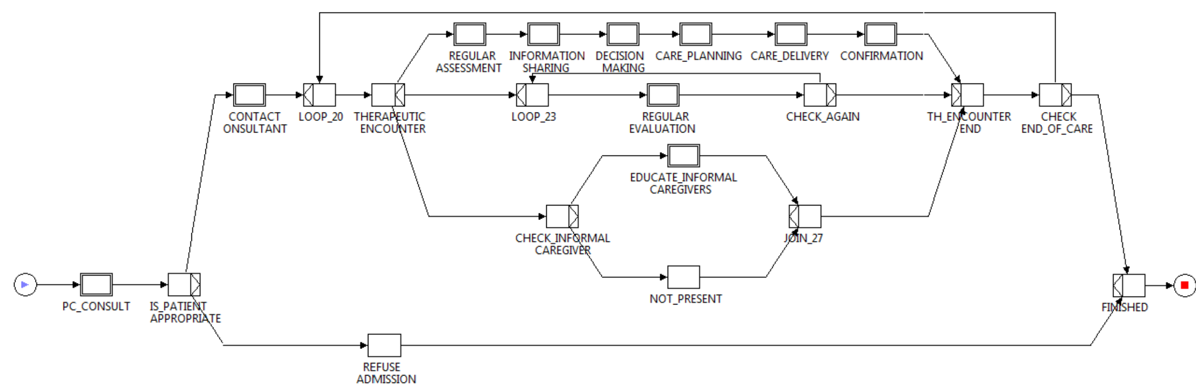

Figure 8.2: Palliative care workflow: Overall

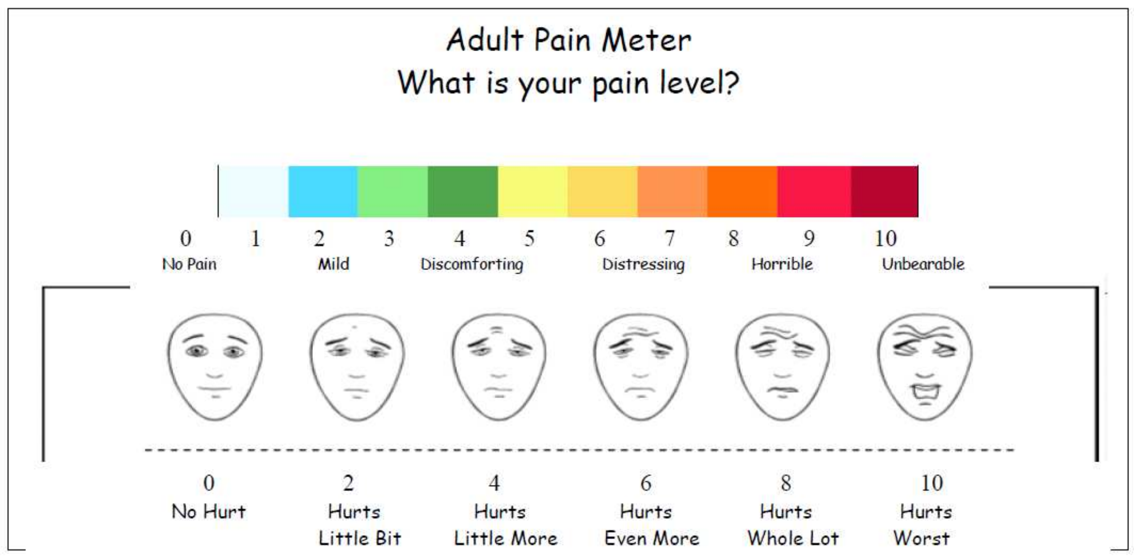

Figure 8.3: GASHA Form: Adult pain meter

Overall model

A patient is referred to the palliative care program from ‘Continuing care’ or ‘Consult team’. When a patient referral is received, it is determined in the PC_Consult subnet whether the patient is eligible for Hospice Palliative Care. If the patient is not eligible, the workflow will end with a proper explanation. Otherwise the patient is sent for the next set of care tasks. The patient consults with a physician and the registration is performed. During each therapeutic encounter the workflow ensures that the basic steps are covered. Thus assessments, care planning, etc. occur regularly. The Information Sharing step identifies the confidentiality limits: what the patient and family already know, what they would like to know, and whether they are ready to listen is established before sharing information. If language is a barrier, translators who understand the medical concepts and terminology will be employed to facilitate information sharing. The patient’s and family’s desire for additional information is assessed regularly. In the Decision Making step the patient’s decision-making capacity, and the patient’s and family’s goals are assessed regularly. In this step, the patient and family will prioritize the importance of each of the identified issues. In addition, requests to withhold or withdraw therapies, and requests to initiate therapeutic interventions that appear to have no potential to benefit the patient and family, and the factors underlying those requests, are discussed openly. Therapies, therapeutic options and patient and family choices are reviewed. After the Decision Making comes the Care Planning step. The plan of care includes strategies for addressing each of the patient’s and family’s issues or opportunities, expectations, needs, hopes and fears, for delivering their chosen therapies, providing backup coverage if care givers are unable to fill their role in the plan of care, and for providing care giver respite, coping with emergencies, planning for discharge,

and etc. In this step, patients and families are assisted by the care team coordinators to select an appropriate setting of care. The plan and setting of care are reviewed by the care team and/or the organization's regional team and adjusted to compensate for changes in the patient's and family's status and choices. When it comes to the Care Delivery step, care is provided by a specific interdisciplinary care team that forms to care for each patient/family unit. Each care team has the leadership it needs to facilitate care team formation and function, and coordinate care planning and delivery. The setting of care is maintained so that it is safe, comforting, and provides ample opportunity for privacy and intimacy. The compatibility of the medicines is checked in this step. Any errors in therapy delivery are reported to supervisors immediately and documented appropriately. 'Regular Evaluation' is conducted regularly by the formal caregivers to assess and reinforce the patient's, family's and informal care giver's understanding of the situations such as the plan of care, the appropriate use of medications, therapies, equipment and supplies.

Registration

Fig. 8.4 shows the registration subnet. A nurse consultant collects the information from patient and makes a patient file. During the execution of the 'Fill_Patient_Info_Form' process two important data (i.e., patient's location and PPS value) are collected which are used in many places of the workflow to make decisions. In the task property file (i.e., 'Fill_Patient_Info_Form.java') these two attributes are set with possible non-deterministic values. The syntax for writing the abstract specification is provided here:

```
public class Fill_Patient_Info_Form extends UncompensableTaskMCImpl
{
```

```
Patient thePatient;
Patient.Location patientsLocation;
PatientData patientData;
Integer ppsValue;
@Override
public void action() {
// TODO Auto-generated method stub
}
@Override
public void finalize() {
thePatient.setPatientLocation(patientsLocation);
patientData.setpPS(ppsValue);
}
@Override
public void initialize() {
patientsLocation = (Patient.Location)Util.getNonDeterministicData(
new Patient.Location[]{Patient.Location.HOME, Patient.Location.HOSPITAL});
ppsValue = (Integer) Util.getNonDeterministicData(new Integer[]{40,50,60});
}
}
```

Intake

Patients issues are identified from the process 'Identify_Issues' and some data (e.g., preferred language, advanced directives, consent to contact other team members) are

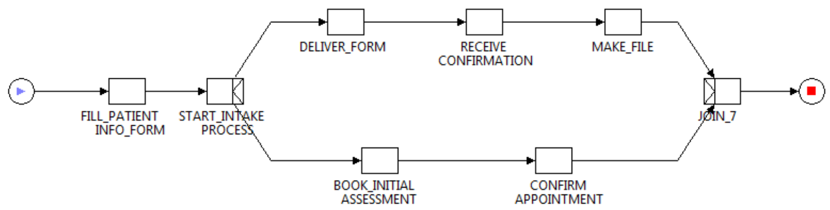

Figure 8.4: Registration

collected from this process. The patients' medication history is collected and distress screening is measured. Fig. 8.5 shows the subnet workflow.

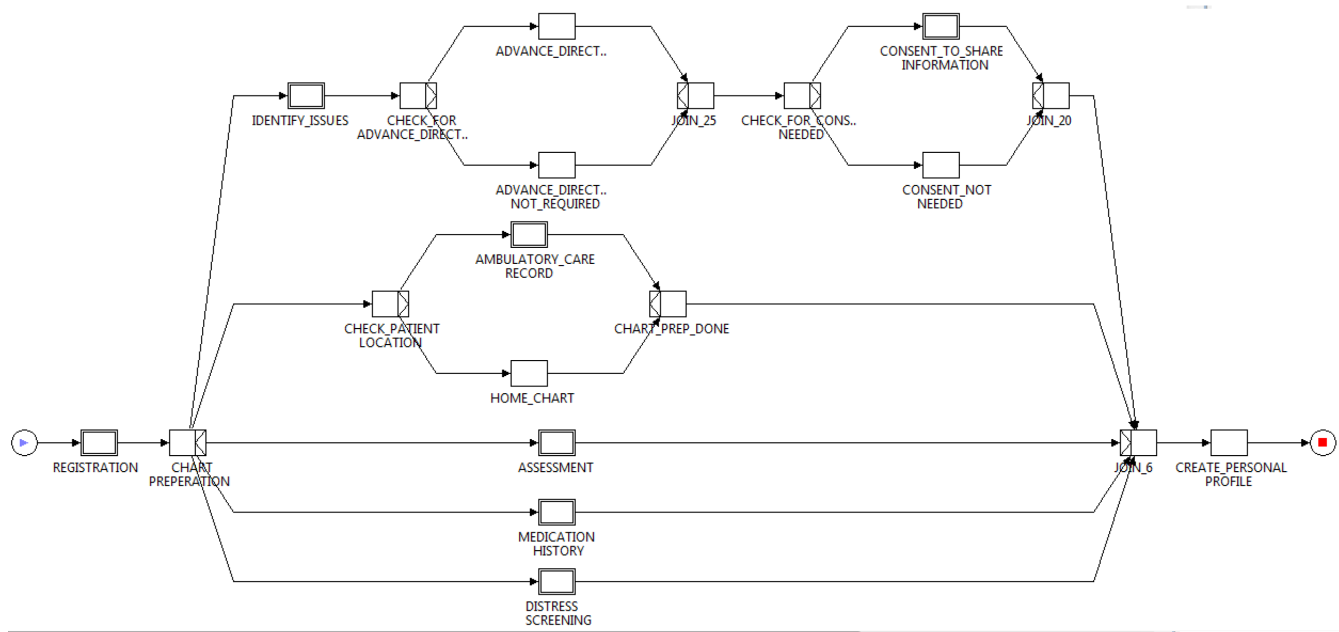

Figure 8.5: Palliative care workflow: Intake

Regular assessment

During each therapeutic encounter, the patient's condition is assessed by various performance metrices and tests in the 'Regular Assessment' subnet. Fig. 8.6 shows the 'Regular Assessment' subnet of the palliative care workflow which consists of 'Adult

pain meter', 'Edmonton assessment', 'Palliative performance scale', 'Distress screening' processes.

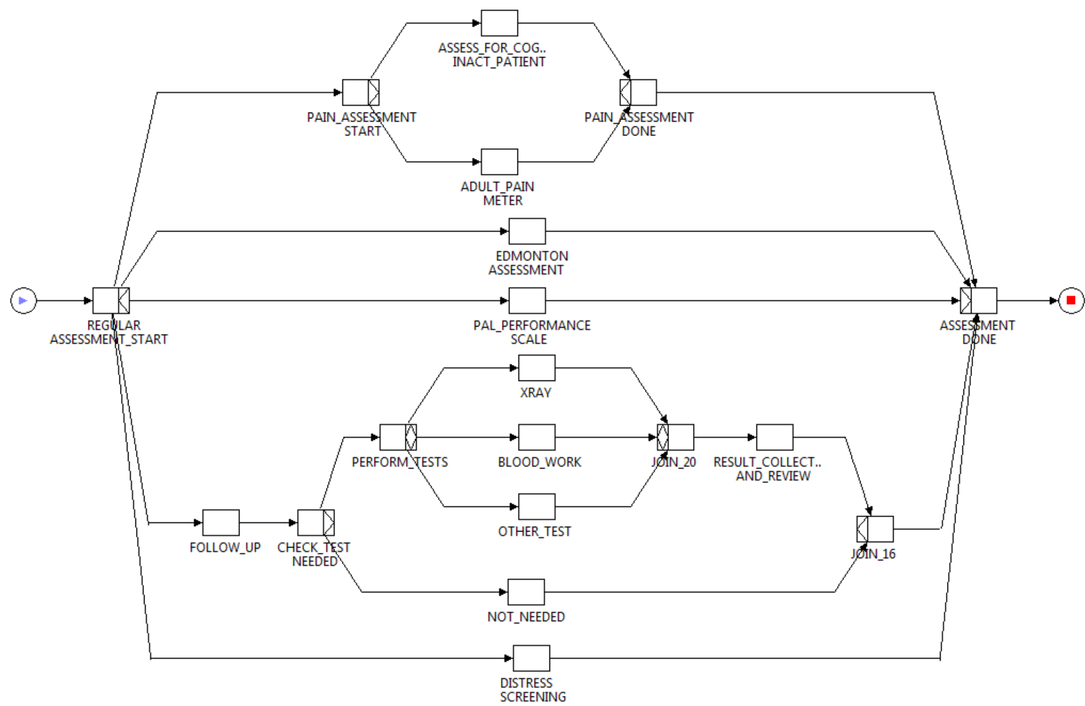

Figure 8.6: Palliative care workflow: Regular Assessment

Team building

A care team is built for a palliative care patient consisting of formal and informal care givers. We modeled the 'Team Building' subnet (Fig. 8.7) with *compensable tasks*. If the patient's location is at home, then a home service is assigned; but if there is no home service available for the given location, the patient must move to the hospital. This flow is designed with an 'Alternative Choice' composition. The code snippet to check the patient's location for the task 'Check_Patient_Location' is given below:

```
public class Check_Patient_Location extends InternalChoiceSplitMCImpl
```

```
{
Patient thePatient;
Patient.Location location;
.......
@Override
public void initialize() {
location = thePatient.getPatientLocation();


}
@Override
public boolean branchCondition(int branchNo) {
if(branchNo == 1)
return location == Patient.Location.HOME;
else
return location == Patient.Location.HOSPITAL;
}
}
```

To assign a *formal caregiver* for the patient an *speculative choice* block was designed. A *family physician* or a *pc physician*, whoever comes first, is assigned to the patient.

8.2 Verification of the palliative care process

Norms from the CHPCA National Model are general statements of guidelines. Some of the properties we verified are listed below, along with the CHPCA norms that they

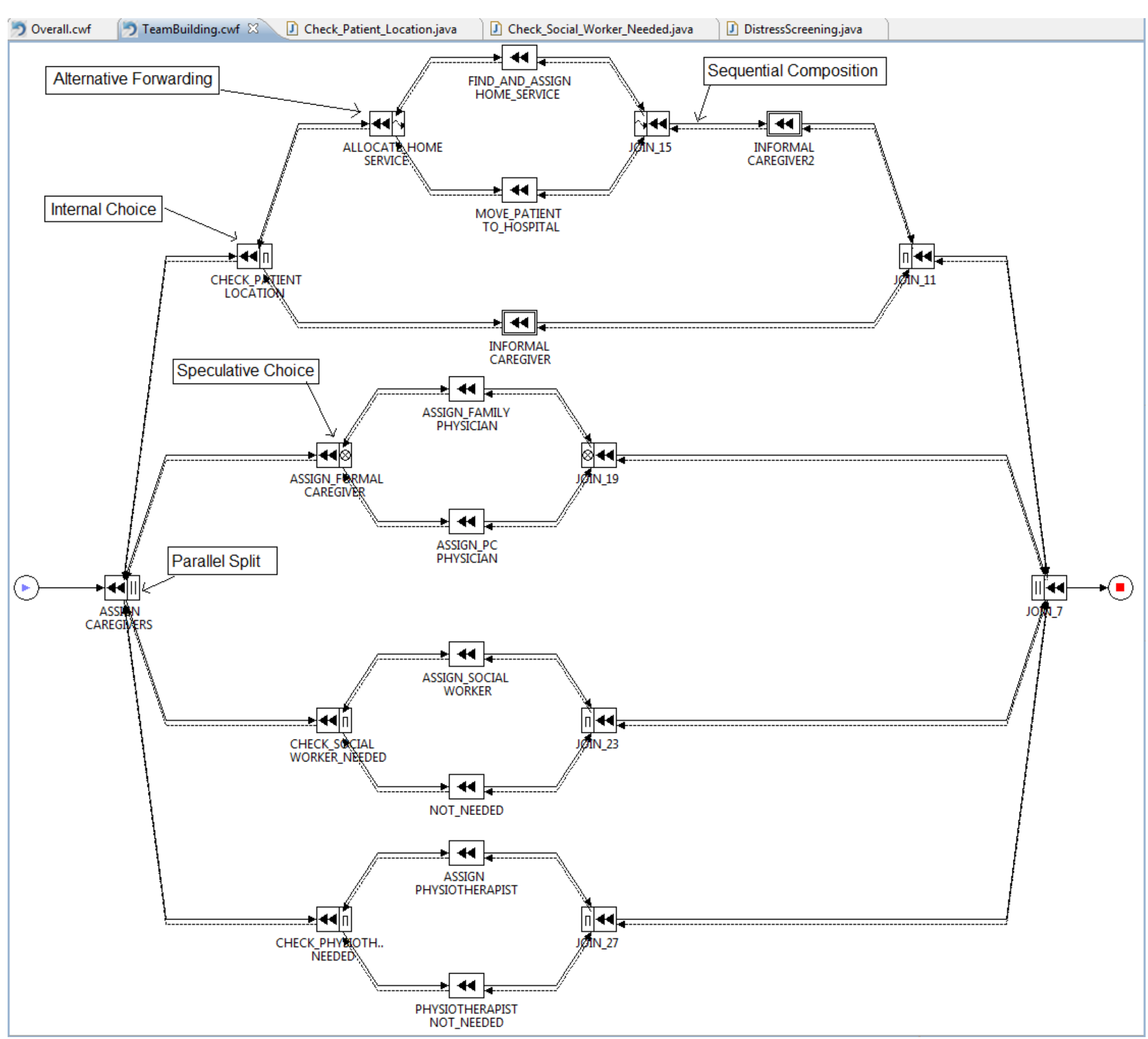

Figure 8.7: Palliative care workflow: Team Building

pertain to:

Prop1 (N5.1, N1.1)- If patient is distressed, then a Social Worker must be assigned in the care team.

```
#define patient_is_distressed (patientsDistressScreening_distressed == 1 )
#define patient_is_not_distressed (patientsDistressScreening_distressed == 0)
#define social_worker_is_assigned  (careTeam_socialWorker_ELEMENT_0_id > 0 )
#define end_of_care_reached patientData_isEndOfCare == 1
#define composition_ok  _Workflow_Composition_Ok == 1


#property ( G composition_ok) ->  G ( patient_is_distressed ->
F ( patient_is_not_distressed || end_of_care_reached ||
social_worker_is_assigned ) )
```

Prop2 (N1.2)- If the patient does not have anyone designated as their Next of Kin, then the “Interview With Family Member” task will not execute.

```
#define nextofkin_not_present (thePatient_nextOfKin_id == 0)
#define interview_done  (assessment_interviewWithFamilyMember > 0 )
#define composition_ok  _Workflow_Composition_Ok == 1


#property ( G composition_ok) ->  G ( nextofkin_not_present ->
F interview_done )
```

Prop3 (N3.5, N4.1, N4.4, N5.3)- If the patient is at home and has no family, then there must be an Informal Caregiver provided. Otherwise, the patient must be moved to the hospital.

```
#define patient_at_home thePatient_patientLocation == 0
#define patient_at_hospital  thePatient_patientLocation == 1
#define home_service_assigned  careTeam_homeServiceCaregivers_ELEMENT_0_id > 0
#define informal_cg_assigned  careTeam_familymemberCaregivers_ELEMENT_0_id > 0
#define composition_ok  _Workflow_Composition_Ok == 1

#property ( G composition_ok) ->  G ( patient_at_home -> F (home_service_assigned
|| informal_cg_assigned || patient_at_hospital ) )
```

Prop4 (N1.2)- If there is a risk for staff who visit the patient's home, then there will be no formal care provided to the patient in their home. They must come to the hospital.

```
#define patient_at_home thePatient_patientLocation == 0
#define patient_at_hospital thePatient_patientLocation == 1
#define there_is_a_risk  preVisitRiskAssessment_isThereARisk_yesOrNo == 1
#define composition_ok  _Workflow_Composition_Ok == 1

#property ( G composition_ok) -> G( there_is_a_risk -> F  patient_at_hospital )
```

Prop5 (N3.5, N4.1, N4.4)- If the patient is evaluated and assigned a PPS of 50% or lower, then they must be moved to the hospital.

```
#define pps_lower_than_50 patientData_pPS <= 50
#define patient_at_hospital thePatient_patientLocation == 1
#define composition_ok  _Workflow_Composition_Ok == 1
```

```
#property ( G composition_ok) -> G ( pps_lower_than_50 -> F  patient_at_hospital )
```

Prop6 (N3.5, N4.1, N4.4)- If the patient is evaluated and assigned a level of 3 or lower, then they must be moved to the hospital.

```
#define patient_at_level_three_or_lower patientData_level <= 3
#define patient_at_hospital thePatient_patientLocation == 1
#define composition_ok  _Workflow_Composition_Ok == 1

#property ( G composition_ok) -> G ( patient_at_level_three_or_lower ->
F  patient_at_hospital )
```

Prop7 (N3.7, N3.8, N3.9)- If the patient is no longer capable of making decisions, then there must be a proxy decision maker assigned.

```
#define patient_is_not_capable patientData_isCapable == 0
#define proxy_is_a_kin patientsAdvanceDirectives_proxy_id == 1
#define proxy_is_not_a_kin patientsAdvanceDirectives_proxy_id == 2
#define proxy_is_not_made patientsAdvanceDirectives_proxy_id == 0
#define composition_ok  _Workflow_Composition_Ok == 1

#property ( G composition_ok) -> G ( patient_is_not_capable ->
F  ( proxy_is_a_kin || proxy_is_not_a_kin ))
```

Prop8 (N2.4)- If the patient needs a translation of the information provided to him/her, then the translation will be provided.

```
#define translation_required patientsIssueLog_log1_preferredLanguage != 0
```

```
#define translation_is_done  patientsIssueLog_log1_isTranslationDone == 1


#property  G (translation_required -> F  translation_is_done )
```

Prop9 (N5.1, N1.1, N1.3)- If patients mobility is changed, then a Physiotherapist will be notified.

```
#define change_in_mobility theCommunicationSheet_changeInMobility == 1
#define physiotherapist_assigned  (careTeam_physiotherapist_ELEMENT_0_id > 0 )
#define composition_ok  _Workflow_Composition_Ok == 1


#property ( G composition_ok) ->  G ( change_in_mobility ->
F physiotherapist_assigned )
```

Prop10 (N2.1)- If the field “Consent to Contact Other Team Member” on ‘Issues Log’ is set to YES, then Consent to Share Information must be filled out.

```
#define advance_directive_is_required patientsIssueLog_advancedDirectives == 1
#define advance_directive_is_filled_out  (patientsAdvanceDirective_id > 0 )


#property   G ( advance_directive_is_required ->
F advance_directive_is_filled_out )
```

Note that Java entity class references and their attributes are translated to DiVinE data type and variables. To write a LTL-property the translated DiVinE variable names are used. The DVE code for Prop1 is shown here:

Property	Acc Cycle	WR + POR			POR		
		States	Memory (MB)	Time (s)	States	Memory (MB)	Time (s)
Prop1	No	107167421	83315.3	305.3	Unknown	Overflow	$> 1 hour$
Prop2	No	24501	220.0	7.9	Unknown	Overflow	$> 1 hour$
Prop3	No	126188210	88619.1	384.3	236576621	143836.2	1860
Prop4	No	13443	285.3	5.0	Unknown	Overflow	$> 1 hour$
Prop5	No	128013744	88920.0	397.9	251323543	153290.3	1931
Prop6	No	127934841	88894.5	396.1	213254702	140215.0	1854
Prop7	No	21234	274.5	6.1	Unknown	Overflow	$> 1 hour$
Prop8	No	12190	4.5	4.1	Unknown	Overflow	$> 1 hour$
Prop9	No	132038485	90285.3	315.0	211347231	139521.1	1833
Prop10	No	13479	230.1	9.7	202233451	125804.1	1803

Table 8.1: Verification results for the DiVinE model checker

All experiments were executed on the Mahone2 cluster of ACEnet, the high performance computing consortium for universities in Atlantic Canada. The tests were performed using DiVinE with 64 CPU's and 3GB memory (per CPU). After several iterations of modeling and verification the properties were verified; the results are shown in Table 8.1.

Chapter 9

Conclusion and future work

Model checking has been successfully applied to verify hardware systems (e.g., embedded system, circuits, communication protocols). It has a number of advantages over traditional approaches of validation/verification that are based on simulation, testing and deductive reasoning. This is a popular technique as it performs the verification process automatically and produces a counter-example that is useful in debugging a system. However, software systems are generally much more complex than hardware systems. To verify a software system using the model checking approach needs a great deal of research as model checking often suffers from the state explosion problem. Building the abstract finite state machine from the given software design and verifying the abstract finite state machine might solve the state explosion problem but it requires a lot of time and effort for modeling which is offen error prone. In this thesis we have presented a tool NOVA Workflow to design a workflow model that supports verification. With it one can graphically design a workflow and write business logic for the tasks. On the other hand abstract specifications (e.g., pre-conditions, actions, abstract values for variables, etc.) for tasks can be written in the task property file which is subject to the verification. The

workflow model is then automatically translated to a model checking program. This will significantly reduce the time and effort required to build an abstract state machine from an enterprise software system. Beside this we have presented a reduction algorithm that reduces the number of concurrent tasks from a workflow model and produces a small workflow model preserving required properties of the system. We have shown the applicability of this tool on a fairly big model for a health-care application. Our case study shows the effectiveness of the tool.

The specific contributions of this thesis are listed here:

- Proof of associativity for t-Calculus operatos ($||, \sqcap, \rightsquigarrow$ and $\otimes$) in section 3.2.9.
- A graphical compensable workflow modeling language based on t-Calculus operators:
  - Definition of atomic task, nonatomic task and compensable task: Definition 4.1 - 4.5;
  - Definition of Compensable Workflow Net in Definition 4.6;
  - Graphical workflow modeling language in Fig. 4.3 and its Petri net representation in section 4.2;
  - Soundness analysis of CWML in section 4.3;
- Automated translation of a workflow model to a model checker DiVinE, along with the proof of correctness:
  - Translation algorithm in Algorithm 1;
  - Proof of correctness in section 5.2.2;
- A Workflow Reduction method and its proof of stuttering equivalence:

  - Wofkflow reduction algorithm in Algorithm 3;
  - Proof of stuttering equivalence in section 6.3;
  - Effectiveness of the reduction studied in section 6.4;

- A workflow management system named NOVA Workflow to design, develop, verify and analyse compensable workflows discussed in (chapter 7). The software may be found in [7].

- Modeling and verification of a palliative care system as a case study, (chapter 8).

In future we will incorporate a sophisticated ontology to guide the workflow and explicit-time description methods [47] [30] into workflow modeling and verify larger models of real-world health-care processes with timing information. We will also incorporate a personalized health-care access control system into the NOVA Workflow.

Bibliography

[1] Divine project, http://divine.fi.muni.cz/. last accessed on nov: 2010.

[2] Eclipse plugin. http://www.eclipse.org/articles/article-plug-in-architecture/plugin_architecture.html/. last accessed, August 2010.

[3] Graphical editing framework. http://www.eclipse.org/gef/. last accessed, August 2010.

[4] Relational persistence for java and .net, http://hibernate.net/. last accessed, November 2010.

[5] Spring framework, http://www.springsource.org/. last accessed, November 2010.

[6] Bpel2pn. http://www2.informatik.hu-berlin.de/top/bpel2pn/. last accessed, February 2011.

[7] Nova workflow. http://logic.stfx.ca/software/nova-workflow/. last accessed, March 2011.

[8] Wsengineer. http://www.doc.ic.ac.uk/ltsa/eclipse/wsengineer/. last accessed, February 2011.

[9] Assaf Arkin and Intalio. Business process modeling language. BPML specification, November 2002.

[10] Ahmed Awad, Gero Decker, and Mathias Weske. Efficient compliance checking using bpmn-q and temporal logic. In *Proceedings of the 6th International Conference on Business Process Management*, BPM '08, pages 326–341, Berlin, Heidelberg, 2008. Springer-Verlag.

[11] Jiri Barnat, Lubos Brim, and Petr Rockai. Scalable multi-core ltl model-checking. In *SPIN*, pages 187–203, 2007.

[12] Tony Buzan. *The Mind Map Book*. Penguin Books, 1996.

[13] E. Clarke, O. Grumberg, and D. Long. Model checking. In *Proceedings of the NATO Advanced Study Institute on Deductive program design*, pages 305–349, Secaucus, NJ, USA, 1996. Springer-Verlag New York, Inc.

[14] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs*, pages 52–71, 1981.

[15] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986.

[16] Frank D. Ferris, Heather M. Balfour, Karen Bowen, Justine Farley, Marsha Hardwick, Claude Lamontagne, Marilyn Lundy, Ann Syme, and Pamela J. West. A model to guide hospice palliative care. *Canadian Hospice Palliative Care Association, 2002.*

[17] Hector Garcia-Molina and Kenneth Salem. Sagas. *SIGMOD Rec.*, 16:249–259, December 1987.

[18] Jifeng He. Modelling coordination and compensation. In *ISoLA*, pages 15–36, 2008.

[19] David Hollingsworth. The workflow reference model. Workflow Management Coalition, January 1995.

[20] Microsoft SAP Siebel IBM, Bea. Business process execution language for web services version 1.1. May 2003.

[21] He Jifeng. Formal methods and hybrid real-time systems. chapter Compensable programs, pages 349–363. Springer-Verlag, Berlin, Heidelberg, 2007.

[22] Edmund M. Clarke Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, 1999.

[23] Saul Aaron Kripke. A semantical analysis of modal logic I: Normal modal propositional calculi. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, 9:67–96, 1963.

[24] Gary T. Leavens, K. Rustan M. Leino, and Peter Muller. Specification and verification challenges for sequential object-oriented programs. *Form. Asp. Comput.*, 19:159–189, June 2007.

[25] Nazia Leyla, Ahmed Mashiyat, Hao Wang, and Wendy MacCaull. Workflow Verification with DiVinE. In *Parallel and Distributed Methods in verifiCation*. PDMC, 2009.

[26] Jing Li, Huibiao Zhu, and Jifeng He. Algebraic semantics for compensable transactions. In *Proceedings of the 4th international conference on Theoretical aspects of computing*, ICTAC'07, pages 306–321, Berlin, Heidelberg, 2007. Springer-Verlag.

[27] Jing Li, Huibiao Zhu, and Jifeng He. Specifying and verifying web transactions. In *Formal Techniques for Networked and Distributed Systems - FORTE 2008*, volume 5048 of *Lecture Notes in Computer Science*, pages 149–168. Springer-Verlag, 2008.

[28] Jing Li, Huibiao Zhu, Geguang Pu, and Jifeng He. A formal model for compensable transactions. In *Proceedings of the 12th IEEE International Conference on Engineering Complex Computer Systems*, pages 64–73, Washington, DC, USA, 2007. IEEE Computer Society.

[29] Jing Li, Huibiao Zhu, Geguang Pu, and Jifeng He. Looking into compensable transactions. *Software Engineering Workshop, Annual IEEE/NASA Goddard*, 0:154–166, 2007.

[30] Ahmed Shah Mashiyat, Fazle Rabbi, Hao Wang, and Wendy MacCaull. An automated translator for model checking time constrained workflow systems. In *FMICS*, pages 99–114, 2010.

[31] Jan Mendling. On the detection and prediction of errors in epc business process models. *EMISA Forum*, 27(2):52–59, 2007.

[32] Tadao Murata. Petri nets: properties, analysis, and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

[33] Carl Adam Petri. Kommunikation mit automaten. PhD thesis, Institut fur instrumentelle Mathematik, Bonn, 1962.

[34] Amir Pnueli. The temporal logic of programs. In *FOCS*, pages 46–57, 1977.

[35] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in cesar. In *Proceedings of the 5th Colloquium on International Symposium on Programming*, pages 337–351, London, UK, 1982. Springer-Verlag.

[36] Fazle Rabbi, Hao Wang, and Wendy MacCaull. Yawl2dve: An automated translator for workflow verification. In *Secure Software Integration and Reliability Improvement*, pages 53–59, 2010.

[37] M.U. Reichert, S.B. Rinderle, U. Kreher, H. Acker, M. Lauer, and P. Dadam. Adept2 - next generation process management technology. In *Proceedings Fourth Heidelberg Innovation Forum*, Aachen, April 2007. D.punkt Verlag.

[38] Wasim Sadiq and Maria E. Orlowska. Applying graph reduction techniques for identifying structural conflicts in process models. In *Proceedings of the 11th International Conference on Advanced Information Systems Engineering*, CAiSE '99, pages 195–209, London, UK, 1999. Springer-Verlag.

[39] Wasim Sadiq and Maria E. Orlowska. Analyzing process models using graph reduction techniques. *Inf. Syst.*, 25(2):117–134, 2000.

[40] W. M. P. van der Aalst and Ter. YAWL: yet another workflow language. *Information Systems*, 30(4):245–275, June 2005.

[41] Wil M. P. van der Aalst. The application of petri nets to workflow management. *Journal of Circuits, Systems, and Computers*, 8(1):21–66, 1998.

[42] Wil M. P. van der Aalst. Business process management demystified: A tutorial on models, systems and standards for workflow management. In *Lectures on Concurrency and Petri Nets*, pages 1–65, 2003.

[43] Wil M. P. van der Aalst and Kees van Hee. *Workflow Management: Models, Methods, and Systems.* MIT Press, 2002.

[44] Wil M. P. van der Aalst, Kees M. van Hee, Arthur H. M. ter Hofstede, Natalia Sidorova, H. M. W. Verbeek, Marc Voorhoeve, and Moe Thandar Wynn. Soundness of workflow nets with reset arcs. *T. Petri Nets and Other Models of Concurrency*, 3:50–70, 2009.

[45] Boudewijn F. van Dongen, Wil M. P. van der Aalst, and H. M. W. Verbeek. Verification of epcs: Using reduction rules and petri nets. In *CAiSE*, pages 372–386, 2005.

[46] Martin Vasko and Schahram Dustdar. A view based analysis of workflow modeling languages. In *Proceedings of the 14th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 293–300, Washington, DC, USA, 2006. IEEE Computer Society.

[47] Hao Wang and Wendy MacCaull. An efficient explicit-time description method for timed model checking. In *PDMC*, pages 77–91, 2009.

Appendix A

A.1 Syntax for writing abstract tast specification

A.1.1 Supported data types

To write an abstract task specification for verification you can only use byte, integer, long and boolean, although these data types can be specified inside a Class, List, Vector or Aggregate Class. *String*, *Float* and *Double* are not supported as they are not supported by the model checker. NOVA workflow handles 'entity class references' and 'variables with primitive data type' in two different ways; entity class references are translated to the 'Global variables' in DVE and variables with primitive data types are translated to 'Local variables'. While writing the specification, the entity class references can be used for interprocess communication.

A.1.2 Details of task property files

Abstract task specifications are written in task property files those are Java classes. NOVA workflow provides different interfaces for writing task specification. Table A.1 shows the list of task types and their interfaces and methods. To initialize a local variable (primitive data type) with non-deterministic values or from any global variable

(entity class reference) the *initialize()* method is used. Statements with assignment or arithmatic operations can be written in the *action()* method. Boolean expressions can be written inside the branchCondition() method; this method is used to write the condition for different branches of a split task. *Finalize()* method can be used to set something into a global variable (entity class reference). Table A.2 lists allowed syntax for different methods.

Task Type	Interface	Abstract Methods
AtomicTask	UncompensableTaskMCImpl	initialize(), action(), finalize()
AndSplitTask	AndSplitMCImpl	initialize(), action(), finalize()
AndJoinTask	AndJoinMCImpl	initialize(), action(), finalize()
XorSplitTask	XorSplitMCImpl,	initialize(), action(), finalize(),
	IMCBranchCondition,	branchCondition(),
	IMCBranchOrder	getBranchOrder()
XorJoinTask	XorJoinMCImpl	initialize(), action(), finalize()
OrSplitTask	ORSplitMCImpl,	initialize(), action(), finalize(),
	IMCBranchCondition	branchCondition()
OrJoinTask	ORJoinMCImpl	initialize(), action(), finalize()
LoopSplitTask	LoopSplitMCImpl,	initialize(), action(), finalize(),
	IMCBranchCondition,	branchCondition(),
	IMCBranchOrder	getBranchOrder()
LoopJoinTask	LoopJoinMCImpl	initialize(), action(), finalize()

Table A.1: Task types and interfaces

Task Type	Interface	Abstract Methods
CompensableTask	CompensableTaskMCImpl	initialize(), action(), finalize(), abortInitialize(), abort(), abortFinalize()
ParallelSplitTask	ParallelSplitMCImpl	initialize(), action(), finalize(), abortInitialize(), abort(), abortFinalize()
ParallelJoinTask	ParallelJoinMCImpl	initialize(), action(), finalize(), abortInitialize(), abort(), abortFinalize()
InternalChoieSplitTask	IntrnlChoiceSplitMCImpl, IMCBranchCondition	initialize(), action(), finalize(), abortInitialize(), abort(), abortFinalize(), branchCondition()
InternalChoiceJoinTask	IntrnlChoiceJoinMCImpl	initialize(), action(), finalize(), abortInitialize(), abort(), abortFinalize()
SpeculativeSplitTask	SpecChoiceSplitMCImpl	initialize(), action(), finalize(), abortInitialize(), abort(), abortFinalize()
SpeculativeJoinTask	SpecChoiceJoinMCImpl	initialize(), action(), finalize(), abortInitialize(), abort(), abortFinalize()
AlternativeSplitTask	AltSplitMCImpl, IMCBranchOrder	initialize(), action(), finalize(), abortInitialize(), abort(), abortFinalize(), getBranchOrder()

Table A.1: Task types and interfaces (Continued)

Task Type	Interface	Abstract Methods
AlternativeJoinTask	AltJoinMCImpl	initialize(), action(), finalize(), abortInitialize(), abort(), abortFinalize()
BackwardHandlerTask	BckHandlerMCImpl	initialize(), action(), finalize()
ForwardHandlerTask	FwdHandlerMCImpl	initialize(), action(), finalize(), abortInitialize(), abort(), abortFinalize()
Programmable CompensationTask	ProgCmpMCImpl	initialize(), action(), finalize()

Table A.1: Task types and interfaces (Continued)

Method Name	Allowed Syntax

initialize(),
abortInitialize()
	localVar = (Integer)Util.getNonDeterministicData(new Integer[]1,2,..);
	localVar = (Long)Util.getNonDeterministicData(new Long[]1,2,..);
	localVar = (Byte)Util.getNonDeterministicData(new byte[]1,2,..);
	localVar = (Boolean)Util.getNonDeterministicData(new Booelan[]true,false);
	localVar = globalVar.getAttribute();
	localVar = globalVar.getAggregateProperty().getAttribute();
	localVar = globalVar.getListAttribute().get(index);
	localVar = globalVar.getListAttribute().get(index).getAttribute();

Table A.2: List of allowed syntax in task property file

Method Name	Allowed Syntax
action(), abort()	Assignment statements using local variables and numbers. Assignment statements can contain: - Numbers, true, false - Parenthesis: (,) -Variable identifiers -Unary operators () -Binary operators ($\|, \&, ==, !=, <, \leq, >, \geq, >>, <<, -, +, /, *, \%$)

finalize(),
abortFinalize()
	globalVar.setAttribute(localVar);
	globalVar.getAggregateProperty().setAttribute(localVar);
	globalVar.getListAttribute().set(index, localVar);
	globalVar.getListAttribute().get(index).setAttribute(localVar);

Table A.2: List of allowed syntax in task property file (continued)

Method Name	Allowed Syntax
branchCondition (*int branchNumber*)	$if(branchNumber == 1)$ return Boolean_Expression; $else\ if(branchNumber == 2)$ return Boolean_Expression; $else$ return Boolean_Expression; Boolean expressions can be written using local variables and numbers. The statements can contain: -Numbers, true, false -Parenthesis: (,) -Variable identifiers -Unary operators () -Binary operators $(|, \&, ==, !=, <, \leq, >, \geq, >>, <<, -, +, /, *, \%)$
getBranchOrder (*int branchNumber*)	$if(branchNumber == 1)$ return 2; $else\ if(branchNumber == 2)$ return 1;

Table A.2: List of allowed syntax in task property file (continued)

A.2 Workflow engine service

Method Name	Functionality
createNewWorkflowInstance	This method is used to create a new workflow instance. It inserts a new record in table WfInstance and generates a unique id for the newly created instance.
getInstance	To know details about an instance this method can be used.
getAllActiveInstances	This method returns all Active instances
getAvailableMethods	There are two overload methods: i) Takes a workflow instance as parameterand returns all InstanceInfo containing taskId and available methods of all active tasks ii)Takes an instanceId and taskId as parameter and returns all available methods for the task

Table A.3: Description of the methods of `IWorkflowEngineService`