SPECIFICATION AND SCHEDULING OF WORKFLOWS UNDER RESOURCE ALLOCATION CONSTRAINTS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
THE MIDDLE EAST TECHNICAL UNIVERSITY

BY

PINAR (KARAGÖZ) ŞENKUL

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

IN

THE DEPARTMENT OF COMPUTER ENGINEERING

JUNE 2003

Approval of the Graduate School of Natural and Applied Sciences.

Prof. Dr. Tayfur Öztürk
Director

I certify that this thesis satisfies all the requirements as a thesis for the degree of Doctor of Philosophy.

Prof. Dr. Ayşe Kiper
Head of Department

This is to certify that we have read this thesis and that in our opinion it is fully adequate, in scope and quality, as a thesis for the degree of Doctor of Philosophy.

Assoc. Prof. Dr. İsmail Hakkı Toroslu
Supervisor

Examining Committee Members

Prof. Dr. Varol Akman

Prof. Dr. Adnan Yazıcı

Assoc. Prof. Dr. Özgür Ulusoy

Assoc. Prof. Dr. Ferda Nur Alpaslan

Assoc. Prof. Dr. İsmail Hakkı Toroslu

ABSTRACT

SPECIFICATION AND SCHEDULING OF WORKFLOWS UNDER RESOURCE ALLOCATION CONSTRAINTS

Şenkul, Pınar (Karagöz)

Ph.D., Department of Computer Engineering

Supervisor: Assoc. Prof. Dr. İsmail Hakkı Toroslu

June 2003, 141 pages

Workflow is a collection of tasks organized to accomplish some business process. It also defines the order of task invocation or conditions under which task must be invoked, task synchronization, and information flow. Before the execution of the workflow, a correct execution schema, in other words, the schedule of the workflow, must be determined. Workflow scheduling is finding an execution sequence of tasks that obeys the business logic of workflow. Research on specification and scheduling of workflows has concentrated on temporal and causality constraints, which specify existence and order dependencies among tasks. However, another set of constraints that specify resource allocation is also equally important. The resources in a workflow environment are agents such as person, machine, software, etc. that execute the task. Execution of a task has

a cost and this may vary depending on the resources allocated in order to execute that task. Resource allocation constraints define restrictions on how to allocate resources, and scheduling under resource allocation constraints provide proper resource allocation to tasks. In this thesis, we present two approaches to specify and schedule workflows under resource allocation constraints as well as temporal and causality constraints. In the first approach, we present an architecture whose core and novel parts are a specification language with the ability to express resources and resource allocation constraints and a scheduler module that contains a constraint solver in order to find correct resource assignments. In the second approach, we developed a new logical formalism, called Concurrent Constraint Transaction Logic (CCTR) which integrates constraint logic programming (CLP) and Concurrent Transaction Logic, and a logic-based workflow scheduler that is based on this new formalism. CCTR has the constructs to specify resource allocation constraints as well as workflows and it provides semantics for these specifications so that validity of a schedule can be checked.

Keywords: workflow, scheduling, logic, constraint programming, resource, resource allocation constraints

ÖZ

KAYNAK AYRIM KISITLARI ALTINDA İŞAKIŞLARININ TANIMLANMASI VE PLANLANMASI

Şenkul, Pınar (Karagöz)

Doktora, Bilgisayar Mühendisliği Bölümü

Tez Yöneticisi: Doç. Dr. İsmail Hakkı Toroslu

Haziran 2003, 141 sayfa

İşakışı, karmaşık ticari işlemleri oluşturmak üzere biraraya gelmiş görevler topluluğudur. İşakışları, aynı zamanda, görevlerin çalıştırılma sıraları, hangi şartlar altında çalışması gerektiği, görev senkronizasyonu ve görevler arası bilgi akışına dair tanımlamaları da içerir. İşakışı çalışmadan önce, doğru bir çalışma şeması, diğer bir deyişle, işakışı planı belirlenmelidir. İşakışı planlama, görevlerin çalışma sıralarının, işakışı mantığına uygun olarak bulunmasıdır. İşakışı modelleme ve planlama üzerine yapılan araştırmalar, zaman ve nedensellik kısıtları üstünde yoğunlaşmıştır. Ancak, kaynak ayrımını tanımlayan diğer bir kısıt tipi de işakışları için eşit derecede önemlidir. Bir işakışı ortamında, görevlerin yapılması için gerekli olan personel, makina, yazılım ve

benzeri ihtiyaçlara kaynak adı verilir. Bir görevin yerine getirilmesinin bir maliyeti vardır ve bu maliyet genellikle, görevin yerine getirilmesi için ayrılan kaynağa bağlı olarak değişir. Kaynak ayrım kısıtları, kaynakların dağılımına dair sınırlamalar tanımlar ve bu kısıtlar altında yapılan işakışı planlaması, görevler için doğru kaynak ayrımını sağlar. Bu çalışmada, zaman ve nedensellik kısıtlarının yanı sıra, kaynak ayrım kısıtları altında işakışı modelleme ve planlamayı sağlamak amacıyla iki yaklaşım sunulmaktadır. Birinci yaklaşımda, temel ve yenilik taşıyan parçaları, kaynakları ve kaynak ayrım kısıtlarını ifade edebilen bir işakışı tanımlama dili ve tanımlanan kısıtlar altında doğru kaynak ayrımını bulmak için, kısıt çözücü içeren bir planlayıcı olan bir işakışı yönetim sistemi mimarisi anlatılmıştır. İkinci kısımda ise, Concurrent Constraint Transaction Logic (CCTR, Eşzamanlı Kısıt İşlem Mantığı) adıni verdiğmiz, Constraint Logic Programming (CLP, Kısıt Mantık Programlama) ve Concurrent Transaction Logic'i (CTR, Eşzamanlı İşlem Mantığı) birleştiren bir formalizasyon ve bu formalizasyonu temel alarak geliştirilen bir işakışı planlayıcısı anlatılmaktadır. CCTR ile kaynak ayrım kısıtlarını tanımlamak, anlamlarını modellemek, böylece bu kısıtlar altındaki bir işakışı planını doğrulamak mümkün olmaktadir.

Anahtar Kelimeler: işakışı planlama, mantık, kısıt programlama, kaynak, kaynak ayrım kısıtları

ACKNOWLEDGMENTS

Firstly, I would like to thank to my thesis advisor, Assoc. Prof. Dr. İsmail Hakkı Toroslu, for his guidance and for the opportunities he provided throughout my research. I would like to thank to also Prof. Dr. Michael Kifer for his support and guidance during my stay at Stony Brook and for responding my mails and thus keeping on his guidance even after I came back to METU. Thanks are also to Prof. Dr. Asuman Doğaç for her support and company during VLDB conference and to our department chair, Prof. Dr. Ayşe Kiper, for providing me the research environment during the last terms of my thesis. I would like to thank to my thesis jury members for their comments. Thanks to my friends, Arzuhan and Hasan, who helped me a lot when I first arrived Stony Brook. I also very much like to thank to my mother and father for always being caring and supportive parents. The last but not the least, thanks are due to my beloved husband, Selçuk, for accompanying me during my stay at Stony Brook and being right beside me at my ups and downs during this thesis work.

To my family

TABLE OF CONTENTS

ABSTRACT iii

ÖZ v

ACKNOWLEDGMENTS vii

DEDICATON viii

TABLE OF CONTENTS ix

LIST OF FIGURES xii

CHAPTER

1 Introduction 1

2 Previous Work 7

2.1 Previous Work on Workflow Modeling and Scheduling 7

2.1.1 Workflow Modeling Using Temporal Logic 7

2.1.2 Workflow Modeling Using Event Algebra 12

2.1.3 Workflow Modeling Using Event-Condition-Action Rules 17

2.1.4 Workflow Modeling Using Extended Transaction Models: ACTA 20

2.1.5 Workflow Modeling Using Petri Nets 24

2.2 Constraint Logic Programming 36

2.2.1 Basic Features of CLP 38

2.2.2 Domains in CLP 41

2.2.3 Consistency Techniques 42

2.2.4 CLP Systems 43

2.3 Previous Work on Resource Modeling in Workflows 45

2.4 Other Related Work 51

3 Modeling of Workflows and Resource Allocation Constraints 53
3.1 Workflow Structure 53
3.2 Workflow Modeling 55
3.3 Modeling Resource Allocation Constraints 58
4 Using CLP to Schedule Workflows Under Resource Allocation Constraints 60
4.1 System Architecture and Specification Language 60
4.1.1 System Architecture 60
4.1.2 Workflow Specification Language: WSL 62
4.1.2.1 Workflow Blocks 62
4.1.2.2 Temporal and Causality Constraints 63
4.1.2.3 Resources 64
4.1.2.4 Resource Allocation Constraints 64
4.1.2.5 Example - House Construction 67
4.2 Defining Workflow Using Constraint Programming 67
4.2.1 Translation from *WSL* to Constraint Language (*Wfto-Con*) 67
4.2.1.1 Sequential Block 68
4.2.1.2 AND Block 68
4.2.1.3 OR Block 68
4.2.1.4 XOR Block 69
4.2.1.5 Iteration Block 69
4.2.1.6 Condition 70
4.2.1.7 Temporal and Causality Constraints 70
4.2.1.8 Defining Resources 71
4.2.1.9 Resource Allocation Constraints 71
4.2.1.10 Example 72
4.2.2 Experiments and Efficiency Issues 76
5 Developing a Logic-based Framework to Schedule Workflows Under Resource Allocation Constraints 81
5.1 Concurrent Constraint Transaction Logic (CCTR) 82
5.1.1 Syntax 83
5.1.2 Semantics 84
5.1.2.1 States and Oracles 84
5.1.2.2 Partial Schedules 85

5.1.2.3 Resource and Resource Assignment 87
5.1.2.4 Constraint Universe 88
5.1.3 Model Theory of CCTR 89
5.2 Wf-Constraint Universe 93
5.3 CCTR as a Workflow Scheduler 98
5.3.1 Transformation Rules and Specifying Constraint System 99
5.3.2 CTR Interpreter 103
5.3.3 Constraint Solver 105
5.3.4 Correctness of the Scheduler 106
6 Discussions and Applications 109
6.1 Comparison of the Approaches 109
6.2 Applications 110
6.2.1 Workflows 110
6.2.1.1 Travel Agency Workflow 111
6.2.1.2 Telecommunication Service Providing Workflow 111
6.2.1.3 Conference Planning 112
6.2.2 Composite Web Services 113
6.2.3 Workflows in Non-cooperative Environments 115
7 Conclusion 118
REFERENCES 120
APPENDIX
A Proofs 127
A.1 Proof of Lemma 5.20 127
A.2 Proof of Theorem 5.21 129
B Definitions for Applications 133
B.1 Travel Agency Workflow 133
B.2 Telecommunication Service Providing Workflow 134
B.3 Conference Planning 136
B.4 Composite Web Services 138
VITA 140

LIST OF FIGURES

1.1 House construction workflow 5

2.1 Dependency automata 9
2.2 Automaton of Figure 2.1(a) augmented with self-looping transitions . . 11
2.3 Scheduler transitions for the dependency D_1 in the traveler workflow . . 15
2.4 A Petri Net representation of a task 26
2.5 Petri Net representation of dependencies 28
2.6 Petri Net representation of logical relations between dependencies 30
2.7 A Petri Net model of a complaint processing workflow 34

3.1 Workflow structure 54
3.2 Workflow modeling frameworks 56

4.1 System architecture 61
4.2 House construction workflow in WSL 66
4.3 Subworkflow for house construction example 72
4.4 Resource allocation cost table 72
4.5 Blocks of house constraint example 73
4.6 Constraints for house constraint example 73
4.7 Resource definitions for house construction example 74
4.8 Solution produced by Oz 75
4.9 Schedule for house construction example 76
4.10 Structures used in the first set of experiments 77
4.11 Structures used in the second set of experiments 77
4.12 Effect of increase in constraint set 79
4.13 Effect of increase in number of tasks 79

5.1 Examples of set constraints and their properties 96
5.2 The big picture 98
5.3 Transformation rules for a workflow scheduler 99
5.4 Transformation for house construction workflow 100
5.5 Template rules for constraint systems 101
5.6 Placeholders for problem-specific predicates and resource assignments . 102
5.7 Placeholders for house construction example 103
5.8 CTR deduction for Example 5.19 104
5.9 Constraints computed for house construction workflow 105

6.1 Telecommunications workflow control flow graph 112
6.2 Conference planning workflow control flow graph 113
6.3 Composite web service control flow graph 114
6.4 Control flow graph for workflow with non-cooperative task 116

A.1 Constraint universe predicates vs. template rules 128
A.2 Signatures for predicates and functions 129

B.1 Constraint definitions for travel planning workflow 134
B.2 Telecommunications workflow in constraint language 135
B.3 Telecommunications workflow constraints in constraint language 135
B.4 Constraint definitions for telecommunications workflow 136
B.5 Definition of *disjoint* constraint 136
B.6 Example sch. and const. sets for telecom. workflow 137
B.7 Conference planning workflow in constraint language 137
B.8 *Before* constraint in constraint language 138
B.9 *Not-parallel* constraint in constraint language 138
B.10 Constraint definitions for conference planning workflow 138
B.11 Definition of *before* constraint 139
B.12 Definition of *not-parallel* constraint 139
B.13 Placeholders for web service composition 140

CHAPTER 1

Introduction

Workflow can be defined as a coordinated set of activities that act together to perform a well-defined and complex process while satisfying a set of constraints which represents the business logic of the workflow. Real life processes such as multi-agent banking transactions, trip planning, catalog ordering and fulfillment processes in an enterprise are typical examples of workflows.

A workflow management system *(WfMS)* [47, 41, 5] defines a model and tools for the specification, analysis and execution of workflows. However, specification models of today's WfMS's are not fully sufficient for the specification of all kinds of constraints a workflow may contain. They ignore a very important class of constraints, those that arise from resource allocation. Physical objects like devices and personnel who carry out tasks can be considered as resources. Sometimes resources are called *agents*. In real life applications, usually costs like time and money are associated with the execution of a task by an agent. Since typically resources not limitless, scheduling of a workflow execution should involve decisions as to which resources to use and when.

Scheduling of workflows is a problem of finding a *correct* execution sequence for the workflow tasks, *i.e.*, execution that obeys the constraints that embody the business logic of the workflow. Most research on workflow scheduling has concentrated on temporal and causality constraints, which specify the correct ordering of tasks ignoring the resource allocation constraints [7, 72, 73, 77, 1, 75, 29, 11]. Although resource management has been recognized as an important aspect of a WfMS [24, 4, 77], most of the work has focused on modeling the various resources [82, 34, 48] with no or little attention devoted to scheduling under the constraints associated with such resources.

Temporal and causality constraints help to describe the execution flow of a workflow. For example constraints like *tasks 1 and 2 must both execute*, and, *if task 1 executes then tasks 2 and 3 must execute as well* are typical examples of temporal/causality constraints. On the other hand, *resource allocation constraints* are needed in order to determine which resources should be used to execute the tasks and when [55, 70]. For example, a human agent or a machine might not be used for executing different tasks simultaneously since undivided attention could be required by tasks defined in parallel. For such cases we should be able to specify that the same agents cannot be assigned to parallel branches of the workflow. Similarly, we might want to define a limit on the cost of agent allocation.

Most of the previous work on workflows concentrated on issues related to run-time check of the constraints. If a workflow is executed before it is verified, its constraints may be checked and a schedule might be obtained incrementally during the execution. However, a workflow specification might be unexecutable because of its incorrect design or conflicting constraints. At some point of the execution, if it is detected that the

workflow cannot be completed because of design errors, some rollback operations might be needed before abandoning the execution of the workflow. Obviously such cases cause waste of resources. Therefore, it is important to verify the given workflow specification, and this verification can be done by obtaining a feasible workflow schedule providing proper resource allocation, prior to the execution.

In this thesis, two approaches have been studied in order to pre-schedule the workflows under resource allocation constraints. The first approach develops a scheduler architecture whose core component is a constraint solver. The second approach is based on a logic-based framework. This framework is realization of a logic-based language developed within the scope of this thesis in order to model and schedule workflows under resource allocation constraints.

In the first approach, we present a workflow management system architecture that provides modules to model resources and resource allocation constraints and to find schedules fulfilling these constraints. In order to find schedules, *constraint programming approach* is used. That is, scheduler incorporates an off-the-shelf constraint solver to obtain a feasible schedule for the workflow satisfying both resource allocation and temporal/causality constraints. *Operations Research (OR)* and *constraint programming* have been successfully used for problems such as *job-shop scheduling*, in which proper machine (i.e., resource) allocation constitutes an important part of the solution. (For more information on constraint programming, reader may refer to [50, 40, 10, 16]). Although we were inspired by these previous work, in this thesis, we study a new domain, namely workflow scheduling under resource allocation constraints, and contrary to most OR problems, our goal is to find a feasible resource allocation rather than

finding the optimal solution.

The main contributions of this method can be summarized as follows:

- As a part of the architecture, we have developed a specification language that can model cost information for resources and specify resource allocation constraints, which is not provided by previous workflow definition languages.

- The architecture contains a translator module that produces a constraint program corresponding to the workflow specification. The scheduler module, which incorporates a constraint solver, gets this program and produces a schedule satisfying resource allocation constraints.

In the second part, we develop a new logical framework, *Concurrent Constraint Transaction Logic* (abbr., *CCTR*), that extends Concurrent Transaction Logic (abbr., *CTR*) [13] by incorporating ideas from Constraint Logic Programming (CLP) [49, 50]. This new framework can be used for the specification, verification and the scheduling of workflows containing resource allocation constraints in addition to ordinary temporal/causality constraints.

The role of CCTR in our framework is to model workflows and specify all kinds of constraints in a rigorous and precise way. The semantics of the CCTR modeling of a workflow represents to a schedule that contains both an execution ordering that the specified workflow can execute, and a set of resource assignments to the tasks of the workflow satisfying all the given constraints. The realization of this framework can be summarized in three components:

- A formula transformer, that transforms the conjunctive CCTR formula which

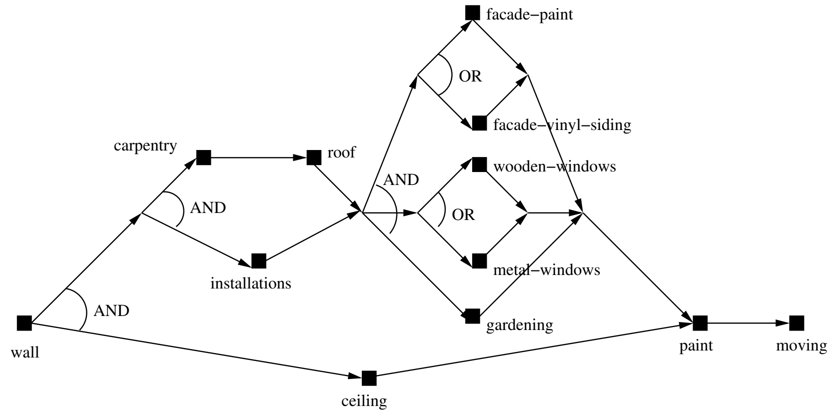


Figure 1.1: House construction workflow

specifies the workflow in conjunction with the cost and control constraints, into a conjunction-free CTR formula that contains exactly the same constraints.

- A CTR interpreter, that solves the non-constraint part of the formula in order to determine the partial schedule of the workflow and extract the constraints.

- A constraint solver, that determines the resource allocations to the tasks of the workflow by solving the constraints.

The combination of the last two steps together constitutes the solution for the original conjunctive CCTR formula that contains both the workflow and the resource allocation constraints.

A Motivating Example. The following example scenario, derived from [69] illustrates the resource allocation constraints defined on workflows.

Example 1.1 House construction company A builds a house, does gardening and moves customer's furniture into the new house. The company subcontracts with other companies for the various subtasks. There can be several candidate subcontractors,

or the same company may be qualified to do several subtasks. To satisfy customer's requirements and to maximize its own profit, company A wants to choose the most appropriate companies to subcontract with. The workflow is shown in Figure 1.1. In the figure, the AND-nodes represent branches of work that can be done in parallel (but all the parallel branches must be finished). OR-nodes represent alternative courses of action. For instance, the facade can be painted or the customer might choose to use vinyl siding. Tasks that must be done in sequence are connected via directed edges.

Resource allocation constraints in this workflow can include:

1. The budget for the construction should not exceed the given amount.
2. The construction should not last longer than the given duration.
3. Different companies must be chosen for parallel tasks (to speed up the construction).

Organization. The organization of this thesis is as follows: Chapter 2 presents the related work. Preliminaries on workflow modeling and resource allocation constraints are explained in Chapter 3. Our first approach to workflow scheduling under resource allocation constraints is presented in Chapter 4. The second approach that involves a logic-based framework is presented in Chapter 5. Comparison of two approaches and some applications are given in Chapter 6. Chapter 7 concludes the thesis.

CHAPTER 2

Previous Work

2.1 Previous Work on Workflow Modeling and Scheduling

The previous work on workflow modeling and scheduling generally involves ordering dependencies among the workflow tasks. There are different approaches to model and schedule under these dependencies. In this chapter, we present approaches that use logic, event algebra, triggers and Petri nets.

2.1.1 Workflow Modeling Using Temporal Logic

In [7], Attie et al. proposed to model workflows as a set of intertask dependencies. Both local and global constraints can be modeled in this way. The tasks in a workflow are described in terms of significant events. A typical event is the beginning or termination of a task, but it can also be sending an email to the boss, printing a report, etc.

When an event is received for execution, it is checked against every dependency and based on that the event might be accepted, rejected, or delayed and scheduled later. The dependencies are specified as formulae in Computational Tree Logic (CTL)

[37]. The scheduler enforces these dependencies by converting them into automata and ensuring that the sequence of scheduled events is accepted by all these automata.

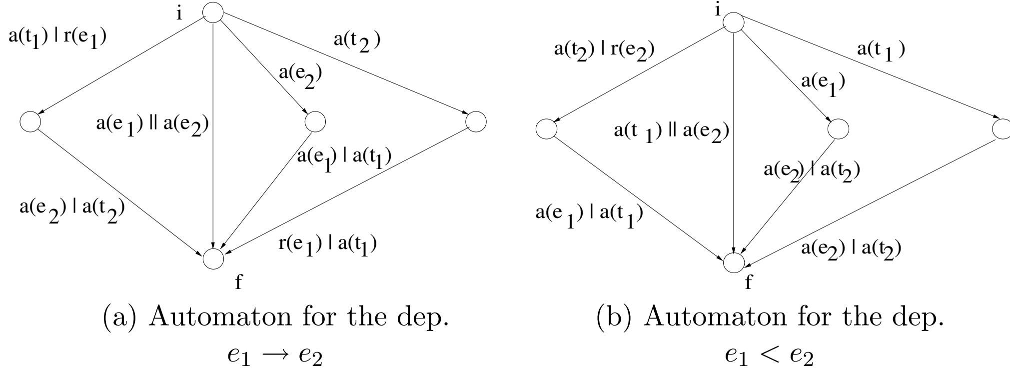
This work does not explicitly deal with the verification issues, such as whether the given set of constraints implies some other constraints, but the major issue is whether implication of workflow dependencies can be tested more efficiently due to the specialized form of these constraints.

Formalization

The formalization is based on the following assumptions: workflows are modeled as streams of significant events such as *start*, *precommit*, *commit*, and *abort*; the unique event assumption holds; and events can be delayable, rejectable, or forcible.

A workflow is specified as a set of dependencies over the events associated with the tasks. If $e_1, e_2, ..., e_n$ are the significant events associated with a number of tasks, then a dependency D involving these events is denoted as $D(e_1, e_2, ..., e_n)$. Computational Tree Logic (CTL) is used to specify these dependencies. For instance, the order dependency, $e_1 < e_2$, is specified in CTL as $\mathcal{A}\Box(e_2 \rightarrow \mathcal{A}\Box\neg e_1)$, *i.e.*, on every path it is always true that if e_2 occurs then e_1 will not occur later on any continuation of that path. A dependency, D, specified in CTL, is compiled into a finite state automaton A_D, which is a tuple $\langle s_0, S, \Sigma, \rho \rangle$, where:

- S is a set of states.
- s_0 is the initial state.
- Σ is a set of event expressions, which can have one of the following forms:
  - $a(e_1, ..., e_n)$ says that the events $e_1, ..., e_n$ are *accepted* by A_D and scheduled



(a) Automaton for the dep. $e_1 \rightarrow e_2$

(b) Automaton for the dep. $e_1 < e_2$

Figure 2.1: Dependency automata

for execution.

- $r(e_1, ..., e_n)$ says that the events $e_1, ..., e_n$ are *rejected* by A_D.
- $\sigma_1 \parallel ... \parallel \sigma_n$ says that the event expressions $\sigma_1, ..., \sigma_n$ are run *concurrently* in an interleaved fashion.
- $\sigma_1; ...; \sigma_n$, where $\sigma_i \in \Sigma$ says that the operations $\sigma_1, ..., \sigma_n$ are run in *sequence.*

- $\rho \subset S \times \Sigma \times S$ is the transition relation.

Figure 2.1 shows two automata for two different dependencies. Here t_1 denotes the significant event of termination (*i.e.*, abort or commit) of task 1 and t_2 denotes the termination event for task 2. Symbols e_1 and e_2 are used to denote other, non-termination events. Because of the special semantics of termination events, no significant events from a task i can arrive once the event t_i has arrived and t_i must be scheduled last.

The symbol | indicates choice — *either* event can cause the corresponding transition. This should be contrasted with the event combinator $\parallel$. For instance, an arc labeled with $a(e_1) \parallel a(e_2)$ means that *both* events, e_1 and e_2, must occur and the corresponding state transition can happen in one of the two ways: either by scheduling $a(e_1)$ first and $a(e_2)$ next or by scheduling these events in the reverse order.

The initial state in every automaton is denoted by i and the final state by f. Every *path* from the initial state to the final state corresponds to a way in which the dependency can be satisfied.

Figure 2.1(a) presents the automaton for the dependency $e_1 \rightarrow e_2$. Some valid paths for this dependency is as follows:

- $r(e_1)\, a(e_2)$ — rejection of e_1 followed by acceptance of e_2.
- $a(t_1)\, a(e_2)$ — termination of task 1 followed by acceptance of e_2.
- $a(e_1)\, a(e_2)$ and $a(e_2)\, a(e_1)$ — because $a(e_1) \parallel a(e_2)$ is a label on one of the arcs, which means that executing e_1 and e_2 in any order can cause the corresponding transition.

Similarly, Figure 2.1(b) is an automaton for the order dependency $e_1 < e_2$. The sequence of events $a(t_1)\, a(e_2)$ is accepted by both automata, since each automaton has a path consistent with this sequence of events. However, the sequence $a(e_1)\, a(t_2)$ is accepted by the automaton for $e_1 < e_2$ only. This is because $a(e_1)\, a(t_2)$ does not correspond to a legal execution sequence in the automaton for the dependency $e_1 \rightarrow e_2$.

To provide a complete automata, a self-loop is added to every state in each automaton. In addition, the initial state of the automaton is also made into an accepting (final) state. The automaton of Figure 2.1(a) transformed in such a way is depicted in Figure 2.2. In this figure, a label such as "*not* e_2, t_2" means that the transition along that arc can be caused by any event expression that does not mention e_2 or t_2. For instance, neither $r(e_2)$ nor $a(e_2)$ can cause the transition, but $a(t_1)$ or $r(e_1)$ can.

Scheduling

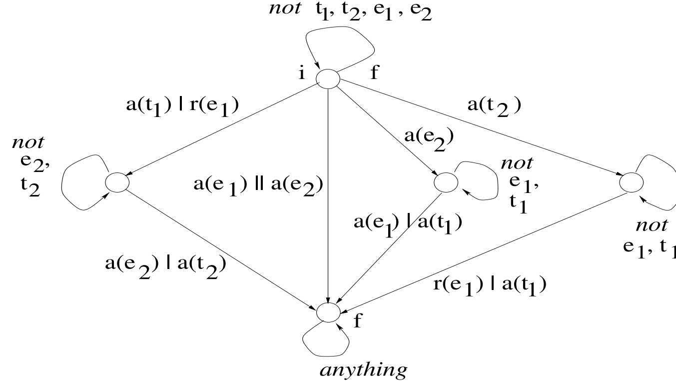


Figure 2.2: Automaton of Figure 2.1(a) augmented with self-looping transitions

On the automata as developed above, given a sequence of events, *seq*, the work of the scheduler is to find a legal execution path, π, such that the events mentioned in the expressions in π are all and only the events that occur in *seq*. If every automaton is of size N and there are m automata, then one can build a product automaton of size N^m. Unfortunately, this might be unacceptable for workflows that have many constraints. To avoid this state explosion problem, the individual automata are checked at run-time, as explained below. The worst time complexity of run-time scheduling is still exponential. However, it is believed that the worst case does not occur in practice [7].

The *global state* of the scheduler is a tuple whose components are the local states of the dependency automata — one state per automaton. The *initial* global state is a tuple of the initial states of these automata. When an event, e, arrives, the algorithm tries to construct an event sequence, π, which is accepted by every *augmented* automaton, such that π includes e and possibly some of the events that have arrived previously but have not yet been scheduled (these are called *delayed* events). In addition, each event on the path must occur at most once. If such a path cannot be found, then the scheduler delays the execution of the event e.

Consider the dependencies $e_1 \rightarrow e_2$ and $e_1 < e_2$ with the automata $A_{\rightarrow}$ and $A_{<}$, respectively, shown in Figure 2.1. Let $A'_{\rightarrow}$ and $A'_{<}$ be the augmentations of these automata. Augmentation for $A_{\rightarrow}$ is shown in Figure 2.2 and augmentation of $A_{<}$ is constructed similarly. Let e_1 be an event submitted to the scheduler. Since there is no path in *both* automata that begins by either accepting or rejecting e_1, the scheduling of e_1 has to be delayed. Now suppose the event e_2 is submitted to the scheduler. Two execution paths can be found in $A_{\rightarrow}$ that accept both e_1 and e_2: $a(e_2)a(e_1)$ and $a(e_1)a(e_2)$. The only path in $A_{<}$ that accepts both e_1 and e_2 is $a(e_1)a(e_2)$. However, in $a(e_1)a(e_2)$ the order of events is different from the path $a(e_2)a(e_1)$ in $A_{\rightarrow}$. Thus, the only legal execution path is $a(e_1)a(e_2)$ — the scheduler can execute e_1 followed by e_2 and satisfy both constraints.

2.1.2 Workflow Modeling Using Event Algebra

[72] defines an algebra, which is suitable for reasoning about constraints over an incoming stream of events. This algebra is expressive to represent very general temporal intertask dependencies, including control flow graphs. But conditions on transitions between tasks in such a graph cannot be expressed. A scheduling algorithm starts with an expression that represents the entire set of constraints and then chips away at these expressions (or *residuates* in the terminology of [72]) as it schedules the arriving events.

Formalization

Execution of a workflow relies on the notion of significant events produced by the tasks that comprise the workflow. Examples of such events are *start, precommit, commit*, and *abort.* A workflow is specified as a set of dependencies between these significant

events. The dependencies are represented as event expressions in the algebra.

The set of symbols that represent significant events is denoted by Σ. An *atomic event expression* is either an event symbol from Σ or its *negation.* If $e \in \Sigma$, then its negation is represented as $\bar{e}$; it represents the assertion that e does not occur in the execution of the workflow.

The language of *event expressions*, denoted by $\mathcal{E}$, is defined as follows:

- $\Gamma = \{e, \bar{e} | e \in \Sigma\} \subseteq \mathcal{E}$. This just states that atomic events are event expressions.
- We distinguish two special event expressions : 0 and $\top$ in $\mathcal{E}$. The event 0 represents the event expression that is always false and the event $\top$ represents the expression that is always true.
- If $E_1, E_2 \in \mathcal{E}$, then $E_1 \cdot E_2 \in \mathcal{E}$. The operator "$\cdot$" denotes *sequencing.*
- If $E_1, E_2 \in \mathcal{E}$, then $E_1 + E_2 \in \mathcal{E}$. The operator "+" denotes *choice* or *disjunction.*
- If $E_1, E_2 \in \mathcal{E}$, then $E_1 | E_2 \in \mathcal{E}$. The operator "|" means *conjunction.*

The event algebra uses denotational style semantics where an event expression represents a set of traces. A *trace* is a sequence of atomic events where

- each event symbol occurs at most once in the same trace;
- an event and its negation cannot occur in the same trace; and
- for each $e \in \Sigma$, either e or $\bar{e}$ occurs in the trace.

An event expression represents a constraint on the execution and the set of traces it represents are those that satisfy this constraint.

The set of traces (or *denotation*) for an event expression E is denoted by $[E]$. Given a set of atomic events, Γ, $U_\Gamma \subset \Gamma^* \cup \Gamma^\omega$ is the set of all finite (Γ^*) and infinite (Γ^ω) traces over the language Γ, *i.e.*, sequences of events that satisfy the three conditions given above. The denotations of the various event expressions are defined as follows:

- $[e] = \{\tau \in U_\Gamma \mid e \in \tau,\ i.e.,\ e \text{ occurs in } \tau\}$
- $[0] = \emptyset$, that is no trace satisfies the expression 0.
- $[\top] = U_\Gamma$, that is every trace satisfies the expression $\top$.
- *Sequencing*: $[E_1 \cdot E_2] = \{\nu\tau \in U_\Gamma \mid \ \nu \in [E_1] \text{ and } \tau \in [E_2]\}$, that is the resulting trace is obtained by concatenation of the traces of E_1 and E_2.
- *Disjunction*: $[E_1 + E_2] = [E_1] \cup [E_2]$.
- *Conjunction*: $[E_1 | E_2] = [E_1] \cap [E_2]$.

For an event expression $E \in \mathcal{E}$ and a trace $\tau \in U_\Gamma$, $\tau \models E$ denotes *satisfiability* of the event expression E by the trace τ, *i.e.*, the fact that $\tau \in [E]$.

Scheduling

Given a set of dependencies specified as a set of event expressions, the job of the scheduler is to find traces that satisfy the dependencies. The scheduler starts with an event expression that represents all dependencies. An incoming event is scheduled when this act is guaranteed to not break the dependencies regardless of which events will arrive in the future. The major insight here is that there is no need to record the past history of scheduled events. Instead, the information that is contained in the history and is relevant to the scheduler can be stored in the *residual event expression* that remains to be satisfied by the future incoming event stream.

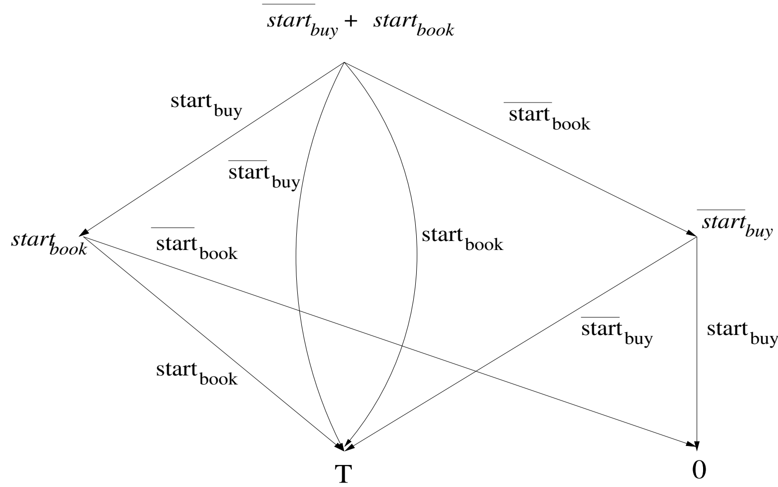


Figure 2.3: Scheduler transitions for the dependency D_1 in the traveler workflow

This storing of history is done through the *residuation* operator. The state of the scheduler is represented by an event expression, D, which remains to be satisfied by the incoming stream of events. When a new event arrives, the residuation of D by e, denoted by D/e, is the new state of the scheduler.

Before giving a formal definition, we illustrate this notion by an example. Figure 2.3 shows the effect of residuation on the dependency $D_1 = \overline{start}_{buy} + start_{book}$ in the travel workflow discussed above. The dependency appears at the top of the figure and each node is labeled with an event expression (which might be a compound expression). Arcs are labeled by atomic event expressions. If the scheduler schedules an event that labels an arc, the result of the residuation would be the expression pointed to by the arc.

Suppose that the scheduler schedules the event $start_{buy}$. Since this implies that from now on all traces will contain this event, the traces represented by the $\overline{start}_{buy}$ will not be possible, so we can remove this part of D_1 and do not need to worry about it. Thus, D_1 is residuated to $start_{book}$. If however, the scheduler decides that *book*

is not allowed to start (*i.e.*, it schedules $\overline{start}_{book}$ because of the need to satisfy some other constraint), then none of the traces that satisfy $start_{book}$ can occur, so we can remove that part of D_1. That is, $\overline{start}_{book}$ is scheduled then D_1 residuates to $\overline{start}_{buy}$, which becomes the dependency left to be satisfied. Informally, this means that if *book* is not allowed to start then the scheduler must ensure that *buy* is not allowed to start either. If the scheduler can schedule either $\overline{start}_{buy}$ or $start_{book}$ then the dependency D_1 is satisfied and it is residuated to $\top$. If a dependency cannot be satisfied then it residuates to 0. For example, suppose the current state of the scheduler is represented by the dependency $\overline{start}_{buy}$ and the event $start_{buy}$ arrives. Since there is no way to schedule this event (now or in the future) and still have the dependency satisfied, it is residuated to 0.

Formally, the residuation operator is defined as follows:

> $v \in [E_1/E_2]$, where E_2 is an atomic event expression, if and only if for every trace $u \in [E_2]$ it holds that $uv \in [E_1]$.

If E has the form where neither "|" nor "+" occur under the scope of the sequencing operator "$\cdot$", then the following rewrite rules provide an algorithm that computes residuation:

1. $0/e = 0$

2. $\top/e = \top$

3. $(E_1|E_2)/e = (E_1/e)|(E_2/e)$, where E_1 and E_2 are event expressions.

4. $(E_1 + E_2)/e = (E_1/e) + (E_2/e)$

5. $(e \cdot E)/e = E$, if $e, \bar{e}$ do not appear in the event expression E.

6. $(e' \cdot E)/e = 0$, if $e \neq e'$ and e occurs in E.

7. $(e' \cdot E)/e = 0$, if $\bar{e}$ occurs in E.

8. $E/e = E$, if e or $\bar{e}$ does not appear in the event expression E. This means that only the dependencies that mention the event e are relevant to residuation when e comes up for scheduling.

A scheduler can now be constructed as follows. Let E be the initial event expression, which is a conjunction of all constraints. When an event, e, arrives, we compute $E' = E/e$. If $E' \neq 0$, the event is scheduled and E' becomes the new constraint that needs to be satisfied.

If $E' = 0$ due to the rule (7) then e cannot be scheduled and we have two choices. If e can be rejected, the scheduler does so and keeps E as its current state. If e is not rejectable, then the event stream cannot be scheduled and an error results.

If $E' = 0$ due to the rule (6) than e cannot be scheduled at this time, but it might be in the future. So, if e is delayable, it is delayed until such time when the dependency can be residuated by e to a non-0. Otherwise, if e is not delayable, the stream of events cannot be scheduled and an error results.

2.1.3 Workflow Modeling Using Event-Condition-Action Rules

Event-condition-action rules contain three components: as the name implies event, condition and action. An *event* is a significant event, *condition* is a query over the state and *action* is a state changing operation. Informally, a rule says that, when an event is detected, if the condition holds, then perform the action. Dependencies between the workflow activities can be defined as a set of event-condition-action rules.

In this section, we discuss about [30], which models workflows using such rules.

In [31], workflows are modeled by using *triggers*, which are simple event-condition-action rules. The workflow activities are represented as transactions and the dependencies between the transactions are represented by triggers. The scheduler executes the transactions in a nested model and defines mechanisms to serialize concurrent rules.

Formalization

The events in a trigger can be either primitive or complex events. A complex event is built from primitive events as follows:

- *Disjunction:* The event $E_1|E_2$ is generated when either one of the events E_1 or E_2 is generated.
- *Sequencing:* The event $E_1 \cdot E_2$ is generated when the events E_1 and E_2 are generated in sequence.
- *Closure:* The event $E*$ is generated when there are non-zero occurrences of the event E.

The proposed model provides different relation types (called *coupling modes*) between the event, condition and action components of a trigger. The coupling modes are defined with respect to the *triggering* and *triggered* transactions. The transaction which triggers the event is called the triggering transaction and the transaction in which action is performed is called the triggered transaction. Given a triggering transaction T which fires a trigger having E as the event, C as the condition and A as the action components, the different coupling modes are:

- *Immediate:* If the $E - C$ mode is immediate, then C is checked within the trans-

action T immediately when E is detected.

- *Deferred:* If the $E - C$ mode is deferred, then C is checked within T but after the last operation in T and before T commits.

- *Decoupled:* If the $E - C$ mode is decoupled, then C is evaluated in a separate transaction. This mode is used for decomposing a long running sequence of triggers into short transactions.

The same coupling semantics apply to the $C - A$ component.

Scheduling

In [31], the triggers are executed by using a nested transaction model. A nested transaction is one which is started inside another transaction. The scheduler use the concept of *subtransaction* to realize the coupling modes. A subtransaction is a nested transaction whose parent is suspended until the child transaction completes.

When an event E occurs, the scheduler creates a nested transaction T' within the triggering transaction T, to evaluate the condition and action components of the trigger associated with E. T' creates another nested transaction, T_C, for evaluating condition C. If T_C commits, then T' creates nested transaction T_A for evaluating A. According to the coupling mode between the components of the trigger, T and T' are scheduled as follows:

- *Immediate:* If the $E - C$ coupling mode is immediate, then T' is created as a subtransaction of T. If the $C-A$ coupling is immediate then T' is a subtransaction of T and T_A is a subtransaction of T'.

- *Deferred:* Execution cycles are created in order to schedule transactions generated

by triggers in deferred coupling mode. The scheduler delays the execution of the triggered transaction. In $cycle_0$, only the triggering transaction T is executed. In $cycle_i$, all the delayed transactions created in $cycle_{i-1}$ are concurrently executed a s separate subtransactions of T. The cycles of execution continue until the last cycle finishes in which no delayed transactions are created. The commit of these delayed subtransactions are conditional upon the commit of T.

- *Decoupled:* If the $E - C$ or the $C - A$ coupling modes are decoupled then the triggered transaction is started as a *nested top transaction*. A nested top transaction has its own transaction tree, can commit independently of its parent and has no special privileges relative to its parent.

The schedule provides a mechanism to serialize multiple concurrently executing triggers. The different approaches incorporated by the scheduler are given as follows:

- Rules are grouped into *priority classes* and triggered transactions are scheduled in the priority order starting from the highest. All transactions in the same priority are scheduled concurrently.

- For decoupled subtransactions, *pipelining* method is used. If a transaction T is serialized after a transaction T', then all decoupled subtransactions created by T are serialized only after all decoupled subtransactions created by T' have been serialized.

2.1.4 Workflow Modeling Using Extended Transaction Models: ACTA

In order to schedule long-running transactions, extended transaction models relax the atomicity, consistency, isolation and durability (ACID) properties of traditional trans-

action models. In these models, transactions commit with a varying degree of atomicity and isolation. In workflows, coordination among the tasks is the primary issue rather than strict atomicity and isolation. For this reason, extended transaction models are suitable to model and schedule workflows. One major drawback of these models is that they cannot be easily used for verification properties of the workflows.

In [17], an extended transaction model, called $ACTA$, is defined in order to model workflows. ACTA is a first order theory of transactions with a set of axioms governing the transactions and a set of first order rules for specifying dependencies between the transactions. In this model, workflow activities are represented as transactions and dependencies between the transactions are specified as rules in terms of significant events associated with the transactions in the workflow.

In ACTA, there is a temporal ordering among the events of a transaction. Events associated with a transaction can be:

- *Significant Events:* The three significant events associated with a transaction are *begin*(initiation event), *commit* and *abort*(termination events).
- *Object Events:* An object event is an operation that acts on a database object and always produces an output.

The executions of transaction is recorded as a history, H, in terms of significant events of the transactions. The occurrence of an event in the history can be constrained as follows:

- $\epsilon \rightarrow \epsilon'$: Event ϵ can occur only after event ϵ'.
- $\epsilon \Rightarrow Condition$: Event ϵ can take place only if $Condition$ is satisfied.

- $Condition \Rightarrow \epsilon$: If $Condition$ holds then event ϵ should be in the history.

Different dependencies between the transaction can be expressed by using the above constraint structures. The type of such dependencies are as follows:

- *Commit Dependency:* $Commit_{t_j} \Rightarrow (Commit_{t_i} \wedge Commit_{t_j})$

  if both t_i and t_j commit then the commit of t_i precedes that of t_j

- *Strong-Commit Dependency:* $Commit_{t_i} \Rightarrow Commit_{t_j}$

  if t_i commits then t_j commits.

- *Abort Dependency:* $Abortt_i \Rightarrow Abortt_j$

  if t_i aborts then t_j aborts.

- *Weak-Abort Dependency:* $Abortt_i \Rightarrow (\neg(Commit_{t_j} \rightarrow Abort_{t_i}) \rightarrow Abortt_j)$

  if t_i aborts and t_j still has not committed then t_j aborts.

- *Termination Dependency:* $\epsilon' \Rightarrow (\epsilon \rightarrow \epsilon')$ where $\epsilon \in (Commit_{t_i}, Abort_{t_i})$ and $\epsilon' \in (Commit_{t_j}, Abort_{t_j})$

  t_j can not commit or abort until t_j has also committed or aborted.

- *Exclusion Dependency:* $Commit_{t_i} \Rightarrow (Begin_{t_j} \Rightarrow Abort_{t_j})$

  if t_i commits and t_j has already started then t_j is aborted.

- *Force-Commit-on-Abort Dependency:* $Abortt_i \Rightarrow Committ_j$

  if t_i aborts then t_j commits.

- *Begin Dependency:* $Begint_j \Rightarrow (Begin_{t_i} \rightarrow Begin_{t_j})$

  t_j cannot begin executing until t_i has begun.

- *Serial Dependency:* $Begint_j \Rightarrow (\epsilon \rightarrow Begin_{t_j})$, where $\epsilon \in (Commit_{t_i}, Abort_{t_i})$

  t_j cannot begin executing until t_i either commits or aborts.

- *Begin-on-Commit Dependency:* $Begint_j \Rightarrow (Commit_{t_i} \rightarrow Begin_{t_j})$

  t_j cannot begin executing until t_i commits.

- *Begin-on-Abort Dependency:* $Begint_j \Rightarrow (Abortt_i \rightarrow Begin_{t_j})$

  t_j cannot begin executing until t_i aborts.

- *Weak Begin-on-Commit Dependency:* $Begint_j \Rightarrow (Commit_{t_i} \Rightarrow (Committ_i \rightarrow Begin_{t_j}))$

  if t_i commits then t_j can begin execution after t_i commits.

The above dependencies are used in modeling the workflow dependencies.

ACTA defines a set of axioms to be satisfied by every atomic transaction. These axioms relax the *ACID* constraints of the traditional transaction, and define new conditions for safe scheduling of transactions. If t is an atomic transaction, p is a transactional operation, and *ob* is a database object, then:

- every *begin* event can be invoked at most once by a transaction.
- only and instantiated transaction can commit or abort.
- an atomic transaction cannot be committed after it has been aborted.
- only in-progress transactions can invoke operations on objects.
- $View_t = H_{current}$: The view of a transaction is the set of objects visible to the transaction at a time point. An atomic transaction can view the current history or the most recent state of objects in the database.

- $ConflictSet_t = \{p_{t'}[ob] | t' = t, Inprogress(p_{t'}[ob])\}$: The conflict set of a transaction is the set of operations from other transactions which can conflict with it. For an atomic transaction, the conflict set of a transaction is the set of operations from other in-progress transactions.

- $Commit \Rightarrow \neg(tC * t)$: An atomic transaction can commit only if it is not part of a cycle C of relations developed through the invocation of conflicting operations. $C*$ is the transitive closure.

- $\exists ob \exists p, Commit_t[p_t[ob]] \Rightarrow Commit_t$: If an operation p of t, on object ob, commits then t must also commit.

- $\exists ob \exists p, Abort[p_t[ob]] \Rightarrow Abort$: If an operation p of t, on object ob, aborts then t must also abort.

- $Commit_t \Rightarrow \forall ob \forall p(p_t[ob] \Rightarrow Commit_t[p_t[ob]]$: If t commits then all the operations invoked by t also commits.

- $Abort_t \Rightarrow \forall ob \forall p(p_t[ob] \Rightarrow Abort_t[p_t[ob]]$: If t aborts then all the operations invoked by t also aborts.

2.1.5 Workflow Modeling Using Petri Nets

Petri Nets are developed for modeling and verifying process behavior. It is basically a bipartite graph that models transitions between the entities of a process. Workflows are also complex processes, therefore there are several works [1, 77, 76] that use Petri Nets to model and verify the correctness of workflows. [1] models workflow dependencies by using Petri Nets. On the other hand, [77, 76] provide a more abstract workflow model

to study the verification issues on certain properties of workflows.

A *Petri Net* is a graph with two types of nodes, called *places* and *transitions*. Edges go either from places to transitions or from transitions to places. At any time a place contains zero or more *tokens*. The state of the Petri Net, referred to as *marking*, denotes the distribution of tokens over places. For example, if a Petri Net has four places p_1, p_2, p_3 and p_4, then the marking $1p_1 + 2p_2 + 1p_3 + 0p_4$ is a state with one token in place p_1, two tokens in p_2, one tokens in p_3 and zero tokens in p_4.

A Petri Net is formally defined as a quadro-tuple $PN = (P, N, T, F, M_0)$ such that:

- P is a finite set of places $\{p_1, .., p_n\}$.
- T is a finite set of transitions $\{t_1, .., t_m\}$. It is required that $P \cap T =$ (bipartite condition) and $P \cup T \neq$ (non-empty condition).
- F is a set of edges, $F \subseteq (P \times T) \cup (T \times P)$. The edge from place p_i to transition t_j is denoted by $f(p_i, t_j)$ and similarly, an edge from transition t_i to place p_j is denoted by $f(t_i, p_j)$.
- M_0 is the initial marking, i.e., the initial distribution of i tokens over places. Given a marking M, the number of tokens in place p_i is denoted by $M(p_i)$.

Given a Petri Net, the input set of places of a transition t_i, denoted by $\bullet t_i$, is the set of places which have an edge leading to t_i, i.e., $\bullet t_i = \{p_j \mid f(p_j, t_i) \in F\}$. The *output* set of places of a transition t_i, denoted by $t_i \bullet$, is the set of places which have an edge coming form t_i, i.e., $\bullet t_i = \{p_j \mid f(t_i, p_j) \in F\}$. Similar definitions are given for a place p_i, denoted by $\bullet p_i$ and $p_i \bullet$. The execution of a Petri Net changes the token distribution according to the following definitions:

- Given a marking M, a transition t_i is said to be *enabled* if and only if for every $p_j \in \bullet t_i$, $M(p_j) > 0$. At any moment, a transition can be either enabled or disabled.

- Firing a transition t_i results in a new marking M' from the current marking M such that $\forall p_j \in \bullet t_i$ and $\forall p_k \in t_i \bullet$, $M'(p_j) = M(p_j) - 1$ and $M'(p_k) = M(p_k) + 1$. Firing a transition removes a token from every input place of the transition and adds one token to every output place of the transition.

- A state M' is *reachable* from a state M, denoted by $M \xrightarrow{*} M'$, if and only if there exists a sequence of firing transitions $\sigma = t_1 t_2 ... t_n$ such that $M \xrightarrow{\sigma} M'$, i.e., $M \xrightarrow{t_1} M_1 \xrightarrow{t_2} M_2 ... \xrightarrow{t_n} M'$.

Modeling Dependencies of a Workflow

In [1], a workflow is represented as a set of tasks and dependencies between these tasks. Tasks of a workflow are represented as a set of states and transitions between these states. When a task is modeled as a Petri Net, places are used to represent the task states and transitions are used to represent the various *significant events* which are operations like begin, commit, etc.

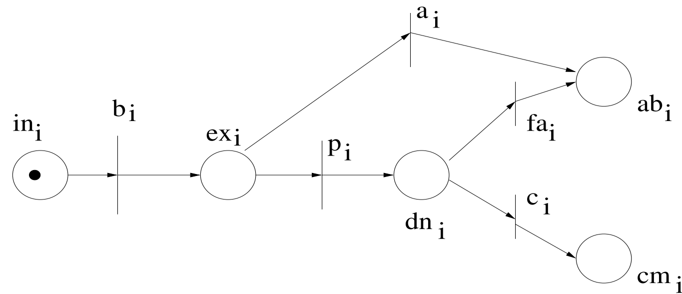


Figure 2.4: A Petri Net representation of a task

Formally, a task is defined as:

- a set of states: *initial*, *execution*, *done*, *commit* and *abort*. Figure 2.4 is a Petri Net representation of a sample task In the initial marking, there is one token in place $initial_i$ and zero tokens in all the other places.

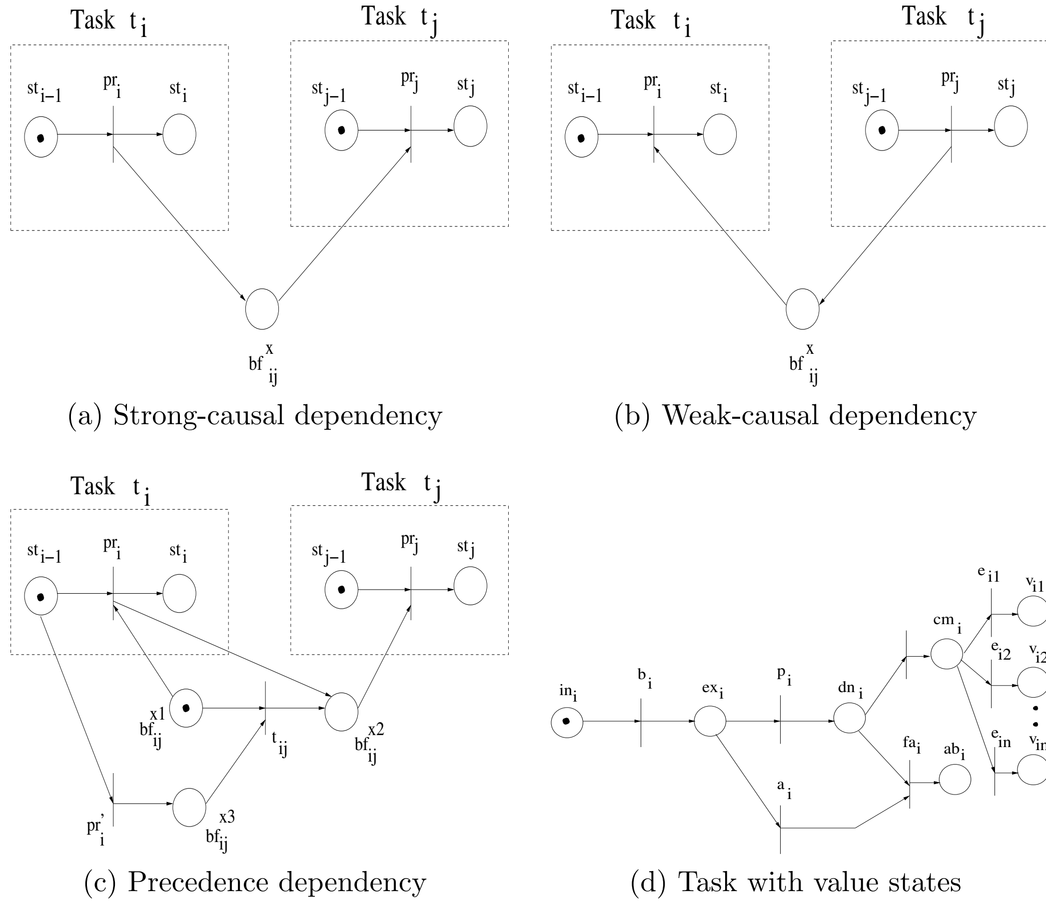
- a set of operations: *begin*, *abort*, *precommit*, *commit* and *force-abort*. Execution of these task operations results in change of state for the task. In Figure 2.4, the enabled transition $begin_i$ can fire and the firing changes the state of the task from $initial_i$ to $execution_i$.

- both the task states and task operations are ordered in a precedence relationship. The following precedence relationships are always satisfied: $initial_i \prec execution_i \prec done_i \prec commit_i$, $dn_i \prec ab_i$ and $ex_i \prec ab_i$. For task operations, any operation must be preceded by a $begin_i$ and must be followed by either a $abort_i$ or a $precommit_i$.

A dependency x between two tasks t_i and t_j is denoted by $t_i \xrightarrow{x} t_j$. [1]. There are three types of dependencies:

- *Control-flow* dependencies: these specify the conditions under which a task t_j, is allowed to enter a state st_j based on whether another task t_i has entered state st_i. Control-flow dependencies are further divided into:
  - *causal* dependencies: imply a logical constraint between t_i and t_j. The different types of causal dependencies are:
    * *strong-causal* dependencies: Task t_j enters state st_j only if task t_i enters st_i. Figure 2.5(a) shows a Petri Net modeling of a strong-causal depen-

[1] Note that these dependencies are equivalent to temporal/causality constraints explained in Chapter 1



(a) Strong-causal dependency (b) Weak-causal dependency

(c) Precedence dependency (d) Task with value states

Figure 2.5: Petri Net representation of dependencies

dency. Initially, task tw_i is in state st_{i-1} and task t_j is in state st_{j-1}. Task t_j can transition to the state st_j only if the transition pr_j fires. Transition pr_j is enabled only if the transition pr_i fires which changes the state of the task t_i to st_i. An example of a strong-causal dependency is the *begin-on-commit* dependency where task t_j can begin only if t_i commits.

∗ *weak-causal* dependencies: If task t_i enters st_i then task t_j must enter st_j. Figure 2.5(b) is a Petri Net modeling of a weak-causal dependency. Note that the only difference is the direction of the arrows adjacent to

the place bf^x_{ij}. An example of a weak-causal dependency is the *strong-commit* dependency between tasks t_i and t_j. In the dependency, if task t_i commits then task t_j must also commit.

Note that causality constraints do not imply any temporal ordering between the events. They are implied by the following precedence constraints.

- *precedence* dependencies: If both states st_i and st_j are entered then t_i must enter st_i before t_j enters st_j. There is no logical relationship between t_i and t_j. Figure 2.5(c) shows a Petri Net model of a precedence dependency. Note that, if transition pr'_i fires then task t_j can transition to state st_j even though task t_i might not be able to transition to state st_i. If however, task t_i transitions to state st_i then it would happen before task t_j can change state to st_j. An example of a precedence dependency is the *commit* dependency between tasks t_i and t_j, where if both tasks commit then t_i commits before t_j.

- *Value* dependencies: Task t_j can enter state st_j only if the value generated by task t_i satisfies a given condition. Figure 2.5(d) is a Petri Net model of a task with different value states. $v_{i1}, ..., v_{in}$ represent the different value states of task t_i and the firing of the transitions $e_{i1}, ..., e_{in}$ depends on the value generated in the task. A value dependency between tasks t_i and t_j can be modeled by using a buffer place between the appropriate value transition e_{ik} in task t_i and the transition in t_j.

- *External* dependencies: these dependencies depend on some external parameter. If time is chosen as the only external factor then these are called *temporal*

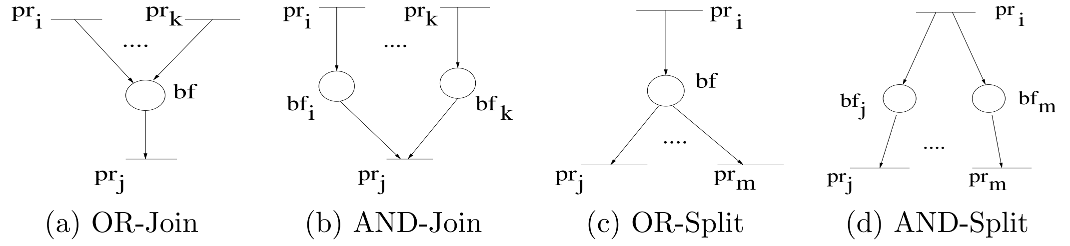


Figure 2.6: Petri Net representation of logical relations between dependencies

dependencies.

Logical relationships among the tasks are represented as follows:

- *OR-Join*: if any one of the transitions pr_i to pr_k fires, then transition pr_j is enabled.

- *AND-Join*: transition pr_j is enabled only if all of pr_i to pr_k fires.

- *OR-Split*: only one of the transitions pr_j to pr_m fires after the firing of pr_i.

- *AND-Split*: all the transitions pr_j to pr_m gets enabled after the firing of pr_i.

An OR-join and an OR-split are used to model conditional relationship between dependencies. Note that the definition of OR-join is essentially an exclusive OR between the tasks pr_j to pr_m. The AND-join and AND-split are used to model parallel relationship between dependencies.

It is possible to model time considerations of a workflow by using Petri Nets. However, an extends version, *Temporal Constraint Petri Net*, is used for this purpose. Every place and transition in this modified Petri Net is associated with a time interval (which is relative for places and absolute for transitions) and every token is associated with an absolute timestamp. A transition is enabled if each of it's input places have at least one available token. However, this is not enough to fire the transition. A transition t_i

can fire only if the time intervals of all tokens in the input places for t_i intersect the enabling time interval for that transition.

[1] discusses the following issues related to verification:

- *consistency*: A workflow specification is inconsistent if there are:

  - inconsistent *precedence* relationships between tasks (e.g. loops).
  - inconsistent *logical* relationships between tasks.

  A *siphon* is a subset S of places in a PN such that $\bullet S \subseteq S\bullet$. The presence of a siphon in a Petri Net indicates a deadlock because it means if there are no tokens in that subset then no place in that subset will ever get a token. Inconsistent specifications among dependencies can be verified by performing siphon detection on the Petri Net.

- *safe*: A workflow is said to be safe if it terminates in any acceptable state. This reduces to *reachability* problem of Petri Nets. The reachability problem is checking whether a sequence of transitions exist such that a marking M_1 is reachable from marking M_0. This problem is known to be DSPACE(exp)-hard for ordinary Petri Nets. However for *acyclic* Petri Nets, reachability can be solved in polynomial time. It can be proved that if a Petri Net is consistent then it is also acyclic. Determining safety reduces to solving a linear equation for every unsafe state using a matrix representation of the Petri Net.

- *schedulable*: A workflow is schedulable if there are no inconsistent temporal constraints. Starting from the initial marking, the firing times of transitions are calculated. When a transition fires, the timestamps of the tokens are updated. If

for a transition, earliest firing time is less than the latest firing time, then that transition cannot be scheduled. If every path from the initial place to the final place contains at least one transition that cannot be scheduled, then the workflow cannot be scheduled.

In [77], workflows have been modeled as tasks and transitions between these tasks. Join and split constructs are used to model constraints between these tasks. However, it is possible to specify only local constraints using these constructs. Triggers have been used to model constraints arising out of external conditions. In order to model constraints based on attribute values and time, a higher level Petri Net extended with the semantics of token color and time is used. The Petri Net model of a task is simpler than in [1]. However, the use of higher level Petri Nets provides more abstraction of the workflow specification. It is possible to check for deadlock , live-lock and proper termination on the Petri Net model of the workflow. Special structural characterizations of Petri Nets have been provided where these properties can be verified in polynomial time.

In this model, a workflow process definition specifies the order in which the appropriate tasks are to be executed. However, even if a task is enabled it may not be able to execute (for example it may require human intervention). In order to distinguish enabling from execution, the concept of *triggering* is introduced. A trigger is an external condition which leads to the execution of an enabled task. The different types of triggers used in this model are:

- *automatic* triggering: the task is triggered the moment it is enabled and it is appropriate for automated applications.

- *user* triggering: the task is triggered by a human user of the system.

- *message* triggering: the task is triggered by an external event like a message.

- *time* triggering: the task is triggered by some time duration or some particular absolute time.

Ordinary Petri Nets are extended with the semantics of *color, time,* and *hierarchy* to make these Nets better suited for the task of workflow modeling. Such Petri Nets are known as *high-level* Petri Nets. The *color* of the token models attributes of the object represented by the token. Thus, a colored Petri Net can be used to specify preconditions on transitions which take the value of the attribute into account. Transitions can also change the values of these attributes by changing the color of the token. In order to model real world complex workflow systems it is essential that a large Petri Net can be composed of simpler Nets. A *hierarchical* Petri Net can be constructed of subnets of Petri Nets each of which is an aggregate of a number of places, transitions and other subnets. A Petri Net extended with *time* can be used to model the temporal behavior of systems.

A task is built up from three states, *start, rollback* and *commit*, which are represented as transitions, and places between them. A Petri Net which models the process dimension of a single case is called a *workflow net* or WF-net. Workflow tasks are represented by transitions and states are represented by places. Tokens in a WF-net correspond to one single case. In general, workflows with many cases can be handled by colored tokens where the color of the token represent the case identifier. In this case, transitions cannot fire unless the color of the tokens in the input places match. A Petri net PN is a WF-net if and only if:

- The Petri Net has two special places i and o such that:
  - $\bullet i$ is empty, i.e. it is the input place.
  - $o\bullet$ is empty, i.e. it is the output place.
- Every place and transition is on a path from i to o.

The initial marking M_i has a token in place i only. The final marking M_o has a token in place o only. A WF-net for a process definition may still contain potential deadlocks and/or live-locks. It is the purpose of verification to detect these conditions.

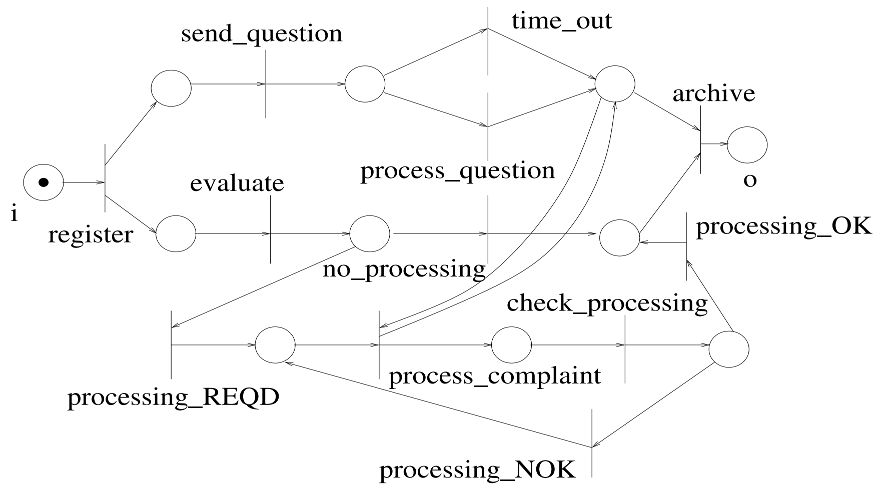


Figure 2.7: A Petri Net model of a complaint processing workflow

Figure 2.7 is an example of a Petri Net model of a workflow for processing complaints. The task *register* enables a complaint to be registered. In parallel, a questionnaire is sent to the complainant in the task *send_question* and the complaint is evaluated in the task *evaluate*. Note that the task *send_question* leads into an OR-Split. If the results of task *send_question* are obtained within a time period then *process_question* fires; else *time_out* fires and the results of the questionnaire are discarded. Based on the results of the task *evaluate*, it is either decided to process the complaint in tasks *process-*

ing_REQD, *process_complaint*, and *check_processing*; or the complaint is not processed at all. Finally, there is an AND-Join in the task *archive* when the results from processing and questionnaire are archived. The transition *register* contains an AND-Split while the transitions *evaluate* and *check_processing* lead into OR-Splits.

In [77], in order to define the correctness property of a workflow, a notion of *soundness* of the workflow net is introduced. A WF-net (P, T, F, M_i), where P is the set of places, T is the set of transitions, F is the transition function and M_i is the initial marking, is *sound* if and only if:

- $\forall M, (M_i \xrightarrow{*} M) \Rightarrow (M \xrightarrow{*} M_o)$: there are no dead ends in the workflow.
- $\forall M, (M_i \xrightarrow{*} M \wedge M(o) \geq 1) \Rightarrow (M = M_o)$: M_o is the only state with a token in o.
- $\forall t \in T \exists M, M' M_i \xrightarrow{*} M \xrightarrow{t} M'$: there are no dead transitions. This is the *liveness* property of Petri Nets.

A concept related to the number of tokens in a place is *boundedness*. A Petri Net is *bounded* if and only if for every place p and for every reachable state the number of tokens in p is bounded (i.e. always less than some natural number n). For simplicity, a classical Petri Net is used for verifying soundness.

The problem of deciding liveness and boundedness is EXPSPACE-hard for arbitrary WF-nets [20, 38]. For complex WF-nets, the decidability of soundness may even be intractable. For this reason, in [77], the soundness property is checked on three groups of WF-nets.

- A Petri net is free-choice Petri net iff for every two transitions t_1 and t_2, $\bullet t_1 \cap \bullet t_2 \neq$

0 implies $\bullet t_1 = bullett_2$.

- A Petri net PN is well-handled iff for any pair of nodes x and y such that one of the nodes is a place and the other a transition and for any pair of elementary paths C_1 and C_2 leading from x to y, $\alpha(C_1) \cap \alpha(C_2) = \{x, y\} \implies C_1 = C_2$. A WF-net PN is well-structured iff the extended net $\overline{PN}$ is well-handled.

- A WF-net PN is S-coverable iff the extended net $\overline{PN} = (P, T, F)$ satisfies the following property. For each place p, there is a subnet $PN_s = (P_s, T_s, F_s)$ such that $p \in P_s, P_s \subseteq P, T_s \subseteq T, F_s \subseteq F, PN_s$ is strongly connected, PN_s is a state machine (i.e., each transition in PN_s has one input and one output arc) and for every $q \in P_s$ and $t \in T : (q, t) \in F \implies (q, t) \in F_s$ and $(t, q) \in F \implies (t, q) \in F_s$.

Free-choice WF-nets and well-handled WF-nets can be checked for soundness in polynomial time. However, the complexity of deciding soundness for an S-coverable WF-net is PSPACE-complete [77, 76].

2.2 Constraint Logic Programming

Constraint programming is a major area of interaction between operations research and computer science. It combines programming language paradigms from computer science, like logic programming and concurrent programming, with efficient constraint solving techniques from mathematics, artificial intelligence, and operations research. Constraint programming allows the user both to build and to solve a model in the same framework. Programming language constructs make it possible to formulate the problem in a natural an declarative way. Constraint handling techniques supporting these language constructs provide for efficiency in problem solving.

Constraint technology has been applied successfully to a large variety of practical problems such as production planning, scheduling, and resource allocation.

Logic programming states the problems declaratively by using logic and solves them by using deduction. This approach provides a neat and easy framework for specification and solution of many problems. However, there are two main limitations of logic programs. Firstly, the set of possible terms are constituted out of functions and variables. Therefore, the objects manipulated by a logic program are uninterpreted structures. The second problem is due to the nature of SLD-resolution, which does depth-first search throughout the problem domain. This leads to inefficient solutions for large search applications.

Constraint Logic Programming (CLP) is the result of attempts to develop a framework that exploits the deductive structure of the logic programming, but overcomes the stated limitations, as well. In CLP, to overcome the first limitations, semantic objects, i.e., arithmetic expressions, are introduced into the system. As a result of this, the unification of the logic programming is augmented with constraint handler for given domain. Against the second limitation, in order to have more efficient solvers, the consistency techniques from Operations Research have been incorporated into CLP. Therefore, the depth-first search replaced or supplemented with more sophisticated and efficient techniques that uses constraints to prune the search space. With CLP systems, many problems such as scheduling, resource allocation, hardware design have been modeled and solved. CLP solves such problems with similar efficiency as other constraint programming and OR frameworks. As an advantage over other mentioned frameworks, due to the declarative structure, it is easier to model, maintain and modify

the problem.

2.2.1 Basic Features of CLP

We can state the key features of CLP as follows:

- Constraints are used to specify the query as well as answers.
- During execution, new variables and constraints are created.
- The collection of constraints in every state is tested as a whole for satisfiability before execution proceeds further and thus allows for control of execution.

As explained above, CLP include semantic objects that have meaning in particular domains. Different problems may demand different domains. Therefore, as explained in [49], a *Constraint Logic programming Scheme* denoted as *CLP(X)* is developed. X denotes the application domain such as *set, integer, real, boolean, finite domain* or even *finite trees*. Therefore, the user has been provided the flexibility and effectiveness to model the objects. Each computation domain is associated with algebraic operations such as addition, multiplication for numeric domains, boolean expressions for boolean domain or set operations for set domains. In addition to this, domain dependent relations on the objects such as equality and inequality are defined as a part of given domain.

Once semantic objects and expressions, relations on these objects, which are called *constraints*, are part of the language, a mechanism to handle the expressions and relations must be provided as well. Therefore, CLP languages contain constraint solvers specific for the given domain. At the very basic level, unification is a constraint solver

that can handle only equalities on uninterpreted objects. Another property of the unification is that it provides the most general solution for equalities, if there exists a solution. Under other domains, where such a most general answer does not exits, the system should be able to continue with manipulating the original set of constraints and tell the user if the constraints are satisfiable or not.

A constraint solver is *complete* if it can tell whether the given set of constraints is satisfiable or not. For efficient execution, constraint solver must be *incremental*, i.e., when adding a new constraint to an already solved set, the solver should not start from the beginning but use the result of the already solved set.

The declarative and operational semantics of CLP(X)-language is based on the semantics of pure logic programs, but parameterized by a choice of X, the domain of computation and constraints. Let X be a 4-tuple (Σ, D, L, E), where

- Σ is a signature,
- D is a Σ-structure,
- E is a first-order Σ-theory

The domain of computation is D, over which constraints of a class L of Σ-formulas can be expressed, and E axiomatizes properties of D.

A CLP(X) program consists of a finite set of constraint rules of the form $h : -b_1, ..., b_n$, where h is an atom and and the b_i are either atom or constraint. A CLP(X) goal is of the form $? - b_1, ..., b_n$.

An operational model that describes the operational semantics of many CLP-systems is given by a transition system on states, i.e., tuples $< A, C >$, where A is

a multiset of atoms and C is a multiset of constraints also called *constraint store*. The special state $\perp$ denotes failure. Intuitively, A is the set of not yet considered goals and C is the set of constraints collected so far. We assume to have a computation rule that selects an element of A and an appropriate transition rule for each state. The transition system is parameterized by a predicate *consistent* and a function *infer* corresponding to the specific constraint solver. An initial CLP-goal $?-b_1,...,b_n$ is represented by the state $(b_1,...,b_n,0)$. Suppose that the computation rule selects a constraint c in A, then rewriting the state $\langle A, C\rangle$ is defined by

$$\langle A, C\rangle = \begin{cases} \langle A-\{c\}, C'\rangle & : \quad consistent(C\cup\{c\}), C' = infer(C\cup\{c\}) \\ \perp & : \quad \neg consistent(C\cup\{c\}) \end{cases}$$

Suppose that the computation rule selects an atom a in A and a constraint rule $h:-b_1,...,b_n$ of the constraint logic program such that a and h have the same predicate symbol, then

$$\langle A, C\rangle = \begin{cases} \langle (A-\{a\})\cup\{b_1,...,b_n\}, C'\rangle & : \quad consistent(C\cup\{h=a\}), and \\ & : \quad C' = infer(C\cup\{h=a\}) \\ \perp & : \quad \neg consistent(C\cup\{h=a\}) \end{cases}$$

If there is an atom a in A, but no constraint rule in CLP-program having the same predicate, we have a failure.

$\langle A, C\rangle = \perp$

The predicate *consistent* checks for consistency of a set of constraints C and is defined by $consistent(C) \Leftrightarrow D \models \exists x : C$

The function *infer* is responsible for simplifying a set of constraints C. We require that $D \models infer(C) \Leftrightarrow C$

2.2.2 Domains in CLP

The set of possible constraint domains for CLP Scheme is quite large. Below, we explain some basic ones.

Real Domain. This domain is generally used in order to model and solve arithmetical expressions. *CLP(R)* [49] was the first CLP language developed to solve arithmetic constraints. One limitation of CLP(R) is that it can only solve linear expressions. In a *linear* expression, multiplication operation can have at most one variable operand and for division operation, denominator is always a constant. In order to solve linear expressions, algorithms such as *Gaussian elimination* and *Simplex* are employed. CLP languages that can solve non-linear expressions over real domain have been developed after CLP(R). These languages employ algorithms for non-linear expression such as *Gröbner bases* or *quantifier elimination* [62]. Some examples for CLP languages in this category are CAL [2], RISC-CLP [46].

Boolean Domain. Boolean constraints are basically used for hardware verification and theorem proving applications. Boolean terms are built from truth values (i.e. *true* and *false*), boolean operations and variables. The only constraint between boolean expressions is the equality. Therefore, solvers of the boolean domain handle only equality check. However, since boolean constraint solving process is NP-complete, solvers have exponential worst case complexity. There are several CLP languages that support boolean domain such as CAL and CHIP [32]. In order to provide comparatively more efficient solvers, each of such languages employ different methods for description of the boolean constraints and solve them over the given description.

Finite Domain. Finite domain constraints are particularly useful for modeling

discrete optimization and verification problems such as scheduling, planning packing, timetabling etc. A finite domain is a subset of the integers and a finite domain constraint denotes a relation over one or more finite domains. All domain variables, i.e., variables that occur as arguments to finite domain constraints get associated with finite domain, either explicitly declared by the program, or implicitly imposed by the constraint solved. The domain of all variables gets narrower and narrower as more constraints are added. If a domain becomes empty, the accumulated constraints are unsatisfiable, and the current computation branch fails. At the end of a successful computation, all domains have usually become singletons, i.e., the domain variables have become assigned. Since the domain size is limited, finite domain solvers can decide on the satisfiability of the constraint set. Finite domain solvers generally employ consistency techniques that are described below.

2.2.3 Consistency Techniques

As opposed to using only depth-first search as employed by logic programming languages, consistency techniques employed pruning of the search space by using the constraints and then do the searching. Pruning is done by propagating information about the variables via the constraints between them. For example, given that $T_1 \in \{1, 2, 3\}$ and $T_2 = \{1, 2, 3\}$ and $T_1 < T_2$, consistency technique deduces that $T_1 \in \{1, 2\}$ and $T_2 \in \{2, 3\}$. This consistency technique, which removes inconsistent values from the domains, is called *arc consistency* [59].

Propagation continues until no further reduction is possible. However, for most applications it is not possible to find the solution solely by propagation. Generally,

there remains combinations of values in the resulting domains which are inconsistent. Therefore, to find a solution, the system performs some search, by labeling a variable with some value in its domain and do further propagation. For example, for the above constraints, T_1 may be labeled with the value $T_1 = 2$, then propagation yields $T_2 \in \{3\}$.

Constraint propagation has been basically used in finite domain solvers. However, the process is extended, firstly, to interval domains and then to any domain. This version of the propagation is called *generalized propagation* [67].

In order to provide further efficiency, instead of plain depth-first search, heuristics have been employed. The correct strategy on the labeling of the constraint variable effects the execution time considerably.

2.2.4 CLP Systems

There is a vast number of research and products on CLP languages. Below, we list only some of them.

CHIP: CHIP developed at European Computer-Industry Research Center (ECRC). It includes finite domain, boolean domain and linear real domain by Simplex algorithm. This system is extensible with user-defined constraints and control facilities over the search [32].

CLP($\mathcal{R}$): This language is developed as a demonstrator of CLP schema at Monash University and Carnegie Mellon University. Its domain is real linear arithmetic and its solver is based on Simplex algorithm [51].

Prolog-III: Prolog III is a product of University of Marseille. It includes linear real arithmetic domain by Simplex algorithm, boolean domain and finite strings domain

[25].

Triology: Triology is a CLP language developed are Complete Logic Systems in Vancouver. Its domain is integer arithmetic. Unlike other languages, this system is not developed on Prolog. It is based on a "theory of pairs" [78].

CAL and GDCC: CAL (Constrainte Avec Logique) is a product of ICOT, Tokyo. It is the first CLP language with nonlinear constraint domain. The parallel version, named GDCC, has also developed at ICOT [3, 2].

BNR-Prolog: BNR-Prolog is a product of Bell-Northern Research, Ottawa. It is based on *relational arithmetic* domain. This domain is based on a new interval variable representing a real number lying between lower and upper bound of this interval [8].

RISC-CLP: This is a prototype system by RISC, Linz. It can handle any real constraints [45].

clp(FD): It is a product of INRIA, France [23]. Its key feature is the use of a single primitive to define finite domain constraints. It is extended to *GNU Prolog* with the ability to express and solve more complex constraints.

Oz: Oz has logic programming and concurrent programming features as well as its constraint specification and solving capabilities [69, 80]. Oz exhibits logic programming and constraint programming features through its logic variables, disjunctive constructs and programmable search strategies [68, 69].

2.3 Previous Work on Resource Modeling in Workflows

Resource management has been recognized as an important issue in a WfMS [24, 4]. In [24], *Workflow Management Coalition (WfMC)* defines an activity as a piece of work

that forms one logical step within a process that requires human and/or machine resources to support process execution. WfMC uses the term *workflow participant* to denote human resources and supports modeling of organizational structures for workflow participants. One important definition given by WfMC is on workflow constraints. WfMC groups constraints in three: *time based*, *resource based* and *cost based*. However, current systems support only the first group of constraints.

There are several works on workflow resource management. However, most of the work in this area have focused on modeling of resources with little attention devoted to scheduling.

In [77], resource classification and resource management issues for the proposed workflow management system model is briefly explained. In this model, the term *resource* refers to the actor that is able to execute a given task. On the basis of the organizational units, resources are grouped in classes. The resources are categorized also on the basis of role groups. Typically, each task is associated with a class and role so that a resource from the given class and role is assigned to the task. [2]

In order to assign a resource to a task, two basic methods exist:

- *push control:* The workflow system selects the resource and directs the task to the selected resource.
- *pull control:* A copy of the task is directed to each capable resource. The task is assigned to the first resource that accepts it and other copies are deleted.

In [77], it is noted that *pull control* is preferable due to its flexible structure. How-

[2] Note that, linking the task directly with a resource blocks the execution of the whole system when the resource is not available. For this reason, the workflow management systems, as much as possible, define a set of candidate resources for each task.

ever, in this work, since resource allocation constraints are the basic element that determines the agent selection, we adopt *push control* mechanism. It is important to note that the task is not directly linked to a certain resource and system can select another capable resource, where available.

[82] summarizes general approaches and requirements in workflow resource modeling and describes a framework of representation of resources and organizational model for workflow systems. In this work, the basic properties that a workflow resource model should have are listed as follows:

- *Robustness:* Changes to the resource models should not affect the workflow model and similarly, changes to the workflow model should leave resource model unaffected.
- *Flexibility:* The resource model should be flexible enough to allow the transfer of existing organizational models.
- *Scalability:* The integration of new hierarchy structures and new permissions should be possible.
- *Domain-independence:* The resource model for a collaborative system should be as domain-independent as possible. It should be able to represent the organizational and resource structures that belong to different environments and have different natures.

In [82], three different ways have been distinguished for resource assignment:

- *Direct Designation.* An activity is assigned to one or more entities of the resource model directly.

- *Assignment by Role.* The resources are grouped under roles and activities are associated with roles. Therefore, any resource in the same role can handle the activity associated with the given role.

- *Assignment by Formal Expression.* In this way of activity assignment, the resource is selected through an expression. An example is *activity_performer = superior(resource(activity(1))).* This expression assigns the supervisor of the resource of the first activity to the activity in consideration. Resource assignment expressions may need further information than the resource and organizational model. It may check the workflow execution history or may determine the resource only in run-time according to the resource assignment for other activities. [9] discusses these expressions in more detail.

In the same work, two strategies for resources modeling have been identified. The first model, which [82] called *technology-driven approach*, presumes no predefined set of resources in the organization. The resources are grouped into roles implicitly according to their authorizations. The second model, *organization-driven approach*, defines a formal organization hierarchy separate from the workflow model. However, this model does not include technical resources.

The main contribution of [82] is the proposal of a generic resource model that amalgamates technology-driven and organization-driven approaches. It provides a framework to model organizational hierarchy and technical resources under the same roof.

[34] is another work on workflow resource management. [34] develops an architecture in order to integrate resource systems that are built by different organizations for different purposes for the overall workflow. The architecture is composed of *Local*

Resource Managers (LRM) that preexist and have their own resource models and communication protocols and *Global Resource Managers (GRM)* that represent integrated views of part or all of the underlying LRMs. GRMs have the same resource model and communication protocol.

In order to support different views of enterprise workflow resources, GRMs are further subdivided into *Enterprise GRMS (ERMSs)* and *Site GRMs (SRM)*. ERMs represent enterprise-wide view of the workflow resources and interface with underlying SRMs, which represent partial views of the workflow resources within an organization and physical boundary. All the SRMs and ERMS have the same following structure:

- *Interface Layer:* This layer allows other components to send requests to the resource manager and defines administrative APIs and uses the underlying security mechanism.

- *Policy Manager and Resource Model:* This layer implements the policy rules and the unified resource model.

- *Request Processing Engine:* This layer takes the requests and routes them to appropriate information source. In addition to this, it assembles all the results coming from different information sources.

- *Integration Layer:* This layer manages all the different protocols spoken by local information sources (i.e., LRMs).

[48] is another work by HP Labs on workflow resource management. In this work, a resource policy handling method for workflow systems is described. Resource policies are general guidelines every individual resource allocation must observe. A policy

manager is a module within a resource manager, responsible for efficiently managing a set of policies and enforcing them in resource allocation.

[48] groups the policies in three categories:

- Qualification Policies: They state a type of resources is qualified to do a type of activities.

- Requirement Policies: They state the conditions that a resource must satisfy in order to be chosen to carry out an activity with specified characteristics. For example, a programming job with number of lines more than 10000 requires a programmer with at least 5 year of experience.

- Substitution Policies: They state that a resource can replace another resource under the condition that it is not available, to carry out an activity.

An SQL-like policy specification language is developed in order to model the above types of policies. [48] explains several query rewriting and policy retrieval techniques for efficient policy handling.

Time is an important dimension of resource allocation cost. There are several work in the literature on time management in workflows. However, in these work, time and duration are defined as properties of activities and they are examined independent of resource allocation.

In [35], the following time management issues are addressed:

- modeling of time at process time

- pro-active time calculation to provide alerts in case of potential time violations

- time monitoring and deadline checking at runtime

- handling of time errors

In order to capture time information in modeling, basic workflow model is augmented with *time points*, *durations* and *deadlines*. A workflow designer can assign execution durations and deadlines to individual activities and to the whole workflow process. When an instance of the workflow model is created, deadline and duration information are recomputed on the basis of the current calendar and time points are created. Time points denote the possible *start* and *end* events of the activities. Actually, for a significant event (i.e. *start* and *end* events), several time points are assigned. These time points are calculated according to a set of forward and backward computation procedures and show the earliest and latest possible time that the event must happen. The result is presented in the form of a timed graph.

At run-time, the time workflow graph is used for monitoring the process. As activities are completed, the time points are refreshed. The status of the workflow is monitored and the state information is presented to the user in terms of colors. Green is the normal state and no deadline violation expected for the completion of the workflow. *Yellow* shows the possible necessity to take some precautions, such as dropping some optional activities. *Red* status shows threat of missing a deadline and exception handling is required.

In [36], this structure is extended to handle the following constraints:

- *Lower-bound constraint:* The duration between events A and B must be greater than or equal to θ.

- *Upper-bound constraint:* The duration between events A and B must be smaller than or equal to θ.

- *Fixed-Date constraint:* Event B can occur on certain dates.

Fixed-date type is defined in order to model fixed-date constraint. For a F is a fixed-date type, the following methods are defined: $F.valid(D)$ returns true if the date D is valid; $F.next(D)$ and $F.prev(D)$ return, respectively, the next and previous valid dates after D; $F.period$ returns the maximum distance between valid dates; and $F.dist(F')$ returns the maximum distance between valid dates of F and F'.

Before creating the timed workflow graph for the workflow instance, first, fixed-date constraints are converted to lower-bound constraints using worst-case estimates. Maximum distance between the valid dates are used for the computations.

In order to create timed graph under new constraints, forward and backward computation procedures are augmented with new algorithms for lower-bound and upper-bound constraints. At run-time, the process is monitored against both deadline and constraint violations.

2.4 Other Related Work

Job-shop scheduling: Operations research (OR) has developed a number of successful algorithms for solving many scheduling problems [16, 10]. Among them, job-shop scheduling is most relevant to workflow scheduling. Constraint logic programming (CLP), which integrates logic-based and OR techniques, has also been successfully used to deal with job-shop scheduling problems. The following works is just a tip of an iceberg of the vast research on the subject [42, 18, 81]. Both of the job-shop scheduling and workflow scheduling problems incorporate resource allocation constraints. However, a workflow can be much more complex than a job-shop. Furthermore, the focus of

our work is orthogonal to the works on constraint solving — we *use* constraint solvers in Stage 3 of our scheduling process.

Planning in AI: As in the scheduling problem in OR, constraint programming is used for planning in AI [14, 66], and there also are works on planning under resource constraints [43, 15]. However, only a small number of works in this area propose planning as a workflow scheduling technique [63, 65]. However, those that do deal with scheduling do not address this problem under resource allocation constraints. Instead, planning techniques are used to schedule dynamically changing workflows.

Agent-based workflow systems: Using agent technology for workflow systems is another related research area [79, 74, 52, 53, 54, 44]. In agent-based workflow systems, execution decisions are based on the communication events that occur when one agent requests services of another. Research on this area has largely concentrated on intelligent agent modeling and communication issues. Only a few [53] briefly mention the issue of scheduling under resource allocation constraints, which they propose to do through negotiations among the agents. However, to the best of our knowledge, details of such techniques have not been worked out.

CHAPTER 3

Modeling of Workflows and Resource Allocation Constraints

In this chapter, we present some preliminary information on workflow structures and workflow modeling. Then we have a closer look at the resource allocation constraints in workflow context.

3.1 Workflow Structure

Following the process definition standards defined by *Workflow Management Coalition (WfMC)* in [24], workflow specifications consists of basic block structures. These block structures, shown in Figure 3.1, are as follows:

- The simplest block being a single task, blocks may have recursive structures.
- In an *AND-block*, all sub-blocks are executed and the block successfully finishes execution when all its sub-blocks are completed successfully. Depending on the resources and the constraints, some or all sub-blocks can execute concurrently.

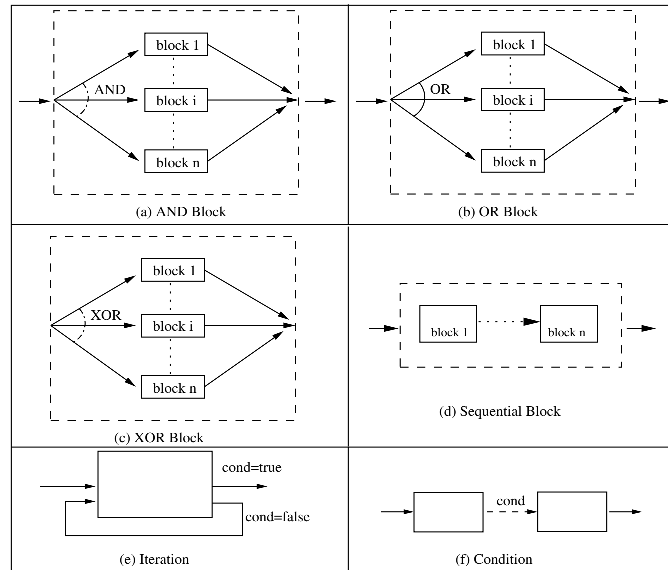


Figure 3.1: Workflow structure

- *OR-block* is completed successfully when at least one of the sub-blocks successfully completes execution.

- In an *XOR-block* (*eXclusive Or*) only one of the sub-blocks can be completed. The block successfully finishes execution when at most and at least one of the sub-blocks is completed successfully.

- In a *Sequential Block* sub-blocks are executed sequentially in the order of definition.

- *Iteration block* may run more than once repetitively until a given condition holds. Thus, several instances of the iteration block may execute sequentially.

The basic component for workflow is a task. In addition to tasks, conditions are also necessary for workflow specification. *Condition* is a logical expression. It is a part of iteration block, or it can be used as a precondition for other blocks. If the condition

is met, the block following the condition can be executed as shown in Figure 3.1 (f).

The above structures capture most of the order and existence dependencies among tasks. However, there may be other order and existence dependencies that can not be defined using these structures, such as the dependencies between two tasks in different blocks. The typical examples of such temporal/causality constraints may be listed as follows:

- Task t_i must execute.
- Task t_i must not execute.
- If t_i executes, t_j must execute as well. (Klein's existence constraint in [56])
- If t_i and t_j both execute, t_i must execute before t_j (Klein's order constraint in [56])

3.2 Workflow Modeling

As explained above, business logic of a workflow is generally specified by using block structures. At the beginning of this chapter, temporal and causality constraints are introduced as the basic and traditional formalisms to model the relations between these block structures. There are several frameworks to represent these constraints. They are sometimes referred to as *order dependencies* and basically they are grouped in two, as *immediate* (or *local*) and *global* dependencies.

A control flow graph represents the immediate ordering dependencies between the tasks of a workflow as a graph. Since it presents the visual representation of overall flow of control, this modeling framework has been incorporated by most of the commercial

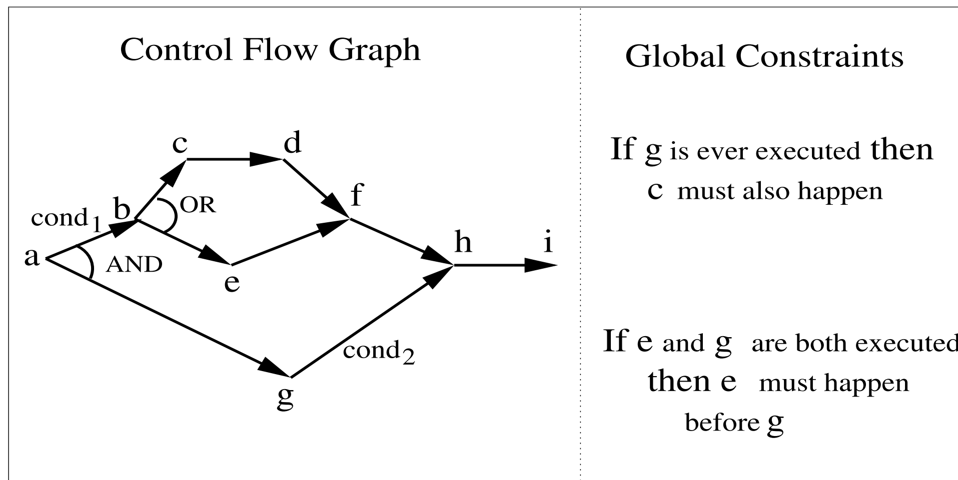


Figure 3.2: Workflow modeling frameworks

workflow management tools. A control flow graph example is given in the first part of Figure 3.2. A typical control flow graph specifies the successor tasks of a given task, initial and final tasks of a workflow, the concurrent and alternative branches of execution. The concurrent branches are denoted by AND and alternatives are denoted by OR.

It is also possible to label the arrows in order to specify the *transition conditions* between the tasks. The condition is checked against the current state. After the task at the tail of the arrow is successfully completed, if the condition is satisfied, the task at the head of the arrow can start executing.

The control flow graph can be extended to model features such as loops, subworkflows, retries etc. However, the major drawback is that only local immediate dependencies can be modeled, however, global dependencies, whose examples are given in the second part of Figure 3.2, can not be modeled in this framework. In many systems, immediate dependencies are modeled by using graphs and other that can not be speci-

fied in control flow graphs are specified as as global dependencies using some temporal constraint language.

In [72], an algebra of temporal constraints is described, which can represent the many of the necessary global constraints in the workflow context. The most important of them are the Klein's constraints [56] which are as follows:

- *Order constraint ($a < b$)* If tasks a and b both execute successfully, then a comes earlier than b.

- *Existence constraint ($a \rightarrow b$):* If task a ever executes, then b must execute as well b (in any order).

Using *ECA rules* (or *triggers*) is another way to model workflows [31]. The ordering relations among the workflow tasks can be modeled as ECA rules. These dependencies capture the normal or expected executions of a workflow process. However, exceptions or errors may arise during the execution as well. One important advantage of using ECA rule is that, actions to be employed in case of abnormal, unanticipated situations can be modeled as well. An example ECA rule may be "if event e occurs and then if condition c holds then do action a". The major drawback of this modeling framework is the lack of validation techniques.

In this work, we use control flow graphs and temporal/causality constraints together for workflow modeling. When it is more practical, we convert the immediate dependencies in control flow graph into temporal/causality constraints.

3.3 Modeling Resource Allocation Constraints

On the contrary to previous workflow scheduling works that deal with temporal/causality constraints, we specify resource allocation constraints as well as temporal/causality constraints and find schedules satisfying all of the given constraints. In this section, we have a closer look at resource allocation constraints.

Resource allocation constraints define restrictions on which resources to allocate and when to allocate them. Some of the constraints directly involve the total execution cost, whereas others define restrictions on agents and relations between agents. From this point of view, we may categorize the resource allocation constraints as *cost* and *control* constraints.

- *Cost Constraints* are defined on resource allocation cost. For instance, *total cost* < 40 is a cost constraint.
- *Control Constraints* are defined directly on the resource. *if task* t_2 *is done by agent* x*, then task* t_5 *must be done by the same agent as well* is an example for control constraints.

Another categorization orthogonal to the above is as follows: Satisfaction of some constraints is necessary in order to obtain a solution, whereas satisfaction of some others is not obligatory for the solution, but they rather facilitate the satisfaction of the constraint set. According to this property, we group resource allocation constraints as *hard* and *facilitating* resource allocation constraints.

1. *Hard Resource Allocation Constraints:* The satisfaction of these constraints is necessary for the solution. For instance, *sum of costs of tasks* t_1 *and* t_2 *must be*

less than 10 is in this category. Similarly, *if task t_2 is done by agent x, then task t_5 must be done by the same agent as well* is also an hard constraint. Majority of the resource allocation constraints defined for workflows are in this category.

2. *Facilitating Resource Allocation Constraints:* Sometimes, satisfaction of a certain resource allocation constraint facilitates the satisfaction of total cost constraint. For instance, *if tasks t_1 and t_3 are done by the same agent, then there is a discount of $300* is such a constraint. *Discount* in this example, is discount by the given amount in the total expenditure. Another constraint may be *if t_1 and t_2 are performed on the same computer, execution time of t_2 is shortened by 2 minutes, due to gain in communication and data transfer.* Note that *discount* and *reduction in execution time* are not actually constraints themselves. For this reason, these constraints are different from *conditional constraints.*

Sometimes, temporal/causality constraints work together with resource allocation constraints. For instance, a temporal/causality constraint may take part in a resource allocation constraint such as *if task t_5 costs more than $1000, then do not execute task t_6*. Similarly, a facilitating resource allocation constraint may contain a temporal/causality constraint as in *if task t_4 is completed before task t_6 begins, then there is a discount of $400.* Thus, it is possible to make the following generalization: for hard resource allocation constraints in the form of *if p then q*, p or q may be a temporal/causality constraint [1]. For facilitating resource allocation constraints *if p then q*, p may be a temporal/causality constraint.

[1] Note that, if both p and q are temporal/causality constraints, *if p then q* is a temporal/causality constraint as well.

CHAPTER 4

Using CLP to Schedule Workflows Under Resource Allocation Constraints

In this chapter, we present a workflow management system architecture that provides modules to model resources and resource allocation constraints and to find schedules fulfilling these constraints. In order to find schedules, *constraint programming approach* is used. That is, scheduler incorporates an off-the-shelf constraint solver to obtain a feasible schedule for the workflow satisfying both resource allocation and temporal/causality constraints.

4.1 System Architecture and Specification Language

4.1.1 System Architecture

Resource specification and management are crucial for workflows. For this reason, we propose an architecture to model and solve resource allocation constraints, which is shown in Figure 4.1. In order to specify workflow, temporal/causality constraints, re-

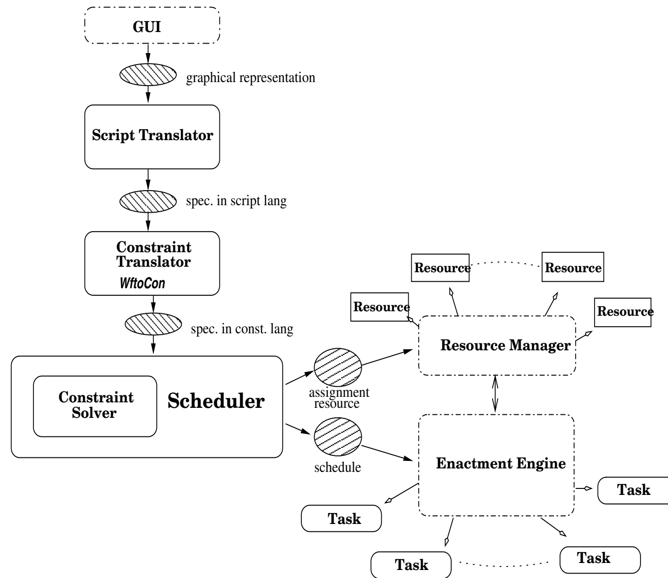


Figure 4.1: System architecture

sources and resource allocation constraints, we have developed a specification language, **W***orkflow* **S***pecification* **L***anguage (WSL)*. In our system, the user directly specifies the workflow in WSL. It is also possible to develop a graphical interface tool, that may be incorporated into the system. By using this tool, a graphical representation of the workflow can be defined by the user and then it is translated into WSL specification.

Since we employed constraint programming in order to find schedules under resource allocation constraints, as the next step, workflow specification in WSL is translated into a program in constraint language. Then, the resulting constraint specification is sent to the scheduler. The core of the scheduler is a constraint solver that can understand and solve the specification in a constraint language. The scheduler produces a schedule and a corresponding resource assignment.

Once the schedule and the resource assignments are produced, enactment engine and resource manager may receive the schedule and the resource assignment, respectively,

from the scheduler, and executes the workflow. Since, in this work, the emphasis is on specification and scheduling part, we have not implemented the resource manager and the enactment engine.

4.1.2 Workflow Specification Language: WSL

In order to specify a workflow, we created a script language, *WSL*, that can model basic workflow blocks, resource information and constraints.

4.1.2.1 Workflow Blocks

The syntax of the basic workflow blocks defined in Section 1 in WSL are as follows:

- a *task* is represented as ⟨*task_name*⟩(⟨*parameterlist*⟩). The information flow between the task is through parameters. However, in this work, we do not explain the details of parameter passing and information flow mechanism. Since, in this work, the emphasis is on scheduling, the tasks given in the examples are defined without parameters.
- **And Block: AND**{⟨ List of Block Definitions ⟩}
- **Or Block: OR**{⟨ List of Block Definitions ⟩}
- **XOr Block: XOR**{⟨ List of Block Definitions ⟩}
- **Sequential Block: SEQ**{⟨ List of Block Definitions ⟩}
- **Iteration Block: WHILE**(⟨*condition*⟩){⟨ Block Definition ⟩}
- **Condition Block: IF**⟨*condition*⟩{⟨ Block Definition ⟩} **ELSE**{⟨ Block Definition ⟩}

An example workflow specification in WSL is given in Figure 4.2.

4.1.2.2 Temporal and Causality Constraints

In most of the script languages (such as given in [57, 19]), temporal/causality constraints can not be specified. On contrary, WSL provides the constructs to define temporal and causality constraints. These structures have a simple and straightforward semantics. They can be listed as follows:

i. **Execute**($\langle task \rangle$)

ii. **NotExecute**($\langle task \rangle$)

iii. $\langle task_i \rangle$ **BEFORE** $\langle task_j \rangle$.

iv. **IF** X **THEN** Y, where X and Y are either temporal/causality constraints as defined above or a set of such constraints.

By using the above syntax, Klein's existence constraint is represented as

IF Execute(t_i) **THEN Execute**(t_j).

Similarly, Klein's order constraint is

IF Execute(t_i), **Execute**(t_j) **THEN** (t_i **BEFORE** t_j).

Note that, *if* part contains two constraints separated by a comma denoting conjunction. For instance, the constraint "if facade painting is done, wooden framed window installation must be done as well" is represented as

IF Execute($paint_facade$) **THEN Execute**($install_wooden_frame$).

Another constraint that is common for workflows is

$$\textbf{IF Execute}(t_i)\textbf{THEN NotExecute}(t_j),$$

meaning either t_i or t_j must execute exclusively.

4.1.2.3 Resources

In addition to the basic blocks, WSL provides structures to define resources in the workflow system as follows:

$\langle Resource_Name\rangle$ **CAPABLE_OF** $\langle task\rangle$ **WITH**

$[(\langle cost_term_1\rangle, \langle cost_1\rangle)...(\langle cost_term_n\rangle, \langle cost_n\rangle)]$

For instance, resource r_1 that can execute task t_2 having cost dimensions of money, time and energy with certain values is defined as

r_1 **CAPABLE_OF** t_2 **WITH** $[(money, 10), (time, 5), (energy, 2)]$

4.1.2.4 Resource Allocation Constraints

One of the basic issues of this work is to specify resource allocation constraint for wokflows. For this purpose, our specification language, *WSL*, provides constructs to model resource allocation constraints, which is not provided by previous workflow specification languages. As explained in Section 3.3, we group the resource allocation constraints as hard resource allocation constraints and facilitating resource allocation constraints. Both types of constraints are modeled in this language as follows:

1. **Hard Resource Allocation Constraints:**

   i. $\langle task\rangle$.**AssignedResource**=$\langle task\rangle$.**AssignedResource**

e.g. *construct_wall*.**AssignedResource**=

construct_ceiling.**AssignedResource**.

ii. $\langle task \rangle$.**AssignedResource** $\neq$ $\langle task \rangle$.**AssignedResource**

e.g. *install_frame*.**AssignedResource**$\neq$*facade*.**AssignedResource**.

iii. $\langle task \rangle$.**AssignedResource**=$\langle resource \rangle$

e.g. *construct_wall*.**AssignedResource**=r_1.

iv. $\langle task \rangle$.**AssignedResource** $\neq$ $\langle resource \rangle$

e.g. *install_frame*.**AssignedResource**$\neq r_1$.

v. $f(\langle task \rangle.\langle costterm \rangle, ..., \langle task \rangle.\langle costterm \rangle)\langle op \rangle$

$f'(\langle task \rangle.\langle costterm \rangle, ..., \langle task \rangle.\langle costterm \rangle)$

e.g. $max((carpentry.Duration + roof.Duration),$

$(elec_inst.Duration + plumbing.Duration) \leq 7$ days

vi. **IF** X **THEN** Y, where X and Y are hard resource constraints. It is also possible that X or Y (but not both) is a temporal constraint.

e.g. **IF** $gardening.Cost > 50$ **THEN** $moving.Cost < 80$

2. **Facilitating Resource Allocation Constraints:**

IF $\langle$ hard constraint $\rangle$ **THEN Discount**($\langle$ cost term $\rangle$, $\langle$ amount $\rangle$)

e.g. **IF** $carpentry.AssignedResource = roof.AssignedResource$

THEN Discount$(totalCost, 100)$.

As explained in Section 3.3, *hard constraint* of the constraint may be replaced by a *temporal constraint*.

```
SEQ {
        wall_installation;
        AND {
                SEQ {
                        AND {
                                SEQ {
                                        carpentry;
                                        roof; }
                                SEQ {
                                        electricity_installation;
                                        plumbing; }}
                        WHILE (there_is_leakage){
                                check_plumbing; }
                        AND {
                                XOR {
                                        facade_painting;
                                        vinyl_siding; }
                                XOR {
                                        wooden_frame_installation;
                                        metal_frame_installation; }
                                gardening; }
                        painting; }
                SEQ {
                        ceiling;
                        OR {
                                fireplace_installation;
                                small_sauna_installation; }}}
        moving_into_the_house; }

r1 CAPABLE_OF wall_installation WITH [(time, 10), (money, 10)]
r2 CAPABLE_OF wall_installation WITH [(time, 12), (money, 20)]

wall_installation.money + carpentry.money + ... + moving.money < 200
IF carpentry.AssignedResource = r1 THEN roof.AssignedResource = r1
IF plumbing.AssignedResource = check_plumbing.AssignedResource
                                                THEN Discount(money, 7)

electricity_installation BEFORE ceiling
IF Execute(facade_painting) THEN Execute(wooden_frame_installation)
```

Figure 4.2: House construction workflow in WSL

4.1.2.5 Example - House Construction

The WSL representation of the house construction workflow is given in Figure 4.2. The first part of the WSL representation defines the blocks constituting the workflow. This is followed by resource definitions. Due to space limitation, we give only a part of the resource definitions for the first task. The resource allocation constraints and temporal/causality constraints of Example 1.1 are also specified in WSL, in the last part of the figure.

4.2 Defining Workflow Using Constraint Programming

4.2.1 Translation from *WSL* to Constraint Language (*WftoCon*)

One of the most important contributions of this work is the translator (*WftoCon*) module of the architecture, which is defined in Section 4.1.1. *WftoCon* translates workflow and constraint specifications into a constraint program in Oz. In general, any constraint language can be used as target language. We defined a mapping from each block and constraint type specification in WSL to a constraint structure. By using this mapping, *WfToCon* produces a constraint set equivalent to WSL specification.

In order to capture the inter and intra-block dependencies, we defined *start*, *end* and *duration* attributes for blocks, denoting *beginning,end* and *duration* of block's execution [1]. $\langle block\rangle.start + \langle block\rangle.duration = \langle block\rangle.end$ is a default constraint for each block. As discussed in the previous section, these attributes are defined as finite-domain objects.

In the rest of this section, the mapping from WSL to constraint language is pre-

[1] Since a task is the simplest block, these attributes are defined also for tasks.

sented. We used a classical mathematical notation for the representation of constraints. Since we used constraint programming *Oz* for realization, this notation is converted to Oz's constraint notation in the implementation.

4.2.1.1 Sequential Block

Sequential block P with the sub-blocks $b_1, ..., b_i, ..., b_n$ is defined as:

$P.start = b_1.start$

$P.end = b_n.end$

$\forall\ i,\ 1 \leq i \leq n{-}1,\ b_i.end \leq b_{i+1}.start$

4.2.1.2 AND Block

AND block P with the sub-blocks $b_1, ..., b_i, ..., b_n$ is defined as:

$\forall\ i,\ 1 \leq i \leq n, P.start \leq b_i.start$

$\forall\ i,\ 1 \leq i \leq n, b_i.end \leq P.end$

4.2.1.3 OR Block

OR block P with the sub-blocks $b_1, ..., b_i, ..., b_n$ is defined as follows:

$(\forall_i, 1 \leq i \leq n, \bigvee(P.start \leq b_i.start \wedge P.end \geq b_i.end)$

In an OR block, at least one sub-block is executed. This is modeled through disjunction. Typically, if a block is executed, its start time is an integer value within the execution time scale. In our representation, a task is executed if its start time is within the execution time range of the block in which it is defined.

4.2.1.4 XOR Block

XOR block P with the sub-blocks $b_1, ..., b_i, ..., b_n$ is defined as:

$$\forall\ i,\ 1 \leq i \leq n, \bigvee((P.start = b_i.start \wedge P.end = b_i.end) \wedge$$
$$\forall\ j\ 1 \leq j \leq n, j \neq i\ (b_j.start = outOfRange))$$

In an XOR block, only one of the sub-blocks is executed. The first part of the constraint shows that a sub-block is chosen for execution and the second part guarantees that once a sub-block is executed, the others are not executed. We represent that a sub-block (or task) is not to be executed by setting a start value for it that is out of the normal execution range (given as *outOfRange* above). This may be a negative value or a value that is higher than the end of scale. The sub-block is chosen non-deterministically and this is represented by disjunction.

If the start time of a block is set to *outOfRange*, this should be reflected to all sub-blocks of this block. Hence, the following constraints are defined for subtasks of OR and XOR blocks.

$\forall\ P$,P is OR/XOR block, $\forall\ Q,\ Q \in P,\ P.start = outOfRange \rightarrow Q.start = outOfRange.$

4.2.1.5 Iteration Block

To represent an iteration block as a set of constraints, the tasks in the block must be renamed for each instance of iteration. These instances are modeled as sub-blocks of a sequential block. In order to determine the number of iterations, a close guess may be possible from the previous executions or we may overestimate this number.

If the iteration will take place n times, then we rename n instances of the iteration

block P as $P_1, ..., P_n$ and model the iteration as follows:

$\forall\ i,\ 1 \leq i \leq n-1, (P_i.end \leq P_{i+1}.start)$

In this model, the sub-blocks in P_i's must also be renamed to differentiate the instances. Multiple copies of the constraints involving the iteration block must be defined as the number of the iteration and the iteration block or its sub-blocks must be renamed for different copies, as well.

4.2.1.6 Condition

A precondition for a block P is defined as

$Pre_Cond_P \rightarrow \langle P \rangle$,

where $\langle P \rangle$ is the set of constraints defining block P. Post conditions also can be defined similarly. If execution of a block Q can make a condition true, then this is represented as

$\langle Q \rangle \rightarrow Cond_Q$

4.2.1.7 Temporal and Causality Constraints

In order to represent temporal/causality constraints, as in OR/XOR block definitions, *outOfRange* is used as the starting time value in order to denote that the block is not executed. The temporal/causality constraint definitions of WSL are translated into constraint language as follows:

- **Execute**(t_i) is represented as ($t_i.start \neq outOfRange$) .
- **NotExecute**(t_i) is represented as ($t_i.start = outOfRange$) .
- t_i **BEFORE** t_j is represented as ($t_i.end \leq t_j.start$).

- **IF** X **THEN** Y is represented as $X \rightarrow Y$.

4.2.1.8 Defining Resources

The resource information given in *WSL* as

$\langle Resource_Name \rangle$ **CAPABLE_OF** $\langle task \rangle$

WITH $[(\langle cost_term_1 \rangle, \langle cost_1 \rangle)...(\langle cost_term_n \rangle, \langle cost_n \rangle)]$

is represented as follows:

$\langle task \rangle.resource = \langle resource \rangle \rightarrow$

$\langle task \rangle.\langle cost_term_1 \rangle \langle cost_1 \rangle, ..., \langle task \rangle.\langle cost_term_n \rangle \langle cost_n \rangle$

It is possible that some tasks in OR and XOR blocks are never executed. We have to show that no resource selection is done for these tasks and resource allocation cost is 0. For this purpose, we make additional definitions with empty resource, as follows:

$\langle task \rangle.resource = Empty \rightarrow \langle task \rangle.\langle cost_term_1 \rangle\ 0, ..., \langle task \rangle.\langle cost_term_n \rangle\ 0$

In addition to this, we have the following constraints to show that no resource allocation is performed for tasks with start value $outOfRange$:

$\langle task \rangle.start = outOfRange \rightarrow \langle task \rangle.resource = Empty$

4.2.1.9 Resource Allocation Constraints

The specification of resource allocation constraints in *WSL* and constraint language are similar with minor changes in the notation. For instance,

$\langle task_i \rangle$.**AssignedResource** = $\langle task_j \rangle$.**AssignedResource**

of *WSL* is represented as

$\langle task_i \rangle.resource = \langle task_j \rangle.resource$

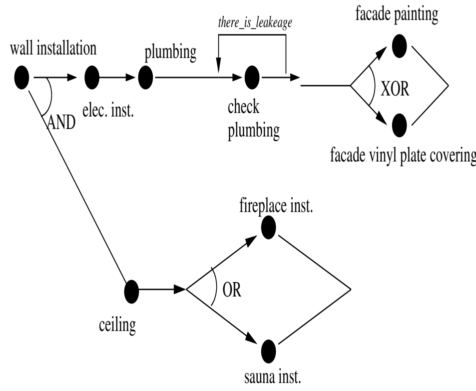


Figure 4.3: Subworkflow for house construction example

tasks	x	y
wall	10	20
elec	20	10
plumb	20	15
checkPlumb	15	10
facadePaint	20	40
facadeVinyl	10	20
ceiling	10	15
fireplace	30	20
sauna	50	60

Figure 4.4: Resource allocation cost table

in constraint language. As another example, in constraint language, facilitating resource allocation constraints are modeled as follows:

$\langle$ hard constraint $\rangle \rightarrow \textit{Discount}.\langle \textit{cost term} \rangle = \langle \textit{value} \rangle$.

4.2.1.10 Example

In this example, we present the constraint language representation of a subworkflow of house construction workflow. We assume that there are two subcontructors (i.e., resources), x and y that are capable of performing the tasks of the workflow given in Figure 4.3 with the financial costs given in Figure 4.4. In order to simplify the example, we assume that the execution time is *1 unit* for all tasks with any resource. The resource allocation constraint is $total_cost < 150$. The constraint language representation of the workflow blocks are given in Figure 4.5, constraints in Figure 4.6 and resource information in Figure 4.7. The number of iterations for *check plumbing* is estimated as 3 and 3 instances of this task is created. As seen from the example, most of the constraints are defined on attributes of the tasks, not the blocks. The constraints on block attributes can be rewritten in terms of task attributes. This reduces the

Duration information

$wall.duration = 1$, $electricity.duration = 1$, ..., $sauna.duration = 1$

Block Definitions

$wall.start + wall.duration \leq electricity.start$

$electricity.start + electricity.duration \leq plumbing.start$

$plumbing.start + plumbing.duration \leq checkPlumbing1.start$

$checkPlumbing1.start + checkPlumbing1.duration \leq checkPlumbing2.start$
$checkPlumbing2.start + checkPlumbing2.duration \leq checkPlumbing3.start$
$checkPlumabing3.start + checkPlumbing3.duration \leq facade.start$

$((facade.start = facadePaint.start \wedge$
$\quad facade.end = facadePaint.start + facadePaint.duration \wedge$
$\quad facadeVinyl.start = outOfRange) \vee$
$(facade.start = facadeVinyl.start \wedge$
$\quad facade.end = facadeVinyl.start + facadeVinyl.duration \wedge$
$\quad facadePaint.start = outOfRange))$

$wall.start + wall.duration \leq ceiling.start$

$ceiling.start + ceiling.duration \leq fireplaceOrSauna.start$

$((fireplaceOrSauna.start = fireplace.start \wedge$
$\quad fireplaceOrSauna.end = fireplace.start + fireplace.duration) \vee$
$(fireplaceOrSauna.start = sauna.start \wedge$
$\quad fireplaceOrSauna.end = sauna.start + sauna.duration))$

Figure 4.5: Blocks of house constraint example

Constraints

$wall.cost + elecricity.cost + plumbing.cost +$
$\quad checkPlumbing1.cost + checkPlumbing2.cost + checkPlumbing3.cost +$
$\quad ceiling.cost + facadePaint.cost + facadeVinyl.cost + firePlace.cost +$
$\quad sauna.cost - discount = total_cost$

$total_cost < 150$

$electricity.end \leq ceiling.start$

$plumbing.resource = check_plumbing1.resource \wedge$
$\quad check_plumbing1.resource = check_plumbing2.resource \wedge$
$\quad check_plumbing2.resource = check_plumbing3.resource \rightarrow discount = 7$

Figure 4.6: Constraints for house constraint example

redundancy in constraint set. Although not given in the example, domain declarations are also a part of the specification.

Resource definitions:

$wall.resource = x \rightarrow wall.cost = 10$
$wall.resource = y \rightarrow wall.cost = 20$

$electricity.resource = x \rightarrow electricity.cost = 20$
$electricity.resource = y \rightarrow electricity.cost = 10$

$plumbing.resource = x \rightarrow plumbing.cost = 20$
$plumbingc.resource = y \rightarrow plumbing.cost = 15$

$checkPlumbing1.resource = x \rightarrow checkPlumbing1.cost = 15$
$checkPlumbing1.resource = y \rightarrow checkPlumbing1.cost = 10$

$checkPlumbing2.resource = x \rightarrow checkPlumbing2.cost = 15$
$checkPlumbing2.resource = y \rightarrow checkPlumbing2.cost = 10$

$checkPlumbing3.resource = x \rightarrow checkPlumbing3.cost = 15$
$checkPlumbing3.resource = y \rightarrow checkPlumbing3.cost = 10$

$facadePaint.resource = x \rightarrow facadePaint.cost = 20$
$facadePaint.resource = y \rightarrow facadePaint.cost = 40$
$facadePaint.resource = Empty \rightarrow facadePaint.cost = 0$

$facadeVinyl.resource = x \rightarrow facadeVinyl.cost = 10$
$facadeVinyl.resource = y \rightarrow facadeVinyl.cost = 20$
$facadeVinyl.resource = Empty \rightarrow facadeVinyl.cost = 0$

$ceiling.resource = x \rightarrow ceiling.cost = 10$
$ceiling.resource = y \rightarrow ceiling.cost = 15$

$fireplace.resource = x \rightarrow fireplace.cost = 30$
$fireplace.resource = y \rightarrow fireplace.cost = 20$
$fireplace.resource = Empty \rightarrow fireplace.cost = 0$

$sauna.resource = x \rightarrow sauna.cost = 50$
$sauna.resource = y \rightarrow sauna.cost = 60$
$sauna.resource = Empty \rightarrow sauna.cost = 0$

$facadePaint.start = outOfRange \rightarrow facadePaint.resource = Empty$
$facadeVinyl.start = outOfRange \rightarrow facadeVinyl.resource = Empty$
$fireplace.start = outOfRange \rightarrow fireplace.resource = Empty$
$sauna.start = outOfRange \rightarrow sauna.resource = Empty$

Figure 4.7: Resource definitions for house construction example

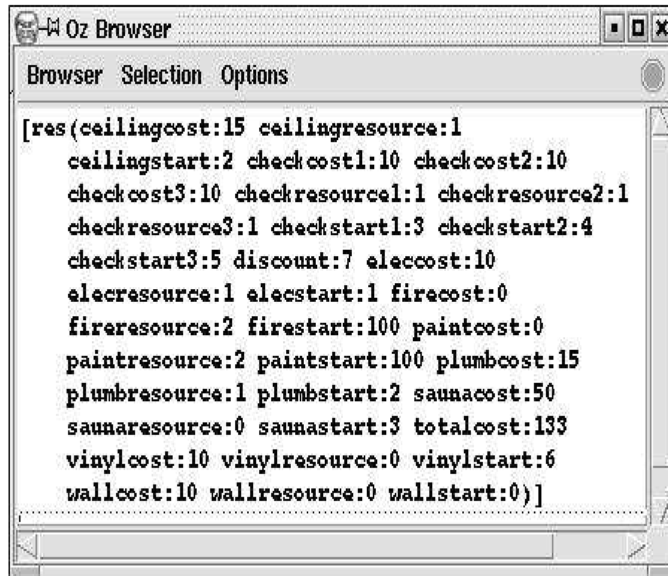


Figure 4.8: Solution produced by Oz

A feasible solution provided by Oz to the house construction workflow of Figure 4.3 is presented in Figure 4.8, which is a snapshot from Oz programming environment. The solution contains resource allocations, costs of resource allocations and start times for each task. The total budget (represented as *totalcost*) is also a part of solution. The content of the solution may be extended or reduced depending on the number of cost dimension attributes of resource allocation. The results are listed in alphabetical order. Start value 100 denotes *outOfRange* value. Resources are represented as integers: 0 denotes resource x, 1 denotes resource y and 2 denotes *Empty* resource allocation. Figure 4.8 represents the schedule given in Figure 4.9. The total cost of this schedule is 133 with a discount of 7 (see Figure 4.8). This solution produced by Oz is one of the feasible solutions that satisfy all the constraints.

In order to activate the workflow according to the provided results, the start times can be sent to the enactment engine and resource assignment information is sent to the

tasks	resource	cost	start
wall	x	10	0
elec	y	10	1
plumb	y	15	2
checkPlumb1	y	10	3
checkPlumb2	y	10	4
checkPlumb3	y	10	5
facadePaint	empty	0	do not execute
facadeVinyl	x	10	6
ceiling	y	15	2
fireplace	empty	0	do not execute
sauna	x	50	3

Figure 4.9: Schedule for house construction example

resource manager. The enactment engine activates the tasks according to the schedule. Just as a task is activated, the enactment engine notifies the resource manager and therefore resource manager allocates the scheduled resource for the task. Although our architecture is designed to provide feasible solution, it can be tuned to find all solutions or the optimal solution. However, as expected, in this case execution time will increase since the whole search space needs to be explored.

4.2.2 Experiments and Efficiency Issues

For most of the constraint problems, the aim is to find the optimal solution which requires the search throughout the whole search space. However, in this work, the aim is to obtain a feasible solution. For this reason, complexity of the problem is reduced when compared to conventional scheduling problems that requires optimal solution.

In order to check the feasibility of constraint programming approach for workflow scheduling under resource allocation constraints and to see the effect of different workflow structures on the execution times, we have conducted some experiments. In the first set of experiments, we tested the effect of increase in constraint set on four types

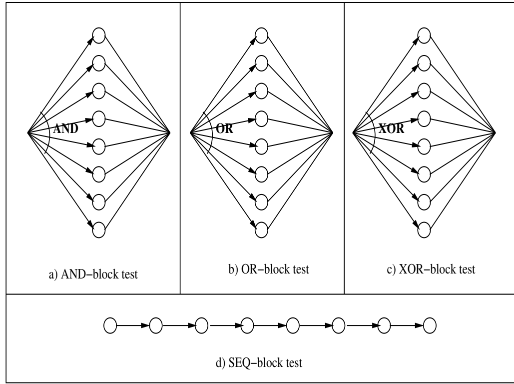


Figure 4.10: Structures used in the first set of experiments

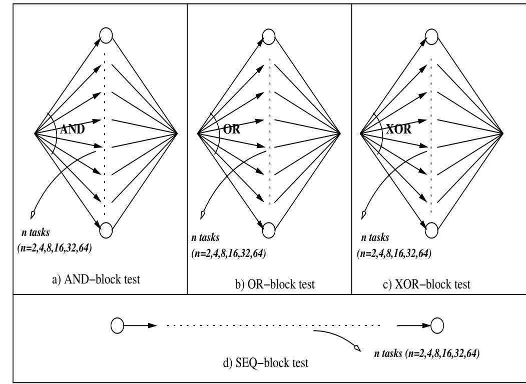


Figure 4.11: Structures used in the second set of experiments

of workflows that include the same fixed number of tasks. The first workflow consists of a single sequential block, second one has a single AND block, third one has a single OR block and fourth one consists of a single XOR block. We have not used iteration blocks, since it is modeled using sequential blocks. The experiment is conducted with constraints sets including 2, 4, 8, 16 and 32 constraints. The workflow structures used in the experiments are shown in Figure 4.10. The result of this first set of experiments is presented in Figure 4.12. The basic results of this experiment are as follows:

- The increase in the number of tasks leads to more increase in execution time compared to the increase in the size of the constraint set. This result is expected due to the fact that each task is represented with a constraint variable and therefore its associated domain and each new task means a new domain to be reduced.
- The increase in the execution time for workflows consisting of OR block and XOR block is higher compared to workflows consisting of SEQ and AND blocks. This is due to the disjunction in the semantics of OR and XOR blocks.
- When we compare OR and XOR blocks, the execution time for XOR block is

slightly longer than that of OR block. The reason for this result is that OR blocks are easier to satisfy. The same set of constraints may fail to find a solution for XOR block.

- The increase in execution time is all polynomial.

On the average, the workflows consist of mostly sequential and AND blocks. The number and the size of XOR and OR blocks appear only occasionally and thus the execution time is expected to be below the OR/XOR curve. For instance, the execution time for House Construction example is 0.90 seconds. More efficient results can be obtained by using specialized solvers.

- The increase in the number of constraints causes a little increase in the execution time.

- The amount of increase is the same for all four types of workflows.

- The ratio between the execution time of XOR/OR blocks vs. SEQ/AND blocks is almost constant and it is related to the number of tasks.

The second set of experiments is conducted to check the effect of increase in the number of tasks. Again we have used four types of workflows including only a sequential block, AND block, OR block and XOR block, respectively. The experiment is repeated with exponentially increasing number of tasks in blocks - using 2, 4, 8, 16 and 32 tasks - under the same constraint set with 5 constraints. The workflow structures used in the experiments are shown in Figure 4.11. The result is given in Figure 4.13. The basic results of this experiment are listed as follows:

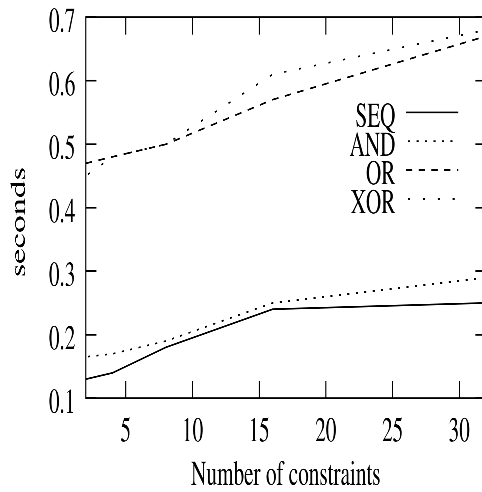


Figure 4.12: Effect of increase in constraint set

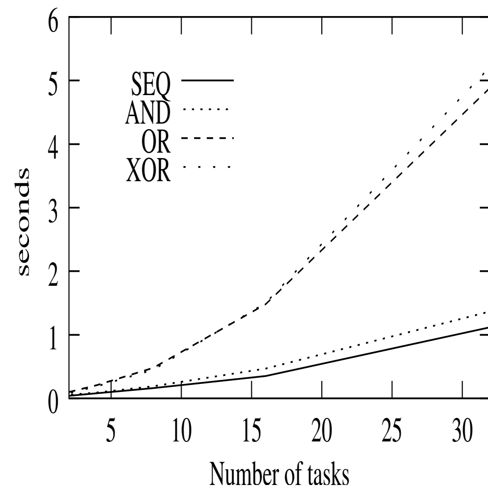


Figure 4.13: Effect of increase in number of tasks

- The increase in the number of tasks leads to more increase in execution time compared to the increase in the size of the constraint set. This result is expected due to the fact that each task is represented with a constraint variable and therefore its associated domain and each new task means a new domain to be reduced.

- The increase in the execution time for workflows consisting of OR block and XOR block is higher compared to workflows consisting of SEQ and AND blocks. This is due to the disjunction in the semantics of OR and XOR blocks.

- When we compare OR and XOR blocks, the execution time for XOR block is slightly longer than that of OR block. The reason for this result is that OR blocks are easier to satisfy. The same set of constraints may fail to find a solution for XOR block.

- The increase in execution time is all polynomial.

On the average, the workflows consist of mostly sequential and AND blocks. The number and the size of XOR and OR blocks appear only occasionally and thus the

execution time is expected to be below the OR/XOR curve. For instance, the execution time for House Construction example is 0.90 seconds. More efficient results can be obtained by using specialized solvers.

CHAPTER 5

Developing a Logic-based Framework to Schedule Workflows Under Resource Allocation Constraints

In this chapter, we present a new logical framework, based on *Concurrent Constraint Transaction Logic* (abbr., *CCTR*). CCTR extends Concurrent Transaction Logic (abbr., *CTR*) [13] by incorporating ideas from Constraint Logic Programming (CLP) [49, 50]. This new framework can be used for the specification, verification and the scheduling of workflows containing resource allocation constraints in addition to ordinary temporal/causality constraints. Shorter version of this work is explained in [70].

The role of CCTR in our framework is to model workflows and specify all kinds of constraints in a rigorous and precise way. The semantics of the CCTR modeling of a workflow represents to a schedule that contains both an execution ordering that the specified workflow can execute, and a set of resource assignments to the tasks of the workflow satisfying all the given constraints. The realization of this framework can be summarized in three components:

- A formula transformer, that transforms the conjunctive CCTR formula which specifies the workflow in conjunction with the cost and control constraints, into a conjunction-free CTR formula that contains exactly the same constraints.

- A CTR interpreter, that solves the non-constraint part of the formula in order to determine the partial schedule of the workflow and extract the constraints.

- A constraint solver, that determines the resource allocations to the tasks of the workflow by solving the constraints.

5.1 Concurrent Constraint Transaction Logic (CCTR)

In this work, we extended the Concurrent Transaction Logic (CTR) with the capability of workflow modeling and scheduling under resource allocation constraints. We called this new formalism *Concurrent Constraint Transaction Logic (CCTR)*. CTR is an extension to first-order logic for programming, executing and reasoning state changing concurrent processes. It was introduced in [13] and it is one of the few formalisms that have been successfully applied to modeling, reasoning about and scheduling workflows [29, 11]. For instance, [29] shows that a large class of temporal and causality constraints can be represented in CTR and that the proof theory of the logic can be used to perform a number of tasks ranging from consistency checking of a workflow to its scheduling subject to the specified constraints. In CCTR, we exploited the workflow modeling, reasoning and scheduling capabilities of CTR and extended these capabilities for the new set of constraints. In this section, we present CCTR language with its syntax, semantics and model theory.

5.1.1 Syntax

In CCTR, the alphabet consists of four countable sets of symbols: a set $\mathcal{F}$ of function symbols, a set $\mathcal{V}$ of variables, a set $\mathcal{P}$ of predicate symbols and a set $\mathcal{C}$ of constraint predicates. As we shall see, constraint predicates are treated differently from other predicates in CCTR.

Each function, predicate and constraint predicate symbol has an *arity*, indicating the number of arguments the symbol takes. Constants are viewed as 0-arity function symbols and propositions are viewed as 0-arity predicate symbols.

CCTR also extends classical logic with four new connectives and modalities (borrowed from CTR) of which the most important are $\otimes$, the *serial conjunction*, and $|$, the *parallel conjunction*. The simplest transaction formulas are *atomic formulas*, which are defined via predicates and terms. In addition, if ϕ and ψ are transaction formulas, then so are the following expressions:

- $\phi \vee \psi, \phi \wedge \psi, \phi \otimes \psi, \phi \mid \psi, \neg\psi$
- $(\forall X)\phi$ and $(\exists X)\phi$, where X is a variable.

Intuitively, the formula $\phi \otimes \psi$ means that the subtransactions ϕ and ψ execute serially. The formula $\phi \mid \psi$ means that subtransactions ϕ and ψ execute concurrently.

To see how CCTR can be used to model a workflow, we show a formula that corresponds to a part of the house building process in Figure 1.1:

$$\begin{aligned}&\texttt{wall}\otimes\\&\quad(((\texttt{(carpentry} \otimes \texttt{roof)} \mid \texttt{installations)} \otimes \textit{the-middle-piece}) \mid \texttt{ceiling})\\&\otimes \texttt{paint} \otimes \texttt{move}\end{aligned} \tag{5.1}$$

Each proposition here represents a task and CCTR operators tell whether to combine the tasks concurrently or serially. The proposition *the-middle-piece* represents the part of the workflow that did not fit and we show it separately:

$$(\texttt{facade-paint} \vee \texttt{facade-vinyl}) \mid (\texttt{wooden-windows} \vee \texttt{metal-windows}) \mid \texttt{gardening}$$

As can be seen from the last formula, $\vee$ represents alternative executions in workflow. For instance, in the above workflow we only need to either paint the facade or cover it with vinyl.

Let wf denote the formula in (5.1) and let c be a constraint expressed using the predicates from the set $\mathcal{C}$. Then $wf \wedge c$ is a complete specification of Example 1.1 in CCTR. The symbol c here denotes a resource allocation constraint, which in our framework has two parts: a control constraint and a cost constraint. These constraints will be discussed later.

It should be noted that many kinds of constraints are already expressible in CTR and don't require CCTR. However, handling control constraints require a different, more expressive semantics and cost constraints require an extension that is analogous to the way constraint logic programming extends regular logic programming.

5.1.2 Semantics

5.1.2.1 States and Oracles

In CCTR, like in Transaction Logic, the specification of the elementary database operations is incorporated into the language as a parameter through a pair of oracles, the *data oracle* and the *transition oracle*. The data oracle specifies a set of primitive database queries (i.e., the *static* semantics of the states) and transition oracle specifies

a set of primitive updates (i.e., the *dynamic* semantics of the states). These oracles are defined with a set of *database states*. Intuitively, a database state is a set of data items, which can be any kind of persistent object, such as a tuple, a disk page, an email queue, or a logical formula. Formally, however, a database state has no structure, and we access it only through the oracles.

Data Oracle. Formally, data oracle, denoted by $\mathcal{O}^d$, is a mapping from states to sets of first-order formulas. Intuitively, if D is a state, then $\mathcal{O}^d(D)$ is the set of formulas that are true of the state.

Transition Oracle. Formally, transition oracle, denoted by $\mathcal{O}^t$ is a mapping from pairs of states to sets of ground atomic formulas. Intuitively, if D_1, D_2 are states and b is a ground atomic formula, $b \in \mathcal{O}^t(D_1, D_2)$ means that b is an elementary update that changes the current state from D_1 to D_2. Further details and examples on data and transition oracle can be found in [12] and [13]

In CCTR, we have one more parameter of the language, *constraint universe*, whose details will be discussed later.

5.1.2.2 Partial Schedules

Paths and Multi-paths. In Transaction Logic [12], formulas are viewed as transactions that execute along a sequence of database states and during the execution they query and change the underlying database state. When a user executes a transaction, the database changes from its initial state to some final state, going through any number of intermediate states. This sequence of states is called a *path*, and the truth value of a transaction is determined with respect to paths. For example, $a.ins \otimes b.ins$ is a

transaction that first inserts a and then b. It happens to be true on the path $\langle D_0 D_1 D_2 \rangle$, where $D_0 = \{\}$, $D_1 = \{a\}$ and $D_3 = \{a, b\}$.

Concurrent Transaction Logic was designed to model concurrent processes; it generalizes the notion of a path to *multi-paths* (abbr., *m-path*). Intuitively, an m-path represents periods of continuous execution, separated by periods of suspended execution. Formally, an m-path is a finite sequence of paths, where each constituent part represents a period of continuous execution. For example, $\langle D_1 D_2 D_3, D_4 D_5, D_6 D_7 D_8 \rangle$ is an m-path. Note that a path can be viewed a special case of m-paths.

Partial Schedule. While m-paths are adequate to model serial and concurrent execution in CTR, they are not sufficient to model resource requirements that may be necessary for those executions to succeed. In CCTR we need to be able to distinguish that two m-paths are part of different concurrent branches of the same execution. To this end, we introduce the notion of a *partial schedule*, which adds certain amount of structure to m-paths.

Partial schedules are composed using two operators: $\bullet_p$ and $\|_p$. The first represents concatenation and is associative; the second does parallel combination of schedules and is both associative and commutative.

Definition 5.1 *A* partial schedule *is defined as follows:*

- *A* simple *partial schedule is just an m-path.*
- *Serial composition of two partial schedules, $\omega_1 \bullet_p \omega_2$, is a partial schedule*
- *Parallel composition of two partial schedules, $\omega_1 \|_p \omega_2$, is a partial schedule*

In CTR, concatenation and interleaving operations are defined on m-paths as fol-

lows: If $\pi = \langle \kappa_1, ..., \kappa_n \rangle$ and $\pi' = \langle \kappa'_1, ..., \kappa'_m \rangle$ are two m-paths, then their *concatenation* is the m-path $\pi \bullet \pi' = \langle \kappa_1, ..., \kappa_n, \kappa'_1, ..., \kappa'_m \rangle$. If π and $\pi_1, ..., \pi_n$ are m-paths, then π is an *interleaving* of $\pi_1, ..., \pi_n$ if π can be partitioned into order-preserving subsequences $C_1, ..., C_n$ such that each C_i is π_i. The set of all interleavings of π_1 and π_2 is denoted $\pi_1 \parallel \pi_2$. We will use these operations to define the association between partial schedule and m-paths.

Definition 5.2 Every partial schedule ω can be associated with a set of m-paths, where each m-path represents a possible full schedule that is consistent with ω.

- if ω_1 and ω_2 are partial schedules and π_1 and π_2 are m-paths, then

$$mpaths(\omega_1 \bullet_p \omega_2) \equiv \{\pi_1 \bullet \pi_2 | \pi_1 \in mpaths(\omega_1), \pi_2 \in mpaths(\omega_2)\} \tag{5.2}$$

- ω_1 and ω_2 are partial schedules and π_1 and π_2 are m-paths, then

$$mpaths(\omega_1 \parallel_p \omega_2) \equiv \bigcup_{\pi_1 \in mpaths(\omega_1), \pi_2 \in mpaths(\omega_2)} (\pi_1 \parallel \pi_2) \tag{5.3}$$

- If ω is a *simple* partial schedule, then

$$mpath(\omega) = \{\omega\}, \text{ recall that a simple partial schedule is an m-path.} \tag{5.4}$$

5.1.2.3 Resource and Resource Assignment

In proposing CCTR, the basic motivation is to provide a mechanism to define and solve resource allocation constraints. For this reason, we define *resource* and *resource assignment* as a part of CCTR as given below.

Definition 5.3 *A* resource *is an object with the attributes* token *and* cost*.*

In workflow modeling, a resource typically represents an agent or a device needed for executing various tasks. The attribute *token* then represents the name of the resource and the attribute *cost* represents the cost of using that resource. The cost does not need to be a number - it can be a structured object such as a list of numbers.

Definition 5.4 *A* resource assignment *is a partial mapping from partial schedules to sets of resources. Any resource assignment, ξ, must satisfy the following conditions:*

- $\xi(\omega_1 \parallel_p \omega_2) = \xi(\omega_1) \cup \xi(\omega_2)$, *if both* $\xi(\omega_1)$ *and* $\xi(\omega_2)$ *are defined*

- $\xi(\omega_1 \bullet_p \omega_2) = \xi(\omega_1) \cup \xi(\omega_2)$, *if both* $\xi(\omega_1)$ *and* $\xi(\omega_2)$ *are defined*

5.1.2.4 Constraint Universe

Constraint universe is the third parameter of CCTR language. It contains the domains that will be used later to define the semantics of constraint predicates.

Definition 5.5 *A* constraint universe $\mathcal{D}$ *is a set of domains together with predicates associated with each domain. The domains in the constraint universe are*

1. Elementary Domains *include scalar domains (e.g. integer), goal domain (i.e., the set of all CCTR goals, which are formulas that represent workflows), the domain of partial schedules, the domain of resource assignments and the domain of resources.*

2. Complex Domains *are domains that are composed out of elementary domains using various set constructors (e.g., goal $\times$ partial schedule, $2^{resource}$).*

Each domain in $\mathcal{D}$ has a set of constraint predicates associated with it.

Example 5.6 Here are some examples of constraint predicates in $\mathcal{D}$:

1. $disjoint(R_1, R_2) \equiv (R_1 \cap R_2 = \varnothing)$ is a predicate on the domain $(2^{resource} \times 2^{resource})$ where $R_1, R_2 \subset 2^{resource}$.

2. $less_than_c(I) \equiv (I < c)$ is a predicate on the integer domain, where $I \in$ *integer domain* and c is an integer constant.

3. $cost_constraint(\omega, \xi) \equiv less_than_c(f(\omega, \xi))$ is a predicate on the domain (*partial schedule* $\times$ *resource assignment*), where ω is a partial schedule, ξ is a resource assignment, $less_than_c$ is defined above, and f is a function of type (*partial schedule* $\times$ *resource assignment* $\mapsto$ *integer*.) A function like this is typically used to define the cost of executing the schedule under the given resource assignment.

Constraint universe contains relations that represent the meaning to the constraint predicate symbols in $\mathcal{C}$. For each constraint predicate symbol $c \in \mathcal{C}$, there is a relation $c_{\mathcal{D}}$ in the constraint universe, which has two extra arguments. One of the new arguments is a partial schedule and the other is a resource assignment.

5.1.3 Model Theory of CCTR

The semantics of CCTR is based on partial schedule structures. A partial schedule structure is a mapping that assigns a regular first-order semantic structure to every partial schedule.

Definition 5.7 (*Partial Schedule Structures*) Let $\mathcal{L}$ be a language of *CCTR* with the set of function symbols $\mathcal{F}$ and let $\mathcal{D}$ be a constraint universe. A *partial schedule structure* $\mathbf{M}$ over $\mathcal{L}$ is a 3-tuple $\langle U, I_{\mathcal{F}}, I_{ps} \rangle$, where

- U is a set, called the *domain* of $\mathbf{M}$,

- $I_{\mathcal{F}}$ is an interpretation of function symbols in $\mathcal{L}$. It assigns a function of type $U^n \mapsto U$ to every n-ary function symbol in $\mathcal{F}$.

  Let $Struct(U, I_{\mathcal{F}})$ denote the set of all classical first-order semantic structures over $\mathcal{L}$ of the form $\langle U, I_{\mathcal{F}}, I_{\mathcal{P}}, I_{\mathcal{C}} \rangle$, where $I_{\mathcal{P}}$ is a mapping that interprets predicate symbols in $\mathcal{P}$ by relations on U. $I_{\mathcal{C}}$ performs a similar function for constraint predicates: it interprets the predicates in $\mathcal{C}$ by relations in the constraint domain in $\mathcal{D}$.

- I_{ps} is a total mapping from the partial schedules in $\mathcal{L}$ to the semantic structures in $Struct(U, I_{\mathcal{F}})$. I_{ps} is subject to the following restrictions:

  - *Compliance with the data oracle:* $I_{ps}(\langle \mathbf{D} \rangle) \models^c \sigma$ for every formula $\sigma \in \mathcal{O}^d(\mathbf{D})$.

  - *Compliance with the transition oracle:* $I_{ps}(\langle \mathbf{D_1 D_2} \rangle) \models^c b$ for every formula $b \in \mathcal{O}^t(\mathbf{D_1 D_2})$.

As in classical logic, a *variable assignment*, v is a mapping $\mathcal{V} \mapsto \mathcal{U}$ that takes variables as input and returns domain elements as output. We extend the mapping from variables to terms in the usual way.

In CCTR, a formula that holds along a partial schedule can be informally understood as being able to execute according to that schedule in a way that satisfies all the resource allocation constraints.

Satisfaction of CCTR formulas. Let ω be a partial schedule, M be a partial schedule structure, ξ be a resource assignment and α be a CCTR goal. $M, \omega, \xi \models \alpha$ (read: α is

true in M along the schedule ω under the resource assignment ξ) is defined as follows:

- If α is a variable-free atomic formula of the form $p(t_1, ..., t_n)$, where $p \in \mathcal{P}$, then $M, \omega, \xi \models_v \alpha$, if and only if ω is a partial schedule and $I_{ps}(\omega) \models^c_{\mathcal{V}} \alpha$. Here $\models^c_v$ stands for entailment in classic first-order logic (recall that $M(\omega)$ is a first-order semantic structure).

- if α is a variable-free constraint predicate of the form $c(t_1, ..., t_n)$, where $c \in \mathcal{C}$, then $M, \omega, \xi \models \alpha$, if and only if $\mathcal{D} \models_{\sqsubseteq} \rfloor_{\mathcal{D}}(\omega, \xi, \sqcup_{\infty}, ..., \sqcup_{\backslash})$, where $c_{\mathcal{D}} = I_{\mathcal{C}}(c)$, is a relation in $\mathcal{D}$ that corresponds to c.

- if α is any atomic formula, then
  $M, \omega, \xi \models \neg\alpha$, iff it is not the case that $M, \omega, \xi \models_v \alpha$.

- $M, \omega, \xi \models_v \alpha \otimes \beta$, if and only if $\omega = \omega_1 \bullet_p \omega_2$, $M, \omega_1, \xi \models_v \alpha$ and $M, \omega_2, \xi \models_v \beta$.

- $M, \omega, \xi \models_v \alpha \mid \beta$, if and only if $\omega = \omega_1 \parallel_p \omega_2$, $M, \omega_1, \xi \models_v \alpha$ and $M, \omega_2, \xi \models_v \beta$.

- $M, \omega, \xi \models_v \alpha \wedge \beta$, if and only if $M, \omega, \xi \models_v \alpha$ and $M, \omega, \xi \models_v \beta$.

- Universal and existential quantification is defined as usual in first-order logic.

Note that satisfaction of a CCTR goal that does not contain any constraint predicates does not depend on resource assignment. Therefore, if α is a goal with no constraint predicates, we can use the notation $M, \omega \models \alpha$, which omits variable assignment.

The first item in the above definition states that an atomic transaction named α (which does not involve constraints) is true along the simple partial schedule ω, if the partial schedule structure M says that α is true along ω in the classical sense. The intuitive meaning of this statement is that α is the name of a transaction that

can "execute" along ω under any resource assignment ξ. The second item states that satisfaction of constraint predicates determined by the constraint universe as explained in Section 5.1.2.4. The third item defines the meaning of negation and says that if α is not true along ω under ξ, then $\neg\alpha$ is true along ω under ξ. The fourth item says that the transaction $\alpha \otimes \beta$ can execute along a schedule ω if and only if this schedule is a concatenation of two schedules and α can execute along the prefix of the schedule, while β can execute along the suffix of the schedule. The fifth item states that a parallel combination of transactions, $\alpha \mid \beta$, can execute along a schedule ω if and only if it is a parallel combination of schedules, $\omega_1 \parallel_p \omega_2$, and α can execute along ω_1 while β can execute along ω_2. The sixth item says that in order to execute $\alpha \wedge \beta$ along a schedule, both α and β must be true along the schedule.

A *CCTR rule* has the form *head* : −*body*, where *head* is an atomic formula which is not a constraint and *body* is a CCTR goal. The semantics of such a rule is analogous to first-order logic: It is satisfied in a partial schedule structure M if, for every partial schedule ω, $M, \omega, \xi \models body$ implies $M, \omega, \xi \models head$.

Definition 5.8 A *CCTR goal* is any formula of the form:

- an atomic formula of the form $p(t_1, ..., t_n)$, where $p \in \mathcal{P}$; or
- $\phi_1 \otimes ... \otimes \phi_n$, where each ϕ_i is a CCTR goal; or
- $\phi_1 \mid ... \mid \phi_n$, where each ϕ_i is a CCTR goal; or
- $\phi_1 \vee ... \vee \phi_n$, where each ϕ_i is a CCTR goal; or
- $\phi \wedge c_1(t_1, ..., t_n) \wedge .... \wedge c_n(t_1, ..., t_m)$, where ϕ is a CCTR goal and $\forall c_i, c_i \in \mathcal{C}$.

A CCTR program consists of two parts: the transaction base **P**, and the initial database state **D**. Recall that database state is a set of data items and transaction base is a finite set of transaction formulas. The transaction base specifies procedures for updating the database and answering the queries. Database is the updatable part of the program. On the other hand, transaction base is immune to the changes. On the basis of this program structure, we define *executional entailment* as given below.

Definition 5.9 Let **P** be a transaction base, let ϕ be a CCTR formula, ξ be a resource assignment, ω is a partial schedule and $D_0, D_1, ..., D_n$ be a sequence of database states. Then

$$\mathbf{P}, D_0, D_1, ..., D_n, \xi \models \phi \tag{5.5}$$

is true iff $M, \omega, \xi \models \phi$ for every model **M** of **P** and $\langle D_0, D_1, ..., D_n \rangle \in mpaths(\omega)$. Related to this is the statement

$$\mathbf{P}, D_0, - - -, \xi \models \phi \tag{5.6}$$

which is true iff Statement 5.5 is true for some sequence of database states.

5.2 Wf-Constraint Universe

Since resource assignment and constraints are part of CCTR, workflow specifications with resource allocation constraints can be modeled and solved directly in CCTR. In this section, we define a constraint universe, called *wf-constraint universe*, in order to model cost and control constraints on a workflow specification. We make the following natural assumption regarding workflow tasks: Each task is modeled as a transaction

that inserts an atom into the database indicating that the task has been executed. for all ω_i such that

Recall that each n-ary constraint predicate symbol c is represented by a (n+2)-ary relation $c_{\mathcal{D}}$ over the constraint universe $\mathcal{D}$. Since resource allocation constraints generally belong to two categories, cost constraints and control constraints, we introduce 0-ary constraint predicate symbols, which we will denote as *cost_constraint* and *ctrl_constraint*. There can be many such predicate symbols in $\mathcal{C}$, and our use of the above name is generic, *i.e.,* for example, *cost_constraint* refers to any cost constraint predicate. Therefore, the constraint system should contain the corresponding 2-ary relations $cost_constraint_{\mathcal{D}}$ and $ctrl_constraint_{\mathcal{D}}$.

Definition 5.10 *A* wf-constraint universe ζ *contains a set of constraint definitions. It consists of two subsystems* ζ_{cost} *and* ζ_{ctrl}.

The ζ_{cost} subsystem is used to specify cost constraints (*e.g.*, this task must execute in less than 1 day); the ζ_{ctrl} subsystem is used to specify control constraints (*e.g.*, the copier on the second floor cannot be used by two concurrent tasks).

Definition 5.11 The constraint subsystem ζ_{cost} consists of relations of the form $cost_constraint_{\mathcal{D}}(\omega, \xi)$, where ω is a partial schedule and ξ is a resource assignment. More specifically, $cost_constraint_{\mathcal{D}}(\omega, \xi)$ has the form $value_constraint(cost(\omega, \xi))$, where *value_constraint* is a relation over a scalar domain (*e.g.*, *integer*) and *cost* is a function with the following properties. Let ω_1 and ω_2 be partial schedules such that both $cost(\omega_1, \xi)$ and $cost(\omega_2, \xi)$ are defined. Then:

$cost(\omega_1 \parallel_p \omega_2, \xi) = op_{|}(cost(\omega_1, \xi), cost(\omega_2, \xi))$

$cost(\omega_1 \bullet_p \omega_2, \xi) = op_{\otimes}(cost(\omega_1, \xi), cost(\omega_2, \xi))$

$cost(\omega, \xi) = cost_of(r)$, where $r \in \xi(\omega)^1$, if ω is an m-path. (Recall that $cost_of$ is a function that determines the cost of using the resource r, as defined right after Definition 5.3)

Here, $op_{\otimes}$ and $op_{|}$ are functions that takes a pair of scalar values and return another scalar value.

Definition 5.12 The constraint subsystem ζ_{ctrl} consists of relations of the form $ctrl_constraint_{\mathcal{D}}(\omega, \xi)$, which satisfy the following conditions. Let $\omega, \omega_1, \omega_2$ be partial schedules and ξ be an assignment such that both $\xi(\omega_1)$ and $\xi(\omega_2)$ are defined. Then $ctrl_constraint_{\mathcal{D}}$ has the form

$$ctrl_constraint_{\mathcal{D}}(\omega_1 \bullet_p \omega_2, \xi) \equiv$$
$$set_constraint_{\otimes}(\xi(\omega_1), \xi(\omega_2)) \land ctrl_constraint_{\mathcal{D}}(\omega_1, \xi) \land ctrl_constraint_{\mathcal{D}}(\omega_2, \xi)$$

$$ctrl_constraint_{\mathcal{D}}(\omega_1 \parallel_p \omega_2, \xi) \equiv$$
$$set_constraint_{|}(\xi(\omega_1), \xi(\omega_2)) \land ctrl_constraint_{\mathcal{D}}(\omega_1, \xi) \land ctrl_constraint_{\mathcal{D}}(\omega_2, \xi)$$

$ctrl_constraint_{\mathcal{D}}(\omega, \xi) \equiv task_constraint(\xi(\omega))$, if ω is an m-path

Here $set_constraint_{\otimes}$ and $set_constraint_{|}$ are relations over the domain $2^{resources} \times 2^{resources}$, which are intended to express conditions on sets of resources, such as disjointness. The relation $task_constraint$ is defined over the domain $2^{resources}$; it restricts executions of individual tasks and can be used to say that, for example, task t_1 cannot be executed by agent A.

To ensure that $cost_constraint_{\mathcal{D}}$ is well-defined [2], we impose the following restrictions on $op_{|}$ and $op_{\otimes}$:

[1] Note that here we consider assignment of a single resource for a task

[2] A relation definition is *well-defined* if it produces the same result independent of the processing order of its arguments

Definition	Com	Dis
$disjoint(R_1, R_2) = (token_of(R_1) \cap token_of(R_2) \equiv \varnothing)$	Yes	Yes
$subset(R_1, R_2) = (token_of(R_1) \subset token_of(R_2))$	No	Yes
$subsumes_c(R_1, R_2) = ((token_of(R_1) \cap token_of(R_2)) \subset token_of(c))$	Yes	Yes

Figure 5.1: Examples of set constraints and their properties

- *Commutativity:* $op_|(X, Y) = op_|(Y, X)$.
- *Associativity:*

$$op_|(op_|(X, Y), Z)) = op_|(X, op_|(Y, Z)) \; op_\otimes(op_\otimes(X, Y), Z)) = op_\otimes(X, op_\otimes(Y, Z)).$$

Lemma 5.13 *The function "cost" in Definition 5.11 is well-defined.*

Similarly, to ensure that $ctrl_constraint_{\mathcal{D}}$ is well-defined, we impose the following restrictions on $set_constraint_\otimes$ and $set_constraint_|$.

- *Commutativity:* $set_constraint_|(R_1, R_2) = set_constraint_|(R_2, R_1)$.
- *Distribution over Union:*

$$set_constraint_\otimes(R_1 \cup R_2, R_3) = set_constraint_\otimes(R_1, R_3) \wedge set_constraint_\otimes(R_2, R_3)$$

$$set_constraint_|(R_1 \cup R_2, R_3) = set_constraint_|(R_1, R_3) \wedge set_constraint_|(R_2, R_3).$$

Lemma 5.14 *The predicate ctrl_constraint in Definition 5.12 is well-defined.*

Although there are no restrictions on the use of the attributes *cost* and *token* in specifying constraints, *cost* is typically used in *cost_constraint* and *token* in *ctrl_constraint*. The functions $op_\otimes$ and $op_|$ are usually aggregates, such as `sum` or `max`, and the relations $set_constraint_|$ and $set_constraint_\otimes$ are various set constraints, such as those in Figure 5.1.

Example 5.15 Let $R_1,\ R_2$ denote sets of resources. Figure 5.1 lists some predicates along with their distributivity and commutativity properties. Those that have both properties can be used as $set_constraint_{|}$ and those that have only distributivity can be used as $set_constraint_{\otimes}$ only.

Example 5.16 Example 1.1 has two cost constraints that would be defined in ζ_{cost} and one control constraint that would be defined in ζ_{ctrl}. For ζ_{cost}, the function $cost()$ of Definition 5.11 should return the costs of the assignment — the construction time and the dollar amount. We can represent this as a list where first element is the time and second is the amount: $cost(\omega, \xi) = cost_of(\xi(\omega)) = [V_1, V_2]$

The following constraint can be used to state that total time should not exceed c_1 and budget should not exceed c_2: $value_constraint([V_1, V_2]) \equiv V_1 < c_1,\ V_2 < c_2$

The functions $op_{|}$ and $op_{\otimes}$ define how the cost of assignment is aggregated. For instance, for parallel executions, maximum of the execution time is used, whereas for dollar costs, payments are added up: $op_{|}([V_1, V_2], [V_1', V_2']) \equiv [V_1 + V_1', max(V_2, V_2')]$

The following set constraint, which could be a part of ζ_{ctrl}, says that the resource sets allocated to parallel branches of a schedule must be disjoint:

$set_constraint_{|}(R_1, R_2) \equiv (R_1 \cap R_2) = \varnothing)$

In order to simplify the notation, in the rest of the paper, we drop the domain subscript $\mathcal{D}$ and specify the constraint definitions in terms of the relations $cost_consraint(\omega, \xi)$ and $ctrl_consraint(\omega, \xi)$

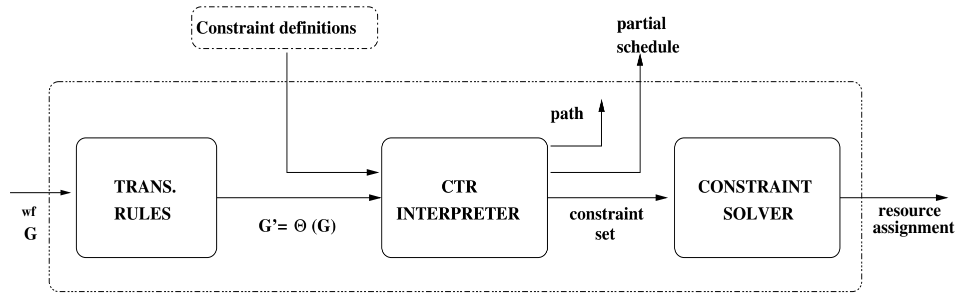


Figure 5.2: The big picture

5.3 CCTR as a Workflow Scheduler

In this section, we present a workflow scheduler that produces schedules conforming to resource allocation constraints. This scheduler accepts a CCTR formula of workflow and resource allocation constraints. Through a series of steps, the system produces a schedule consisting of an execution ordering and a resource assignment that satisfy the constraints. The scheduling process under resource allocation constraints is depicted in Figure 5.2. The process has three main components:

- the transformation rules and templates
- the inference system of CTR
- an off-the-shelf constraint solver

In the consecutive subsections, each of these components are described. A prototype of the system has been implemented by using the CTR interpreter and a constraint solver. The prototype has a graphical interface for the user to define the workflow control and resource allocation constraints. The interface provides facilities to see the intermediate results of the process as well as the final result (i.e., the execution order and final resource assignment). By default, this system returns the first solution, but

	TRANSFORMATION RULES
(1)	$\mathbf{\Theta}(G \wedge C_1 \wedge ... \wedge C_n) \;\equiv\; \mathbf{\Theta}(G) \otimes C_{1_{\mathcal{D}}}(\mathbf{\Theta}(G)) \otimes ... \otimes C_{n_{\mathcal{D}}}(\mathbf{\Theta}(G))$ $C_{i_{\mathcal{D}}}$ is the relation in $\mathcal{D}$ that represent the meaning of $C_i \in \mathcal{C}$. The realization of $\mathcal{D}$ is the constraint template definitions.
(2)	$\mathbf{\Theta}(A) \;\equiv\; A \otimes (resource_asg(A, Agents))$ where A is an atomic task, $resource_asg$ is a resource assignment predicate for task A and $Agents$ is a new variable or a list of variables
(3)	$\mathbf{\Theta}(G_1 \vert G_2) \;\equiv\; (\mathbf{\Theta}(G_1) \vert \mathbf{\Theta}(G_2))$
(4)	$\mathbf{\Theta}(G_1 \otimes G_2) \;\equiv\; (\mathbf{\Theta}(G_1) \otimes \mathbf{\Theta}(G_2))$
(5)	$\mathbf{\Theta}(G_1 \vee G_2) \;\equiv\; \mathbf{\Theta}(G_1)$
(6)	$\mathbf{\Theta}(G_1 \vee G_2) \;\equiv\; \mathbf{\Theta}(G_2)$

Figure 5.3: Transformation rules for a workflow scheduler

all solutions are returned through backtracking.

5.3.1 Transformation Rules and Specifying Constraint System

The transformation process mentioned in Figure 5.3 takes a constrained workflow specification represented as a CCTR formula $G \wedge cost \wedge ctrl$ and produces a CTR formula that does not involve the $\wedge$ connective. The motivation for this step is the resulting CTR formula can be handled using CTR proof theory [13] and evaluated using CTR interpreter such as [26].

The operator $\mathbf{\Theta}$ inserts a predicate in order to add the resource allocation information into the CTR formula. The predicate can be of the form $resource_asg(task, Agents)$,

$step_1$	$\mathbf{\Theta}(((c \otimes r)\|(i\|g)) \wedge cost_constraint \wedge ctrl_constraint)$
$step_2$	$\mathbf{\Theta}((c \otimes r)\|(i\|g)) \otimes$ $cost_constraint(\mathbf{\Theta}((c \otimes r)\|(i\|g))) \otimes ctrl_constraint(\mathbf{\Theta}((c \otimes r)\|(i\|g)))$
$step_3$	$(\mathbf{\Theta}(c \otimes r)\|\mathbf{\Theta}(i\|g))$ $\otimes\ cost_constraint(\mathbf{\Theta}((c \otimes r)\|(i\|g))) \otimes ctrl_constraint(\mathbf{\Theta}((c \otimes r)\|(i\|g)))$
$step_4$	$((\mathbf{\Theta}(c) \otimes \mathbf{\Theta}(r)) \mid (\mathbf{\Theta}(i)\|\mathbf{\Theta}(g)))$ $\otimes\ cost_constraint(\mathbf{\Theta}((c \otimes r)\|(i\|g))) \otimes ctrl_constraint(\mathbf{\Theta}((c \otimes r)\|(i\|g)))$
$step_5$	$(((c \otimes resource_asg(c, W)) \otimes (r \otimes resource_asg(r, X)))\|$ $((i \otimes resource_asg(i, Y)) \mid (g \otimes resource_asg(g, Z))))$ $\otimes\ cost_constraint(((c \otimes resource_asg(c, W)) \otimes (r \otimes resource_asg(r, X)))\|$ $((i \otimes resource_asg(i, Y)) \mid (g \otimes resource_asg(g, Z))))$ $\otimes\ ctrl_constraint(((c \otimes resource_asg(c, W)) \otimes (r \otimes resource_asg(r, X)))\|$ $((i \otimes resource_asg(i, Y)) \mid (g \otimes resource_asg(g, Z))))$

Figure 5.4: Transformation for house construction workflow

where *task* is a constant that represents the task to which resources are assigned and *Agents* is a new variable or a list of new variables. In addition to this, constraint predicates, C_i, are replaced with classical predicates $C_{i_{\mathcal{D}}}$ that are the realizations of the relations in $\mathcal{D}$.

Example 5.17 Consider a subset of Example 1.1: $(carpentry \otimes roof) \mid (installations \mid gardening)$

The scheduling transformation of this subworkflow is shown in Figure 5.4, where the long task names are abbreviated to c, r, i, and g, respectively. The transformed workflow is given in $step_5$.

(1) $ctrl_constraint(G \otimes cost_constraint(X)) : -ctrl_constraint(G)$

(2) $ctrl_constraint(G \otimes ctrl_constraint(X)) : -ctrl_constraint(G)$

(3) $ctrl_constraint((G_1 \mid G_2)) \ : - \ \mathbf{set_constraint}_{|}(G_1, G_2),$

$ctrl_constraint(G_1), ctrl_constraint(G_2)$

(4) $ctrl_constraint((G_1 \otimes G_2)) \ : - \ \mathbf{set_constraint}_{\otimes}(G_1, G_2),$

$ctrl_constraint(G_1), ctrl_constraint(G_2)$

(5) $ctrl_constraint(G) \ : - \ task(G), \mathbf{task_constraint}(G)$

(6) $cost_constraint(G) \ : - \ cost(G, V), \mathbf{value_constraint}(V)$

(7) $cost((G \otimes cost_constraint(X)) \ : - \ cost(G)$

(8) $cost((G \otimes ctrl_constraint(X)) \ : - \ cost(G)$

(9) $cost((G_1 \otimes G_2), V) \ : - \ cost(G_1, V_1), cost(g_2, V_2), \mathbf{op}_{\otimes}(V_1, V_2, V)$

(10) $cost((G_1 \mid G_2), V) \ : - \ cost(G_1, V_1), cost(G_2, V_2), \mathbf{op}_{|}(V_1, V_2, V)$

(11) $cost(A \otimes resource_asg(A, Agents), V) \ : - \ \mathbf{cost_of}(resource_asg(A, Agents), V)$

Figure 5.5: Template rules for constraint systems

In order to specify the resource allocation constraint definitions in CCTR Wf-constraint universe, we introduce rule templates, which the user can instantiate in order to describe the constraint system appropriate for the application at hand. Details of the Wf-constraint universe of CCTR introduced in Section 5.2 can vary greatly, but their general properties can be realized as a single set of Prolog rule *templates* shown in Figure 5.5. In the figure, the boldface predicates are *placeholders* for functions and constraints that the user can specify to adapt the template to a particular application domain. These placeholders are explained in Figure 5.6. Later we illustrate the use of these templates on a number of nontrivial examples.

	Resource Assignment:
	$resource_asg(T, Agents)$ – placeholder for a user-specified term, which associates resources to a task. T denotes an atomic task that can be represented by a single variable, and $Agents$ denotes the resource that can be represented by a single variable or a list of variables (in case of multiple resources).

	User Predicates (typically defined via user-supplied rules):
(1)	**cost_of**($resource_asg(T, Agents), V$) – placeholder for predicate that tells the costs associated with the resources. V has the data type of the costs attribute of the resource. It can be a single variable or a list of variables.
(2)	$\mathbf{set_constraint}_{\mid}(G_1, G_2)$ – placeholder for a control constraint for sequential composition of tasks.
(3)	$\mathbf{set_constraint}_{\otimes}(G_1, G_2)$ – placeholder for a control constraint for parallel composition of tasks.
(4)	**task_constraint**(G) – placeholder for a constraint on individual tasks.
(5)	**value_constraint**(V) – placeholder for a predicate used to define cost constraints. V has a user-defined data type.
(6)	$\mathbf{op}_{\mid}(V_1, V_2, V)$ – placeholder for aggregate operator that tells how to compute the the cost (V) of a parallel composition of subworkflows from the costs (V_1, V_2) of those subworkflows. Used in the definition of cost constraints. V, V_1, and V_2 must the same user-defined data type.
(7)	$\mathbf{op}_{\otimes}(V_1, V_2, V)$ – similar to $\mathbf{op}_{\mid}$, but used for serial compositions of subworkflows.

Figure 5.6: Placeholders for problem-specific predicates and resource assignments

Let the placeholder $resource_asg(T, Agents)$ be of the form $rsrc(T, A)$, where T represents task and A the agent

$\mathbf{cost_of}(rsrc(T, A), [V, U])\ :-\ duration(T, A, V),\ price(T, A, U)$

$\mathbf{value_constraint}([V, U])\ :-\ V < c_1,\ U < c_2$

$\mathbf{set_constraint}_{|}(G_1, G_2)\ :-\ disjoint(G_1, G_2)$

$\mathbf{set_constraint}_{\otimes}(G_1, G_2)\ :-\ true$

$\mathbf{task_constraint}(G)\ :-\ true$

$\mathbf{op}_{|}([V_1, U_1], [V_2, U_2], [V, U])\ :-\ V \text{ is } max(V_1, V_2),\ U \text{ is } U_1 + U_2$

$\mathbf{op}_{\otimes}([V_1, U_1], [V_2, U_2], [V, U])\ :-\ V \text{ is } V_1 + V_2,\ U \text{ is } U_1 + U_2$

Figure 5.7: Placeholders for house construction example

Rules 1 to 3 in Figure 5.5 define *ctrl_constraint* — the control constraint for |-branches, ⊗-branches, and tasks of the workflow formula, respectively. Rules 4 to 7 define *cost_constraint* — the cost constraint. Again, this is done separately for each branch type of the workflow formula.

In our prototype, the user can select previous constraint definitions or to create a new constraint definition. For new definitions, the system opens a template with only rules f Figure 5.5 and some guidelines about user-defined rules, which are going to be defined by the user.

Example 5.18 The placeholder definitions for the constraints of Example 1.1 are shown in Figure 5.7.

5.3.2 CTR Interpreter

The transformation module produces a CTR goal that includes constraint predicates. In our prototype, we used CTR interpreter, which is the realization of CTR proof

theory in [13].

In Section 5.1, it was stated that the CTR goals are satisfied along m-paths. However, the inference system produces a solution for a complete system that does not interleave with other executions, therefore it produces a path satisfying the given CTR goal. The following example presents an informal overview of the CTR inference mechanism.

Example 5.19 Suppose CTR goal is $((a.ins \otimes b.ins) \mid (c.ins \otimes d.ins)) \otimes e.ins$, where each predicate inserts an atom into the database (e.g., $a.ins$ inserts propositional atom a.) Figure 5.8 presents one of the several possible executions for this transaction. The inference system manipulates expressions of the form $P, D - -- \vdash \varphi$, called *sequents*. Each sequent in the table of Figure 5.8 is derived from the one below by an inference rule, reaching an axiom at the bottom. When carried out top-down, the deduction corresponds to execution sequence (i.e., a *schedule*) for the predicates of the given transaction. The execution sequence for the deduction in Figure 5.8 is $c.ins, a.ins, b.ins, d.ins, e.ins$.

Sequent
$P, \{\} - -- \vdash ((a.ins \otimes b.ins) \mid (c.ins \otimes d.ins)) \otimes e.ins$
$P, \{c\} - -- \vdash ((a.ins \otimes b.ins) \mid (d.ins)) \otimes e.ins$
$P, \{c, a\} - -- \vdash (b.ins \mid d.ins) \otimes e.ins$
$P, \{c, a, b\} - -- \vdash d.ins \otimes e.ins$
$P, \{c, a, b, d\} - -- \vdash e.ins$
$P, \{c, a, b, d, e\} - -- \vdash \{\}$

Figure 5.8: CTR deduction for Example 5.19

cost constraints	V_3 is $V_1 + V_2$, U_3 is $U_1 + U_2, V_6$ is $max(V_4, V_5)$, U_6 is $U_4 + U_5$, V is $max(V_3, V_6)$, U is $U_3 + U_6, V < c_1$, $U < c_2$
ctrl const.	$Y \neq Z$, $W \neq Y$, $W \neq Z, X \neq Y$, $X \neq Z$

Figure 5.9: Constraints computed for house construction workflow

CTR interpreter produces an execution sequence for the tasks of the given workflow, as explained in Example 5.19. In Section 5.1.2.2, it is explained that while m-paths are adequate to model serial and concurrent execution in CTR, they are not sufficient to model resource requirements necessary for those executions to succeed. In addition to the path, CTR interpreter instantiates the resource allocation term. Since it captures the structure of the partial schedule and it carries the resource assignment information, this term is the one that is actually used for constraint solving process. On the other hand, path gives us a valid serialization of the execution.

Constraint predicates *cost_constraint* and *ctrl_constraint* of the transformed goal include constraints that are going to be solved by a constraint solver. CTR interpreter does not have the constraint solving capability by itself. Therefore, it finds a schedule for the constraint-free part of the goal and sends the constraints to a solver. However, in order to facilitate the constraint solving process, instead of sending the *cost_constraint* and *ctrl_constraint* predicates to the solver, it collects and stores the atomic constraints in the definition of constraint system. Then, sends this set of atomic constraints to the solver. Figure 5.9 shows the set of atomic constraints for house construction workflow.

5.3.3 Constraint Solver

The area that deals with defining and solving constraint problems is *constraint programming* and it provides framework for both defining and solving problems and it is

effectively used in many areas such as production planning or scheduling [10, 16, 81]. Constraint programming uses algorithms from mathematics, artificial intelligence and operations research and provides constraint solvers that implement these algorithms. Constraint Logic Programming (*CLP*) [50, 40] is a branch of constraint programming in which logic is used as the declarative modeling language for the constraints. CLP supports several constraint domains and their corresponding solvers are able to model different applications and problems.

In our system, in order to solve the constraint set, which is accumulated by CTR interpreter module, a constraint solver is used. The solution to the constraint set is a valid resource assignment for the schedule obtained. Since our system provides a logic-based framework for the representation of workflow and constraints, a CLP solver is the natural choice as the constraint solver. Our implementation uses the constraint solver provided by XSB,[3] since the CTR interpreter is realized as an XSB application. However, any off-the-shelf constraint solver that is compatible with the defined constraints can be incorporated into the system. The set of candidate environments s very rich. We can list some of them as SICStus Prolog [71], CHIP [22], clp-fd [23], Oz [69].

5.3.4 Correctness of the Scheduler

In the scheduling mechanism we have explained, the transformer module takes any CCTR goal ϕ and returns a new conjunction-free CTR goal ϕ'. Then the scheduler module finds an execution $D_0 ... D_n$, $P, D_0 ... D_n \models \phi'$ and a resource assignment ξ on ϕ' such that $P, D_0 ... D_n, \xi \models \phi$. Mechanisms that satisfy this property are called *correct*

[3] XSB is a high-performance deductive database and Prolog system available at `http://xsb.sourceforge.net/`

scheduler.

Before the correctness of the scheduler presented, the following lemma tells about the isomorphism of constraint system and constraint template.

Lemma 5.20 *The constraint template in Figure 5.5 defines a wf-constraint universe in the sense of Section 5.2, provided that the actual predicates that replace the boldface placeholders have the appropriate associativity and commutativity properties stated in that section.*

Proof. It is given in Appendix A.1.

Let ζ be a constraint system and $\mathcal{D}$ be a constraint domain, which can be represented using the template rules in Figure 5.5 (plus the additional definitions for the boldface placeholders). Then, the transformation $\boldsymbol{\Theta}$ in Figure 5.3 and the system explained constitute a correct scheduler. This is formally given in the following theorem.

Theorem 5.21 *The presented system is a correct scheduler, i.e., let P be a transaction base, $D_0, ..., D_n$ be a sequence of states, ξ be a resource assignment, ϕ be a CCTR goal and ϕ' be CTR goal. Then, $P, D_0...D_n, \xi \models \phi$ iff*

$\Theta(\phi) = \phi'$

$P, D_0...D_n \models \phi'$ *and*

ξ *is obtained from* ϕ'

Proof. It is given in Appendix A.2.

In addition to the correctness of the scheduler, we find it worthwhile to state another property of the scheduler.

Theorem 5.22 *For non-disjunctive workflow goals, this scheduler finds the solution without backtracking on the execution order, i.e., there is a resource assignment for all possible execution orders (paths) iff there is a solution for one particular execution order.*

Proof. The different valid executions order for non-disjunctive goals differ only in the order of parallel tasks. The set of scheduled tasks is always the same. If a resource r is allocated for some task along one valid execution order, when we allocate the same resource for the same task for another valid execution, since the structure of the goal does not change, the constraint definitions will produce the same result as in the previous execution order and thus is still a valid resource allocation schema. □

CHAPTER 6

Discussions and Applications

In this chapter, we present a discussion on the comparison of the approaches presented and some of the possible applications.

6.1 Comparison of the Approaches

The common point of the two presented approaches is the use of constraint logic programming. Constraint logic programming has been successfully used for several resource allocation problems. Therefore, we chose to adopt it for the problem of scheduling under resource allocation constraints. However, there are several basic differences about the way CLP is used in the proposed approaches and about the structure and the main aim of the approaches.

The first approach demonstrates the applicability of constraint logic programming for workflow scheduling problems. The proposed architecture can be used together with any constraint solver. In addition to this, the architecture proposes the use of a graphical user interface for workflow modeling. Therefore, the system facilitates the

modeling step for the end-user.

In the first approach, constraint logic programming is the core of almost the whole architecture. However, the goal of the second approach is the development of a formalism that can define the semantics of resource allocation constraints, validation of schedules under these constraints and finding such schedules. The formalism involves constraint logic programming only for determining the correct resource allocation. Although this system can be extended with a graphical interface to facilitate workflow modeling step, the modeling of resource allocation constraints may need little experience on the system.

The approaches have some differences in the workflow structures as well. The iteration and execution of at least one of the alternative blocks can be modeled in the first approach. Although, currently not supported, CCTR can be extended to model these features.

6.2 Applications

Although our main target domain is workflows, we have successfully applied our frameworks to some other domains as well.

6.2.1 Workflows

We have already presented the modeling and scheduling of the house workflow of Example 1.1 by using both of the approaches. In this section, we present three more examples that demonstrate workflow scheduling under resource allocation constraints for different applications. The definitions for the examples are given in Appendix B.

6.2.1.1 Travel Agency Workflow

The resources of the workflow applications can vary greatly. In this example, we model another common workflow example; travel arrangement to a given destination. This workflow is composed of serial composition of a **f**light booking task, a **h**otel reservation task and a **c**ar rental task, in the given order. The resource allocation constraints are total cost constraints $cost < \$1000$, $duration < 7days$ and $quality > 2$ (ranked over 5). Therefore, the scheduler must check the duration of the flight plans and comfort and quality ratings for the hotels and transportation, in order to develop a valid travel schedule. Although the workflow and resource allocation constraint specifications are quite similar to that of house construction example, this time the nature of the resources are different. In this workflow, airways, hotels and car rental companies are the resources for flight booking, hotel reservation and car rental tasks, respectively. As a result of this resource modeling, scheduler produces a resource assignment showing which airway, hotel and car rental company to choose for the travel. The definitions for this example in both of the frameworks are given in Appendix B.

6.2.1.2 Telecommunication Service Providing Workflow

The resource allocation constraints may be affective on the selection of the tasks in addition to selection of the resources. This is possible only in workflow goals including disjunctive subgoals. In this example, we have a telecommunication company that provides line installation service. Line installation process is composed of several subprocess, each subprocess can be handled by one of the several alternative tasks. The control flow graph for this workflow is given in Figure 6.1.

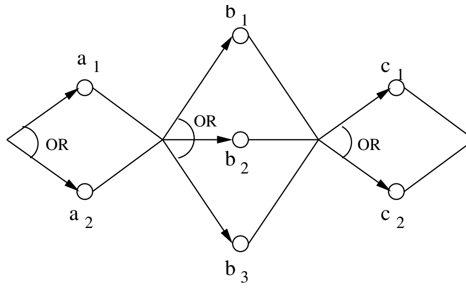


Figure 6.1: Telecommunications workflow control flow graph

The resource allocation constraints on this workflow is a given limit on the total duration and the requirement that the parallel tasks must be handled by different resources. Due to resource allocation constraints, some tasks may be eliminated by the scheduler. For example, if task a_1 can be done only by resource r_1 and similarly, r_1 is the only resource that can do task c_1, due to the control constraint it is impossible to schedule both a_1 and c_1 in the same execution sequence. The details about the specification and processing of the constraints are given in Appendix B.

6.2.1.3 Conference Planning

In a conference planning task, the conference organization committee decides on placement of the sessions to the proper time slots. Sometimes, this task can be unexpectedly complex due to the constraints resulting from the situations such as the attendance of the same presenter to several sessions or the content dependence among the sessions.

In this example, we model a workshop with four time slots having three parallel sessions at each time slot, as a workflow having four sequential subworkflows, each of which has three concurrent tasks. In this example, sessions are the resources to be allocated for the tasks. Therefore, the conference workflow is represented as given in

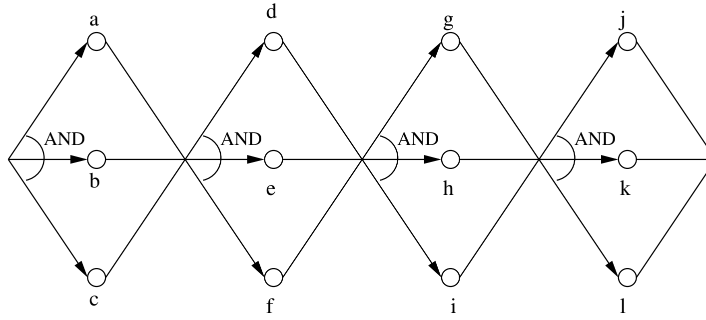


Figure 6.2: Conference planning workflow control flow graph

Figure 6.2. Each resource is allocated at least and at most one task and there are additional constraints such as:

session 4 must take place before session 11.

session 1 must not take place at the same time as session 2.

Since sessions are considered to be the resources for the workflow, the above constraints restrict the allocation ordering for the resources. In this workflow, we have only control constraints. Therefore, cost constraint definitions are omitted. In addition to the *resource disjointness* constraint, two new types of constraints, *resource* r_1 *must be allocated before* r_2 and *resource* r_1 *can not work in parallel with resource* r_2, appear in the control constraint definitions. The details of the constraint definitions are given in Appendix B.

6.2.2 Composite Web Services

Internet has grown to be more than a media containing a huge amount of information. In addition to information, some services such as reservation, translation, buying/selling etc. are also provided through this media. Such services that can be invoked through

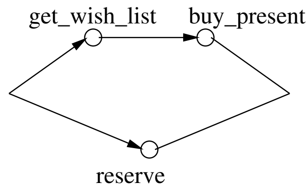


Figure 6.3: Composite web service control flow graph

Internet is generally called as *web services* [28]. Currently, web services are based on human interaction. However, the goal is automatic invocation of these services, and building complex services that are actually the composition of individual services. For this reason, there is a great effort on defining machine-readable representation of the services, semantic issues of these representations in order to find and invoke web services automatically and to build composite web services [6, 61, 21, 58, 27, 60, 64, 33].

Consider the following request from an Internet user: "For our wedding anniversary, I want to buy a present for my spouse and make a dinner reservation". In order to get the "wish list" of the spouse, certain sites are to be visited, then an item is picked and a reservation is made. However, the user may want to have certain cost constraints such as "I don't want to spend more than a certain amount of money but I want to have dinner at a restaurant with quality above average". Ordinarily, the user has to visit each of the sites and make a decision on the present and restaurant so that the constraints are satisfied. Due to the nature of the constraint, it is hard to eliminate the possibilities directly, it is necessary to check the overall cost. The mechanism proposed to build composite web services do not provide a solution for such constraints.

The presented frameworks may be used as a tool to build a system that can select web services and model a composite web service under given resource allocation con-

straints. The immediate dependencies for this composite service can be represented as in Figure 6.3. In a composite service building system, the web services are the resources that will fulfill the user's demand and our frameworks can find the proper resource assignments for workflow tasks. Among the set of available web services, the individual web services that will take part in the composite service are chosen so that the composite service satisfies user's constraints. The details of the definitions are presented in Appendix B.4.

If such a system is built using the first approach, WSL is used as the composite service modeling language. In order to search and invoke service, WSL representation can be translated into XML later on. In the second approach, CCTR may serve as the modeling language. As stated above, web services are the resources of the system. In workflow examples, the set of candidate resources were pre-defined and limited. For composite services, in order to build a set of candidate web services, a search module either searches Internet for available and capable services (e.g., restaurants in the given city), or looks for such services from a directory like *UDDI*. The candidate web service set can be further pruned by using some of the resource allocation constraints (e.g., search for restaurants in the city with $quality > 3$). Our framework chooses the proper web services and the selected services constitute the composite service that meets user's constraints.

6.2.3 Workflows in Non-cooperative Environments

In most of the workflow modeling and scheduling research, it is assumed that the workflow is composed of *purely* cooperative tasks. However, sometimes, external tasks

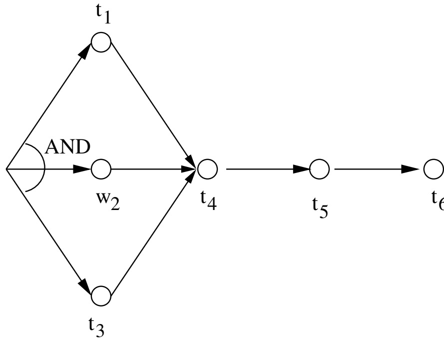


Figure 6.4: Control flow graph for workflow with non-cooperative task

that the system has little or no control over, or the tasks that works adversely take part in the workflow. In order to find assignment satisfying the resource allocation constraints, the scheduler must take the constraints and assignments of external tasks into consideration. Workflows operating in *open* environment could involve in such cases. One simple scenario is as follows: A company has several departments and all departments share the printers in the building. Each department has its own workflow, however departments are in constant interaction. The production department needs the number of sales on certain products and within its workflow, it calls services from sales department (possibly from the other departments as well for other tasks). However, the requested external service has the constraint to use printers with highest printing quality but that is slightly slower than the others. On the other hand, the production workflow has total duration constraint.

Assume that the tasks of the production workflow for a given item are "getting the number of products in the inventory" (t_1), "getting the latest sale numbers" (w_2), "downloading production schema for the item" (t_3), "producing parts" (t_4), "combining them" (t_5) and sending to inventory (t_6). The control flow graph for this workflow is

shown in Figure 6.4. Note that the external service "getting the latest sale numbers" (w_2) may be a complex service including several tasks. Therefore, this subflow must be considered together with it own resource allocation constraints. We can represent this situation as a CCTR formula as follows:

$$w = (t_1 \mid w_2 \mid t_3) \otimes t_4 \otimes t_5 \otimes t_6.$$

Including department's constraint, the workflow goal becomes

$$w' = w \wedge c = ((t_1 \mid w_2 \mid t_3) \otimes t_4 \otimes t_5 \otimes t_6) \wedge c.$$

However, we have to consider external service's constraints and assignments. Therefore, external service w_2 must be extended with sub-constraints, to $w_2' = w_2 \wedge c_2$. Therefore, the overall workflow becomes

$$w' = w \wedge c = ((t_1 \mid w_2' \mid t_3) \otimes t_4 \otimes t_5 \otimes t_6) \wedge c.$$

Although we have shown the CCTR specification to present the workflow structure more clearly, it is also possible to model and solve this workflow by using the first approach. This picture can be extended with different other resource allocation constraints for the external service and with other tasks having partially or fully adverse to the constraints of the overall workflow.

CHAPTER 7

Conclusion

In this thesis, resource allocation constraints are defined as a new scheduling criteria for workflows. This idea has been motivated by the necessity of resource distribution control for tasks according to the task execution ordering and lack of mechanism for such controls. Current approaches for scheduling tasks in a workflow provide no mechanism to reason about the relative costs of schedules. Scheduling under resource allocation constraints provide decisions on the assignment of resources to tasks, as well as correct execution sequence of tasks. Hence, more efficient workflow schedules can be obtained for business environments. To accomplish this, we have studied on two approaches for workflow scheduling under resource allocation constraints.

The first approach develops an architecture to model and schedule workflows with resource allocation constraints as well as with the traditional temporal/causality constraints. We use constraint programming to schedule workflows with resource allocation constraints. Workflow specification together with resource information and constraints, including both resource allocation and temporal/causality constraints, are translated to

finite-domain constraints. For the implementation, constraint programming language *Oz* is used. Oz is a multi-paradigm programming language that has logic programming and concurrent programming features as well as constraint specification and solving capabilities.

The main contribution of this approach is the proposed architecture which provides a specification language that can model resource information and resource allocation constraints, and a scheduler model that incorporates a constraint solver in order to find proper resource assignments. Contrary to the classical constraint programming problems that seeks the optimal solution, our approach finds a feasible solution satisfying the constraints.

In the second approach, a logical framework based on Concurrent Constraint Transaction Logic (CCTR), for scheduling workflows under resource allocation constraints has been presented. We developed CCTR language, which integrates Concurrent Transaction Logic [13] with Constraint Logic Programming [49, 50], in order to provide a basis that can express the syntax and semantics for both workflow and the resource allocation constraint. We developed our framework on the basis of this formal modeling. In the first step of the framework, by using the transformation algorithm, initial workflow specification and a set of resource allocation constraints represented in CCTR are transformed into a new workflow in CTR, such that every execution of that workflow is guaranteed to satisfy the constraints. In the next step, CTR's inference engine is used to determine the schedule of the initial workflow. In the final step, constraints are also extracted and solved by the off-the-shelf constraint solver to determine the resource assignments for the tasks. A prototype of the system was developed by using the CTR

interpreter available at `www.cs.toronto.edu/~bonner/ctr/index.html`. We present correctness of this system. In addition to the correctness, we state one more property of this scheduler on efficiency issue. For non-disjunctive goals, the scheduler does not need to backtrack on the partial schedule.

As a future work, both of the frameworks can be extended with special-purpose constraint solvers that are optimized for our frameworks. In addition, scheduling under resource allocation constraints for dynamic workflows and scheduling concurrent workflow instances under resource allocation constraints are also interesting topics for future work. In this work, we have not considered soft resource allocation constraints and preferences on resource allocation constraints. Workflow scheduling under such constraints and preferences can be another future work subject.

REFERENCES

[1] N. R. Adam, V. Atluri, and W.-K. Huang. Modeling and analysis of workflows using Petri Nets. In *Journal of Intelligent Information Systems 10(2)*, 1998.

[2] A. Aiba and R. Hasegawa. Constraint logic programming systems - CAL,GDCC and their constraint solvers. In *In Proceedings of the International Conference on Fifth Generation Computing Systems (FGCS)*, pages 113–131, 1992.

[3] A. Aiba, K. Sakai, Y. Sato, D. J. Hawley, and R. Hasegawa. Constraint logic programming language CAL. In *In Proceedings of the International Conference on Fifth Generation Computing Systems (FGCS)*, pages 263–276, Tokyo, 1988.

[4] G. Alonso, D. Agrawal, A. El Abbadi, M. Kamath, R. Günthör, and C. Mohan. Advanced transaction models in workflow contexts. In *International Conference on Data Engineering*, New Orleans, Louisiana, February 1996.

[5] G Alonso, D. Agrawal, A. El Abbadi, and C. Mohan. Functionality and limitations of current workflow management systems. In *IEEE-Expert. Special issue on Cooperative Information Systems*, 1997.

[6] A. Ankolenkar, M. Burstein, J.R. Hobbs, O. Lassila, S.A. McIlraith D.L. Martin, S. Narayanan, M. Paolucci, T. Payne, K. Sycara, and H. Zeng. Daml-s: A semantic markup language for web services. In *Intl. Semantic Web SYmposium (SWWS)*, July 2001. http://www.semanticweb.org/SWWS/program/full/paper2.pdf.

[7] P. Attie, M.P. Singh, A.P. Sheth, and M. Rusinkiewicz. Specifying and enforcing intertask dependencies. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, 1993.

[8] BNR-prolog user guide. Technical report, Bell-Northern Research Ltd., 1988.

[9] E. Bertino, E. Ferrari, and V. Atluri. The specification and enforcement of authorization constraints in workflow management systems. *ACM Transactions on Information and System Security*, 2(1):65–104, 1999.

[10] J. Blazewicz, K. H. Ecker, E. Pesch, G. Schmidt, and J. Weglarz. *Scheduling Computer and Manufacturing Processes*. Springer-Verlag, 1996.

[11] A.J. Bonner. Workflow, transactions, and datalog. In *ACM Symposium on Principles of Database Systems*, Philadelphia, PA, May/June 1999.

[12] A.J. Bonner and M. Kifer. Transaction logic programming. In *Intl. Conference on Logic Programming*, pages 257–282, Budapest, Hungary, June 1993. MIT Press.

[13] A.J. Bonner and M. Kifer. Concurrency and communication in transaction logic. In *Joint Intl. Conference and Symposium on Logic Programming*, pages 142–156, Bonn, Germany, September 1996. MIT Press.

[14] P. Van Breek and X. Chen. CPlan: A constraint programming approach to planning. In *Sixteenth National Conference on Artificial Intelligence (AAAI-99)*, pages 585–590, 1999.

[15] M. Brenner. A formal model for planning with time and resources in concurrent domains. In *IJCAI Workshop on Planning with Resources*, Seattle, USA, August 2001.

[16] P. Brucker. *Scheduling Algorithms*. Springer-Verlag, 1995.

[17] P. K. C. and K. Ramaritham. ACTA: The SAGA continues. In *Transaction Models for Advanced Database Applications*. Morgan Kaufman, 1992.

[18] Y. Caseau and F. Laburthe. Improved CLP scheduling with task intervals. In *Intl. Conference on Logic Programming*, 1994.

[19] S. Ceri, P. W. P. J. Grefen, and G. Sanchez. WIDE: A distributed architecture for workflow management. In *RIDE*, 1997.

[20] A. Cheng, J. Esparza, and J. Palsberg. omplexity results for 1-safe nets. In R. K. Shyamasundar, editor, *Foundations of Software Technology and Theoretical Computer Science*, pages 326–337. Springer-Verlag, Berlin, 1993. Volume 761 of Lecture Notes in Computer Science.

[21] R. Chinnici, M. Gudgin, J.-J. Moreau, and S. Weerawarana. Web services description language (wsdl) version 1.2. Technical report, W3C, July 2002.

[22] CHIP- Constraint Handling In Prolog. http://www.cosytec.com/production_scheduling/chip/chip_technology.htm.

[23] clp(FD,s) language. http://contraintes.inria.fr/ georget/software/clp_fds/clp_fds.html, 2002.

[24] Workflow Management Coalition. Terminology and glossary ver 3.0. Technical Report (WFMC-TC-1011), Workflow Management Coalition, Brussels, February 1999.

[25] A. Colmerauer. An introduction to prolog-III. *Communications of the ACM*, 33(7):69–90, July 1990.

[26] Ctr interpreter. www.cs.toronto.edu/∼bonner/ctr/index.html.

[27] F. Curbera, Y. Goland, J. Klein, F. Leymann, D. Roller, S. Thatte, and S. Weerawarana. Business process execution language for web services, version 1.0. Technical report, IBM, July 2002.

[28] F. Curbera, W. Nagy, and S. Weerawarana. Web services: Why and how. In *OOPSLA 2001 Workshop on Object-Oriented Web Services*. ACM, 2001.

[29] H. Davulcu, M. Kifer, C.R. Ramakrishnan, and I.V. Ramakrishnan. Logic based modeling and analysis of workflows. In *ACM Symposium on Principles of Database Systems*, pages 25–33, Seattle, Washington, June 1998.

[30] U. Dayal, M. Hsu, and R. Ladin. Organizing long-running activities with triggers and transactions. In *ACM SIGMOD Conference on Management of Data*, 1990.

[31] U. Dayal, M. Hsu, and R. Ladin. Organizing long running activities with triggers and transactions. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, 1990.

[32] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berhier. The constraint logic programming language CHIP. In *In Proceedings of the International Conference on Fifth Generation Computing Systems (FGCS)*, Tokyo, 1988.

[33] A. Dogac, I. Cingil, G.B. Laleci, and Y. Kabak. Improving the functionality of UDDI registries through web service semantics. In *3rd VLDB Workshop on Technologies for E-Services (TES-02)*, Hong Kong, China, August 2002.

[34] W. Du, j. Davis, Y. Huang, and M. Shan. Enterprise workflow resource management. In *Int'l Workshop on Research Issues in Data Engineering*, pages 108–115, Sydney, Australia, 1999.

[35] J. Eder, E. Panagos, H. Pezewaunig, and M. Rabinovich. Time management in workflow systems. In *Int. Conf. on Business Infomation Systems*, pages 265–280, Poznan, Poland, 1999.

[36] J. Eder, E. Panagos, and M. Rabinovich. Time constraints in workflow systems. In *Conference on Advanced Information Systems Engineering*, pages 286–300, Germany, 1999.

[37] E. A. Emerson. Temporal and modal logic. In *Handbook of Theoretical Computer Science, vol B*, 1990.

[38] J. Esparza and M. Nielsen. Decidability issues for Petr-nets. Technical Report BRICS RS-94-8, BRICS, Department of Computer Science, University of Aarhus, 1994.

[39] T. Fruhwirth et al. Constraint logic programming - an informal introduction. Technical Report ECRC-93-5, Europian Computer-Industry Research Center, 1993.

[40] F. Fages. Constraint logic programming. In *Europian Summer School in Logic, Language and Information*, 2001.

[41] D. Georgakopoulos, M. Hornick, and A. Sheth. An overview of workflow management: From process modeling to infrastructure for automation. *Journal on Distributed and Parallel Database Systems*, 3(2):119–153, April 1995.

[42] H. Goltz and U. John. Methods for solving practical problems of job-shop scheduling modelled in clp(fd). In *Conf. on Practical Application of Constraint Technology*, London, April 1996.

[43] P. Halsum and H. Geffner. Heuristic planning with time and resource. In *IJCAI Workshop on Planning with Resources*, Seattle, USA, August 2001.

[44] M. Hannebauer. From formal workflow models to intelligent agents. In *AAAI-99 Workshop on Agent Based Systems in the Business Context*, pages 19–24, Orlando, USA, 1999.

[45] H. Hong. *Improvements in CAD-based Quantifier Elimination*. PhD thesis, Ohio State University and Information Science Research Center, 1990.

[46] H. Hong. Non-linear constraint solving over real numbers in constraint logic programming (introducing RISC-CLP. Technical report, RISC, Linz, 1992.

[47] Special issue on workflow systems. *Bulletin of the Technical Committee on Data Engineering (IEEE Computer Society)*, 18(1), March 1995.

[48] Y. Huang and M. Shan. Policies in a resource manager of workflow systems: Modeling, enforcement and management. In *International Conference on Data Engineering*, 1999.

[49] J. Jaffar and J.-L. Lassez. Constraint logic programming. In *ACM Symposium on Principles of Programming Languages*, Munich, W. Germany, 1987.

[50] J. Jaffar and M. J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19/20, May/July 1994.

[51] J. Jaffar, S. Michaylov, P. Stuckey, and R. Yap. The CLP(($\mathcal{R}$) language and system. Technical report, IBM Research Division, 1990.

[52] N.R. Jennings, P. Faratin, M.J. Johnson, and P. O'Brien ad M.E. Wiegand. Using intelligent agents to manage business processes. In *Int'l Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology*, pages 345–360, London, 1996.

[53] N.R. Jennings, T.J. Norman, and P. Faratin. Adept: An agent-based approach to business process management. *ACM SIGMOD Record*, 27(4):32–39, 1998.

[54] N.R. Jennings, T.J. Norman, P. Faratin, P. O'Brien, and B. Odgers. Auotonomous agents for business process management. *Int'l Journal of Applied Artificial Intelligence*, 14(2):145–189, 2000.

[55] P. Karagoz. Adding constraint to logic-based workflows to obtain optimized schedules. In *In Proceedings of the Extended Database Technology PhD Workshop*, Konstanz, Germany, March 2000.

[56] J. Klein. Advanced rule-driven transaction management. In *IEEE COMPCON*. IEEE, 1991.

[57] N. Krishnakumar and A. P. Sheth. Managing heterogeneous multi-system tasks to support enterprise-wide operations. *Distributed and Parallel Databases*, 3(2):155–186, 1995.

[58] F. Leymann. Web services flow language (wsfl 1.0). Technical report, IBM, May 2001. http://www-4.ibm.com/software/solutions/webservices/pdf/WSFL.pdf.

[59] A. K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977.

[60] S. McIlraith and T.C. Son. Adapting Golog for programming the semantic web. In *Proc. 5th Symposium on Logical Foundations of Commensense Reasoning*, 2001.

[61] B. McKee, D. Ehnbuske, and D. Rogers. UDDI Version 2.0 API Specification. Technical report, UDDI.org, June 2001. http://www.uddi.org/.

[62] E. Monfroy. Non linear constraints: A language and a solver. Technical report, ECRC, Munich, Germany, 1992.

[63] K.L. Myers and P. M. Berry. At the boundary of workflow and ai. In *AAAI-99 Workshop on Agent Based Systems in the Business Context*, Orlando, USA, 1999.

[64] S. Narayanan and S. McIlraith. Simulation, verification and automated composition of web services. In *Proc. of 11th Internation Conference on World Wide Web*, pages 77–88, 2002.

[65] A. Nareyek. Applying local search to structural constraint satisfaction. In *IJCAI Workshop on Intelligent Workflow and Process Management: The New Frontier for AI in Business*, Stockholm, Sweden, August 1999.

[66] A. Nareyek. AI planning in a constraint programming framework. In G. Hommel, editor, *Communication-Based Systems*, pages 163–178. Kluwer Academic Publishers, 2000.

[67] T. Le Provost and M. Wallace. Domain independent propagation. In *In Proceedings of the International Conference on Fifth Generation Computing Systems (FGCS)*, pages 1004–1012, Tokyo, 1992.

[68] P. Van Roy. Logic programming in Oz with Mozart. In *Intl. Conference on Logic Programming*, pages 38–51, Las Cruces, New Mexico, 1999.

[69] C. Schulte and G. Smolka. Finite domain constraint programming in Oz: A tutorial. Version 1.1.0, February 2000.

[70] P. Senkul, M. Kifer, and I.H. Toroslu. A logical framework for scheduling workflows under resource allocation constraints. In *Intl. Conference on Very Large Data Bases*, August 2002.

[71] SICStus prolog. http://www.sics.se/isl/sicstus.html.

[72] M. P. Singh. Semantical considerations on workflows: An algebra for intertask dependencies. In *Proceedings of the International Workshop on Database Programming Languages*, 1995.

[73] M.P. Singh. Synthesizing distributed constrained events from transactional workflow specifications. In *Proceedings of 12-th IEEE Int'l Conference on Data Engineering*, pages 616–623, New Orleans, LA, February 1996.

[74] M.P. Singh and M. H. Huhns. Multiagent systems for workflow. *Int'l Journal of Intelligent Systems in Accounting, Finance and Management*, 8:105–117, 1999.

[75] G. Trajcevski, C. Baral, and J. Lobo. Formalizing (and reasoning about) the specifications of workflows. In *Proceedings of the International Conference on Cooperative Information Systems (CoopIS)*, September 1996.

[76] W.M.P. van der Aalst. Verification of workflow nets. In *Application and Theory of Petri Nets*, 1997.

[77] W.M.P. van der Aalst. The application of Petri Nets to workflow management. In *The Journal of Circuits, Systems and Computers 1(8)*, 1998.

[78] P. Voda. The constraint language triology: Semantics and computations. Technical report, Complete Logic Systems, North Vancouver, BC, Canada, 1988.

[79] F. Wan, K. Roustogi, J. Xing, and M.P. Singh. Multiagent workflow management. In *IJCAI Workshop on Intelligent Workflow and Process Management: The New Frontier for AI in Business*, Stockholm, Sweden, August 1999.

[80] J. Wurtz. Constraint-based scheduling in Oz. In *Selected Papers of the Symposium on Operations Research (SOR 96)*, pages 218–223, Braunschweig, Germany, September 1996.

[81] J. Wurtz. Oz scheduler: A workbench for scheduling problems. In *Proceedings of International Conference on Tools with Artificial Intelligence (ICTAI)*, 1996.

[82] M. zur Muhlen. Resource modeling in workflow applications. In *Workflow Management Conference*, pages 137–153, Muenster, Germany, November 1999.

APPENDIX A

Proofs

A.1 Proof of Lemma 5.20

The parallelism between the wf-constraint universe definitions and template rules is straightforward. For this reason, we just present these definitions as a table that shows wf-constraint universe definition and corresponding template rule on the same row. This table is given in Figure A.2. The signatures of the user-defined parts are presented similarly, as well, in Figure A.1. Note that the parameter types match as well as the semantics of the definitions. Since it is *boolean* by default, return type is not stated for template placeholders. Another point to be noted is that *goal* corresponds to *partial schedule* $\times$ *assignment* and 2^{rsrc}. This is due to the fact that functions with the domain *partial schedule* $\times$ *assignment* are defined in terms of other functions with the domain *(partial schedule* $\implies$ *assignment)* $\implies$ 2^{rsrc}.

The *goal* - *partial schedule* $\times$ *assignment* correspondence is due to the following. Transformation Θ adds the predicate $resource_asg(A, Agents)$ for each atomic goal, in

order to model the resource allocation information corresponding to the task. Therefore, an atomic goal a is represented with the transformed goal $a \otimes resource_asg(a, Agent_a)$. $Agent_a$ is created as a different variable for each atomic goal. In *cost* subsystem of Wf-constraint universe, cost of resource assignment for an atomic goal is found with the function $cost_of(\omega, \xi) = cost_o f(r), r \in \xi(\omega))$. The corresponding rule in constraint template is $cost(A \otimes resource_asg(A, Agent), V) = cost_o f(resource_asg(A, Agents), V)$. The term $resource_asg(A, Agents)$ corresponds to $r \in \xi(\omega))$, where ω is a partial schedule that satisfies a.

Similarly, in *ctrl* subsystem of Wf-constraint universe, in order to specify the constraints on resource assignment of a single task, the user defines the function $task_constraint(\omega, \xi)$, where ω is an m-path satisfying the task under consideration. The constraint is to be defined on resource $r \in \xi(\omega))$ The corresponding placeholder in the constraint template is $task_constraint(A \otimes resource_asg(A, Agent)$. The user specifies the constraint on the variable *Agent*, which denotes the resource to be allo-

	Constraint System Predicates	Template Rules
1	$cost_constraint(\omega, \xi) \equiv$ $value_constraint(cost(\omega, \xi))$	$cost_constraint(G)$:− $cost(G, V)$, $\mathbf{value_constraint}(V)$
2	$cost(\omega_1 \bullet_p \omega_2, \xi) \equiv$ $op_{\otimes}(cost(\omega_1, \xi), cost(\omega_2, \xi))$	$cost(G_1 \otimes G_2, V)$:− $cost(G_1, V_1), cost(G_2, V_2), \mathbf{op}_{\otimes}(V_1, V_2, V)$
3	$cost(\omega_1 \Vert_p \omega_2, \xi) \equiv$ $op_{\vert}(cost(\omega_1, \xi), cost(\omega_2, \xi))$	$cost(G_1 \mid G_2, V)$:− $cost(G_1, V_1), cost(G_2, V_2), \mathbf{op}_{\vert}(V_1, V_2, V)$
4	$ctrl_constraint(\omega_1 \Vert_p \omega_2, \xi) \equiv$ $set_constraint_{\vert}(\xi(\omega_1), \xi(\omega_2)) \wedge$ $ctrl_constraint(\omega_1, \xi) \wedge$ $ctrl_constraint(\omega_2, \xi)$	$ctrl_constraint(G_1 \mid G_2))$:− $\mathbf{set_constraint}_{\vert}(G_1, G_2)$, $ctrl_constraint(G_1)$, $ctrl_constraint(G_2)$
5	$ctrl_constraint(\omega_1 \bullet_p \omega_2, \xi) \equiv$ $set_constraint_{\otimes}(\xi(\omega_1), \xi(\omega_2)) \wedge$ $ctrl_constraint(\omega_1, \xi) \wedge$ $ctrl_constraint(\omega_2, \xi)$	$ctrl_constraint(G_1 \otimes G_2)$:− $\mathbf{set_constraint}_{\otimes}(G_1, G_2)$, $ctrl_constraint(G_1)$, $ctrl_constraint(G_2)$

Figure A.1: Constraint universe predicates vs. template rules

	Constraint Uni. Predicates/Functions	Template Placeholders
1	$cost_constraint$: $partial\ schedule \times asg. \rightarrow boolean$	$cost_constraint(G)$, $G : goal$
2	$value_constraint : scalar \rightarrow boolean$	$value_constraint(V)$, $V : scalar$
3	$op_{\otimes} : scalar \times scalar \rightarrow scalar$	$op_{\otimes}(V_1, V_2, V)$, $V_1, V_2, V : scalar$
4	$op_{\vert} : scalar \times scalar \rightarrow scalar$	$op_{\vert}(V_1, V_2, V)$, $V_1, V_2, V : scalar$
5	$cost_of : rsrc \rightarrow scalar$	$cost_of(G, V), G : goal,\ V : scalar$
6	$ctrl_constraint$: $partial\ schedule \times asg. \rightarrow boolean$	$ctrl_constraint(G)$, $G : goal$
7	$task_constraint$: $simple\ p.sch. \times asg. \rightarrow boolean$	$task_constraint(G),\ G : atomic\ goal$
8	$set_constraint_{\otimes}$: $2^{rsrc} \times 2^{rsrc} \rightarrow boolean$	$set_constraint_{\otimes}(G_1, G_2), G_1, G_2 : goal$
9	$set_constraint_{\vert}$: $2^{rsrc} \times 2^{rsrc} \rightarrow boolean$	$set_constraint_{\vert}(G_1, G_2), G_1, G_2 : goal$

Figure A.2: Signatures for predicates and functions

cated for task A. The same approach holds for $set_constraint_{\otimes}$ and $set_constraint_{|}$, as well. □

A.2 Proof of Theorem 5.21

The correctness of the expression $P, D_0...D_n, \xi \models \phi$ iff depends on three items:

- $\Theta(\phi) = \phi'$
- $P, D_0...D_n \models \phi'$ and
- ξ is obtained from ϕ'

The first module of the system transforms any CCTR goal ϕ into a CTR goal ϕ' by transformation Θ.

The execution ordering $D_0...D_1$ is found by CTR interpreter for ϕ'.

As explained in Section A.1, transformation Θ introduces the resource assignment information for tasks into ϕ' in the form of predicates $resource_asg(A, Agents)$. The

variable *Agents* corresponds to resource $r \in \xi(\omega)$, where ω satisfies A. Therefore, if ω is a partial schedule satisfying a non-atomic goal ϕ and $\xi(\omega)$ is the resource allocation for the goal ϕ, then set of predicates $resource_asg(A, Agents)$ constitute the resource allocation. According to Lemma 5.20, constraint template and placeholder definitions are equivalent to Wf-Constraint Universe definitions, then the solution provided to the constraint system that grounds the predicates $resource_asg(A, Agents)$ is the solution to Wf-Constraint Universe definitions, which is ξ for ϕ.

Having obtained the execution order $D_0...D_n$ and resource assignment ξ, we have to show that $P, D_0...D_n, \xi \models \phi$ is true. This expression is true if for every model M of P, $M, \omega, \xi \models \phi$ and $\langle D_0, ..., D_n \rangle \in mpaths(\omega)$. Therefore, we show that there exists such a ω through induction, as follows:

Base Case.

- if $\phi = a$, where $a \in \mathcal{P}$, then $\phi' = a \otimes resource_asg(a, X)$. If the path for ϕ' is found as $\pi = \langle D_0 D_1 \rangle$, then let ω be $\langle D_0 D_1 \rangle$. Since ϕ does not have any constraint predicate $M, \omega, \xi \models \phi$ under any ξ and $\pi \in mpaths(\omega)$. Therefore, $P, D_0 D_1, \xi \models \phi$.

- if $\phi = a \otimes b$, where $a, b \in \mathcal{P}$, then $\phi' = (a \otimes resource_asg(a, X)) \otimes (b \otimes resource_asg(b, Y))$. If the path for ϕ' is found as $\pi = \langle D_0 D_1 D_2 \rangle$, then let ω be $\langle D_0 D_1 \rangle \bullet_p \langle D_1 D_2 \rangle$. Since ϕ does not have any constraint predicate $M, \omega, \xi \models \phi$ under any ξ and $\pi \in mpaths(\omega)$. Therefore, $P, D_0 D_1, \xi \models \phi$.

- if $\phi = a \mid b$, where $a, b \in \mathcal{P}$, then $\phi' = (a \otimes resource_asg(a, X)) \mid (b \otimes resource_asg(b, Y))$. If the path for ϕ' is found as $\pi = \langle D_0 D_1 D_2 \rangle$, then let ω be $\langle D_0 D_1 \rangle \parallel_p \langle D_1 D_2 \rangle$ or $\langle D_1 D_2 \rangle \parallel_p \langle D_0 D_1 \rangle$. Since ϕ does not have any

constraint predicate $M, \omega, \xi \models \phi$ under any ξ and $\pi \in mpaths(\omega)$. Therefore, $P, D_0 D_1, \xi \models \phi$.

Induction.

- Let ϕ is some CCTR goal, ϕ' the transformed goal, ξ be the resource assignment and $D_0...D_n$ be the path satisfying ϕ'. Then we assume that ω is a partial schedule such that $M, \omega, \xi \models \phi$ and $\langle D_0, ..., D_n \rangle \in mpaths(\omega)$.

Generalization.

- if $\phi = G_1 \otimes G_2$, where G_1 and G_2 are CCTR goals, then $\phi' = G_1' \otimes G_2'$, where G_1' and G_2' transformed goals corresponding to G_1 and G_2, respectively. If the path for ϕ' found as $\pi = \langle D_0...D_{m-1} D_m...D_n \rangle$ and ξ is the resource assignment, then let ω be $\omega_1 \bullet_p \omega_2$, where $M, \omega_1, \xi \models G_1'$ $\langle D_0...D_{m-1} \rangle \in mpaths(\omega_1)$ and $M, \omega_2, \xi \models G_2'$, $langle D_m...D_n \rangle \in mpaths(\omega_2)$. Then $M, \omega, \xi \models \phi$ and $\pi \in mpaths(\omega)$. Therefore, $P, D_0...D_n, \xi \models \phi$.

- if $\phi = G_1 \mid G_2$, where G_1 and G_2 are CCTR goals, then $\phi' = G_1' \mid G_2'$, where G_1' and G_2' transformed goals corresponding to G_1 and G_2, respectively. If the path for ϕ' found as $\pi = \langle D_0...D_n \rangle$ and ξ is the resource assignment, then let π_1 and π_2 be two order preserving partitions of $\langle D_0...D_n \rangle$, ω be $\omega_1 \bullet_p \omega_2$, where $M, \omega_1, \xi \models G_1'$ $\pi_1 \rangle \in mpaths(\omega_1)$ and $M, \omega_2, \xi \models G_2'$, $\pi_2 \rangle \in mpaths(\omega_2)$. Then $M, \omega, \xi \models \phi$ and $\pi \in mpaths(\omega)$. Therefore, $P, D_0...D_n, \xi \models \phi$.

- if $\phi = G_1 \vee G_2$, where G_1 and G_2 are CCTR goals, then $\phi' = G_1' \vee G_2'$, where G_1' and G_2' transformed goals corresponding to G_1 and G_2, respectively. If the path for ϕ' found as $\pi = \langle D_0...D_n \rangle$ and ξ is the resource assignment, then let

ω be a partial schedule so that $M, \omega, \xi \models G_1$, $\pi \in mpaths(\omega)$ or $M, \omega, \xi \models G_2$, $\pi \in mpaths(\omega)$. Therefore, $P, D_0 ... D_n, \xi \models \phi$.

- if $\phi = G \wedge c$, where G is a CCTR goal and $c \in \mathcal{C}$, then $\phi' = G' \otimes (c_{\mathcal{D}})(G')$, where G' is the transformed goal corresponding to G. If the path for ϕ' found as $\pi = \langle D_0 ... D_n \rangle$ and ξ is the resource assignment, let ω be a partial schedule so that $M, \omega, \xi \models \phi$ and $\pi \in mpaths(\omega)$. Since ξ is the solution to constraint definitions which is equivalent to $\mathcal{D}$, then $\mathcal{D} \models \rfloor_{\mathcal{D}}(\omega, \xi)$. Therefore, $P, D_0 ... D_n, \xi \models \phi$.

[13] shows that CTR proof theory is complete for concurrent serial goals. Except for constraint predicates (which are solved by the constraint solver, not by CTR proof theory), the goals we are dealing with are concurrent serial goals. Therefore, the first part of the scheduler is complete. In our system, we use an off-the-shelf constraint solvers. Therefore, completeness of the system depends on the completeness of constraint solver incorporated into the system. However, if finite-domain solvers are used, since the domain is limited, the solver is complete [39, 50]. Any existing solution can be found, through the search of the entire domain, in the worst case scenario. □

APPENDIX B

Definitions for Applications

B.1 Travel Agency Workflow

WSL definition for travel agency workflow is $SEQ\{b, h, c\}$. The same workflow is translated into the constraint language as follows:

$b.end \leq h.start, h.end \leq c.start$

For this workflow, cost constraints are defined on three dimensions of the total resource allocation cost. However, no control constraints are specified. The cost constraints are represented as:

$b.cost + h.cost + c.cost < 1000$

$b.dur + h.dur + c.dur < 7$

$b.qual + h.qual + c.qual > 2$

The constraint definitions in the constraint language has the same structure as given above.

The same workflow is specified in CCTR as $b \otimes h \otimes c$. Constraint definitions for

$resource_asg := rsrc(T, X)$
$\textbf{cost_of}(rsrc(T, X), [V_1, V_2, V_3]) : -duration(T, X, V_1), cost(T, X, V_2), quality(T, X, V_3)$
$\textbf{value_constraint}([V_1, V_2, V_3]) : -V_1 < 1000, V_2 < 7, V_3 > 2$
$\textbf{numericfunc}_{|}([V_1, V_1', V_1''], [V_2, V_2', V_2''], [V, V', V'']) : -$
$V \text{ is } max(V_1, V_2), V' \text{ is } V_1' + V_2',\ V'' \text{ is } V_1'' + V_2''$
$\textbf{numericfunc}_{\otimes}([V_1, V_1', V_1''], [V_2, V_2', V_2''], [V, V', V'']) : -$
$V \text{ is } V_1 + V_2, V' \text{ is } V_1' + V_2',\ V'' \text{ is } V_1'' + V_2''$

Figure B.1: Constraint definitions for travel planning workflow

travel agency workflow is given in Figure B.1. Recall that there are only cost conditions on this workflow. Together with the constraint predicate the goal is represented as $(b \otimes h \otimes c) \wedge cost_constraint$

The initial goal $G = (b \otimes h \otimes c) \wedge cost_constraint$ is transformed into the following goal:

$G' :\ W\ \otimes\ cost_constraint(W)$, where
$W : ((b \otimes rsrc(b, X) \otimes (h \otimes rsrc(h, Y)))\ \otimes (c \otimes rsrc(c, W)))$

The resulting constraint set is as follows:

$\{duration(b, X, Dur_1), cost(b, X, Cost_1), qual(b, X, Qual_1),$

$duration(h, Y, Dur_2), cost(h, Y, Cost_2), qual(h, Y, Qual_2),$

$Dur_{12} \text{ is } Dur_1 + Dur_2, Cost_{12} \text{ is } Cost_1 + Cost_2, Qual_{12} \text{ is } Qual_1 + Qual_2,$

$duration(c, Z, Dur_3), cost(c, Z, Cost_3), qual(c, Z, Qual_3),$

$Dur_{123} \text{ is } Dur_{12} + Dur_3, Cost_{123} \text{ is } Cost_{12} + Cost_3, Qual_{123} \text{ is } Qual_{12} + Qual_3\}$

B.2 Telecommunication Service Providing Workflow

The WSL representation of this workflow is

$AND\{SEQ\{OR\{a_1, a_2\}, OR\{b_1, b_2, b_3\}\}, OR\{c_1, c_2\}\}.$

The constraint language equivalence of this representation is given in Figure B.2.

$(a_block.start \leq a_1.start \wedge a_block.end \geq a_1.end) \vee$
$(a_block.start \leq a.start \wedge a_2_block.end \geq a_2.end)$
$(b_block.start \leq b_1.start \wedge b_block.end \geq b_1.end) \vee$
$(b_block.start \leq b.start \wedge b_2_block.end \geq b_2.end) \vee$
$(b_block.start \leq b.start \wedge b_3_block.end \geq b_3.end)$
$(c_block.start \leq c_1.start \wedge c_block.end \geq c_1.end) \vee$
$(a_block.start \leq c.start \wedge c_2_block.end \geq c_2.end)$
$a_block.end \leq b_block.start$
$wf.start \leq a_block.start, wf.end \geq a_block.end$
$wf.start \leq c_block.start, wf.end \geq c_block.end$

Figure B.2: Telecommunications workflow in constraint language

$a_block \neq No_resource \wedge$
$(a_block.resource = a_1.resource \vee a_block.resource = a_2.resource)$
$b_block \neq No_resource \wedge$
$(b_block.resource = b_1.resource \vee b_block.resource = b_2.resource \vee$
$b_block.resource = b_3.resource)$
$c_block \neq No_resource \wedge$
$(c_block.resource = c_1.resource \vee c_block.resource = c_2.resource)$
$(a_block.end > c_block.start \wedge c_block.end > a_block.start) \rightarrow$
$a_block.resource \neq c_block.resource.$
$(b_block.end > c_block.start \wedge c_block.end > b_block.start) \rightarrow$
$a_block.resource \neq c_block.resource.$

Figure B.3: Telecommunications workflow constraints in constraint language

The workflow has a single cost constraint and a single control constraint. The constraint on total duration is specified as follows:

$$a_1.dur + a_2.dur + b_1.dur + b_2.dur + b_3.dur + c_1.dur + c_2.dur \leq limit.$$

The control constraint requires the disjointness of the resource allocation for parallel tasks. Among the several ways to express this constraint, one possible representation is given in Figure B.3.

CCTR representation of this workflow is $((a_1 \vee a_2) \otimes (b_1 \vee b_2 \vee b_3)) \mid (c_1 \vee c_2)$ The constraint template definitions are given in Figure B.4. Note that, in this example, cost has a single dimension. Therefore, it is represented as a single variable instead of a list. Control constraint contains only resource disjointness restriction. Definition of $disjoint$ is given in Figure B.5.

$resource_asg := rsrc(T, X)$
$\textbf{cost_of}(rsrc(T, X), V) : -cost(T, X, V)$
$\textbf{value_constraint}(V) : -V < constant$
$\textbf{numericfunc}_{|}(V_1, V_2, V) : -V \text{ is } V_1 + V_2$
$\textbf{numericfunc}_{\otimes}(V_1 V_2, V) : -V \text{ is } V_1 + V_2$
$\textbf{set_constraint}_{|}(G_1, G_2) : -disjoint(G_1, G_2)$
$\textbf{set_constraint}_{\otimes}(G_1, G_2) : -true$
$\textbf{task_constraint}(\mathbf{T}) : -true$

Figure B.4: Constraint definitions for telecommunications workflow

$$
\begin{aligned}
&disjoint(T_1 \otimes rsrc(T_1, X_1, V_1), T_2 \otimes rsrc(T_2, X_2, V_2)) : -X_1 \neq X_2.\\
&disjoint((G_1 \otimes G_2), T \otimes rsrc(T, X, V)) \ : -\\
&\qquad disjoint(G_1, T \otimes rsrc(T, X, V)), disjoint(G_2, T \otimes rsrc(T, X, V)).\\
&disjoint((G_1 \mid G_2), T \otimes rsrc(T, X, V)) \ : -\\
&\qquad disjoint(G_1, T \otimes rsrc(T, X, V)), disjoint(G_2, T \otimes rsrc(T, X, V)).\\
&disjoint(T \otimes rsrc(T, X, V), (G_1 \otimes G_2)) \ : -\\
&\qquad disjoint(T \otimes rsrc(T, X, V), G_1), disjoint(T \otimes rsrc(T, X, V), G_2).\\
&disjoint(T \otimes rsrc(T, X, V), (G_1 \mid G_2)) \ : -\\
&\qquad disjoint(T \otimes rsrc(T, X, V), G_1), disjoint(T \otimes rsrc(T, X, V), G_2).\\
&disjoint((G_1 \otimes G_2), (G_3 \otimes G_4)) \ : -\\
&\qquad disjoint(G_1, G_3), disjoint(G_1, G_4), disjoint(G_2, G_3), disjoint(G_2, G_4).\\
&disjoint((G_1 \otimes G_2), (G_3 \mid G_4)) \ : -\\
&\qquad disjoint(G_1, G_3), disjoint(G_1, G_4), disjoint(G_2, G_3), disjoint(G_2, G_4).\\
&disjoint((G_1 \mid G_2), (G_3 \otimes G_4)) \ : -\\
&\qquad disjoint(G_1, G_3), disjoint(G_1, G_4), disjoint(G_2, G_3), disjoint(G_2, G_4).\\
&disjoint((G_1 \mid G_2), (G_3 \mid G_4)) \ : -\\
&\qquad disjoint(G_1, G_3), disjoint(G_1, G_4), disjoint(G_2, G_3), disjoint(G_2, G_4).
\end{aligned}
$$

Figure B.5: Definition of *disjoint* constraint

The initial goal $G = (((a_1 \vee a_2) \otimes (b_1 \vee b_2 \vee b_3)) \mid (c_1 \vee c_2)) \wedge cost_constraint \wedge ctrl_constraint$ is transformed into the following goal:

$$
\begin{aligned}
&G' : W \ \otimes \ cost_constraint(W) \ \otimes \ ctrl_constraint(W), \text{ where}\\
&W : ((((a_1 \otimes rsrc(a_1, X)) \vee (a_2 \otimes rsrc(a_2, X)))\otimes\\
&\qquad ((b_1 \otimes rsrc(b_1, Y)) \vee (b_2 \otimes rsrc(b_2, Y)) \vee (b_3 \otimes rsrc(b_3, Y)))) \ \mid\\
&\qquad ((c_1 \otimes rsrc(c_1, Z)) \vee (c_2 \otimes rsrc(c_2, Z))))
\end{aligned}
$$

In this example, due to disjunctions in the workflow goal, several schedules and their corresponding constraint sets may be produced. Some of them are listed in Figure B.6.

B.3 Conference Planning

WSL representation of this workflow is

schedule	constraint set
a_1, b_1, c_1	$\{cost(a_1, X, C_1), cost(b_1, Y, C_2), V_1$ is $C_1 + C_2, cost(c_1, W, C_3),$ V is $V_1 + C_3, V < c, X \neq Z, Y \neq Z\}$
a_2, b_2, c_1	$\{cost(a_2, X, C_1), cost(b_2, Y, C_2), V_1$ is $C_1 + C_2, cost(c_1, W, C_3),$ V is $V_1 + C_3, V < c, X \neq Z, Y \neq Z\}$
a_2, b_3, c_2	$\{cost(a_2, X, C_1), cost(b_3, Y, C_2), V_1$ is $C_1 + C_2, cost(c_2, W, C_3),$ V is $V_1 + C_3, V < c, X \neq Z, Y \neq Z\}$

Figure B.6: Example sch. and const. sets for telecom. workflow

$block1.start \leq a.start \wedge block1.end \geq a.end$
$block1.start \leq b.start \wedge block1.end \geq b.end)$
$block1.start \leq c.start \wedge block1.end \geq c.end)$

$block2.start \leq d.start \wedge block2.end \geq d.end$
$block2.start \leq e.start \wedge block2.end \geq e.end)$
$block2.start \leq f.start \wedge block2.end \geq f.end)$

$block3.start \leq g.start \wedge block3.end \geq g.end$
$block3.start \leq h.start \wedge block3.end \geq h.end)$
$block3.start \leq i.start \wedge block3.end \geq i.end)$

$block4.start \leq j.start \wedge block4.end \geq j.end$
$block4.start \leq k.start \wedge block4.end \geq k.end)$
$block4.start \leq l.start \wedge block4.end \geq l.end)$

$block1.end \leq block2.start\ block2.end \leq block3.start\ block3.end \leq block4.start$

Figure B.7: Conference planning workflow in constraint language

$$SEQ\{AND\{a, b, c\}, AND\{d, e, f\}, AND\{g, h, i\}, AND\{j, k, l\}\}.$$

The corresponding constraint language definitions are shown in Figure B.7.

There are no cost constraints on this workflow. The control constraints are resource disjointness, restriction on parallel allocation of some resources and enforcing allocation of certain resources before certain other resources. The specification of disjointness constraint is as in the previous example. We can rephrase the constraint *session 4 must take place before session 11* as *session 4 must not take place after or parallel to session 11.* Assuming that the parallel session all have the same constraint, we may represent the rephrased constraint for a subworkflow that includes first two time slots

$a.resource = r_4 \leftrightarrow b.resource \neq r_{11}$	$d.resource = r_4 \leftrightarrow e.resource \neq r_{11}$
$a.resource = r_4 \leftrightarrow c.resource \neq r_{11}$	$d.resource = r_4 \leftrightarrow f.resource \neq r_{11}$
$b.resource = r_4 \leftrightarrow c.resource \neq r_{11}$	$e.resource = r_4 \leftrightarrow f.resource \neq r_{11}$
$d.resource = r_4 \rightarrow a.resource \neq r_{11}$	$e.resource = r_4 \rightarrow a.resource \neq r_{11}$
$d.resource = r_4 \rightarrow b.resource \neq r_{11}$	$e.resource = r_4 \rightarrow b.resource \neq r_{11}$
$d.resource = r_4 \rightarrow c.resource \neq r_{11}$	$e.resource = r_4 \rightarrow c.resource \neq r_{11}$
$f.resource = r_4 \rightarrow a.resource \neq r_{11}$	
$f.resource = r_4 \rightarrow b.resource \neq r_{11}$	
$f.resource = r_4 \rightarrow c.resource \neq r_{11}$	

Figure B.8: *Before* constraint in constraint language

$a.resource = r_1 \rightarrow b.resource \neq r_2$
$a.resource = r_1 \rightarrow c.resource \neq r_2$
$b.resource = r_1 \rightarrow c.resource \neq r_2$
$d.resource = r_1 \rightarrow e.resource \neq r_2$
$d.resource = r_1 \rightarrow f.resource \neq r_2$
$e.resource = r_1 \rightarrow f.resource \neq r_2$

Figure B.9: *Not-parallel* constraint in constraint language

$resource_asg := rsrc(T, X)$
$\textbf{set_constraint}_{\otimes}(G_1, G_2) : -disjoint(G_1, G_2), before(G_1, G_2).$
$\textbf{set_constraint}_{|}(G_1, G_2) : -disjoint(G_1, G_2), not_parallel(G_1, G_2), before(G_1, G_2)$

Figure B.10: Constraint definitions for conference planning workflow

as given in Figure B.8.

For the same subworkflow, the constraint *session 1 can not be held in parallel to session 2* is specified as shown in Figure B.9.

CCTR representation of this workflow is $(a \mid b \mid c) \otimes (d \mid e \mid f) \otimes (h \mid i \mid f) \otimes (j \mid k \mid l)$. The user defined constraints for conference planning example are given in Figure B.10.

Definition for *disjoint* is same as in the previous examples. Definitions for *before* and *not_parallel* are given in Figure B.11 and Figure B.12, respectively.

B.4 Composite Web Services

WSL representation of this composite service is

$AND\{SEQ\{get_wish_list, buy_present\}, reservation\}.$

$before(T_1 \otimes rsrc(T_1, A_1, V_1), T_2 \otimes rsrc(T_2, A_2, V_2)) \;:-\; A_2 = 4 \rightarrow A_1 \neq 11.$
$before((G_1 \otimes G_2), T \otimes rsrc(T, A, V)) \;:-$
$\quad before(G_1, T \otimes rsrc(T, A, V)), before(G_2, T \otimes rsrc(T, A, V))$
$before((G_1 \mid G_2), T \otimes rsrc(T, A, V)) \;:-$
$\quad before(G_1, T \otimes rsrc(T, A, V)), before(G_2, T \otimes rsrc(T, A, V))$
$before(T \otimes rsrc(T, A, V), (G_1 \otimes G_2)) \;:-$
$\quad before(T \otimes rsrc(T, A, V), T_1), before(T \otimes rsrc(T, A, V), T_2)$
$before(T \otimes rsrc(T, A, V), (G_1 \mid G_2)) \;:-$
$\quad before(T \otimes rsrc(T, A, V), T_1), before(T \otimes rsrc(T, A, V), T_2)$
$before((G_1 \otimes G_2), (G_3 \otimes G_4)) \;:-$
$\quad before(G_1, G_3), before(G_1, G_4), before(G_2, G_3), before(G_2, G_4)$
$before((G_1 \otimes G_2), (G_3 \mid G_4)) \;:-$
$\quad before(G_1, G_3), before(G_1, G_4), before(G_2, G_3), before(G_2, G_4)$
$before((G_1 \mid G_2), (G_3 \otimes G_4)) \;:-$
$\quad before(G_1, G_3), before(G_1, G_4), before(G_2, G_3), before(G_2, G_4)$
$before((G_1 \mid G_2), (G_3 \mid G_4)) \;:-$
$\quad before(G_1, G_3), before(G_1, G_4), before(G_2, G_3), before(G_2, G_4)$

Figure B.11: Definition of *before* constraint

$not_parallel(T_1 \otimes rsrc(T_1, A_1, V_1), T_2 \otimes rsrc(T_2, A_2, V_2)) \;:-$
$\quad A_1 = 1 \rightarrow A_2 \neq 2,\; A_2 = 1 \rightarrow A_1 \neq 2$
$not_parallel(G_1 \otimes G_2), T \otimes rsrc(T, A, V)) \;:-$
$\quad not_parallel(G_1, T \otimes rsrc(T, A, V)), not_parallel(G_2, T \otimes rsrc(T, A, V))$
$not_parallel(G_1 \mid G_2), T \otimes rsrc(T, A, V)) \;:-$
$\quad not_parallel(G_1, T \otimes rsrc(T, A, V)), not_parallel(G_2, T \otimes rsrc(T, A, V))$
$not_parallel(T \otimes rsrc(T, A, V), (G_1 \otimes G_2)) \;:-$
$\quad not_parallel(T \otimes rsrc(T, A, V), G_1), not_parallel(T \otimes rsrc(T, A, V), G_2)$
$not_parallel(T \otimes rsrc(T, A, V), (G_1 \mid G_2)) \;:-$
$\quad not_parallel(T \otimes rsrc(T, A, V), G_1), not_parallel(T \otimes rsrc(T, A, V), G_2)$
$not_parallel((G_1 \otimes G_2), (G_3 \otimes G_4)) \;:-$
$not_parallel(G_1, G_3), not_parallel(G_1, G_4) not_parallel(G_2, G_3), not_parallel(G_2, G_4)$
$not_parallel((G_1 \otimes G_2), (G_3 \mid G_4)) \;:-$
$not_parallel(G_1, G_3), not_parallel(G_1, G_4) not_parallel(G_2, G_3), not_parallel(G_2, G_4)$
$not_parallel((G_1 \mid G_2), (G_3 \otimes G_4)) \;:-$
$not_parallel(G_1, G_3), not_parallel(G_1, G_4) not_parallel(G_2, G_3), not_parallel(G_2, G_4)$
$not_parallel((G_1 \mid G_2), (G_3 \mid G_4)) \;:-$
$not_parallel(G_1, G_3), not_parallel(G_1, G_4) not_parallel(G_2, G_3), not_parallel(G_2, G_4)$

Figure B.12: Definition of *not-parallel* constraint

$resource_asg := rsrc(T, X)$
$\mathbf{cost_of}(rsrc(T, A), V)\ :-\ cost(T, A, V)$
$\mathbf{value_constraint}(V)\ :-\ V < c,$
$\mathbf{set_constraint}_{|}(G_1, G_2)\ :-\ true$
$\mathbf{set_constraint}_{\otimes}(G_1, G_2)\ :-\ true$
$\mathbf{task_constraint}(rsrc(T, A))\ :-\ (T = reservation \rightarrow (quality(T, A, Q), (Q >= q))$
$\mathbf{op}_{|}(V_1, V_2, V)\ :-\ V \text{ is } V_1 + V_2$
$\mathbf{op}_{\otimes}(V_1, V_2, V)\ :-\ V \text{ is } V_1 + V_2$

Figure B.13: Placeholders for web service composition

The total cost constraint can be represented as

$get_wish_list.cost + buy_present.cost + reservation.cost < c.$

Since get_wish_list is just a query, it does not affect the total cost. Constraint on the quality for the dinner is represented as

$reservation.quality > q$

CCTR representation for example composite service is $(get_wish_list \otimes buy_present) \mid reservation$. Constraint definitions for composite web services application is given in Figure B.13.

VITA

Pınar (Karagöz) Şenkul was born on February 24, 1974 in Ankara. She has received her BS degree from Middle East Technical University, Computer Engineering Department in 1996. The same year, she started working as a teaching assistant in the same department. She received her MS degree from the same department in January 1998. In 2000, she has started working as a research assistant in the department for the NSF-Tubitak project," Logic-based Modeling and Specification of Workflows". For the same project she worked as a research associate in State University of New York at Stony Brook between October 2000 and July 2001. Since February 2003, she has been working as an instructor at Bilkent University, Computer Engineering Department.