

FINITE SAFETY MODELS FOR HIGH-ASSURANCE SYSTEMS

by

John C. Sloan

A Dissertation Submitted to the Faculty of
The College of Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy

Florida Atlantic University

Boca Raton, Florida

August 2010

Copyright by John C. Sloan 2010

FINITE SAFETY MODELS FOR HIGH-ASSURANCE SYSTEMS

by

John C. Sloan

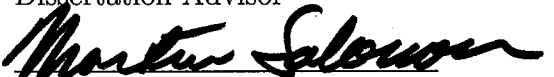
This dissertation was prepared under the direction of the candidate's dissertation advisor, Dr. Taghi M. Khoshgoftaar, Department of Computer and Electrical Engineering and Computer Science, and has been approved by the members of his supervisory committee. It was submitted to the faculty of the College of Engineering and Computer Science and was accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

SUPERVISORY COMMITTEE:

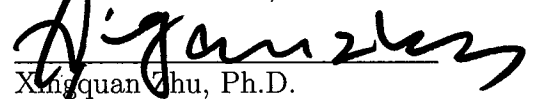


Taghi M. Khoshgoftaar, Ph.D.

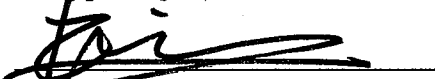
Dissertation Advisor



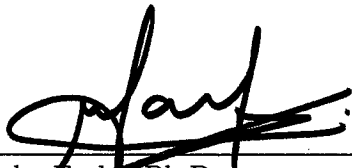
Martin K. Solomon, Ph.D.



Xingquan Zhu, Ph.D.

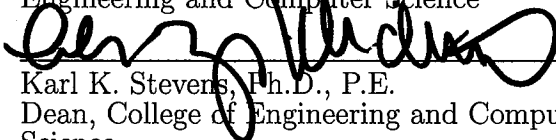


Hanqi Zhuang, Ph.D.



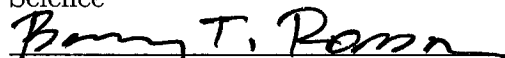
Borko Furht, Ph.D.

Chair, Department of Computer and Electrical
Engineering and Computer Science



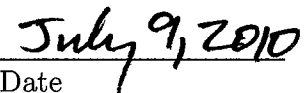
Karl K. Stevens, Ph.D., P.E.

Dean, College of Engineering and Computer
Science



Barry T. Ross, Ph.D.

Dean, Graduate College



Date

ACKNOWLEDGMENTS

I would like to thank Dr. Taghi M. Khoshgoftaar for his invaluable guidance and support of my research and graduate studies at Florida Atlantic University. He is truly a world-class researcher and mentor. I also thank Dr. Martin K. Solomon, Dr. Xingquan Zhu, and Dr. Hanqi Zhuang for serving on my dissertation committee. Finally, I wish to thank Dr. W.M.P (Wil) van der Aalst of Technische Universiteit Eindhoven, and Dr. J.J.M.M. (Jan) Rutten, Dr. Farhad Arbab, and Dr. Dave Clarke of Centrum Wiskunde & Informatica, all of whom encouraged me to study the topics that led to this dissertation. My research at FAU would not have been possible without initial funding from the U.S. Department of Defense and more recent funding from the FAU Center for Ocean Energy Technology (COET).

ABSTRACT

Author: John C. Sloan
Title: Finite Safety Models for High-Assurance Systems
Institution: Florida Atlantic University
Dissertation Advisor: Dr. Taghi M. Khoshgoftaar
Degree: Doctor of Philosophy
Year: 2010

Preventing bad things from happening to engineered systems, demands improvements to how we model their operation with regard to safety. Safety-critical and fiscally-critical systems both demand automated and exhaustive verification, which is only possible if the models of these systems, along with the number of scenarios spawned from these models, are tractably finite. To this end, this dissertation addresses problems of a model's tractability and usefulness. It addresses the state space minimization problem by initially considering tradeoffs between state space size and level of detail or fidelity. It then considers the problem of human interpretation in model capture from system artifacts, by seeking to automate model capture. It introduces human control over level of detail and hence state space size during model

capture. Rendering that model in a manner that can guide human decision making is also addressed, as is an automated assessment of system timeliness. Finally, it addresses state compression and abstraction using logical fault models like fault trees, which enable exhaustive verification of larger systems by subsequent use of transition fault models like Petri nets, timed automata, and process algebraic expressions. To illustrate these ideas, this dissertation considers two very different applications - web service compositions and submerged ocean machinery.

DEDICATION

To my wife Judy S. Montgomery for her cheerful and gracious encouragement and support, and to my late parents Mr. and Mrs. John and Marie Sloan.

FINITE SAFETY MODELS FOR HIGH-ASSURANCE SYSTEMS

LIST OF TABLES	xiii
LIST OF FIGURES	xv
1 INTRODUCTION	1
1.1 Motivation	2
1.1.1 The Compositionality Assumption	2
1.1.2 Compositionality and Web Services	3
1.1.3 Compositionality and Ocean Machinery	4
1.1.4 Relaxing the Compositionality Assumption	6
1.1.5 Safety and Liveness	7
1.2 Issues and Approaches	7
1.2.1 Test Versus Verification	8
1.2.2 Use Cases for Verification	8
1.2.3 Automating Model Capture	9
1.2.4 Automating Model Presentation	9
1.2.5 Assuring Timeliness	10
1.2.6 State Compression and Abstraction	11
1.2.7 Case Study – Ocean Turbines	11
1.3 Organization	12

2	TEST VERSUS VERIFICATION	15
2.1	Chapter Introduction	16
2.1.1	Motivation	16
2.1.2	Context	17
2.1.3	Contributions	18
2.2	Tradeoffs	20
2.2.1	Triaxial Charts	21
2.2.2	External Tradeoffs	22
2.2.2.1	Engineering	23
2.2.2.2	Commercial	25
2.2.2.3	Entertainment	27
2.2.3	Internal Tradeoffs	28
2.2.3.1	Tool Placement	29
2.2.3.2	Interpreting the Landscape	31
2.2.4	Section Critique	32
2.3	Simulating Web Services with Agents	34
2.3.1	Web Service Orchestrations	35
2.3.2	Anatomy of an Agent	37
2.3.3	Mapping between Agents and Services	39
2.3.4	Travel Agency Case Study	43
2.3.5	Section Critique	48
2.4	Practical Impact	50
2.4.1	Deployment Time Testing	50
2.4.2	Changing Baselines	52
2.4.3	Scripted Testing	54
2.4.4	Process Improvement	55
2.5	Chapter Summary	55

3	USE CASES FOR VERIFICATION	58
3.1	Chapter Introduction	59
3.2	Advanced Usage Trends	60
3.2.1	Syndication Time Modeling	61
3.2.2	Service Evolution	61
3.2.3	Incremental Coverage Testing	64
3.2.4	Integrating Instrumentation	65
3.2.5	Stateful Web Services	66
3.2.6	Visualizing State Spaces	67
3.3	Architectural Features	68
3.3.1	Decouple Functionalities	69
3.3.2	Emphasize Soundness and Completeness	70
3.3.3	Streamline Model Capture	71
3.3.4	Control Level of Abstraction	72
3.4	A Predicted Architecture	73
3.5	Chapter Summary	76
4	AUTOMATING MODEL CAPTURE	77
4.1	Chapter Introduction	78
4.1.1	Motivation	78
4.1.2	Contributions	79
4.1.3	Overview	81
4.2	Structure	82
4.2.1	Sub-language	83
4.2.2	Size Estimation	86
4.2.3	Exclusions	87
4.3	Behavior	89
4.3.1	Instantiation and Properties	89
4.3.2	Reactivity	90
4.3.3	Interaction	90
4.3.4	Sequence	91

4.3.5	Choice	91
4.3.6	Parallelism	92
4.3.7	Interprocess Dependencies	94
4.3.8	Assumptions	95
4.3.9	Limitations	97
4.4	Purchase Order Case Study	98
4.5	BPEL to Promela	101
4.5.1	Formulating Rules	103
4.5.2	Generating Declarations	105
4.5.3	Generating Services	107
4.5.4	Generating Orchestrations	111
4.6	Related Work	114
4.7	Chapter Summary	117
5	AUTOMATING MODEL PRESENTATION	129
5.1	Chapter Introduction	130
5.1.1	Context	131
5.1.2	Contributions	132
5.1.3	Organization	132
5.2	Related Work	133
5.2.1	Verification Tools	134
5.2.2	Conversion Tools	137
5.2.3	Graph Drawing	140
5.2.4	Starting Point	142
5.2.4.1	Infrastructure	143
5.2.4.2	Basic Activities	144
5.2.4.3	Link-related activities	144
5.3	Mining Source Artifacts	147
5.3.1	Partner Links	149
5.3.2	Data-related Flows	150
5.3.3	BPEL Variables	151
5.3.4	BPEL Links and Defaults	152

5.3.5	CPN Subnet Names	152
5.3.6	Data Cleansing	153
5.4	Embedding Subnets	155
5.4.1	Strategy	157
5.4.2	Processing	158
5.4.3	Implementation	159
5.5	Generating the CPN	161
5.6	Simulation Results	163
5.7	Chapter Summary	164
6	ASSURING TIMELINESS	170
6.1	Chapter Introduction	171
6.2	Modeling BOINC	173
6.3	Modeling BOINC with UPPAAL	176
6.3.1	Mapping the Petri Net to Timed automata	176
6.3.2	Extending the Timed Automata	177
6.3.3	Model Check Timed Automata	182
6.4	Experiences and Conclusions	182
7	STATE COMPRESSION AND ABSTRACTION	186
7.1	Chapter Introduction	187
7.2	Mathematical Structure	188
7.3	Refinements	191
7.3.1	Multistate Systems	191
7.3.2	Node Commonality	193
7.3.3	State Coherence	194
7.4	Algorithm Design and Analysis	196
7.4.1	The Evaluation Problem	197
7.4.2	Construction Problems	199
7.5	Chapter Summary	200

8	CASE STUDY – OCEAN TURBINES	203
8.1	Chapter Introduction	204
8.2	Physical Design	206
8.3	Reliability Concerns	210
8.4	Related Work	214
8.4.1	Potential	214
8.4.2	Environment	215
8.4.3	Machine Condition Monitoring	221
8.4.4	Wind Turbines	224
8.5	Anti-fouling Topographies	225
8.5.1	Self-similar Surfaces	227
8.5.2	Fabrication Issues	230
8.6	Chapter Summary	232
9	CONCLUSION AND FUTURE WORK	237
9.1	Conclusions	238
9.1.1	Transition fault models	238
9.1.2	Logical fault models	241
9.1.3	Case study	242
9.2	Future Work	243
9.2.1	Transition Fault Models	243
9.2.2	Logical Fault Models	245
9.2.3	Case study	245
	BIBLIOGRAPHY	249

LIST OF TABLES

4.1	From BPEL variables to Promela channel statements	119
4.2	From BPEL <flow> to parallel Promela processes	119
4.3	From BPEL control links to a Promela pattern	120
4.4	BPEL (abbreviated version) of the Purchase Order Process	121
4.5	Generate Promela code from a BPEL artifact	121
4.6	Generate Promela Declarations	122
4.7	Promela declarations for the Purchase Order Process	123
4.8	Generate a model of services from a BPEL artifact	123
4.9	Promela environment for the Purchase Order Process	125
4.10	Generate a model of orchestration from a BPEL artifact	126
4.11	Promela orchestration for the Purchase Order Process	127
4.12	Promela orchestration for the Purchase Order Process (contd)	128
5.1	Excerpt of purchase order process (top) and its PNML silhouette (bottom)	166
5.2	Activity and infrastructure namings	167
5.3	Link-related node namings	168
6.1	Model Checked Properties	183
7.1	Properties of state snapshots	202

7.2	Properties of cut sets	202
7.3	Properties of fault trees	202
8.1	Physical features of turbine and moorings	235
8.2	Reliability concerns	236
8.3	Roughness parameters by surface type	236

LIST OF FIGURES

2.1	Assuring Quality of Orchestrations	19
2.2	SOA Testing Tradeoffs	23
2.3	External Tradeoffs	26
2.4	Internal Tradeoffs	30
2.5	A Web Service Orchestration	36
2.6	An Agent with Two-phase Commit	38
2.7	A Mapping between Agents and Services	41
2.8	Query phase to an Online Travel Agency	44
3.1	Proposed Development Cycle	74
5.1	Silhouettes of generated CPN's	140
5.2	A typical BPEL basic activity in CPN	145
5.3	A basic activity as an origin of a link	146
5.4	A link construct	147
5.5	A basic activity as a link's destination	148
5.6	Closeup of top-level CPN	151
5.7	Equivalent graphs: (a) original, (b) palm tree and (c) visibility representation	156
6.1	A Petri net model of BOINC.	175

6.2	UPPAAL model of service portal.	179
6.3	UPPAAL model of service provider.	180
6.4	UPPAAL simulation environment showing one portal and three providers.	181
8.1	Moorings for an ocean turbine	207
8.2	Nacelle and adjoining structures	208
8.3	Surfaces ranked by r_E : (a) shallow and wide, (b) deep and narrow, (c) lattice drained	219
8.4	Generation of Koch curve	228

CHAPTER 1

INTRODUCTION

This dissertation was an outgrowth of my many years of software development and design engineering experience. Since earning my M.S. in Computer Science from University of Central Florida in 1988, I participated in successful software development projects, eventually becoming a *fixer* for a software solutions provider. In that capacity, I was called in to rescue failing projects. One project was so beyond redemption I was required to layoff the entire development staff. Of the projects that could be fixed, I noted what made these different from either the failed or the successful projects. Subsequent years of soul-searching and study, made me realize that reliability is the most consistently underrated endeavor in not only software development shops, but in ocean systems as well. Section 1.1 describes why this is so, with Section 1.2 outlining issues and approaches concerning reliability. Concluding this chapter, Section 1.3 briefly lists the theme of each subsequent chapter and how each came into being.

1.1 Motivation

The 90's saw decision makers staking claim to new frontiers of the Internet. In the 00's they sought to secure both frontiers and infrastructure from attack. As the Deepwater Horizon oil rig disaster continues to unfold, we are finding that many of the failures over the past decade were simply due to bad or incomplete design and testing. Internet (web) services and novel (ocean) machinery will continue to be doomed to the same fate unless we learn something new.

With the line of research outlined in this dissertation, I predict that the 10's will be known as the decade of *safety* – the decade in which we keep bad things from happening to systems by properly designing them in the first place. Over the past four years and four journal papers [121, 122, 123, 126], I have been taking aim at keeping bad things from happening to two very different types of system artifacts. The first includes orchestrations of web services written in the Business Process Execution Language (BPEL), and the second includes novel types of machinery like ocean turbines.

1.1.1 The Compositionality Assumption

For different reasons, both web service orchestrations and ocean machinery are *black box* systems that defy scrutiny of the internal structure and behavior of their component parts. The correctness of a web service orchestration assumes that each of its constituent web services behave as advertised, with vendors keeping internal details

of their particular web service a secret. Likewise, correct functioning of some machine cannot physically be determined since examination of its internals during operation does not bear scrutiny. In both cases, we must settle for observing their behavior from the outside. In doing so, one is tempted to make at least one strong assumption concerning *compositionality*. For the purposes of this dissertation, the compositionality assumption posits that if each part is free of defects then, if properly assembled, the whole will also be free of defects. Thus, according to this assumption, all that is needed is to verify proper operation of the assembly without needing to scrutinize the reliability of individual parts, once each of their reliabilities have been established. Although formal methods may be used to design compositionality into systems being developed from scratch, virtually all legacy systems do not bear sufficient scrutiny to provably determine whether the compositionality assumption holds.

1.1.2 Compositionality and Web Services

For web services, the compositionality assumption is not empirically tenable, since hidden dependencies can be overlooked. Hence, two seemingly unrelated but well-tested web services inside a composition may from time to time exhibit anomalous behavior. Unbeknownst to the orchestration, both services may have happened to have shared the same resource. The compositionality assumption fails because of logical dependencies – dependencies that can be modeled in terms of discrete mathematics. Pinpointing such intermittent failures requires tracking a system’s behavior

over a sequence of discrete state snapshots over time to identify sequences of operations that lead to failure. Hence, this dissertation adapts *transition fault models* like Petri nets and timed automata¹ to assuring the safety of web service orchestrations.

1.1.3 Compositionality and Ocean Machinery

For ocean machinery the compositionality assumption is not empirically tenable as well, but for different reasons. In general, the *useful life* of mechanical assemblies had long been observed to be shorter than the *rated life* of even its most fault prone components. This was due in large part to interactions between disparate rolling elements like gears, bearings, and shafts.

These physical dependencies have traditionally been modeled in terms of a mathematics of a continuous variable, usually as differential equations. However, rolling elements too often break because the parts encounter some *singularity* in which its systems of equations cease to be well-conditioned due to the presence of higher-order derivatives brought on by some sudden physical impact affecting numerous parts in an assembly.

Anomalous vibrations judged to be noise provides additional evidence of the absence of compositionality. For example, resonance of a machine's containment may as likely be the result of external vibrations, as it would be from internal interactions

¹ Markov chains are another example of transition fault models, although this dissertation does not address them.

between rolling elements. In this case, a closed world assumption cannot be made even though it underlies the compositionality assumption.

A related problem entails vibrations from one component that dominates those from other components in the assembly. Otherwise normal vibrations from internal combustion dwarf the more telling yet subtle vibrations from reciprocating elements like pistons and rolling elements like cams, gears, and drive shafts. Epistemologically, we do not know if each part is free of defects since high impact waves from internal combustion is a normal phenomenon, while high impact waves from rolling elements are not. Such masking prevents us from asserting that each part will be free of defects, hence the entire premise of the compositionality assumption ceases to hold.

These problems motivated use of techniques from discrete mathematics that model a machine's state, with state transitions prompted by an event like a sudden impact. State transition systems that reify some fault transition model can have tractability and expressiveness problems. In our preliminary work, the state of an individual ocean turbine may contain between 64 and 128 Boolean variables. Hence, construction of transition fault models directly from the system state will result in an intractably large state space. Furthermore, such naively constructed models are expressed at too low a level of abstraction. To remedy these issues, this dissertation applies a *logical fault model* like fault trees which, given a state snapshot, will abstract that snapshot to zero or more fault types. Such fault types realize semantic compression and abstraction of individual state snapshots so that finite and tractable

transition fault models can be feasibly verified during both design and deployment of ocean machinery.

1.1.4 Relaxing the Compositionality Assumption

When formally verifying service compositions and machinery, we must relax the compositionality assumption. Doing so for the former, requires selection of a *glue language* that can express dependencies between services, hence motivating our choice of BPEL and of a case study (i.e., the Purchase Order Process) that contains such dependencies. For the latter, we have been working with the Center for Ocean Energy Technology (COET) at FAU, on the design of an ocean turbine that will generate electricity from the momentum flux of the Gulf Stream located 30 to 50km offshore [39]. Assessing the health of such machinery requires use of sensors mounted on the housings of rolling elements like the shaft, gearbox, and generator. In addition to vibration data of the closest element being monitored, a sensor will also detect vibrations in adjacent and non-adjacent elements, as well as vibrations from outside the machine like those emanating from a motor boat passing overhead. Indeed, when you are outside looking in, the world is rarely compositional. The whole is rarely a partition induced on the mereological sum of its individual parts. Compositionality has too often been used as a convenient fiction to make the mathematics work out neatly.

1.1.5 Safety and Liveness

In the section that follows, I provide a top-level perspective on the *safety* of high-assurance systems with respect to *finite* models. Safety concerns whether or not an assembly that initially operates correctly will *always* do so. Only occasionally will I be alluding to *liveness* properties like progress, fairness, and efficiency. Liveness questions concern whether an orchestration *eventually* services all requests fairly and efficiently, or whether the turbine farm over a one-year period will *eventually* attain some aggregate uptime target. Such questions of liveness ultimately concern Quality of Service (QOS) and are left for future work. Indeed, the notions of *always* and *eventually* have mathematically precise characterizations in temporal logic, which is used in software tools known as *model checkers*. Yet, for a model checker to exhaustively verify that safety and liveness properties hold, the model of the system being checked must be *finite* and, in a practical sense, tractable. The following section outlines some issues and approaches to assuring system safety.

1.2 Issues and Approaches

Each chapter has a three to five word motif that names a scientific issue or engineering approach. Hence, each of the following sections previews the content of the chapters that follow.

1.2.1 Test Versus Verification

In [122] we specify a number of decision criteria that guide whether to formally and exhaustively verify some design or whether to informally and non-exhaustively test. The context of that paper was implementation of various Service-Oriented Architectures (SOA) subject to external and internal tradeoffs. External (mission) tradeoffs between assurance, performance, and flexibility were paired up to characterize safety-critical, fiscally-critical, and entertainment-critical systems. We then matched each of these classes of systems to internal (technical) tradeoffs that pertain to the feasibility of verification versus testing. Internal tradeoffs between assurance, scale, and detail became the technical drivers in the decision whether to exhaustively verify or non-exhaustively test. In the process, we determined that only entertainment-critical systems are exempt from exhaustive verification considering their requirements for flexibility and performance at the expense of assurance. For safety- and fiscally-critical systems, we identified the need to decouple high-assurance portions from those that attempt either to maximize the performance of the former, or the flexibility of the latter.

1.2.2 Use Cases for Verification

Given these mission-driven external tradeoffs and technology-driven internal tradeoffs, [119] surveys use cases for model checkers for high-assurance web service orchestrations. We noted how use of model checking tools had evolved to address the

need to verify code that coordinates activities between a distributed collection of web services. The need for architectural features like decoupling of functionalities, use of formal ontologies, automating model capture and presentation, and controlling level of abstraction will further increase the effectiveness of formal verification.

1.2.3 Automating Model Capture

Of the architectural features surveyed in [119], model capture [127] and presentation [121] are to be further examined. Automating model capture for a subset of BPEL, had already been done over the past several years by others. Our contribution is in the generation of understandable and parsimonious models suitable for simulation and model checking that are also extendable with assumptions. Assumptions that impact state space size and hence feasibility of exhaustive verification entail *atomicity*, *synchrony*, and *parallelism*. An initial prototype translator from BPEL to PROMELA – the modeling language used by the Spin model checker – performs translation with respect to these assumptions. In a version of [127] for journal submission we also describe how pessimistic assumptions, like whether a service is fault-prone, would be modeled by the prototype.

1.2.4 Automating Model Presentation

Making a model suitable for human interpretation is the topic of [121]. Using an existing software tool for translating a BPEL artifact into a Petri net, along with our own machine-readable requirements, we created a translator for thence generating

a Colored Petri Net (CPN). The resulting CPN includes notions of data type and hierarchy while improving the layout of the original Petri net. In particular, the prototype partitions the Petri net into a subnet for each web service in addition to the subnet representing the orchestration. It lays out each subnet, colors the places of each subnet with data type, and generates the XML file for the CPN for import into the CPN Tools software package. Our results include depictions of subnets produced and initial simulation results for a well-known case study.

1.2.5 Assuring Timeliness

When considering tangible systems like ocean machinery, we must temper safety requirements of *always* to ones with respect to some specified period of time. Hence, a safety requirement for ocean machinery might stipulate maintenance-free operation over a one-year time period. Another safety requirement might be that the motion of an ocean turbine’s propeller always occurs subject to some minimum rotational velocity. Formally representing and thence verifying such real-valued time constraints requires the use of a timed automaton, as described in [126]. Furthermore, fleet management of ocean turbines requires an SOA infrastructure similar to the e-science volunteer computing infrastructure of our case study in [126], requiring the same timed automata to be used for both simulation and verification. That paper informally outlines a translation of a CPN representing startup/shutdown behavior of units in a fleet, to an application in UPPAAL – a software tool for the simulation

and verification of timed automata. Automating translations into UPPAAL is left for future work.

1.2.6 State Compression and Abstraction

Enhancements to the transition fault models in the previous paragraphs are irrelevant if the otherwise finite model is neither tractable nor at an appropriate level of abstraction. Correcting these shortcomings, we present a formalism [125] for a type of logical fault model known as the *fault tree*. The potential expressive power of fault trees lie in their ability to relate multiple anomalous values of possibly disparate state variables to a smaller more understandable and actionable collection of higher-level faults or failures. Logical fault models like fault trees provide a mechanism for simultaneous state compression and abstraction of system faults. Real time evaluation of a fault tree given system state, results in a marking for a fault transition model like a Petri net. Composition of logical fault models into transition fault models is currently under way.

1.2.7 Case Study – Ocean Turbines

We provided a comprehensive reliability assessment [123] in response to recent interest and funding of an experimental ocean turbine prototype by FAU’s Center for Ocean Energy Technology (COET). This assessment required extensive collaboration from co-authors, since it spanned disciplines outside the scope of my research, most notably in Ocean and Mechanical Engineering. As in most real-world systems, the

topic of any one researcher’s interest comprises only a minute portion of the entire problem space. My research is no exception.

For this case study, much of the previously discussed work can be plugged into four narrowly defined areas in ocean turbine reliability. The first three areas are topics for future work, since the initial prototype is currently under construction. The first area involves deriving both coordination and computational baselines as a means of characterizing what constitutes normal operation. The second involves enhancements to an annunciator panel at some shore side control center, by depicting turbine operation in terms of a timed automata as was done in [126]. The third topic proposes further enhancements to the annunciator panel by depicting the progression of a turbine’s operating state in terms of CPN models as was done in [121]. The fourth area concerns fault trees, which although not considered in the reliability assessment in [123], will provide a fault detection, localization, and explanation capabilities. The fifth and final topic pertains to finite models in preventing biofilm formation. In [123], we identified current research into biomimetic surfaces and propose refinements to the design and fabrication of such self-similar surfaces. Future work in these areas are described in Section 9.2.

1.3 Organization

Understanding how this dissertation is organized requires considering the provenance of each chapter. Unlike more traditional dissertations, this work grew out of a

number of journal articles [121, 122, 123, 126] and conference papers [119, 127]. These journal articles, in turn, grew out of conference papers [11, 120, 124]. Although related to logical fault models in [125], ideas from conference papers [41, 42, 142, 143] for which I was a minor author were not included in this dissertation. Consequently, each chapter can be read as a self-contained unit with each chapter including its own literature review and any mathematical preliminaries. In a note of irony, this dissertation treats the topic of finite safety models as if the topic was compositional!

This dissertation is organized as follows: Chapter 2 examines tradeoffs associated with testing versus formal verification. Chapter 3 identifies use cases for formal verification, particularly model checking, for assuring safety of SOA's. Chapter 4 seeks to automate model capture by defining a translation from BPEL artifacts to finite state models used by the Spin model checker. Chapter 5 automates the construction of a human readable model in a software tool for colored Petri nets, given a classical Petri net that had been generated from a BPEL artifact. Chapter 6 considers real-valued time constraints modeled in timed automata, as opposed to the weaker ordinal-valued time constraints inherent in fair transition systems like classic Petri nets. Chapter 7 considers logical fault models like fault trees, in contrast to the transition fault models treated thus far, by providing a purely set-theoretic formulation. Chapter 8 provides a real world application into which the techniques in the previous chapters may be inserted. In particular, this chapter assesses the reliability of novel ocean systems known as ocean turbines. Finally, Chapter 9 summarizes conclusions

and previews future work.

CHAPTER 2

TEST VERSUS VERIFICATION

Based on the paper titled *Testing and Formal Verification of Service Oriented Architectures* [122], we examine two open engineering problems in the area of testing and formal verification of internet-enabled service oriented architectures (SOA). The first problem involves deciding when to formally and exhaustively verify versus when to informally and non-exhaustively test. The second concerns scalability limitations associated with formal verification, to which we propose a semi-formal technique that uses software agents. Finally, we assess how these findings can improve current software quality assurance practices.

Addressing the first problem, we present and explain two classes of tradeoffs. *External tradeoffs* between assurance, performance, and flexibility are determined by the business needs of each application, whether it be in engineering, commerce, or entertainment. *Internal tradeoffs* between assurance, scale, and level of detail involve the technical challenges of feasibly verifying or testing an SOA. To help decide whether to exhaustively verify or non-exhaustively test, we present and explain these two classes of tradeoffs.

Identifying a middle ground between testing and verification, we propose using software agents to simulate services in a composition. Technologically, this approach has the advantage of assuring the quality of compositions that are too large to exhaustively verify. Operationally, it supports testing these compositions in the laboratory without access to source code or use of network resources of third-party services. We identify and exploit the structural similarities between agents and services, examining how doing so can assure the quality of service compositions.

2.1 Chapter Introduction

In the sections that follow, we describe the motivation, context, and contributions of this chapter.

2.1.1 Motivation

A Service Oriented Architecture (SOA) is an approach to the development, deployment, and evolution of distributed black-box software processes known as *services*. Each service asynchronously communicates either with users or with other services strictly through its standards-compliant interface. Including a service in a composition of interacting services should not require knowledge of exactly how the service was implemented. Glue code can be used to assemble these loosely coupled and reusable services into a *composition*. A composition may be a centrally controlled *orchestration* or a peer-coupled *choreography*.

Although SOA frees developers from concerns over platform, implementation, and versioning, these freedoms preclude test and quality assurance techniques that require access to source code. Unexpected behavior emerging from unforeseen usage scenarios and implicit assumptions may cause race conditions that end in deadlock or other undesired modes of interaction.

2.1.2 Context

Based on our previous work, we identify the circumstances that require agents instead of formal verification tools like *model checkers* to test a composition of services [120]. Once we identified a broad class of use cases for this non-exhaustive mode of testing, we examine how an agent can simulate a service in such a composition [124]. Finally, we assess how this work can impact current quality assurance practices.

To gain contextual insight, we first introduce the most common form of SOA, namely web services, by discussing their standards, their development, and how they are tested. An artifact written in the Web Services Description Language (WSDL) defines the interface between public and private sides of a web service [28]. An artifact written in the Web Services Business Process Execution Language (BPEL) implements the composition as a centrally controlled orchestration [4]. Similarly, an artifact written in the Web Services Choreography Description Language (WS-CDL) implements the composition as a peer-coupled choreography [96].

Development and quality assurance of a web service orchestration follows the pattern shown in Figure 2.1. In Step 1, service providers register their services with a web service syndicator, providing an interface specification like a WSDL artifact. This step makes these artifacts available to a possibly distinct group of application builders known as solution providers. In Step 2, solution providers consult this registry to compose web services into an orchestration. Testing in Step 3 can either be black-box testing [24], or a process of formal verification [78] like *model checking*. Finally in Step 4, orchestrations free of detected faults are deployed. Otherwise, Steps 2 and 3 are repeated with either a debugged composition or different but equivalent services.

2.1.3 Contributions

We identify both external and internal sets of tradeoffs that involve quality assurance, in terms of a mirrored pair of triaxial charts. External enterprise-driven tradeoffs between assurance, performance, and flexibility involve three classes of enterprise that include engineering, commerce, and entertainment. These tradeoffs occupy the left-hand portion of Figure 2.2. A typical web service composition in engineering or commerce has only a *portion* of its services that are either safety or fiscally critical. From an enterprise standpoint, we must assure that non-critical services do not interfere with the operation of critical ones.

Although critical portions of a composition must be exhaustively verified, doing so for the entire composition may not be practical. To understand what is feasible,

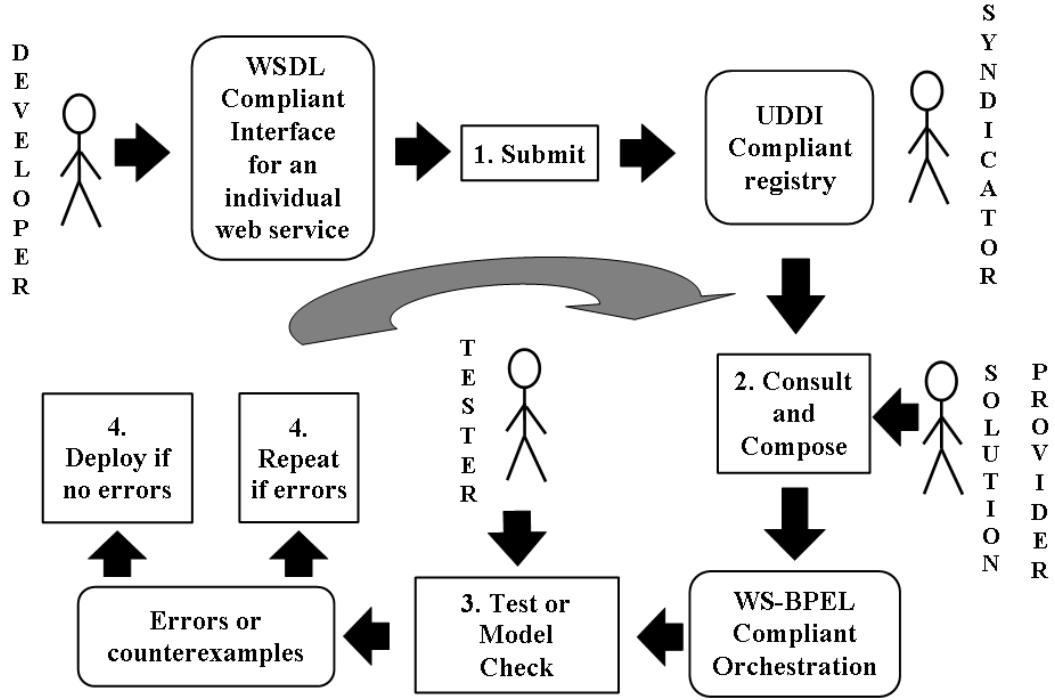


Figure 2.1: Assuring Quality of Orchestrations

we identify internal technology-driven tradeoffs between degree of assurance, scale, and level of detail. These tradeoffs appear on the right-hand side of Figure 2.2. Model checking is a high-assurance technique that exhaustively verifies compositions. It automatically lists every combination of interactions, checking each combination for violations of specified properties. Such a listing can get intractably large. From a technology standpoint, state space size drives the choice between exhaustive verification and non-exhaustive forms of testing.

We identified a technique that occupies a middle ground between exhaustive

verification and non-exhaustive forms of testing. For intractably large compositions comprised of individually machine-verified services, we propose simulating each service as a software agent. The behavior of each agent is driven by the model previously used to formally verify it. In all cases, each agent must use some form of *assume-guarantee* reasoning. Since we wish to implement reliable composition, we chose a *two-phase commit* interaction discipline. We describe how this may be done using existing web services standards, illustrating these ideas with a simple case study that involves composing a set of web service compositions.

The rest of this chapter is organized as follows: Section 2.2 discusses the tradeoffs that govern choice of quality assurance strategy. Section 2.3 presents a semi-formal technique for testing web service orchestrations using software agents. Section 2.4 examines technological impact of the proposed decision strategy and testing method, concluding with Section 2.5.

2.2 Tradeoffs

This section first explores the use of triaxial charts to represent tradeoffs between mutually conflicting goals or choices. We then describe a mirrored pair of triaxial charts in Figure 2.2. The left-hand side shows external business-driven goals commonly associated with *validation*. The right-hand side denotes internal technological choices associated with *verification*. Quality assurance is the one goal in common between the two sides. *External tradeoffs* between assurance, performance,

and flexibility are determined by the business needs of each application, whether it be in engineering, commerce, or entertainment. *Internal tradeoffs* between assurance, scale, and level of detail involve the technical challenges of feasibly verifying or testing an SOA. To help decide whether to exhaustively verify or non-exhaustively test, this section presents and explains these two classes of tradeoffs.

2.2.1 Triaxial Charts

Triaxial charts like those in Figure 2.2 can depict phenomena involving proportions of three items. They are used in fields as diverse as soil geology [32] and medicine [62]. Each point inside the chart, is associated with some unique stochastic vector. For example, a point equidistant from three vertices will be vector $\langle 1/3, 1/3, 1/3 \rangle$, while a point on one of its three vertices will have its coordinate equal to 1 and the remaining two coordinates equal to zero. Likewise, the locus of points along the face opposite any vertex will each have its coordinate for that vertex equal to zero.

The pair of triaxial charts in Figure 2.2 depict two sets of tradeoffs. The left-hand portion shows external tradeoffs such as the need for performance versus assurance versus flexibility. These are dictated by external market and societal forces. The right-hand portion shows internal (technological) tradeoffs such as scale versus assurance versus level of detail. These are determined by the properties inherent in the composition of services.

The choice between test and exhaustive verification on the right hand side of Figure 2.2 will depend on the purpose of the orchestration on the left hand side. An orchestration that is meant to entertain shown on the left hand side may only need testing, while orchestrations that involve money or public safety may require exhaustive verification.

This choice may not always be clear-cut. For example, not all portions of e-commerce or e-science applications require exhaustive verification. A web service for credit card processing may need formal verification, but a web service that recommends the purchase of related products may not. Nonetheless, an orchestration that includes both credit card processing and product recommendation, each of which is shown in Figure 2.3, needs to formally verify that the recommender service does not interfere with operation of the credit card processing service. The next section will lend insight into what externally drives the required degree of quality assurance.

2.2.2 External Tradeoffs

These are driven by the enterprise needs of each application domain, be it in engineering, commerce, or entertainment as shown in Figure 2.3. Tradeoffs inherent to each domain can be described by some colloquial catch phrase that can be couched in the negative. The subsections that follow describe the tradeoffs inherent to each application domain.



2.2.2.1 Engineering

Engineering applications must assure safe and timely operation with the catchphrase “.. *no surprises*”. By “no surprises”, engineering managers eschew the need for having to postpone until run time the choice of what specific services belong in a composition. Implied here is the need to assess both error-free operation and performance of a composition *prior* to its deployment. The paragraphs that follow, describe examples of engineering applications that appear in Figure 2.3.

Grid applications, by nature, are distributed and subject to the vagaries of

their network infrastructure. Deploying such applications in an SOA involves striking a balance between error-free operation and performance. An error-prone grid-enabled service need not produce an incorrect answer as much as to be unavailable. By signaling to its orchestration, the fact that its computation is taking longer than expected, the service provides its orchestration with the opportunity to modify its behavior at run time. This may involve either partitioning the problem and delegating it to multiple service instances, or to assign the problem to a faster but otherwise identical service. Since this process requires some degree of run time flexibility, Figure 2.3 places grid applications toward the center of the figure.

Volunteer supercomputing infrastructures like the Berkeley Open Infrastructure for Network Computing (BOINC) offer a different perspective on the tradeoff between performance and assurance. As the deployment platform for a large number of e-Science portals such as Climateprediction.net, Einstein@home, and the ever-popular SETI@home, BOINC sets out to solve the type of problem that can be partitioned into myriads of subproblems, each of which can be solved on an Internet-enabled personal computer (PC) [6].

Its service providers are myriads of otherwise idle home PC's, each located a mile beyond the isolated fringes of an unreliable Internet². A typical provider at any one time can be temporarily put out of service for a variety of reasons. Such events are often of little consequence to the BOINC server since each provides but one service –

² Known as the *Last Mile Problem*, this refers to the performance and reliability degradation associated with delivering broadband services over its final leg to the home.

raw computing power. So when providers register with an e-Science portal, the only questions asked are its operating system platform and scheduling preferences.

As an SOA, each service provider is equipped with a service artifact known as a BOINC client. The BOINC client focuses on high performance by dedicating the resources of its PC once the PC goes into screen saver mode. Many problems can arise over the last mile of the Internet, so the BOINC client occupies its place near the performance vertex but far from the assurance vertex in Figure 2.3.

The BOINC server, on the other hand focuses more on error-free operation than on performance. Indeed, BOINC work units are batched, with no expectation by the research community of real time response. Much of the logic in the BOINC server is dedicated to spanning the gap between performance and assurance. In one strategy, the server sends the same work unit to three clients, with the first two returning the same answer being credited with providing the correct answer. Consequently, these two clients will be favored to receive subsequent work units. Thus, we place the BOINC server closer to the assurance vertex than to the performance vertex in Figure 2.3.

2.2.2.2 Commercial

Commercial web service applications must assure sound fiscal management in a changing marketplace. Corporate managers are not expressing false humility when they use phrases like “.. *this is not rocket science*”. Instead, they are eschewing

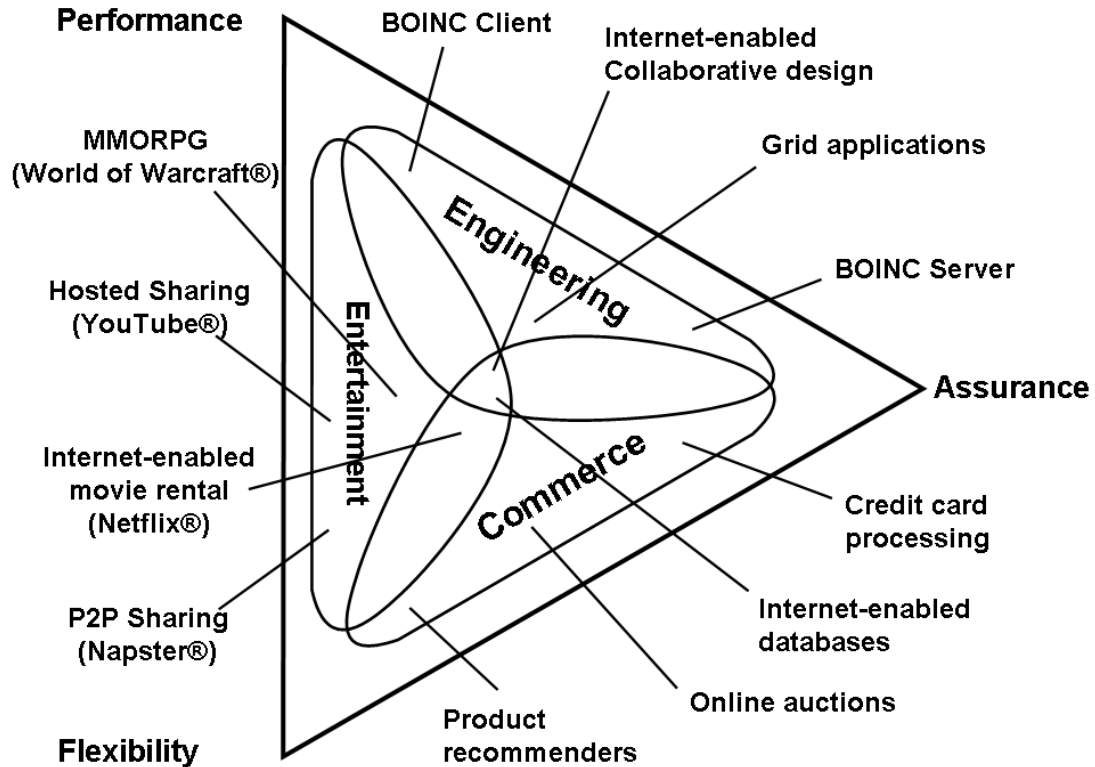


Figure 2.3: External Tradeoffs

the need for high performance services, if doing so means a higher failure rate or an inability to adapt to changing markets. In business, failure rate equates to losing customers or money, where an inability to adapt, equates to having to spend money to *replace* rather than *upgrade*.

Corporate managers know better than anyone else, that the race is not to the swift, that being second to market gives them the advantage of hindsight, and that being averse to risk is to favor quality assurance. Time-to-market and just-in-time operational constraints are not as much about run time performance, as they are

about reliably meeting customer expectations. Meeting these expectations entails a good measure of quality assurance and flexibility, even if it involves sacrificing some degree of performance.

In addition to the credit card processing that depends on error-free operation, and product recommenders that depend on flexibility, online auctions in Figure 2.3 occupy a place in between. Auctions involve participants in financial transactions, thus requiring a higher level of assurance than for recommenders. An auction service by itself does not perform the actual exchange of funds and goods between buyer and seller. Instead, an auction service might bundle in a credit card processing service along with a shipping service as part of its web service composition. Thus, the point for online auctions in Figure 2.3 can be considered the centroid in a constellation of points, each of which is an interacting web service.

2.2.2.3 Entertainment

Entertainment applications assert that the show must go on with the catch phrase “.. *despite the glitches*”. Directors and producers would rather compromise error-free operation than to diminish an audience’s expectation of surprise and immediacy. Varying degrees of flexibility and performance are needed to fulfill these expectations. Figure 2.3 shows some examples, which we describe in the paragraphs that follow ³.

³ Trade names including Napster, YouTube, Macromedia Flash, World of Warcraft, Netflix, and iPod are all registered trademarks of their respective holders.

Peer-to-peer (P2P) file sharing, pioneered by Napster, aimed for flexibility at the expense of performance and error-free operation. Its major use involved exchange of sound files for music sharing between Internet-enabled PC's. Most often the PC's for both peers occupied the low-performance and low-reliability fringe of the Internet. Once downloaded, the file in MP3 format, may be replayed many times, or exchanged with yet other peers. Despite its asynchrony and ad hoc nature, Napster and its successors satisfies the public's desire for discovering, collecting, and enjoying artifacts of entertainment value. Rather than tap into some global *halo* of on-demand music and entertainment this practice of collecting such artifacts has persisted to this day with the iPod and other albeit *network-hosted* devices.

Unlike P2P networks, YouTube uses a centralized or network-hosted form of video file sharing, supporting a greater degree of performance, enabling a user to play the video using Macromedia Flash animation on demand. Judging by its heavy use by the public, the surprise and sense of immediacy in these videos outweigh their low resolution, low sound quality, and high lag.

2.2.3 Internal Tradeoffs

Internal tradeoffs pertain to the feasibility of testing an SOA to a level of assurance appropriate to each of these three classes of enterprise. This section describes how testing and model checking observes *internal* tradeoffs between assurance, scale, and level of detail shown in Figure 2.4.

Due to the state space explosion problem, model checking can only be used for compositions that are either sufficiently small (low scale) or sufficiently abstract (low detail). Most web service compositions occupy some point between scale and detail. In any event, these compositions must be model checked if technically and operationally feasible.

2.2.3.1 Tool Placement

Each model checking tool is adapted to some degree of scale versus level of detail. Three model checkers with freely available binaries that also appear in the web services literature are shown in Figure 2.4. If we were budgeted with a certain number of machine cycles for model checking, each tool will feasibly model check to ranges of all three variables shown in Figure 2.4 as shaded-in areas.

The Labeled Transition System Analyser (LTSA) [48] abstracts away the notion of communication channels by enabling user-specification of message sequence charts (MSC) to model orchestrations. The MSC formalism is easily extended to model checking choreographies [49], a more macro-scale mode of composition. At its core, however, LTSA supports a lower-level process algebraic formalism known as Finite State Process (FSP) notation.

The Spin model checker [66] by far appears most frequently in the SOA verification literature. Both it and LTSA-FSP express interaction in terms of channels. Both however, abstract away the ability to specify real-valued time constraints. The

UPPAAL verification environment [13] supports the most detailed level of modeling, enabling the user to specify real-valued timing constraints on each place and channel in their timed automata models.

Finally, we denote the realm of both traditional and agent-based testing techniques, which for the budgeted number of cycles, has profound limitations on degree of assurance. The wide variety of these techniques however enable test generation from either abstractly stated use cases [43, 136], or other means [23].

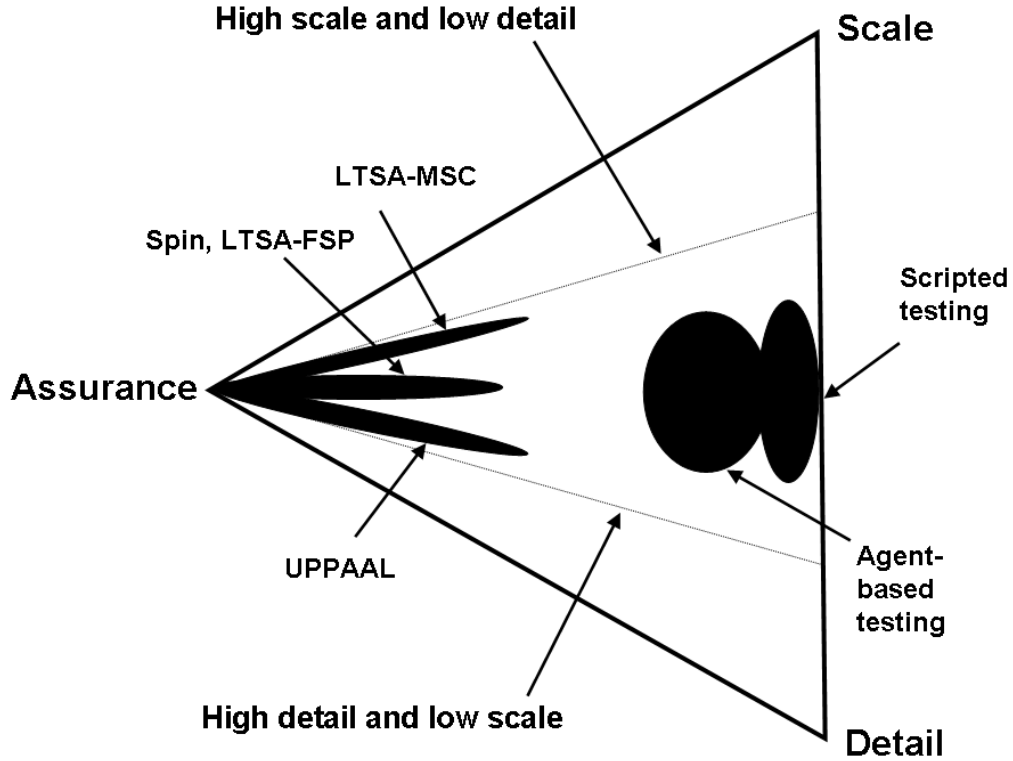


Figure 2.4: Internal Tradeoffs

2.2.3.2 Interpreting the Landscape

Exploring the space inside Figure 2.4, we see how assurance is compromised at the expense of increasing scale or level of detail. Toy problems, used in the teaching of model checking techniques, reside close to the assurance vertex. Such problems are useful when affecting comparisons between model checking tools [103]. As problems become increasingly complex, as when moving from orchestrations to choreographies, so does the need for an abstraction that sacrifices some level of detail. This is not without some risk of *false negative* errors. Suppose we make the apparently reasonable assumption that the middleware underlying an orchestration operates flawlessly. Abstracting away the threading configuration of servlet containers residing in the middleware, as was done in [47], resulted in an observable deadlock, even though model checking on the abstracted model did not detect such a violation of safety properties.

In our discussion of engineering applications, specifically grid computing, we briefly alluded to the use of timers. BOINC and many, if not most, practical SOA applications in all three application domains also make use of timers. On an unreliable network, timers can be indispensable to assure the progress of an activity. Both Spin and LTSA use untimed models that abstract away timers, consequently enabling them to model check larger scale compositions than timed model checkers like UPPAAL [79]. This abstraction may result in a *false positive* error in which Spin or LTSA would detect a deadlock that otherwise could have been resolved in real life

(or in UPPAAL) by expiration of timers. Thus, we see that increasing scale while decreasing detail may lead to both false negative and false positive errors.

2.2.4 Section Critique

We depicted the tradeoffs that govern our choice of quality assurance strategy using triaxial charts. Assurance is the only vertex in common between the charts in Figure 2.2, so we establish a correspondence between application domain and quality assurance technique on the basis of assurance only. We do not imply any correspondence between Performance and Scalability, or between Flexibility and Detail.

The choice between test and exhaustive verification on the right hand side of Figure 2.2 will depend on the purpose of the service composition on the left hand side. Not surprisingly, compositions that are meant to entertain shown in Figure 2.3 occupy the side opposite the Assurance vertex of external tradeoffs. For these systems, an appropriate level of quality assurance corresponds to the side opposite the Assurance vertex of internal tradeoffs shown in Figure 2.4. Thus, an entertainment application may only require non-exhaustive testing.

Compositions that are either fiscally-critical or safety-critical occupy the portions of Figure 2.2 nearer the Assurance vertex, and will require some means of formal verification like model checking. Real-life service compositions have portions that require high assurance, and other portions that don't. When formally verifying the overall composition, care must be taken to avoid having the non-critical portions

interfere with the operation of any of the critical portions. This suggests an asymmetric approach to model checking that may facilitate exhaustive verification of larger scaled or more detailed compositions than the (present) symmetric approach. Further exploration of this approach is reserved for future work.

The notion of quality assurance may itself differ between enterprise and technological perspectives. From the standpoint of an enterprise, assurance may be measured externally using probabilistic indicators like mean time to failure. For complex engineering systems involving machinery, or commercial systems involving financial or distribution networks, such probabilistic measures may be appropriate. From the technological perspective, assurance can be measured discretely in terms of finite state models. If such a model can be constructed and checked exhaustively then one may be tempted to assert that the composition is error free. This would depend on some strong assumptions, one being that individual services in a composition are themselves error free. As models get larger and we have to sacrifice detail for a higher level of abstraction, this assumption becomes less tenable. For arbitrarily complex compositions, even an exhaustively verified yet highly abstracted model becomes less useful for finding faults than would certain non-exhaustive forms of testing. The next section describes a semi-formal approach to non-exhaustive testing using algebraically-guided software agents.

2.3 Simulating Web Services with Agents

Motivated by the rise of the Internet and e-business, development of web services focuses on cost-effective programming-in-the-large of globally interconnected and precisely orchestrated applications. Such development requires widespread attention to standards compliance, reuse, and dependability. Extensive use of third-party services, however, poses unique quality assurance challenges, many of which have been described by Canfora and Di Penta [23]. Opacity of third-party services necessitates black box testing. Residing in a foreign infrastructure, performance and even correctness of web services can be subject to load from other users, network congestion, and choice of middleware. By leveraging best practices like modularity and abstraction, web service standards like BPEL and WSDL enhance the flexibility, reuse, and interoperability of web service orchestrations.

Another technology, one important to robotics and ubiquitous computing, is software agents and agent oriented software engineering. As an outgrowth of artificial intelligence research [145], Agent Oriented System (AOS) technology can also be used to test orchestrations of web services. With notable exceptions [117], little work relates AOS with web service orchestrations, particularly in how to test orchestrations in which each service is an agent.

This section outlines how each service may be simulated by an agent, with the orchestration under test supporting interactions between these agents. Imbuing each service with externally observable behavior that appears uniform, like two-phase

commit, orchestrations can be made more reliable, but at the expense of performance and flexibility.

2.3.1 Web Service Orchestrations

As a software development and distribution paradigm, service oriented architectures (SOA) support registration, discovery, composition, and subsequent requests for services over a network. As its most prominent implementation, web services additionally involve development of standards-compliant artifacts deployed over the web. Figure 2.5 summarizes salient architectural features of a web service orchestration. The center of this figure shows the BPEL artifact that can orchestrate up to four types of web service. These include legacy systems, third party web services, in-house developed web services, and possibly other orchestrations. Each of the four types of web service can be implemented by an agent. Each agent can then simulate a web service so that we may test how well *glue language* artifacts written in BPEL can coordinate their interactions. If BPEL and WSDL supported more robust forms of compositionality, then the entire composition in Figure 2.5 could have its own WSDL interface. Making this composition part of some larger composition still presents a number of challenges [112, 129].

More generally, web service languages have a number of shortcomings as simulation languages. A Web service orchestration is built from WSDL artifacts, each of

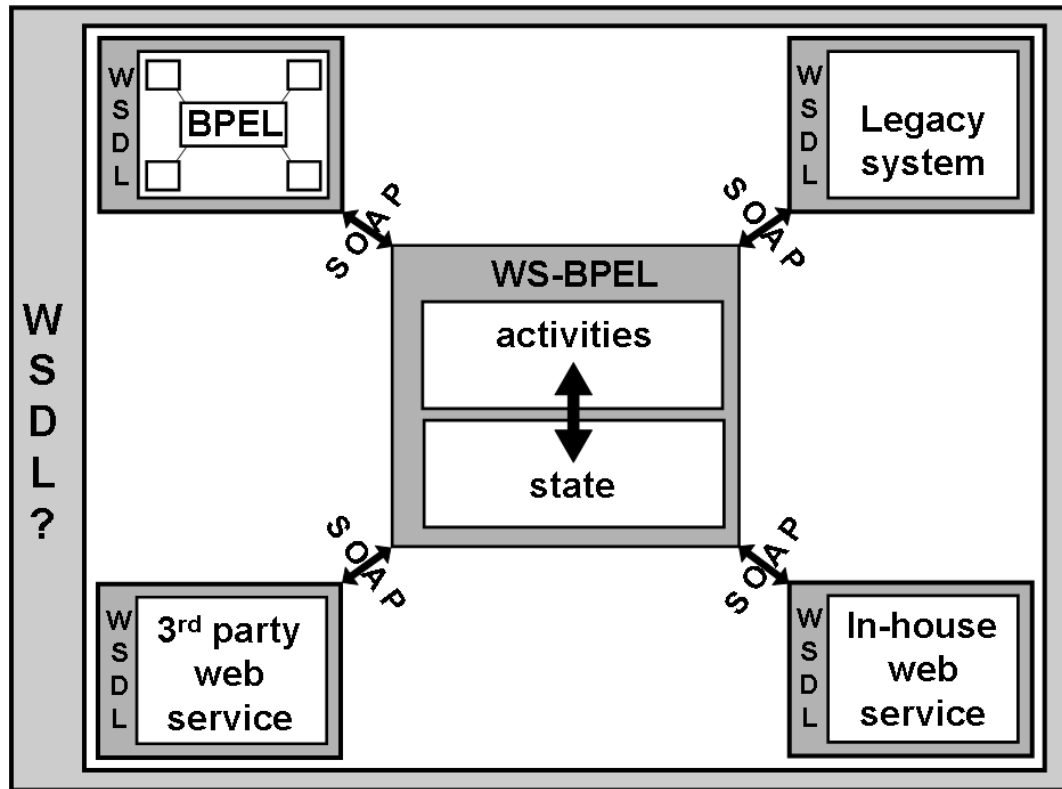


Figure 2.5: A Web Service Orchestration

which provides an interface to its individual web service. The machine-readable portions of WSDL artifacts include type and number of input/output messages, but not their usage context. BPEL artifacts describe the activities, control flows, and states of the composition, while Simple Object Access Protocol (SOAP) messages form the units of information exchange between each service and its BPEL artifact. None of these three types of artifacts describe the contexts under which certain types and numbers of SOAP messages are to be expected or produced. To simulate a web service with an agent a formal specification, one that may require some ontology, will be

needed to unambiguously specify these contexts. Ontologies themselves can get unwieldy without some set of behavioral norms supporting a form of *assume-guarantee* reasoning. Since many web service compositions set out to implement reliable *transactions*, the two-phase commit provides a suitable basis for assume-guarantee reasoning.

2.3.2 Anatomy of an Agent

Agents are defined by Wooldridge [146], and later echoed by Weiß [145] as:
”.. a[n encapsulated] computer system that is situated in some environment, that is capable of [flexible,] autonomous action in this environment in order to meet its design objectives.”

Like web services, agents do not necessarily originate from the same developer nor follow the same internal development methodology.

Based on formal definitions of agents appearing in [58] and [89], Figure 2.6 shows a typical agent that can process a transaction using a two-phase commit. The agent processes inputs and outputs according to some formal specifications included in either a Web Ontology Language for Semantic Web Services (OWL-S) process model [132], or in a Web services Description Language for Semantics (WSDL-S) interface specification [2]. A walkthrough will illustrate why the agent concept is architecturally suited to simulating a web service.

First, the agent filters input based on some invocation condition ι which defines the input portion of an interface describing how the agent may be influenced by system-wide *social* state. Our example requires that the agent be receptive to either a *Start*, *Commit*, or *Rollback* type signal and nothing else. The agent ignores any

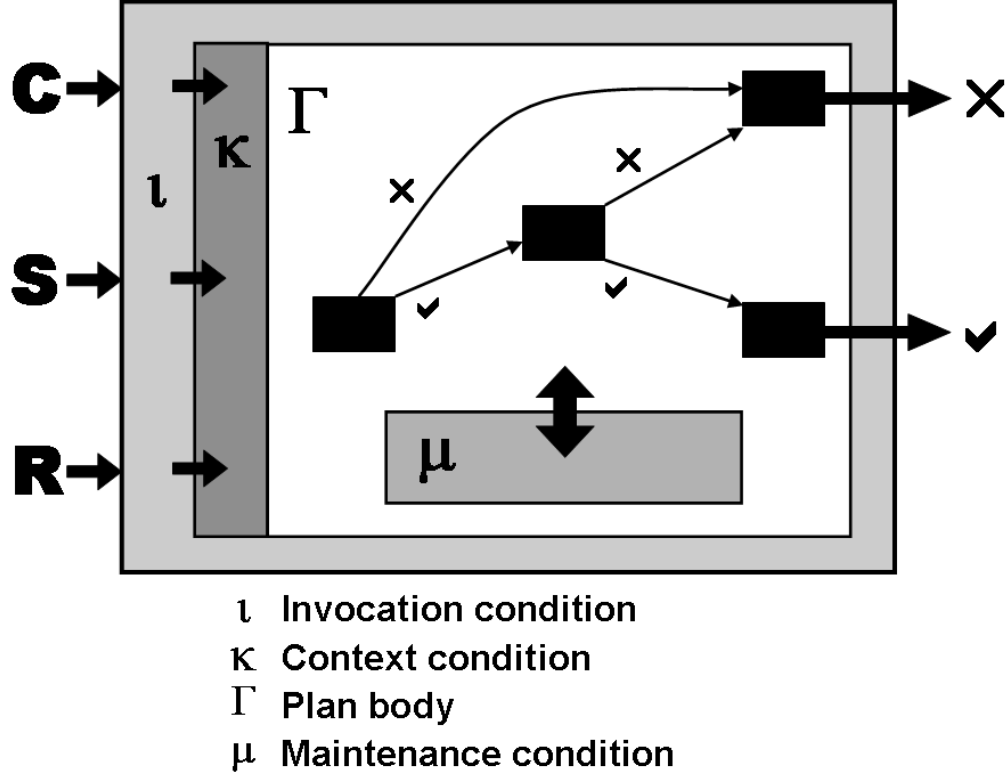


Figure 2.6: An Agent with Two-phase Commit

other type of signal, presuming that the signal was not intended for it. If a signal satisfies the invocation condition, it becomes subject to context condition κ that influences what some authors refer to as the *mental* state of the agent.

Context condition κ indicates whether the agent is ready to process a particular type of signal at a particular time (hence, context) and can be governed by a formal specification involving modal operators *always* and *eventually* from temporal logic.

Informally, κ for a two-phase commit might state:

The agent *always* receives exactly one start signal S , followed *eventually* by emitting either a tick \checkmark or a cross \times signal, unless pre-empted by

receiving a commit C or rollback R signal which it will *eventually* receive prior to receiving the next start signal.

The \surd or \times emanates from nodes within the computation graph (plan body) Γ that was activated by κ indicating success or failure respectively. These two nodes determine the manner in which the agent may influence system-wide *social* state. Maintenance condition μ specifies execution-time invariants. If during runtime, any node in Γ produces a result that violates μ , a \times will be emitted from a node and thence from the agent. In a two-phase commit, it is conceivable that an individual agent may fail, yet later receive a commit signal, since some orchestrations may only require that some of its services succeed. Conversely, an individual agent may succeed, yet later receive a rollback signal due to the failure of some other agent, the success of which was necessary for the process instance (and hence transaction) to succeed.

In a hierarchical composition of web services, each node in Γ could have been an agent that simulates a web service, making plan body Γ an agent-oriented abstraction representing a web service orchestration. In this case, each agent in Γ can be glued together by BPEL code, where the BPEL code performs *coordination*, while each individual agent can be regarded as performing black-box *computation*.

2.3.3 Mapping between Agents and Services

Agents are architecturally suited to simulating services, since both use modularity and abstraction to facilitate design, improve agility, and foster cooperation. Both exhibit structural similarities. Here we describe how agents can be used for this

simulation.

Testing an orchestration involves encoding the behavior of each constituent service into its own intelligent and autonomous agent. As the successor to the DARPA Agent Markup Language for Semantics (DAML-S), OWL-S specifications provide a common nexus between web services on the one hand [135] and agents on the other [91, 132, 135]. OWL-S supports three distinct tasks: (i) discovery, (ii) invocation, and (iii) composition and interoperation [91]. By focusing on (iii), we will describe in Section 2.3.4 how an orchestration of services in an online travel agency behaves.

Figure 2.7 maps each portion of an agent to corresponding portions of a web service. Invocation condition ι in an agent maps to a WSDL specification of a web service. The presence of a sufficient number of messages of the required types at the interface of a service will be required for invocation. A one-to-one relation between an OWL-S *grounding* and a WSDL artifact can be defined [132]. The extension of the WSDL Standard known as WSDL-S [2], may provide an alternative to OWL-S. Finally, an agent's context condition κ can map to the web service's state variables and the logic for setting them.

Strengthening the context condition in the agent will suggest how to do the same in the web service artifact, whether in the wrapper code that bulletproofs individual web service or in the code that implements the orchestration. The net effect will be to reduce the latitude of what messages and permissible message interleaving the agent or its service will be expected to respond to. Including such conditions in

the BPEL artifact, can reduce the size of its state space, making model checking of successively larger compositions more feasible. Section 2.2 explored this and other quality assurance tradeoffs in greater detail.

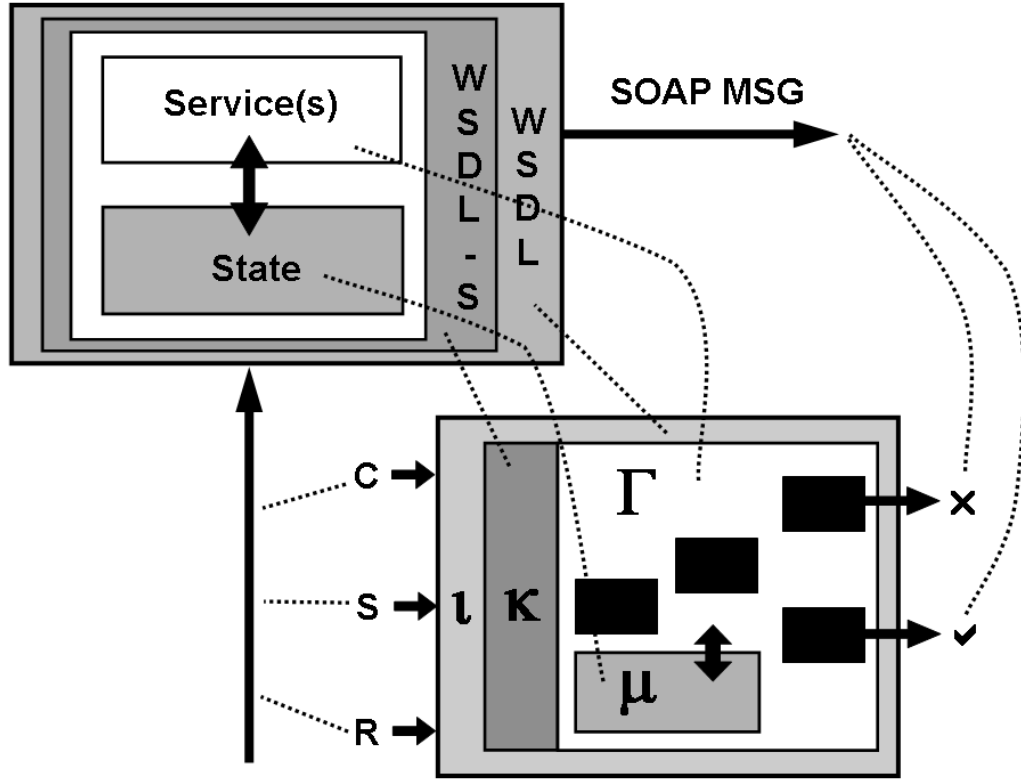


Figure 2.7: A Mapping between Agents and Services

Formulation of κ , Γ , and μ by far pose the most formidable challenges to simulating third-party web services when the source artifacts are not available. For κ , this task is mitigated by improved tool support for specification and formal verification using the model checker BLAST [70], Spin and NuSMV [152], and various testing tools and techniques [71, 135]. Research into advanced process mining techniques

from event logs [34] may make specifying and verifying plan body Γ easier. Finally, messages to and from an agent can be encoded as SOAP messages using an OWL-S *grounding* that can imbue WSDL artifacts with a machine-processable semantics. The discipline of always emanating either success or failure messages from agents as suggested by [89], should be made into a best practice when bulletproofing a web service.

Smaller or highly abstracted compositions of web service artifacts can be model checked prior to deployment. Formal verification of larger or more detailed compositions, however, will eventually encounter the state space explosion problem. These compositions, especially ones that change during run time, as in *Weakly closed* systems [20], may require testing by an agent simulating a web service. Each agent can be coded to behave according to its formal specification but must be isolated so that no two agents internally share either state or computing resources. If event logs for previous or similar orchestrations are available, usage scenarios may be ranked by frequency of occurrence, and can be simulated by the various agents that simulate web services. Although no computational method, formal or otherwise, can exhaustively verify arbitrarily large orchestrations, this *situated* agent approach improves the likelihood that fault detection will remain a step ahead of runtime failures involving race conditions, deadlock, or unspecified receptions.

2.3.4 Travel Agency Case Study

Using the correspondence described previously, we present a case study that examines how this approach can both assure quality of service compositions, and suggest reliability improvements to individual web services. The case study in Figure 2.8 illustrates an on-line travel agency that orchestrates activities between users, airlines, and hotels. A user initiates a transaction by submitting a query to the travel agent via its \surd channel (1). An agent simulating a user generates SOAP messages that could have been mined from event logs of similar sites that exercise this or similar orchestrations. The travel agent reformulates and relays the query to the flight and hotel services through its \surd channels (2). The glue code connecting the three services is modeled and coded as a BPEL artifact. The flight service emanates its results through its \surd channel (3) if at least one available flight had been found. Otherwise it emanates an error through its \times channel (4) rolling back the pending transaction from services awaiting either a \surd or \times . The hotel service emanates its results through its \surd channel (3). If the flight and optionally the hotel services emanated their results through their \surd channels (3), then all services will receive a signal through their commit channels (6). The user is presented with flight information and any available hotel information through its commit port. If, for whatever reason, the user is not satisfied with the results returned, the user rolls back the pending transaction through \times channels (7) and (8). Agents simulating flight or hotel services may access a database with the same schema as the production version, if known. This database

could contain just enough instances to exercise usage scenarios encountered during production processing.

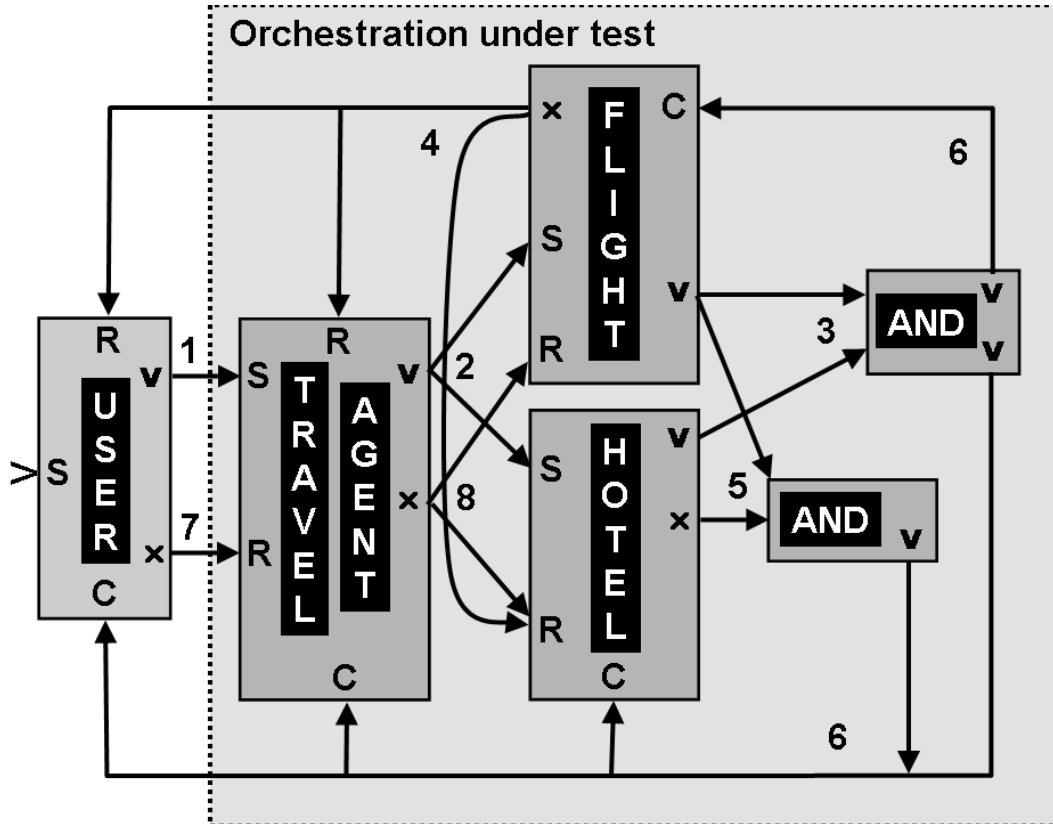


Figure 2.8: Query phase to an Online Travel Agency

Notice that an unspecified reception occurs when a hotel was found but a flight was not, in which case the hotel \vee channel (3) is enabled but not the flight's \vee channel (3). Worse yet, the system is in an inconsistent state since not all services have been rolled back. In particular, the hotel service received neither a commit nor rollback signal. One can modify the orchestration by connecting a \times channel (4) to the R port of the hotel service.

In our example, the hotel service (or agent) emanates a \times which simply means there are no matching records in the database. This example assumes that any service emanating a \times will roll itself back to its state prior to receiving the start signal for this query. Additionally, a snapshot of this error state can be preserved as is currently supported by BPEL. Care must be taken to prevent a rollback from erroneously making this service available for more transactions, unless the service is in some usable state. This case study provides a motivating example of where formal machine-processable specifications are needed to constrain interaction among a society of software agents responsible for completing a transaction [151].

In systems with sufficient redundancy, a faulty service should remain in its error state indefinitely pending either an explicit rollback signal or a more thoroughly debugged replacement. Such redundancy can be modeled as timed automata and implemented using timeouts [126]. Alternatively, this dynamic replacement can be modeled by an agent that functions as an interpreter for expressions in the *Pi Calculus*. LTSA-FSP does support a limited form of dynamic replacement that requires a listing of each possible configuration. In such a setup, each agent is a *tabula rasa* that receives its behavioral specifications from the orchestration engine in the form of a process algebraic specification. From thence forward, the agent behaves according to its specification until it receives another expression that will modify its behavior. This runtime service substitution can be done by a portion of the simulation environment. Expressions in this substitution should be *weak bisimulation equivalent* rather

than *identical*, to model the substitution of approximately equal and presumably interchangeable services.

Use of agents to discover *sufficiently equivalent* services was described in [69] where OWL-S specifications supplement the service's entries in the Universal Description Discovery and Integration (UDDI) registry. The technique of *bisimulation equivalence* to detect sufficiently equivalent web services has also been proposed in [78]. To illustrate this notion of *close enough*, suppose the hotel service in our case study is itself an orchestration of competing or nation-specific implementations of web services. Each such implementation may involve slight differences in message construction and behavior but all must provide the information appropriate to the transaction using the same interaction pattern.

The case study assumes the need to roll back the transaction for all services. In long-running transactions, as in case management, localized commits or rollbacks may be necessary to minimize cascaded rollback. The case study hints at this need when considering only the querying phase of making a reservation. The rollback through channels (7) and (8) may have either been user-initiated or through a *deferred choice* workflow construct [113], initiated at the expiration of a 24 hour timer. In other words, any query results not followed up on, will be flushed from the system after 24 hours, which corresponds to standard industry practice. In the meantime, the user can embark on the second phase of reservations that may involve comparing prices and convenience of similar itineraries, which will reset the system timer. The second phase

will provide its own scope of commit and rollback, as would phase three involving confirmation and payment. One may proceed through the first two phases, and initiate but cancel the third phase by choosing another payment method. Cancelling the third step simply to change payment method should not roll back results of the previous two steps. Sequentially composing compositions of web services into distinct *phases*, with each constituent composition made up of web services that observe the two-phase commit discipline, enables localizing commits or rollbacks to each phase.

From an operational standpoint, case studies like this can be implemented in the laboratory since database size, user load, middleware intricacy, and network unreliability have been abstracted away. By simulating each service with an agent, and executing these transactions at a rate substantially faster than that in live operation will likely detect many coordination faults before they materialize into failures. By separating computation from coordination, one may use agents to test coordination requiring fewer computational resources and less knowledge of the inner workings of any services being simulated [64].

Some legacy systems do not follow this two-phase commit discipline. Encapsulating such a system inside a service, (like that shown in Figure 2.5), that enforces this two-phase commit discipline can be prototyped as an agent. The specific way in which this can be done depends on which of the five message types are not supported by the legacy system. For example, a legacy system emitting only positive responses, will require the service to implement a *deferred choice* workflow construct to emit

a \times upon expiration of a timer. More often, legacy systems do not support some form of compensation handling involving commit or rollback, even though the BPEL standard readily supports this. Adding exception and compensation handling into the orchestration without propagating the change to the offending service may leave that service in a state inconsistent with the rest of the composition.

2.3.5 Section Critique

This section defined both web service orchestrations and Belief Desire Intention (BDI) type software agents. We mapped each part of an agent to the corresponding part of the web service being simulated. Using a case study, we described a means of testing an orchestration by simulating each service using an agent.

Testing an orchestration using an agent at each service not only improves the BPEL artifact under test, but also points to reliability improvements in each service being simulated by an agent. Such testing requires use of a suitable type of agent (i.e., BDI), implemented in a suitable ontology (i.e., OWL-S), using a suitable interaction discipline (i.e., 2-phase commit). The following paragraphs temper this sanguine view of suitability.

The precise role of ontology standards in web services is still unclear. Since a preponderance of web service literature emphasizes web service standards, we considered the use extensions to existing standards like the WSDL-S extension to WSDL. It may still be too early to determine whether *replacement* of web service standards by

OWL-S or *extension* of existing standards with ontologies will become the prevailing practice.

Our advocacy of the two-phase commit as a best practice for all web services may also be disputed. To encapsulate a legacy system as a web service that realizes a two-phase commit can first be modeled and tested by our proposed simulation technique. Although fault tolerance of the resulting composition may be improved, getting the service to support all five message types associated with a two-phase commit discipline may involve timeouts, some notion of pre-emption, and *message idempotency*⁴. Modeling these notions in detail may make even our agent-oriented approach impractical. Abstracting them away may cause us to overlook significant performance issues.

The need to support standards-based behavioral specifications and a uniform interaction discipline may be necessary to hierarchically compose web services. Each service in an orchestration can itself be an orchestration. As an example, the flight service and hotel service each may involve an elaborate orchestration involving multiple airlines or hotel chains, respectively. Furthermore, each orchestration can be part of some larger orchestration. The orchestration developed in our case study can itself be treated as an individual web service representing the query phase of trip planning. This phase could then be part of a larger three-step sequential composition involving query, confirmation, and payment for an orchestration that eventually produces a

⁴ Message idempotency refers to a property of a system wherein system state is not altered with the receipt of duplicate copies of the same message.

trip itinerary and debits the user’s account. Behavioral specifications are required by our proposed framework so that we can represent and guide each agent’s behavior according to some finite state model. Hierarchical composition requires a notion of *compositionality* involving assume-guarantee reasoning implemented as some uniform interaction discipline [1]. Development of provably correct methods for constructing such a hierarchical composition of web service orchestrations, is left as future work.

2.4 Practical Impact

This section treats four areas of practical impact, namely: (i) deployment time testing, (ii) changing baselines, (iii) scripted testing, and (iv) process improvement. We describe how our proposed approach using *in situ* agents can be adapted for deployment time testing of compositions with fixed number and type of services but with modifications to the composition under test (i.e., Weakly Closed systems). We discuss the persistent problem with compositions that change baselines, namely those with variable number of services, or of services with roles that vary (i.e., *Opened* systems). We contrast our approach to the current practices involving scripted testing. Finally, we describe how our work fosters process improvements.

2.4.1 Deployment Time Testing

Tradeoffs described in Section 2.2 yoke the notions of *validation* with respect to external requirements to *verification* with respect to internal requirements, by depicting validation and verification as a mirrored pair of triaxial charts. Validation

sets forth tradeoffs between assurance and the remaining factors of flexibility and performance. Deployment time testing using agents is indicated if performance or flexibility is required, however, quality assurance will suffer. We warn that safety or fiscally critical systems must depend on exhaustive forms of verification and not on any non-exhaustive deployment form of testing.

In Section 2.3, we proposed an approach to testing using agents operating *in situ* within a web service composition under test. Deployment-time testing of this operational glue code (i.e., BPEL) artifact, can be implemented by redundant triples of *partner links* for each service in the composition. The first link is to an agent that simulates the web service. The second link is to a presently implemented production-level service having a known set of capabilities. The third link is to a proposed implementation having, say, enhanced capabilities. Typically the first and second links should exhibit equivalent behaviors, with the third link exhibiting the behavior of a *beta* version of the service. The beta version needs to exhibit the same behavior with respect to the set of interleavings successfully handled by the agent and the production versions. Once this is accomplished, the agent version can be enhanced to simulate the coordination behavior of the beta version. If the simulating agent consequently exhibits undesired behavior, then either the agent or the beta versions need to be modified. This undesired behavior typically surfaces when the less-tested version of the service (i.e., the beta version) induces a malfunction in the composition under test.

In addition to avoiding undesired behavior, a successful agent simulation of the beta version can support capabilities not present in the production version. Nonetheless, all message types present in the production version of the web service need to be simulated by both agent and beta versions prior to upgrading the system to the beta version. Thus, one last regression test needs to be performed, since any undesired interaction will compromise the operation of other services and service compositions, like the one we are testing.

Keep in mind that any service can itself be a composition of web services, and that the system under test is just the visible top-level glue code artifact which we built. It is conceivable, however, that the beta version of a third-party service differs from the production version by differences in its own glue code – code that is not available for us to maintain. Further discussion of this situation falls under the topic of changing baselines.

2.4.2 Changing Baselines

In a web services context, a change in the glue code artifact constitutes a change in the baseline, since such changes may involve one or more of the following: (i) include additional services that need coordination with existing services, (ii) exclude services that may be critical to desired interaction, (iii) restructure the composition which may induce undesired interaction between services, or (iv) modify the way exceptions are handled, violating some assume-guarantee condition. Further analysis

of changing baselines involve two scenarios, the first being non-visible changes and the second being visible ones. The following paragraph discusses the former scenario while the paragraph after that discusses the latter one.

Glue code changes become problematic if hidden behind some third-party web service. Under this circumstance, compositional or modular program reasoning is not possible, since correctness of the top-level visible web service composition depends on correctness of the lower-level non-visible web service implementations [1]. Section 2.3 approaches the problem of changing baselines by guiding the behavior of each agent by a process algebraic expression, the model for which can be checked to some set of temporal logic properties. Whereas the behavior of each individual agent may be tractable to model checking, compositions having services, each modeled by an agent, may become intractable. Use of such agents, however may detect non-compliant glue code modifications to third-party web services before they contaminate existing versions.

Changes to visible top-level glue code artifacts only become problematic if they were themselves wrapped in a web service interface to become third-party services for other orchestrations. Since changing baselines have traditionally been driven by external forces, the former scenario remains the most prevalent, persistent, and problematic. It is worth noting that Section 2.2 suggests that high-assurance systems be comprised of third-party components that evolve slowly enough so that their compositions can be model checked.

2.4.3 Scripted Testing

Traditionally, testing involves the manual construction of *test scripts*, often aided by software tools. These scripts typically reside outside of the system under test, and pose as the environment for that system. Such a script will need to test the top-level glue code artifact and the services it uses, and represents a top-down approach to testing. For compositions involving a tractable number of permitted interleavings, Section 2.2 recommends the use of model checking rather than testing, particularly for safety or fiscally critical systems. Otherwise, agents may be used with the hope that each test agent remain a step or two ahead of humans when detecting faults. Section 2.3 proposes a bottom-up approach in which each agent acts *in situ*, interacting with the web service composition under test as if the agent was an individual web service. This approach is formal since the behavior of each agent is guided by process a algebraic expression that satisfies properties expressed in some temporal logic. It is informal since it depends on a simulation that non-exhaustively tests combinations of interactions.

An alternative method not explored here involves test agents in which each agent operates according to a script. Unfortunately, this approach relies on humans to derive use cases and test scenarios, whereas our approach involves formal specification of these use cases resulting in automatic listing of test scenarios. These formal specifications can then be interpreted by otherwise identical agents, each of which functions as a real-time process algebraic interpreter.

2.4.4 Process Improvement

Section 2.2 describes the realm of validation in terms of industry type be it Engineering, Finance, or Entertainment, and what is required by each industry type. It describes the realm of verification by describing what forms of formal verification are feasible and under what circumstances will either manual or agent-based testing would be acceptable. Distinguishing between these two realms and the tradeoffs inherent in each will assist future selection of verification or test strategies.

Section 2.3 focuses on one form of semiformal verification that includes elements of formal verification, but used in a framework for non-exhaustive testing. This approach helps fill the gap between testing and formal verification for moderate sized orchestrations that require a moderate degree of quality assurance.

2.5 Chapter Summary

We depict the tradeoffs that govern our choice of quality assurance strategy using triaxial charts, in which one chart represents the realm of validation and the other verification. Assurance is the only vertex in common between these two charts, so we establish a correspondence between application domain and quality assurance technique on the basis of assurance only.

The choice between test and exhaustive verification on the verification side will depend on the purpose of the service composition on the validation side. On the validation side, the side representing external tradeoffs, compositions that are meant

to entertain occupy the side opposite to the Assurance vertex. For these systems, an appropriate level of quality assurance corresponds to the side opposite the Assurance vertex of the verification side representing internal tradeoffs. Thus, an entertainment application may only require non-exhaustive testing.

Compositions that are either fiscally-critical or safety-critical occupy the portions of verification side nearer the Assurance vertex, and will require some means of formal verification like model checking. Real-life service compositions have portions that require high assurance, and other portions that don't. When formally verifying the overall composition, care must be taken to avoid having the non-critical portions interfere with the operation of any of the critical portions. This suggests an asymmetric approach to model checking that may facilitate exhaustive verification of larger scaled or more detailed compositions than the (present) symmetric approach. Technological tradeoffs in the realm of validation involving the sacrifice of scale for level of detail, will for successively larger models, become less useful for finding coordination faults. This indicates a need for a semi-formal strategy involving software agents.

Testing an orchestration using an agent at each service not only suggests improvements to the BPEL artifact under test, but also points to reliability improvements in each service being simulated. We established a correspondence between agents and the web services they simulate, critically examining the suitability of BDI type agents operating in some ontological framework using some uniform interaction

discipline. Finally, we assess impact on current practice that includes deployment-time testing, changing baselines, scripted testing, and process improvements.

In summary, modeling and testing a far-flung Internet application either exhaustively or as a society of agents, reduces the human effort from testing and the human suffering from not testing. When exhaustive verification is not practical, modeling each service in an orchestration as an agent assures the quality of web service orchestrations that have been developed in-house.

CHAPTER 3

USE CASES FOR VERIFICATION

Due to their high level of abstraction, web services blur the traditional distinction between *model* and *code*. Consequently, executable artifacts can now be written to a level of abstraction that more closely approximates that of models used for model checking. Unlike traditional localized white-box software artifacts, the distributed black-box nature of web services demand exhaustive verification. Recent usage trends for model checkers coupled with the rise of web services, suggest a different approach to this computationally intensive form of verification. This chapter outlines these trends and describes a web-enabled approach to verification of web services. We outline how activities associated with model checking may be migrated into a web services framework, enabling practitioners to more easily incorporate model checking into their solutions. Based on the paper titled *Toward Model Checking Web Services over the Web* [119], this chapter explores these usage trends and proposes a web-enabled verification architecture.

3.1 Chapter Introduction

As a composition of loosely coupled autonomous services, each having a known interface, advertised functionality, and specified behavior, Service Oriented Architectures (SOA) are most commonly implemented as web services. Among other benefits, SOA's free developers from concerns over platform, implementation, and versioning. These freedoms, however, render most traditional testing techniques ineffective. Without access to source code of services that may not behave as advertised, unforeseen usage scenarios and implicit assumptions can cause race conditions that end in deadlock or other undesired interaction. In this setting, management would be reluctant to deploy safety or fiscally-critical applications as web services.

To remedy this, Nakajima [98] applied model checking to compositions of web services. Given glue code artifacts written to a predecessor of Web Services Business Process Execution Language (WS-BPEL) and Web Services Description Language (WSDL), Nakajima first proposed converting these artifacts to a finite state model suitable for model checking. Model checking is a technique for exhaustively verifying compositions by automatically listing every usage scenario, checking each for violations of specified properties. More precisely, given a finite state model of a system, model checking is a means of verifying if certain properties hold for reachable states within the state space generated from that model. A survey of formal methods, including model checking, appears in [30].

We describe how model checkers have recently been used, from which we delineate features of a proposed SOA for model checking web services, and suggest one such architecture. This work will facilitate development of best-of-breed verification environments by embedding model checking techniques into web-enabled applications.

The remainder of this chapter is organized as follows: Section 3.2 describes how model checkers have been recently used. Based on these usage trends we describe improved means of generating checkable models from process definitions in Section 3.3, followed by a top-level architecture in Section 3.4 concluding with Section 3.5.

3.2 Advanced Usage Trends

The web services test literature abounds with uses of model checking that were not fully anticipated by the original tool developers. This section reviews recent usage patterns of model checkers to delineate problems and trends that may influence future architectures. Figure 2.1 depicts how model checking is presently used in web services development. Note that model checking occurs last, after composing web services in step 3 and performing service discovery in step 2. Modeling earlier during syndication, checking later during deployment, paying increased attention to long-running transactions, and incorporating incremental model checking into a software evolution paradigm, are but a few trends that will change this picture within the next few years. We discuss each trend in what follows.

3.2.1 Syndication Time Modeling

The need for more targeted automatic discovery and composition of web services suggests additions to web service descriptions. The authors of [71] predict that Universal Description Discovery and Integration (UDDI) descriptions will be supplemented with automata-based specifications that include the model used for model checking and the Web Ontology Language for Semantics (OWL-S) based specifications containing temporal constraints.

During web services syndication and discovery, one would like to identify services that perform an "equivalent task" as a means of finding alternative web services. This implies the need to detect various forms of *bisimulation equivalence* – a functionality found in some verification environments that also happen to support model checking [17, 78].

3.2.2 Service Evolution

Assuring the quality of a web service composition as it continues to evolve *after* its initial deployment, requires postponing the checking phase of model checking until after Step 4 in Figure 2.1. The CHARMY project [20] model checks software components that evolve, but must retain *correctness by construction*. This software architecture based approach addresses already implemented software components. Incremental model checking, first described in [128] and later used in CHARMY, may lead to techniques for augmenting and visualizing the state space as one component

dynamically replaces another.

The feasibility of model checking as a system evolves depends on the type of system, which includes Closed systems, Weakly-closed systems, Weakly-opened systems, and Opened systems. The authors focused on the former two, considering the latter two as less feasible.

Closed systems have a fixed set of components and a fixed connector or glue code artifact. Their web service artifacts can be "set in stone", making them the most feasible to model check. Evolution is limited to upgrading any component or its connector with those having *identical* interface and message exchange behavior. Web service artifacts remain unchanged while the implementation that operates behind the WSDL artifact (i.e. choice of middleware platform, or updated executables) changes. Its composition may be best implemented as an orchestration in WS-BPEL and model checked with existing tools at design time. Closed systems do not impact Figure 2.1. Although least malleable, the ability for each constituent web service to be independently upgradable showcases the strengths of SOA.

Weakly-closed systems can undergo types of reconfiguration that will require run time model checking. Such a system has a fixed set of component types or roles, yet its glue code still can dynamically bind to different yet *purportedly equivalent* services. Over time, evolution involves swapping a service instance of a given type with another instance of the same type. In web services, the peers in a long-running transaction often change but their fundamental roles, relationships, and behaviors

do not, and may be best modeled as a *choreography*. For example, long-running transactions in hospital case management will always require an attending physician for any hospitalized patient. If that physician is unavailable, a different physician must temporarily assume the *role* of attending physician. The new attending physician, however, may perform rounds at a different time of day, potentially impacting related workflows. Thus in weakly-closed systems one is concerned with evolving systems using *sufficiently* equivalent services, while checking that each substitution does not adversely impact existing workflows.

Since such systems operate asynchronously, their state spaces can become intractably large, necessitating the use of incremental model checking. Although [20] proposes one of a number of strategies for dynamic composition, reconfiguration, and verification of weakly-closed systems, two facts stand out. Firstly, this dynamic incremental verification will require use of a highly optimized C Language-based model checking tool, which motivates their choice of the Spin model checker. Secondly, they observe that any dynamic reconfiguration strategy will not be appropriate for real-time systems or for systems subject to hard time constraints. This is because the order of complexity of model checking often exceeds that of the application it models.

Thus, users must choose model checking approaches based on a three-way tradeoff between flexibility, assurance, and performance. The best one can hope for is to find an approach that optimizes on these three variables, given the real world requirements for some specific web service application. In this case, service evolution

can only maintain assurance at the expense of performance.

3.2.3 Incremental Coverage Testing

Since a model is an abstraction, it cannot be directly used as a testable artifact. However, model checkers have been used to generate test cases [46] that cover either all or at least the most common situations. Incremental coverage testing involves generating only new or deprecated test cases, to be examined on an exception basis, to determine whether the new or deprecated cases were what stakeholders had intended. The authors of [67] and [108] additionally wished to generate both positive examples and negative (counterexamples) for each temporal logic formula for each path in the state space [67], or each decision in the source code reflected in the state space [108].

Positive examples can be generated by model checking to the negation of these temporal logic properties. Not all positive examples for all coverage criteria can be generated using the Linear Temporal Logic (LTL) like that supported by Spin. Since Computational Tree Logic (CTL) more directly handles this negation, [67] used the SMV model checker, although both the UPPAAL and CPN-Tools verification environments also support CTL [13, 73]. Furthermore, a user needs to know if a change will require generating an intractably large number of new and/or deprecated cases, suggesting the need for a more incremental revision in the composition. If a sufficiently small increment cannot be specified, then a subset of test cases can be generated based on use cases observed in event logs, but at the expense of assurance.

Thus, use of model checkers as test case generators necessarily requires a number of workarounds. Decoupling functionalities described in Section 3.3 will streamline this mode of use.

3.2.4 Integrating Instrumentation

Instrumentation provides feedback that describes how web service compositions actually behave. Gravel, et. al. [55] consider using Spin to model check web service orchestrations, in conjunction with use of their proposed Execution Analysis tool for WS-BPEL (EA4B). This addresses three issues: (i) lack of tool support for understanding the manner in which WS-BPEL actually executes, (ii) enable exploration of service execution to identify the source of an erroneous service, and (iii) assist the user in making informed decisions on correctness of observed behavior. They propose instrumenting WS-BPEL artifacts so that post execution debugging and verification can be performed, or used for near real-time monitoring, or integrated with Spin. In the latter use case, their Web Services Analysis Tool (WSAT) [71] can translate WS-BPEL artifacts to Promela source code for input to Spin. Spin can then generate both positive and negative test cases, which can be executed by the web service composition for tracing and visualization using EA4B.

Process mining uses event logs to discover various perspectives on a process, including data flows, social interactions, and control flows. An interesting approach for generic workflow applications involves use of noise-tolerant genetic algorithms

described by de Medeiros [34]. For web services, this discovery can be made easier since the block structured nature of WS-BPEL supports implementation of only a subset of workflow patterns [113]. Faithful reconstruction of a WS-BPEL artifact using process mining tests both the extent and quality of instrumentation. It provides confidence that the corpus of coverage test cases appearing in the event log used for reconstruction is complete.

3.2.5 Stateful Web Services

Web services must retain state as parties to a long-running transaction enter and exit, or when a web service composition must consistently maintain the state of its variables.

A different use case for model checking, involves conformance checking to some set of norms encoded in a standard like WS-BusinessActivity. Ramsokul et. al. [105] propose a test bed for transaction-oriented (i.e. stateful) web service compositions supplemented by artifacts from the WS-BusinessActivity standard. Over time, such compositions may encounter varying subsets of parties to a long-running transaction, as in case management workflows implemented as web service choreographies. Using event logs, discrepancies between it and the process model are analyzed with respect to some set of assumptions that must not change throughout the lifespan of the case. Implied in this approach is the ability to formally model and state properties in WS-BusinessActivity artifacts.

Zheng et. al. [152, 153] propose a means of automated generation of test cases from WS-BPEL artifacts that manage data dependencies. They use model checkers to automatically generate data flows for each variable by formulating test criteria as trap properties – the negations of the original properties being verified, which is a similar approach taken by [67] and [108]. When generating tests for stateful web services, [153] considers WS-BPEL *variables* and *links*, describing in detail their use of web service automata as an intermediate representation that could then be converted into either Promela or SMV.

3.2.6 Visualizing State Spaces

Visualization can provide insights into the structure and behavior of concurrent systems. With the availability of exchange formats described in Section 3.3, visualization may become more routinely used. Motivated by this need, [56] proposes the use of the freely available mCRL2 verification toolkit that provides visualization support for massive state spaces. It can be downloaded at: [http: / / www.mcrl2.org / wiki / index.php / Home](http://www.mcrl2.org/wiki/index.php/Home).

The specification formalism for mCRL2 is the *Pi-Calculus*, known for its ability to describe concurrent processes having configurations that may change during run time.

The mCRL2 toolkit is intended for general purpose test and verification ranging from microscale embedded systems to macroscale web services. It provides intriguing 3D visualization support for state spaces based on three topological properties: (i) proximity of each state to the initial state, (ii) the size of visual elements as proportional to size of node clusters, and (iii) retention of the symmetry of the resulting state space. On the surface, these visualizations resemble those governing the formation and organization of biological organisms, particularly dendritic phenomenon ⁵.

The toolkit includes model checking capabilities, providing state space compression for otherwise large but regularly structured concurrent systems [17]. Deadlock or other property violations are color coded, enabling developers to visualize the context of modeling or compositional errors, while test traces can display the traversal path over the state space from initial state to some designated state. Considering the trend toward decoupling, there will be a need for generating state spaces in other tools to some standard interchange format for import into the visualization component of mCRL2.

3.3 Architectural Features

By observing usage trends for model checking web services, we identify four features that must be incorporated into any SOA used for verifying web services. The

⁵ Dendritic phenomenon are typically modeled using diffusion-limited aggregation processes. Such processes may be discretely modeled using Tarjan's algorithm [133] for linear-time construction of strongly connected components of a graph. Then again, these visualizations may merely be artifacts of the underlying algorithms used for their rendering.

first involves *decoupling functionalities* offered by model checkers so that users can orchestrate their own service verification environment. The second feature emphasizes *soundness* and *completeness* of web service compositions by representing web service artifacts so that machine verifiable models and properties can be inferred. The third requires *streamlining pre-processing*, which occupies the vast majority of time and effort. Finally, a model checking environment should *control its level of abstraction*, so that the models and properties are defined at an appropriate level of detail, subject to the size of the state space.

3.3.1 Decouple Functionalities

As an end-to-end process, model checking web services involves (i) converting a web service artifact to a model for model checking, (ii) converting the model to a state space, (iii) mining the state space for conformance to temporal logic properties, (iv) producing counterexamples, and (v) feeding these back through the executable web service artifact. Presently, model checkers lump together steps (ii) through (iv).

Syndication time modeling and deployment time checking requires decoupling step (iii) from (ii). For visualization to operate as its own service, it must not depend on how the state space was generated in (ii) or evaluated in (iii). Likewise, a variety of state space generation and mining techniques that optimize on size, performance, or level of abstraction have already been implemented in a number of model checkers,

and should each be available as its own service. Decoupling model checking functionalities is required by the additional architectural features described in the remaining subsections.

Decoupling entails the derivation of interchange formats. Step (i) will require a SOAP message type that can represent any finite state model suitable for model checking. At minimum, this message type must be capable of expressing models used by model checkers recently appearing in the literature. These include the untimed models of Spin and SMV, the Message Sequence Charts (MSC) of the Labelled Transition System Analyser (LTSA) verification tool, and the timed automata of UPPAAL. Step (ii) will require an encoded representation of a state space that supports efficient mining of that space. The need for distribution to or access by multiple hosts for step (iii) must be considered. Step (iv) will require generating an event log containing SOAP messages suitable for the web service composition under test.

3.3.2 Emphasize Soundness and Completeness

Sound and complete interface definitions and behavioral specifications for each service will be needed in a decoupled model checking framework. A model is *sound* if a property that is preserved in all executions of its program is also preserved in all traces of a model. That is, given a model and program, a model checker must not produce false positives. A false positive is where a faulty trace detected by the model can either not be reproduced by the program, or it can be reproduced but is

correctly handled by the program. The converse of soundness is completeness. A model is *complete* if a property that is preserved in all traces of a model, is also preserved in all executions of its program. That is, given a model and program, a model checker must not produce false negatives. A false negative is where a faulty program execution cannot be detected by the model and model checker.

Verifying soundness and completeness must require only WSDL and WS-BPEL artifacts. WSDL is used for defining the interface of a web service between its public and private sides, while WS-BPEL is used for composing these web services into an orchestration. Due to the semi-formal semantics of WS-BPEL, a number of proposals also favored supplementing these artifacts with a formal ontology encoded in OWL-S typified by [70, 71, 144]. A similar effect may be achieved by defining a discipline for coding WS-BPEL artifacts that are correct by construction. The richness of the resulting artifacts, and hence the need for OWL-S specifications, needs further investigation.

3.3.3 Streamline Model Capture

Extensive tool support will be needed for converting WS-BPEL artifacts to models suitable for checking. Currently the most prominent such tool is WSAT for converting to Spin and SMV [71]. LTSA uses an Eclipse plug-in to do this conversion into their internal representation which can thence be compared to that generated from user-specified message sequence charts [49]. Tool support for conversion for

other model checkers is otherwise sparse. As yet, no systematic study compares how well each conversion tool preserves the semantics originally encoded in the WS-BPEL artifact.

Model capture can avoid work later in the model checking cycle. For example, by modeling asynchronous processes as if they were synchronous can be done in identifiable cases, obviating the need to produce such large state spaces [21].

3.3.4 Control Level of Abstraction

What constitutes an appropriate level of abstraction has been open to debate. It becomes relevant when (i) model checking incremental changes in code and (ii) determining a suitable degree of instrumentation for process mining. If an incremental change in code caused a failure, but neither the model nor its specification changed, then either the model or the specification needs further refinement so that the code change bringing about the failure triggers counterexamples from the model checker. In process mining, a test case generated by a model checker not otherwise appearing in an event log will suggest the need for more detailed instrumentation. Likewise a test case appearing in an event log not otherwise appearing in the test cases generated by a model checker suggests the need for a more detailed model. State space size, however, constrains any such refinement.

3.4 A Predicted Architecture

Model-driven development of SOA's typified by the work of Heckel [61] and Lohmann [86] can be applied to developing an SOA for model checking web services over the web. A model-driven approach makes modeling complex systems accessible to practitioners. In this environment, the by-product of creating a WS-BPEL artifact may be a model suitable for model checking. Model checking, in turn, can generate suites for testing the WS-BPEL artifact.

Here we sketch a model-driven approach to assuring quality of web service compositions. A web service composition has three views: (i) *artifact view* showing user-defined artifacts or graphical renderings thereof, (ii) *model view* showing its automata and property specifications, and (iii) *instrumentation view* showing an event log that, when mined, faithfully reconstructs the web service artifact. Manipulating one view will propagate changes to the remaining views. Likewise, any deleted views can be reconstructed from any remaining view.

Figure 3.1 shows two areas of change. The first involves step 2 in which feedback is solicited by the syndicator from the user and is further described in [131]. The second area involves the model checking process in step 4. Presently, composition is a one-off process shown in Figure 2.1. Figure 3.1 makes web service compositions reusable by requiring in step 5 the formulation of a WSDL specification for the composition, prior to interning both back into the UDDI registry. A WS-BPEL artifact must also be coded to contain sufficient information for extraction of the remaining

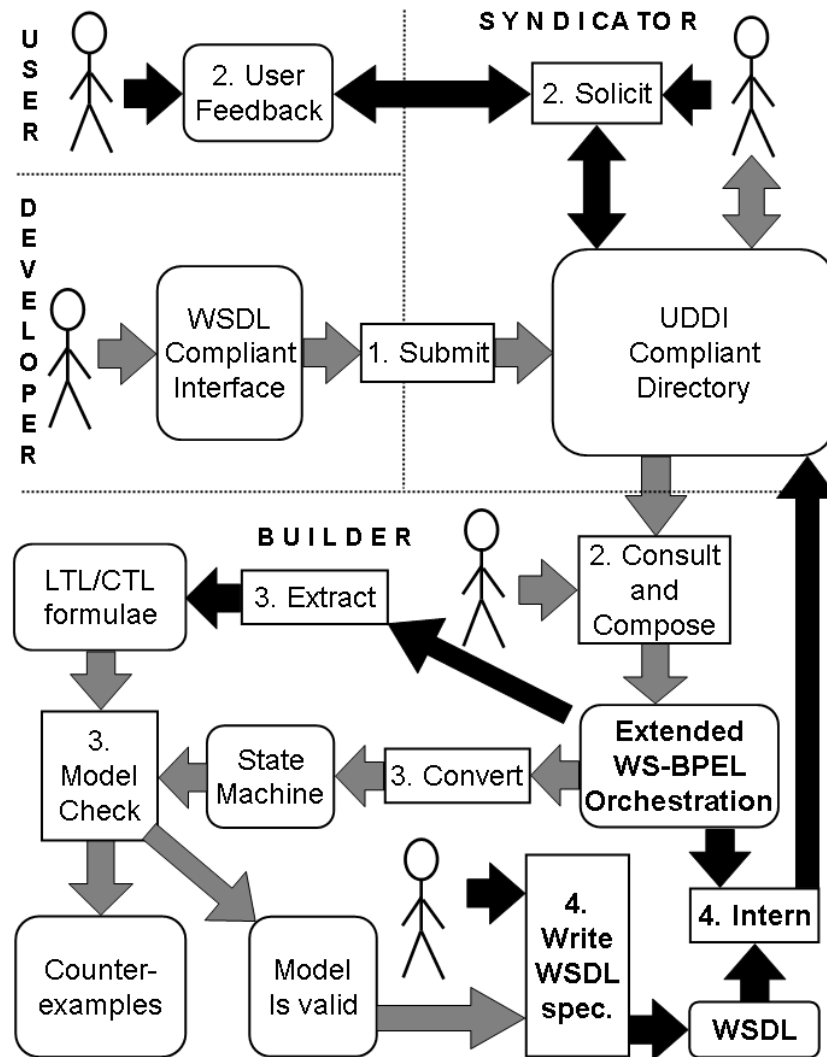


Figure 3.1: Proposed Development Cycle

views. Hence step 3 of Figure 3.1 may require artifacts written to some instrumented extension of WS-BPEL. Note how the application builder assumes sole responsibility for verification, placing responsibility closer to the source of decision-making and aligning each role to market structures.

The result is the ability to intern valid web service compositions in addition to the descriptions of individual services that have traditionally made up the UDDI registry. Consequently, one can realize a form of hierarchical composition in which each WS-BPEL artifact can operate behind its own WSDL artifact without the knowledge of builders who may subsequently use that composition.

Someone wishing to build a web service composition works as usual with WSDL artifacts, without concern for whether any service is itself a composition. Applying this design to the Travel Agency Problem, subsidiary services of airfare and hotel can each be implemented as a non-trivial orchestration involving multiple service providers [124]. This design can compartmentalize an audition [14] or a parallel test phase to each subsidiary service composition.

The instrumentation view requires an event log that can faithfully reconstruct a WS-BPEL artifact. Serving as an alternative representation, this view must contain all coverage test cases. Whereas test suites can be generated from finite state models using model checkers, in process mining, models may be generated from test suites.

3.5 Chapter Summary

This chapter outlines usage trends and architectural features, suggesting a model-driven approach to verification of web services using web services. Among other things, this approach realizes a form of hierarchical composition requiring only glue code and interface artifacts. This form of composition can compartmentalize how subsidiary service compositions are deployed.

Deploying currently known state space generation and optimization techniques as individual services will enhance collaboration, by making mining and visualization of ever larger state spaces feasible. Using web service standards and ontologies to characterize the way in which state spaces are produced, encoded, represented, and mined will enable derivation of sound and workable interchange formats. This will pave the way for seamlessly integrating model checking functionalities into a web-enabled SOA.

CHAPTER 4

AUTOMATING MODEL CAPTURE

Based on an expanded version of [127], this chapter seeks to automate Model Capture for a Subset of Web Service Orchestrations. Generating a machine-verifiable model from a Business Process Execution Language (BPEL) artifact had drawn recent interest, partly because BPEL's level of abstraction approximates that of feasibly verifiable models. To date, no tool has been proposed that enables an analyst to generate a model that is not only understandable, parsimonious, and suitable for simulation, but also extendable with the assumptions needed for feasible machine verification. First, we characterize the subset of BPEL amenable to the proposed translation into PROMELA, the modeling language for the model checker SPIN. Next, we identify behavioral patterns that such a translation must enact along with assumptions that trade state space size for model fidelity. Finally, we present translation algorithms and production rules for each of the three parts of a PROMELA model. We apply these algorithms and rules to a well-known case study, while sharing our experiences with the development of an initial prototype.

4.1 Chapter Introduction

Model checking can exhaustively verify if a BPEL program correctly orchestrates activities amongst a collection of web services. We describe how to automate construction of a machine verifiable model given a BPEL program and a set of modeling assumptions.

4.1.1 Motivation

Orchestrating web services involves combining loosely coupled autonomous services, each of which has its own interface, advertised functionality, and specified behavior. As a popular and well-supported language, BPEL enables the developer to specify orchestration behavior at a suitably high level of abstraction. Such orchestration of black-box services frees developers from low-level concerns involving platform, implementation, and versioning. These freedoms render white-box testing techniques ineffective and, for safety- or fiscally-critical systems, suggest the use of exhaustive verification techniques like model checking. Based on its popularity, maturity, and highly optimized code, we chose to use the SPIN model checker as the destination for our translation from BPEL. Additionally, tool support is available to translate SPIN's PROMELA modeling language artifacts into other modeling and verification formalisms [57].

To make exhaustive verification feasible, we must address three areas that affect state space size. First, a tractable subset of BPEL containing only non-iterative

structured activities will be considered for translation. Iterative activities like `<while>`, `<repeatUntil>`, and `<forEach>` can result in either unbounded or intractably large state spaces. Second, although model capture must be done at a low enough level of abstraction to be useful, it must be done at a high enough level to be tractable [120, 122]. Third, implicit assumptions need to be made explicit and need to be confined to specific portions of a composition [127].

Recently, a number of efforts have been made to automate the generation of machine verifiable models from web service artifacts [49, 50, 76, 87, 100, 114, 121, 141]. Such automation seeks to minimize both human effort and judgment in model capture. These efforts, while laudable, have resulted in modeling artifacts that are difficult to understand, simulate or troubleshoot. None of these approaches enable the analyst to extend these models with assumptions that affect their tractability. Furthermore, existing techniques do not elaborate on how specific behaviors represented in BPEL carry over into the corresponding PROMELA model.

4.1.2 Contributions

This chapter outlines an extendible approach to automating the translation of BPEL source code to a machine-verifiable target model, including these specific contributions and features. We list these by order of exposition, rather than by perceived importance:

- Using a recurrence, we provide a compact characterization of the subset of

BPEL we intend to translate. This demonstrates the utility of recurrences as second-order theories when defining regular languages that are large and difficult to conceptualize. The recurrence also suggests traversal strategies for translating source code to a machine-verifiable model.

- Using a typed CSP-like process algebra, we describe several behavioral patterns inherent in BPEL that a finite state model must be able to represent. We then relate the algebraic representation to portions of both BPEL and PROMELA artifacts. This contribution lays the groundwork for a more extensive examination of workflow patterns listed in [113], by lending language-independent insights into how to orchestrate web service interactions.

- We identify three types of assumptions that affect state space size, and provide a means of confining these to specific *portions* of the target model. Hence, those portions of the composition that have been deemed reliable can be abstracted to models having smaller state spaces, while newer, less reliable portions can be modeled in greater detail. We expect this feature to blend with engineering practices that encourage incremental changes to artifacts under test.

- We present translation algorithms structured according to regular language L^n that represents the dialect of BPEL defined by applications to n levels of the recurrence formulae. Furthermore, we define production rules and associate each to some *leaf* routine inside our algorithms, lending a degree of extendibility to the translator. Using this design, the tool development cost of adapting larger subsets of

BPEL is expected to increase only incrementally.

4.1.3 Overview

We automate model capture by first defining a recurrence relation that formally identifies a sublanguage L^n of BPEL amenable to conversion. We then refine the recurrence relation's level of abstraction from the *activity* level down to the *attribute* level. We then encode assumptions about the behavior *implied* by the BPEL execution model, by adding attributes to the BPEL artifact.

Finally, we present top-level algorithms and detail-level production rules for converting BPEL artifacts to models in PROMELA. In the case study, we added modeling assumptions concerning atomicity, synchrony, and parallelism, running our prototype for all combinations on these assumptions [140]. Additionally, we describe how to extend this prototype to model certain pessimistic assumptions about whether a service is fault-prone or whether to suppress or propagate join failure.

Using a case study, we provide the needed intuition into the translation process. This case study is a simplified version of the Purchase Order Process which initially appeared in the WS-BPEL Specification [4]. We developed a prototype utility for translating BPEL artifacts into PROMELA. The case study discussed in this chapter exercised both parallel and sequential (i.e., `<flow>` and `<sequence>`) constructs, while another case study developed in [140] exercised choice constructs (i.e., `<pick>` and `<if>`). These case studies lent both empirical validity and insight into

the implementation of this utility.

The rest of this chapter is organized as follows: Section 4.2 describes the syntax and structure of the subset of BPEL artifacts to be translated into PROMELA. Section 4.3 identifies a set of behavioral patterns in BPEL that must be carried over to the model. Section 4.4 presents a case study to lend insight into the translation process. Section 4.5 formulates production rules and presents the translation algorithms used as their scaffolding. Section 4.6 examines work related to automated translation of BPEL to verifiable models. Finally, Section 4.7 provides a summary and brief description of future work.

4.2 Structure

Translating a BPEL artifact to a PROMELA model entails translating a structural artifact into a behavioral one. A purely syntactic first step converts BPEL's prefix notation to the infix notation used by PROMELA. That is, BPEL uses prefix notation to announce the types of structured or basic activities or types of XML elements being declared. The sub-language L^n of BPEL defined by Formula 4.1 lists these activities or types as infix operators.

L^n is only defined at the *activity* level of abstraction – one level down from the *process* level. To model interactions between an orchestration and its partner services – to list all its interleavings – requires further refinement down to the attribute level. BPEL attributes that reference variables are converted to channel statements

in PROMELA. Refinement below the attribute level must consider data values, their propagation, and influence on control flows. The desirability and feasibility of generating models at that lowest level of abstraction may be considered for future work.

A final syntactic translation involves *lifting* a BPEL artifact's collection of lexically nested XML elements to a collection of lexically flat first-class and peer-coupled process declarations in PROMELA. An early problem that employed this technique involved the conversion of programs from Pascal to the C programming language. Among other things, the algorithms in the later sections of this chapter incorporate this technique.

4.2.1 Sub-language

The sub-language of BPEL of interest can be described at the activity level of abstraction by a recurrence that conservatively extends the context-free grammar of matching parentheses. As in the case of that grammar, if the maximum of depth i is fixed at n , such a recurrence can generate an, albeit long, regular expression.

$$L_{\circ}^i = \begin{cases} \langle [[p,]^* [\nu,]^*] + [[\nu,]^* [p,]^*] \rangle L_{\circ}^{i+1} + L_{\parallel}^{i+1} + \langle \beta[; \beta]^* \rangle & \text{if } [i \equiv 0] \\ \langle [L_{\circ}^{i+1} + L_{\parallel}^{i+1} + \langle \beta[; \beta]^* \rangle] \{ [L_{\circ}^{i+1} + L_{\parallel}^{i+1} + \langle \beta[; \beta]^* \rangle] \}^* \rangle & \text{if } [0 < i < n] \wedge [\circ \equiv ;] \\ \langle [l,]^* [L_{\circ}^{i+1} + L_{\parallel}^{i+1} + \langle \beta[; \beta]^* \rangle] [\parallel [L_{\circ}^{i+1} + L_{\parallel}^{i+1} + \langle \beta[; \beta]^* \rangle]^*] \rangle & \text{if } [0 < i < n] \wedge [\circ \equiv \parallel] \\ \langle \beta[; \beta]^* \rangle & \text{if } [i \geq n] \end{cases} \quad (4.1)$$

The recurrence in Formula 4.1 generates a regular expression for L^n . In regular expressions, metasyMBOL '+' lists syntactically valid choices, one of which must be

satisfied for any string representation of a BPEL artifact w to belong in L^n . Membership also involves evaluating "zero or more occurrences" of those subexpressions, denoted by the Kleene Star * . Subexpressions are scoped by bracket metasymbols '[' and ']. The alphabet of L^n pertains to constructs that are specific to BPEL, and includes the set $\{\langle, \parallel, \circ, \rangle, p, l, \nu, \beta\}$. The first four symbols pertain to structured activities with ' \langle ' representing the activity's opening portion, ' \parallel ' parallel composition (i.e., $\langle\text{flow}\rangle$), ' \circ ' any one of a number of non-parallel types of composition (e.g., $\langle\text{sequence}\rangle$ or $\langle\text{pick}\rangle$), and closing portion ' \rangle '. The remaining symbols denote partner link declarations ' p ', control link declarations ' l ', application-related variables ' ν ', and basic activities ' β '.

Generating the regular expression for this recurrence is subtly different from interpreting the resulting regular expression. The recurrence ascribes absolutely no meaning to any of these symbols as it strings them out on a clothesline for further examination and interpretation. It does so by expanding values of recursion variables g_j^i at nesting level i for formula j using the string to its right and subject to the guard condition to its left. In generating this regular expression, the guard portions of this recurrence use logical symbols $\wedge \vee$ and \equiv for logical AND OR and ISEQUAL, respectively. Its guards use $i \equiv 0$, and $i \equiv n$ to refer to root and leaf portions of the tree structured regular expression under construction.

The first line of Formula 4.1 generates the outermost portion of the regular

expression and represents the top level of the tree structured BPEL artifact. It generates partner links p followed by application-related variables ν or vice versa, followed by an expression that, for $0 < i \leq n$ gets expanded at the next lower level on *all three* recurrence variables g_1^i , g_2^i , and g_3^i . The reader is reminded that the choice operator '+' in the regular expression being generated is passive. It does not guide expansion of the recurrence in any way. Rather, expansion is solely guided by recurrence variable g_j^i as nesting level i proceeds from 0 downward to n .

The second line of Formula 4.1 generates the portion of the subexpression for one or more non-parallel constructs in BPEL. These BPEL constructs include `<sequence>` denoted by ';', and *choice* (i.e., `<pick>` or `<if>`) denoted by '|'. Choice constructs '|' in the space of BPEL artifacts, are distinct from the choice operator '+' in the space of regular expressions. The third line of Formula 4.1 generates the BPEL subexpression for parallel compositions, namely the `<flow>` element, and involves any number of control links l , followed by one or more child activities. Each child activity, in turn, may either be structured or basic. Finally, the fourth line of Formula 4.1 generates some non-parallel composition involving purely basic activities at the bottom level of nesting.

In addition to these syntactic considerations, a PROMELA artifact must capture the behavior of a BPEL composition. These behaviors will differ depending on whether the BPEL activities are structured or basic, and if structured, whether the activities must proceed in parallel. The high level of abstraction of Formula 4.1 also

hides a number of key elements needed for modeling orchestration behavior. Status variables are a case in point. Since a BPEL code artifact does not explicitly declare the status variables used by its execution engine, these are not shown in the formulae. Nonetheless, a PROMELA model needs to include these status variables to properly enact simulation and machine verification. For example, δ is used to evaluate the *join condition* at the destination activity of any control link, while σ specifies how the result of the join condition should be propagated. A fuller discussion of these variables appears in Section 4.4. Before discussing behavior – the topic of Section 4.3 – we close with remarks on size estimation of these regular expressions and list what features of BPEL we do not set out to translate.

4.2.2 Size Estimation

Conceptualizing language L^n as a regular expression is not practical. Even at the high (activity) level of abstraction and at a depth n barely sufficient for even modestly sized BPEL artifacts, regular expressions are difficult to comprehend. This can be seen by Formulas 4.2 and 4.3, which show the application of the recurrence for $n \equiv 0$ and $n \equiv 1$, respectively. For $n \equiv 0$, Formula 4.2 contains 57 symbols which can be minimized to 39.

$$L^0 = \left\langle \left[[p,]^* [\nu,]^* \right] + \left[[\nu,]^* [p,]^* \right] \right\rangle \left[\langle \beta[\circ \beta]^* \rangle + \langle \beta[\circ \beta]^* \rangle + \langle \beta[\circ \beta]^* \rangle \right] \quad (4.2)$$

For $n \equiv 1$, however, Formula 4.3 contains 190 symbols but can be minimized to 100. Even a modest size example, like the Purchase Order Process used in this chapter, can only be recognized by a regular expression involving two *additional* levels of nesting. At each successive level, the number of symbols more than doubles. So for $n \equiv 3$, the regular expression that can recognize our case study $w \in L^3$ will have 532 symbols. Since middle two lines of Formula 4.1 are the only two formulas in the recurrence that expand the expression to arbitrary levels, the parse tree generated will contain $O(2^n)$ symbols. Hence, a recurrence of fixed size can represent arbitrarily long regular expressions to some arbitrarily deep nesting level n .

$$\begin{aligned}
L^1 = & \left\langle \left[[p,]^* [\nu,]^*] + [[\nu,]^* [p,]^*] \right] \right. \\
& \left\langle [\langle \beta[\circ \beta]^* \rangle + \langle \beta[\circ \beta]^* \rangle + \langle \beta[\circ \beta]^* \rangle] \right. \\
& \quad \left. [\circ [\langle \beta[\circ \beta]^* \rangle + \langle \beta[\circ \beta]^* \rangle + \langle \beta[\circ \beta]^* \rangle]]^* \right\rangle + \\
& \left\langle [l,]^* [\langle \beta[\circ \beta]^* \rangle + \langle \beta[\circ \beta]^* \rangle + \langle \beta[\circ \beta]^* \rangle] \right. \\
& \quad \left. [\parallel [\langle \beta[\circ \beta]^* \rangle + \langle \beta[\circ \beta]^* \rangle + \langle \beta[\circ \beta]^* \rangle]]^* \right\rangle + \\
& \left. \left\langle \langle \beta[\circ \beta]^* \rangle + \langle \beta[\circ \beta]^* \rangle + \langle \beta[\circ \beta]^* \rangle \right\rangle \right\rangle \quad (4.3)
\end{aligned}$$

4.2.3 Exclusions

Language L^n has a number of exclusions and limitations. Some basic activity types, like `<assign>`, have no interactions through partner links, so the prototype

simply indicates its presence as documentation inside the PROMELA code. Furthermore, activities like `<assign>` are sufficiently short-lived that they may be allowed to complete rather than be interrupted during any forced termination. Hence, we model only activities involving inbound or outbound messaging between web services and their orchestration, by using PROMELA channel statements. Language L^n does allow degenerate forms in which a structured activity like a `<flow>` or `<sequence>` may have only one child. The BPEL Validator included in the BPEL2OWFN tool also considers these as valid, but specific BPEL engines may not allow these degenerate forms.

Other activity types have not been implemented such as the `<scope>` activity. Currently, the translator recognizes only process scope and no user-defined scopes. User-defined scopes enable the analyst to confine declarations, error propagation, and compensation handling to specific portions of BPEL code. The recurrence can be extended to generate regular expressions for user-defined scoping by encapsulating the last three lines of Formula 4.1 into some larger and similarly structured recurrence. Such an extension is left for future work. Finally, activities involving iteration, like `<while>` `<RepeatUntil>` or `<ForEach>`, cannot generate a tractable model without severe restrictions on the number of iterations. Hence, implementation of these constructs are also excluded.

4.3 Behavior

Thus far we only considered the *structure*, rather than the *behavior* of BPEL processes. Given a language and a level of abstraction this section defines behavioral patterns or *idioms* that a translator must implement. These idioms simulate some behavior of a BPEL process using PROMELA and include reactivity, instantiation and global properties, interaction, sequence, choice, parallelism, interprocess dependencies, and implicit assumptions. This list is by no means exhaustive, so this section briefly lists limitations that can be overcome with either additional or more generalized idioms.

Enactment of these idioms results in a model of a reactive always-live web service for each partner link that interacts with an orchestration under test. The focus of modeling and verification is how structured activities behave within an orchestration, particularly for the `<flow>` construct.

4.3.1 Instantiation and Properties

The top-level process is initiated by a `createInstance` attribute, typically included inside a `<receive>` activity. This activity receives and processes data from a partner link which conceptually represents a customer or client. Inside the PROMELA process definition for that partner link, the translator places a `'progress:'` label so that SPIN can verify progress by detecting if that label can be visited infinitely often. After receiving and processing data from its partner link, the orchestration contains

a corresponding `<reply>` activity that outputs the result over that same partner link. The orchestration then destroys the instance, simulated in PROMELA by an `'end:'` label. SPIN uses this label to verify deadlock freedom by detecting if there exists a trace that does not reach this label, due typically to termination with no statements enabled. The top-level orchestration loops back to its `<receive>` activity, which blocks until the next inbound transaction arrives.

4.3.2 Reactivity

PROMELA represents each BPEL process or subprocess P as an infinite `do` loop enacting a recursively defined process of the form: $P = \dots ; P$. A BPEL `<process>` element, child activity of a `<flow>`, or service associated with a `<partnerLink>` which takes on the following form `< P > ... < /P >` results in the following PROMELA code segment: `active proctype P() {do ::{ .. } od}`. Nowhere in the body of the PROMELA `'do'` loop is there an exit condition, so on a correct verification, SPIN should always indicate the end state for process P as unreachable. In later sections we will describe the generation of process definitions for web services that constitute the environment (or test harness) of the orchestration along with any of its concurrent sub-processes.

4.3.3 Interaction

Channel statements that listen `'?'` or send `'!'` typed messages over asynchronous channels in PROMELA, model interactions between the orchestration and its partner

web services. These interactions can make an orchestration deadlock. Modeling these requires an attribute-level abstraction – one that is finer grained than the one for activities. The more detailed translation of an individual basic activity β in BPEL results in zero or more of these attribute-level statements. The translator uses attributes that represent input and output variables in the `<invoke>` activity or variables in the `<receive>` or `<reply>` activities. Table 4.1 shows a form for an `<invoke>` activity. Activities like `<receive>` and `<reply>` have otherwise identical code except that variable type attributes are used to generate channel statements.

4.3.4 Sequence

The `<sequence>` construct in BPEL which we symbolize by `';`' is identical in behavior to that in PROMELA. Each activity in a sequence is executed in lexical order, each terminating before the next begins. PROMELA uses `';`' as an infix operator unlike languages like C or Java that use the semicolon as a delimiter.

4.3.5 Choice

BPEL implements choice type activities like `<if>` with internal (nondeterministic) choice `' \sqcap '` and `<pick>` with external (deferred) choice `'|'`. Although the `<pick>` selects a child activity for execution based on some *external* event like the expiration of a timer, we nonetheless model both it and `<if>` nondeterministically. This allows us to exhaustively test all interleavings using a somewhat simpler and more readable model. Further refinements may involve modeling external events at the data

value level of abstraction. The larger state space resulting from a finer grained model may be partially mitigated by the removal of non-determinism in certain places. The feasibility of such finer grained modeling is left for future work.

Suppose a and b are either basic or structured activities guarded by the same always-true predicate. Choice activity $a \sqcap b$ would be expressed in BPEL as `<if>< a... >< b... ></if>`. It is modeled in PROMELA as `if ::(1)->a .. ; ::(1)->b .. ; fi;`. Since we wish to exhaustively check all execution traces, we model both `<pick>` and `<if>` non-deterministically as an internal and seemingly arbitrary choice. Hence, we chose to ignore external factors that influence a given selection.

4.3.6 Parallelism

To express parallelism, BPEL uses the `<flow>` activity which enacts the AND-SPLIT and AND-JOIN workflow patterns described in [113]. Since PROMELA has no direct means of modeling parallelism, we model this idiom in PROMELA using channels. These channels are not declared anywhere inside the BPEL composition, and are hence hidden. This entails launch, parallel execution, and completion of process instances, each represented by immediate children of the `<flow>` activity. Formula 4.4 provides a process-level representation for the `<flow>` activity. At this high level of abstraction, process Q_i^0 launches children Q_j^1 and Q_k^1 to run in parallel, each finishing at their own pace.

$$Q_i^0 = \langle_i \dots Q_j^1 \parallel Q_k^1 \dots \rangle_i \quad (4.4)$$

At lower levels of abstraction, this representation appears as Formulas 4.5 and 4.6. In Formula 4.5, each process is launched by its own sending operation, while the ordering of process completion is chosen non-deterministically.

$$Q_i^0 = \dots t_j! \tau_i; t_k! \tau_i; \{ \{t_j? \tau_{i'}; t_k? \tau_{i'}\} \sqcap \{t_k? \tau_{i'}; t_j? \tau_{i'}\} \} \dots \quad (4.5)$$

Formula 4.5 launches each sub-process by signaling that process through its respective t channel using the same message type τ . Signaling operations using t and τ are *hidden* from how BPEL syntactically represents its `<flow>` construct. Since this signaling is non-blocking, both sub-processes can be modeled in PROMELA as running in parallel. On simulation, the message sequence chart generated by SPIN would show interleavings of messaging events between more than one sub-process, indicating parallel execution.

$$Q_j^1 = t_j? \tau_i; ..; t_j! \tau_{i'}; Q_j^1 \quad (4.6)$$

Q_j^1 in Formula 4.6 receives launch message τ_i through its channel t_j , executes its activities inside the ellipsis ' \dots ', and finally sends completion message $\tau_{i'}$ back through t_j . Q_k^1 is not shown, but is identical to Q_j^1 except that activation and completion uses channel t_k and activities inside its ellipsis may be different.

Thus, we can simulate behavior of the BPEL `<flow>` in PROMELA as an ordered launch of two or more processes, parallel execution of those processes, and eventually by some unordered completion. All messaging is done via a synchronous channel assigned to each process launched. Table 4.2 shows Formula 4.5 in both its original BPEL (top half) and its PROMELA implementation (bottom half). Maintaining a counting semaphore can provide an alternative approach to model launch and completion sequences. In that case the semaphore gets incremented with each child launched and decremented with each child completing.

4.3.7 Interprocess Dependencies

BPEL control links enact dependencies between pairs of basic activities such that each activity is associated with a distinct partner link and distinct child element inside some `<flow>`. Orchestration process Q mediates interaction between source web service P_s and its target P_t as described in Formulas 4.7 - 4.9. In Formula 4.7 P_s simulates a control link fault through a non-deterministic selection of link status in which 0 indicates a fault. Formula 4.8 has Q listen over channel a for link status γ . It then relays a join condition that is a Boolean function B of γ and possibly other variables over channel d to P_t .

In target service P_t , statement $d?e(\delta)$ listens for join condition δ over channel d . Immediately after this statement, we could have modeled exception/compensation handling if δ indicated a fault. Doing so is reserved for future work. Eventually,

some successor activity in P_t will be sending its result and status back to Q . The value returned will depend on a guarded choice ' $|$ ' involving the suppress join failure attribute σ in the target activity. If join failure is being suppressed, signified by σ , then return status from all successor activities will be a '1' indicating a suppression of a fault. If we wish to propagate any join failure, signified by the guard $\neg\sigma$, then δ will be propagated to all successor activities inside P_t . Finally, Q receives the results from successor activities.

The modeling of interprocess dependencies added considerable complexity and brittleness to the prototype. Ongoing work focuses on more cleanly implemented generators of machine verifiable models.

$$P_s = \dots \{a!b(0) \sqcap a!b(1)\} \dots; P_s. \quad (4.7)$$

$$Q = \dots a?b(\gamma); d!e(B(\gamma, \dots)); \dots d?f(\delta); \dots; Q. \quad (4.8)$$

$$P_t = \dots d?e(\delta); \dots \left\{ \left\{ \sigma \rightarrow d!f(1) \right\} \mid \left\{ \neg\sigma \rightarrow d!f(\delta) \right\} \right\} \dots; P_t. \quad (4.9)$$

Table 4.3 shows an excerpt of BPEL code for the links construct followed by its PROMELA model. The message type b and link status c do not appear in the BPEL artifact. Instead, they are generated from control link a .

4.3.8 Assumptions

Two classes of assumptions are considered here; those that impact state space size and those that reflect fault-proneness. The prototype addresses the former class

of assumptions with the latter reserved for future work. Syntactically, assumptions are encoded as attributes (i.e., name-value pairs). Since these are not defined in the BPEL specification, its schema may need to be extended to include these names, or less elegantly, these may be included as markup inside the `<documentation>` element.

Model assumptions that impact state space size concern atomicity, synchrony, and parallelism. The BPEL execution model recommends that each basic activity execute atomically, prohibiting any interleaving of basic activities. Hence by default, the prototype encapsulates channel statements of a basic activity inside a PROMELA *atomic* clause. Not all BPEL execution engines observe this assumption. Relaxing this assumption by omitting this clause will allow interleaving that can result in a less tractable model.

The BPEL execution model also presumes that orchestration of web services is done asynchronously, via buffers. Strengthening this assumption to synchronous or non-buffered interaction may result in a smaller state space. Determining under what circumstances this is allowed was described in [21, 51].

Assumptions that determine how PROMELA should simulate parallelism concern launch and completion sequences for child activities of a `<flow>`. Ordered launch, parallel execution, and ordered completion results in the smallest state space, but this is the least realistic amongst the four combinations. For example, assuming an ordered completion of parallel activities imposes a lock-step synchronization that assumes that all child activities must complete and do so in the specified sequence.

More realistically, unordered completion models child processes completing at their own pace, but at the expense of a larger state space. Section 4.5 describes how a BPEL markup with assumption type attributes gets translated into PROMELA.

4.3.9 Limitations

Although we currently do not model fault-prone services or basic activities, they nonetheless can be modeled in a manner similar to how BPEL control links model and propagate join conditions. Recall the situation in which join failure was not suppressed. In that case, activities within the target service following the location of failure propagated the result of the join condition. This modeled cancellation of successor activities in BPEL and can be adapted to modeling fault-prone services. Positing a fault-prone service can be accomplished by inserting the attribute `fault-Prone="yes"` into the `<partnerLink>` declaration. Similarly a fault-prone basic activity can propagate its error condition to all basic activities dependent on its results.

Modeling cancellation interruption or premature termination by an external error signal is currently not done. These may require a two-place buffer (or two single-place buffers) for each activity inside some scope. This mounting of a virtual *stop button* onto an orchestration requires implementation of some form of priority queue [76]. Cancellation may require modeling missing reply type faults caused by orphaned inbound messaging activities like `<receive>` to preserve the progress property. Extending the prototype to consider fault proneness and cancellation is left for

future work.

We do not implement channel mobility like that described by service interaction Pattern 11 *Request with referral* in [9]. This pattern can be specified in the Pi-Calculus; however, the feasibility of sending the name of a channel over a channel to emulate runtime selection of a certain candidate web service is left for future work. Our prototype currently inserts PROMELA labels that support verification-time detection of deadlock and progress. However, the coding of any orchestration-specific assertions and temporal logic properties is presently done manually. When modeling parallelism, generation of all $k!$ permutations of completion sequences for k children of the `<flow>` element is currently not done by the prototype. Doing so would result in an intractably large model as the number of children increase. Addressing this or alternate implementations (i.e., counting semaphores) is left for future work.

4.4 Purchase Order Case Study

As a running example, we used an abbreviated version ⁶ of the BPEL artifact for the Purchase Order Process which appears in the WS-BPEL Specification [4]. It is reproduced here as Table 4.4, showing only those elements and attributes used by the prototype translator. For example, attributes representing the preamble inside the `<process>` element including lines 02-11 were omitted since they do not influence the construction of the model. Among the basic activities, attributes in the `<assign>`

⁶ Download from: <http://www.osoa.org/display/Main/Relationship+between+SCA+and+BPEL>

activity were omitted, since they would not have resulted in any channel-related statements in the model.

We defined one service per partner link, abstracting away operation and port-Type attributes. A finer grained model would have defined a PROMELA proctype for each combination of partner link, port type and operation. This fine granularity would have abstracted away any dependencies we wish to capture between operations inside any given web service. By defining one service per partner link as we do here, we can later replace that service's automatically generated sequential composition with its own orchestration. In this way, machine verification of hierarchical compositions become possible.

The expression in Formula 4.10 denotes this case study as word $w \in L^3$. Note that the regular expression for L^3 was defined in Formula 4.1 at the activity level, effectively hiding interactions with some hypothetical web services.

$$w = \left\langle_{01} p_{13}, p_{15}, p_{17}, \nu_{24}, \nu_{25}, \nu_{26}, \nu_{27}, \right. \\ \left. \left\langle_{30} \beta_{31}; \left\langle_{36} l_{41}, \langle_{43} \beta_{44}; \beta_{50} \rangle_{43} || \langle_{59} \beta_{60}; \beta_{67}; \beta_{72} \rangle_{59} \right\rangle_{36}; \beta_{76} \right\rangle_{30} \right\rangle_{01} \quad (4.10)$$

All basic activities except the `<assign>` on line 44 of Table 4.4 involve orchestrating the sending and/or receiving of messages through partner links. All basic activities in Formulas 4.11 - 4.16 bind some channel operation to a variable name, or more specifically, the message type of the variable name.

$$\beta_{31} = p_{13}! \nu_{24} \quad (4.11)$$

$$\beta_{50} = p_{17}? \nu_{24} ; p_{17}! \nu_{27} ; \{l_{41}! \nu_{41}(1) \sqcap l_{41}! \nu_{41}(0)\} \quad (4.12)$$

$$\beta_{60} = p_{15}? \nu_{24}(\gamma_{60}) \quad (4.13)$$

$$\beta_{67} = p_{17}! \nu_{26} \quad (4.14)$$

$$\beta_{72} = \{\sigma_{60} \rightarrow p_{15}! \nu_{25}(1)\} | \{\neg \sigma_{60} \rightarrow p_{15}! \nu_{25}(\gamma_{60})\} \quad (4.15)$$

$$\beta_{76} = p_{13}? \nu_{25} \quad (4.16)$$

Formula 4.11 describes `<receive>` activity β_{31} that uses the '!' operator to *send* a purchase order ν_{24} over channel p_{13} representing the partner link for the OrderProcessing Service. The `<variable>` name attribute in a `<receive>` activity causes the prototype to generate a *sending* operation from web service to orchestration. Using the '?' operator, the corresponding expression on the orchestration side will be otherwise identical except the channel expression will be *listening* for the contents of ν_{24} . Similarly, the `<reply>` activity β_{76} models the Order Processing service as listening for invoice ν_{25} .

Control links enable the specification of interprocess dependencies. Basic activities β_{50} , β_{60} , and β_{72} involve control link channel l_{41} , with the β_{50} addressing the control link explicitly. Recall from the BPEL listing in Table 4.4 that the source of l_{41} is the `<invoke>` activity β_{50} . After receiving purchase order ν_{24} and responding with availability ν_{27} , activity β_{50} non-deterministically decides if ship-to-invoice ν_{41} is

correct indicated by the qualifier (1) or incorrect by (0).

The invoke activity β_{60} located at the destination of l_{27} , implicitly refers to that control link as join condition γ_{60} . Although our case study involves only one control link, in general, the join condition is the result of some Boolean expression over multiple control links in the target activity. Join condition δ_{60} indicates success or failure of the control link, while the suppressJoinFailure attribute σ_{60} , controls propagation of any failure to subsequent channel statements. By default the failure is not suppressed and will be propagated to the remaining channel statements inside the web service. In this case, the failure propagates to the <receive> activity β_{72} . Recalling the earlier discussion of interprocess dependencies in Section 4.3, an actual model distributes tasks specified in activities β_{50} , β_{60} , and β_{72} between the orchestration and its web services.

4.5 BPEL to Promela

As an implementation of a typed CSP-like process algebra, a PROMELA artifact is an list of message types, channels, and process declarations. The idea is to map the structural specification offered by the BPEL artifact along with assumptions and hidden variables to a behavioral specification in PROMELA as we have done for individual behavioral patterns in Section 4.3.

We generate a three-part PROMELA model of how BPEL orchestrates its interaction with its environment, with each part requiring its own algorithm. As the

scaffolding from which rules hang, these algorithms provide a context in which these rules will fire. Hence, when discussing each algorithm, we will be presenting selected production rules.

The first algorithm generates declarations of message types, channels, and variables, each drawn from BPEL partner link and control link declarations. Generating PROMELA declarations only requires access to the BPEL declarations. The second algorithm generates a model of each service that interacts with the orchestration. A service is represented as a sequential composition of all basic activities that reference a given partner link, based on an in-order traversal of the BPEL artifact. Generating a service process definition requires access to both basic activities and BPEL declarations. The third algorithm generates a model of the orchestration being verified. Its construction will depend on the type of structured activity (i.e., `<flow>` or not). Generating orchestration process definitions is the most complex of the algorithms, requiring access to basic activities, their enclosing structured activities, and BPEL declarations. At the top level, Algorithm BPEL2PROMELA in Table 4.5 is a three-step process, with each step described in the sub-sections that follow. Since all three algorithms use production rules inside their leaf-level subroutines, we formulate these rules in the next sub-section.

4.5.1 Formulating Rules

This sub-section introduces the principles underlying the production rules used for generating PROMELA code. Evaluating the left-hand side (i.e., guard condition) of each rule entails examining the contents of specific BPEL elements. The right-hand side of each rule generates a snippet of PROMELA code, given the portions of BPEL code referenced on the left-hand side. These rules are encapsulated in the various children of the PRINT subroutines located throughout the three algorithms.

We can conceptualize any BPEL artifact, including attributes representing modeling assumptions, as a collection of four-tuples in some relation $R \subseteq C \times T \times A \times V$. For our prototype, we chose to pre-process the artifact by converting it into a four column table. Other implementations (e.g., using XPATH expressions) would be equally satisfactory. Now suppose tuple $r = (c, t, a, v) \in R$. The first entry, context $c \in C$, denotes that tuple's ancestry of element identifiers. For example, if each element is identified by its line number, then by inspection of Table 4.4, the collection of tuples comprising `<invoke>` activity β_{50} share ancestry `01:30:36:43:50`. The ancestry for the `<flow>` activity is simply `01:30:36`. Thus the context entry c enables us to refer to an entire BPEL element $F \subseteq R$, be it for all attributes inside some basic activity, or for all basic activities inside some structured activity. The second entry, tuple type $t \in T$, denotes the role of tuple $r \in F$, that can be either 'partnerLink', 'variable', 'structOpen', 'structClose', 'link', 'basic', and 'assume'. Finally, the third and fourth entries $a \in A$ and $v \in V$ denote attribute name and value copied directly

from the BPEL source code for one of its elements F .

Given the definitions in the previous paragraph, we describe rules that fire within the context of each of the three algorithms listed inside Algorithm BPEL2PROMELA. Each rule is comprised of an antecedent and a consequent. The antecedent is a Boolean expression over a set of terms, where each term is an *assertion* involving one or more attributes. We treat each attribute reference as an atom in a first-order logic, expressing that atom as pair (t_i, a_i) , or for assumptions as triple (t_i, a_i, v_i) . We define a primitive assertion as one that tests for the *existence* of some tuple $r_k = (c_k, t_k, a_k, v_k) \in F_k$ which evaluates to true only if $(t_i \equiv t_k) \wedge (a_i \equiv a_k)$. Additionally, we define a non-primitive assertion as one that involves some comparison between the *values* of two 4-tuples. In addition to asserting the existence of each tuple in the comparison, non-primitive assertion " $(t_i, a_i) \equiv (t_j, a_j)$ " further implies the equivalence of their respective *values* $v_i \equiv v_j$. If the Boolean expression over these assertions evaluates to 'true', then the antecedent is said to be *satisfied*, and all attributes in all matching elements will become *visible* to the consequent.

The consequent is comprised of a sequence of tuple references and unquoted string constants that specify the PROMELA expression to be generated. For tuple references inside the consequent, our prototype outputs the value v_k corresponding to some tuple reference (t_k, a_k) . Beneath each rule, we provide an example of its application from our case study. These rules are not applied in isolation. Rather, they are evaluated inside children of the PRINT subroutines which occupy the leaf

level of the algorithms that follow.

4.5.2 Generating Declarations

PROMELA declarations collectively refer to message types, channels, and variables. Algorithm GENERATEDECLARATIONS in Table 4.6 produces each type of declaration using an in-order traversal of BPEL `<variable>`, `<partnerLink>` and `<link>` type elements. The traversal uses an XML primitive of the form:

$$d' = \text{LOCATENEXTELEMENT}(F, t, d)$$

This primitive searches inside BPEL element F for the next occurrence of a child element of type t starting at displacement d . It returns displacement d' of the next occurrence, or 0 if no additional occurrences were found. The child occurrence pointed to by d' is then used in the appropriate child routine of PRINT to generate declarations in PROMELA.

Generation of variable and channel declarations from `<flow>` type activities starting at line 16 is less trivial. For each BPEL `<flow>` construct, subroutine GENFLOWMTYPES on line 19 generates two message types, one for activating each parallel process and the other indicating that each process had completed. Each child process spawned by the `<flow>` activity has its own channel through which activation and completion messages pass, which are generated by GENFLOWDECL.

Rules that generate declarations are by far the simplest, usually having antecedents comprised of a single primitive assertion. Subroutine GENCHANDCL generates channel declarations and contains Rule 1, which produces a relatively interesting

consequent. A closer look at channel declarations provides insight into assumptions concerning synchrony. Implied in the execution model for the BPEL Language is that each orchestrated web service operates asynchronously. We can encode this assumption as the attribute at line 017 of the BPEL listing in Table 4.4 as `buffsize=0`, which overrides the BPEL default for `buffsize` of 1. This causes Rule 1 to generate a rendezvous or synchronous or zero-place channel. Related work [21, 51] identifies the conditions under which one may model web services *as if* they operated synchronously. If one or more services in a composition is synchronizable, and if that portion was observed to be reliable, then we can realistically model that portion as a composition of synchronous *rendezvous* channels. During verification, a model with such zero-place channels will tend to have a smaller and more tractable state space.

Table 4.7 lists the PROMELA declarations for the Purchase Order Process. Services located at the target end of each link will have their channels declared with an additional variable for the join condition being propagated. This can be seen for channel `pltPay` on line 15 for the Payment service. Although control link message type `xSti` is declared in the BPEL source, subroutine `GENLINKDCL` generates declarations for both channel `ltxStI` and variable `vxStI` to enact the interprocess dependencies described in Section 4.3. Since there can be more than one set of control links, with each set associated with its target activity, the prototype suffixes each variable name with that activity's location *actId*. Similarly, there can be more than one `<flow>` activity each with multiple children, hence it also suffixes these declarations to assure

uniqueness of names.

4.5.3 Generating Services

A PROMELA artifact must model how the BPEL orchestration under test interacts with the web services that make up its environment. This subsection describes how PROMELA code is generated for each web service.

In the version of the Purchase Order Process used in our case study, the environment includes three services: an order processing service operating via partner link *pOP*, warehouse service via *pWhs*, and payment service via *pPay*. Each of these three services are modeled as a separate process (i.e., PROMELA *proctype*). A PROMELA process definition modeling a web service includes a process declaration, followed by a body that includes a sequence of sending '!' or receiving '?' channel statements, finally followed by a closing portion of code. Modeling each service as a sequential composition of channel statements reflects the assumption that each service maintains its state as an ordered succession of transitions. This is in keeping with the requirement that all web service compositions, including interacting web services, must have unique starting and terminating message events. However, interactions in between do not necessarily need to implement a sequential composition. Services having more complex modes of interaction can be substituted for these automatically generated process definitions and then model checked.

Lacking any further information, we define a service as an endpoint of a partner

link that exchanges messages sequentially with an orchestration. Hence, the prototype generates channel statements as a sequential composition in the same order as a partner link's variables are referenced. For the case study, these services are defined in Formulas 4.17 - 4.19. Each formula is in two parts. The upper part is at the activity level while the lower part is at the attribute level. We derive the latter expressions by substitution of Formulas 4.11 - 4.16 into the former.

$$\begin{aligned}
P_{13} &= \beta_{31} ; \beta_{76} ; \textit{progress} ; P_{13}. \\
P_{13} &= p_{13}! \nu_{24} ; p_{13}? \nu_{25} ; \textit{progress} ; P_{13}. \tag{4.17}
\end{aligned}$$

Order Processing service P_{13} sends purchase order ν_{24} over channel p_{13} to the orchestration and awaits receipt of invoice ν_{25} . Intuitively, this service represents a customer's process interacting with both endpoints in the orchestration. A rule that fires in the presence of the **createInstance** attribute of the **<receive>** activity will generate a **progress** label in PROMELA. This label represents the customer's perception of a live or active orchestration.

$$\begin{aligned}
P_{15} &= \beta_{60} ; \beta_{72} ; P_{15}. \\
P_{15} &= p_{15}? \nu_{24}(\gamma_{60}) ; \{ \sigma_{60} \rightarrow p_{15}! \nu_{25}(1) \mid \neg \sigma_{60} \rightarrow p_{15}! \nu_{25}(\gamma_{60}) \} ; P_{15}. \tag{4.18}
\end{aligned}$$

Both Payment service P_{15} and Warehouse service P_{17} interact with the orchestration in a manner that is hidden from customer view. The Payment service

propagates join condition γ_{60} based on **suppressJoinFailure** attribute σ_{60} by the inter-process dependency pattern described earlier. The Warehouse service operates at the source end of the control link, hence the non-deterministic selection of outcome to be sent over control link l_{41} .

$$\begin{aligned}
P_{17} &= \beta_{50}; \beta_{67}; P_{17}. \\
P_{17} &= p_{17}?\nu_{24}; p_{17}!\nu_{27}; \{l_{41}!\nu_{41}(0) \sqcap l_{41}!\nu_{41}(1)\}; p_{17}!\nu_{26}; P_{17}. \quad (4.19)
\end{aligned}$$

Algorithm GENERATESERVICES in Table 4.8 generates code for these services. After compiling a list of services to generate in line 2, the algorithm represents each service as a separate PROMELA process. The following paragraphs provide a rule for each of the subroutines GENPROCESSHEADER, GENBASICSVCSTMTS, and GENPROCESSTRAILER.

As part of subroutine GENPROCESSHEADER, Rule 2 generates the process declaration at the first element for which its antecedent is satisfied. Its closing form, Rule 2' has an otherwise identical antecedent, except the antecedent is negated and included in subroutine GENPROCESSTRAILER. Rule 2' fires when no more basic activities involving the partner link can be found. The rules in between generate channel-related statements, with some constructs generated based on assumptions encoded in the BPEL artifact.

Subroutine GENBASICSVCSTMTS generates the body of a web service definition, which include channel statements and may also include an atomic clause or

progress label. Basic activities like the ones in this case study must execute atomically [100, 139], so by default the prototype generates an atomic clause unless otherwise specified by attribute `atomic=no`. Rule 3 and its closing form produce the PROMELA code for modeling atomic behavior. The scoping of these and other assumptions can be confined to specific basic activities, or alternatively, all basic activities in a composition, depending on the placement of the assumption type attribute inside the (hierarchical) BPEL artifact.

As pointed out by [47, 76], one cannot always assume that each basic activity must execute atomically. Some implementations of middleware layers can permit interleaved execution of more than one basic activity via the same partner link. Relaxing the atomicity assumption produces a more realistic model, at the expense of a larger state space. If only one service has been observed to violate the atomicity assumption, it would be useful to model only its channel events as not being atomic.

Channel statements are generated by rules similar to Rule 4. This particular rule generates an expression that listens over a channel (e.g., `pltWhs`) for a message (e.g., `mtPO`) inside service `pWhs`. The sending end of this channel resides in the orchestration. It is generated by a complementing form Rule 4c that is otherwise identical to the listening end, except that the channel operator in the consequent is reversed. Notice that *input* and *output* variables as in β_{50} are so named from the standpoint of the service rather than orchestration. Thus, it is the *service* that listens using the '?' operator for the BPEL input variable `inVar`.

The remaining two channel expressions within the atomic scope of service `pWhs`, which supports the example `<invoke>` activity on line 50 of the BPEL artifact, were each generated by its own rule. The rule for the first expression generates a send of an availability message of type `mtAv1` over channel `pltWhs` back to the orchestration. The rule for the second channel expression generates code for the sending or *source* end of a control link (i.e., `ltxStI`). Control links always operate synchronously, causing the target activity (i.e., β_{60}) to block until the source activity (i.e., β_{50}) completes. The rule that generates the expression for its source (sending) end, located inside the service, is always placed last in any sequence of messages for that basic activity. The rule that generates the expression for its target (listening) end, located inside the orchestration, is placed before any channel statements that interact with any other services. Doing so blocks the start of the activity at the destination or target end until the activity at the source end completes. Since all three message events in the service occurs inside the same basic activity β_{50} , and since we assume atomic execution, they are all placed inside the scope of the same atomic construct, as can be seen in the `pWhs` service in Table 4.9.

4.5.4 Generating Orchestrations

An orchestration can be modeled in PROMELA as a collection of processes, one for the `<process>` scope and one for each child of `<flow>` activities. Each orchestration process (i.e., PROMELA proctype) in turn mediates interaction between web

services.

$$\begin{aligned}
Q_{30} &= \beta_{31}; t_{43}!\tau_{36}; t_{59}!\tau_{36}; \\
&\quad \{ \{t_{43}?\tau_{36'}; t_{59}?\tau_{36'}\} \sqcap \{t_{59}?\tau_{36'}; t_{43}?\tau_{36'}\} \}; \\
&\quad \beta_{76}; end; Q_{30}. \\
\\
Q_{30} &= p_{13}?\nu_{24}; t_{43}!\tau_{36}; t_{59}!\tau_{36}; \\
&\quad \{ \{t_{43}?\tau_{36'}; t_{59}?\tau_{36'}\} \sqcap \{t_{59}?\tau_{36'}; t_{43}?\tau_{36'}\} \}; \\
&\quad p_{13}!\nu_{25}; end; Q_{30}. \tag{4.20}
\end{aligned}$$

Formulas 4.20 - 4.22 showcase the modeling of parallelism via channels. After completion of β_{31} in which a purchase order is received, process Q_{30} implements the *<flow>* in two stages. Q_{43} and Q_{59} are sequentially launched via channels t_{43} and t_{59} , respectively. After executing in parallel the two processes terminate by a non-deterministically chosen sequence of channel statements. Finally, Q_{30} performs β_{76} by returning an invoice over channel p_{13} to the order processing service which initiated this transaction.

$$\begin{aligned}
Q_{43} &= t_{43}?\tau_{36}; \beta_{50l}; t_{43}!\tau_{36'}; Q_{43}. \\
\\
Q_{43} &= t_{43}?\tau_{36}; p_{17}!\nu_{24}; p_{17}?\nu_{27}; t_{43}!\tau_{36'}; Q_{43}. \tag{4.21}
\end{aligned}$$

Child processes Q_{43} and Q_{59} are otherwise straightforward sequential compositions, each launched and completed via their own synchronous channels.

$$\begin{aligned}
Q_{59} &= t_{59} ? \tau_{36} ; \beta_{60} ; \beta_{67} ; \beta_{72} ; t_{59} ! \tau_{36'} ; Q_{59}. \\
Q_{59} &= t_{59} ? \tau_{36} ; \\
&\quad l_{41} ? \nu_{41}(\gamma_{41}) ; p_{15} ! \nu_{24}(\gamma_{41}) ; p_{17} ? \nu_{26} ; p_{15} ? \nu_{25}(\gamma_{60}) ; \\
&\quad t_{59} ! \tau_{36'} ; Q_{59}.
\end{aligned} \tag{4.22}$$

Algorithm GENORCHESTRATION in Table 4.10 generates orchestration process definitions from the BPEL artifact. It is by far the most intricate, due to `<flow>` constructs which, true to the recurrences of Formula 4.1, require special treatment. The overall approach is to perform a breadth first traversal starting at the `<process>` scope and proceeding through the children of each `<flow>` construct. As each immediate child is encountered the algorithm generates code for non-flow activities, along with any launch and completion sequences, then places that child onto a queue named *agenda*. Once the activities and launch and completion sequences of all children are complete, the closing portion of the process definition is generated. The algorithm then proceeds to the next child on the agenda, treating the child's element as the current scope, generating a process definition, and placing any of the current child's children onto the queue. The algorithm terminates when the queue becomes empty. The next paragraph describes this algorithm in further detail.

Lines 1-7 generate the top-level process header, naming the process after the first structured activity type encountered. The while loop on lines 8-41 does one or more of the following three tasks: (i) generate a portion of the process body on lines

10-31, (ii) produce the process trailer on lines 32-36, or (iii) produce the header for the next child process on lines 37-41. Which portions of the process body get generated will depend on whether the activity is a `<flow>`, a choice, an opening or closing portion of a structured activity, or a basic activity. Each of these cases correspond to the behavioral patterns described earlier in Section 4.3.

Table 4.11 shows the PROMELA code for the top-level orchestration process, while Table 4.12 shows the child orchestration processes. Top-level process `sequence30` is so-named for the `<sequence>` activity on line 30 of Table 4.4 which encloses all other activities in the composition. Its first and last channel statement communicates with the outside world. In between are statements generated by the `<flow>` construct beginning on line 36. As the result of the breadth-first traversal over children, processes `sequence43` and `sequence59` follow. If these processes were to have encapsulated a `<flow>`, there would have been more process definitions after these. The `printf` statements were manually inserted to provide instrumentation for the control link construct.

4.6 Related Work

The authors assessed tool support for the formal verification of safety- or fiscally-critical service oriented architectures (SOA) [120], e-science SOA [126], and use cases that demand automating the process of conversion [119]. Model capture was addressed by the authors in an earlier work involving colored Petri nets [121]. That

work augmented one of the existing utilities reported in [100] for converting BPEL to Petri nets, to further reflect hierarchy and data type (color in CPN terminology), while improving model layout.

To date, the most mature conversion tool support is offered by the WSAT utility [50] for converting BPEL to modeling languages for the SPIN and SMV model checkers. WSAT uses XPath guards to define a guarded finite state automata (GFSA) that serves as an intermediate representation. From there, WSAT performs a *synchronizability* analysis to determine if a BPEL composition can be treated as if that entire composition were a collection of synchronously communicating web services [51]. A synchronizable composition results in a model having a smaller state space. Next, WSAT performs a realizability analysis to synthesize a set of GFSA that function as peers that communicate with a single GFSA that orchestrates interaction. Finally, WSAT translates these XPath expressions into PROMELA.

We observed that the PROMELA code so generated does not make clear the assumptions used, nor does the code seem intended for human interpretation. An error trace from SPIN becomes difficult for a human to interpret when faced with a rather large artifact. For example, WSAT translated the BPEL artifact for the Loan Approval Process which appears in the WS-BPEL Specification into a PROMELA artifact containing almost 200 guards.

A prototype tool for translating BPEL into PROMELA was mentioned in [76]. It

supports parameterization by degree of asynchrony based on a hierarchy of communication models. It generalizes on earlier work in synchronizability [51]. Of interest in [76] is their description of tool support for identifying the simplest model (i.e., in terms of queuing assumptions) that nonetheless retains some specified property (e.g., boundedness). This work did not address the atomicity assumption, nor was there a clear description of how their code might be extended to address it. Furthermore, there was no detailed description of how they went about simulating parallelism.

A means of translating BPEL to PROMELA via an open workflow net was described in [87]. As part of the TOOLS4BPEL initiative⁷, the BPEL2OWFN workbench employs a form of flexible model generation into its intermediate form, providing a tailored approach to generate a compact model. The specifications were coarser-grained than our approach. Furthermore, a current incarnation of the tool used for translating an open workflow net to PROMELA resulted in a single PROMELA *proctype*, making it difficult to simulate in Spin.

The LTSA modeling and verification toolset is available for download⁸ and runs inside the Eclipse environment. It is accompanied by an introductory text [90], and provides tool support for conversion from BPEL to a CSP-like process algebraic notation known as Finite State Process (FSP). LTSA uses an Eclipse plug-in known as WS-Engineer to do this conversion, the results of which can be compared to those generated from user-specified message sequence charts [49]. It can model assumptions

⁷ <http://www2.informatik.hu-berlin.de/top/tools4bpel/>

⁸ <http://www.doc.ic.ac.uk/ltsa/>

concerning the presence or absence of synchrony, atomicity, and parallelism, but it is not clear whether this tool supports scoping these assumptions to specific basic or structured activities.

Conversion of BPEL to workflow type Petri nets is implemented in the tool BPEL2PNML [100, 114]. It can model a robust set of concerns that involve all three classes of assumptions described earlier. This approach does not seek to exploit the explicit channel semantics of PROMELA, nor was it obvious if different parts of a composition can be subject to different assumptions.

Once a model is expressed in PROMELA, there are a number of tools for generating test suites [153] or for converting PROMELA to models suitable for other verification environments. One such tool offered by the VeriTech Project provides a wide range of tool choices to manage, reframe, or effectively sidestep issues like the state space explosion problem [57]. In addition to the widespread use of SPIN [66], the VeriTech tool helped motivate our choice of PROMELA, since PROMELA can provide a gateway to other verification formalisms.

4.7 Chapter Summary

We described an extendible mapping of BPEL artifacts to machine-verifiable models written in PROMELA – the modeling language used by the SPIN model checker. Extendibility allows the analyst to confine assumptions to portions of a BPEL artifact to influence the state space size of its model. Such a mapping first required us to

characterize the sublanguage of BPEL amenable to mapping to a finite state model. We characterized this sublanguage using recurrences that, in turn, generated the regular expression for that sublanguage. In addition to identifying the scope of applicability for this translation, that characterization later guided development of the top-level control structures used by the translation algorithms.

Since we wished to model orchestration *behavior*, given the *structure* of the BPEL artifact, we did these five things:

- Define the characterization from the *activity* level down to the *attribute* level of abstraction.
- Generate a model for web services that operate at the other end of each partner link.
- List several behavioral patterns we wish to model.
- Present an algorithm for generating each of the three parts of a PROMELA model that include declarations, services, and orchestration processes.
- Define production rules scoped within leaf-level routines appearing in each algorithm that enable us to fine tune models without major source code modifications.

Doing these five things enabled us to model how a BPEL artifact orchestrates interaction with its web services. We illustrated these ideas with a version of a well-known case study.

```

<partnerLink name= $a$  partnerLinkType= $b$  />
...
<variable name= $c$  messageType= $d$  />
<variable name= $e$  messageType= $f$  />
...
<invoke partnerLink= $a$  inputVariable= $c$  outputVariable= $e$  ... >
..

```

```

active proctype a() { .. b?d ; b!f ; .. }

```

Table 4.1: From BPEL variables to Promela channel statements

```

<process name= $Q_i^0$  ... >
{ ...
  <flow><  $Q_j^1$  > ... <  $Q_k^1$  >< /flow> }
  ...
}

```

```

..
active proctype Qj() { .. a?b ; .. a!d ; .. }
..
active proctype Qk() { .. c?b ; .. c!d ; .. }
..
active proctype Qi()
{
  ..
  a!b ; c!b ;
  /* Qj and Qk are now running in parallel */
  if ::(1) -> a?d ; c?d ;
    ::(1) -> c?d ; a?d ;
  fi;
  ..
}
..

```

Table 4.2: From BPEL <flow> to parallel Promela processes


```

...
<flow>
  <links> <link name=a /> </links>
  <structured>
    ...
    <basic ... >
      <sources> <source linkName=b /> </sources>
    </basic>
    ...
  </structured>
  ...
  <structured>
    ...
    <basic partnerLink=d... >
      <targets>
        <joinCondition> c and 1</joinCondition>
        <target linkName=b />
      </targets>
    </basic>
    ...
  </structured>
  ...
</flow>
...

```

```

/* source service */
..
if
:: (1) -> a!b(0); /* source */
:: (1) -> a!b(1); /* source */
fi;
..
/* orchestration */
..
a?b(c); /* target */
d!e(c && 1); /* target */
..
d?f(g); /* successor */
..
/* target service */
..
d?e(g); /* target */
..
if
:: (h==1) -> d!f(1); /* successor */
:: (h!=1) -> d!f(g); /* successor */
fi;
..

```

Table 4.3: From BPEL control links to a Promela pattern

```

01  <process name="purchaseOrderProcess"
..  >
12    <partnerLinks>
13      <partnerLink name="pOP" partnerLinkType="pltOP" .. />
15      <partnerLink name="pPay" partnerLinkType="pltPay" .. />
17      <partnerLink name="pWhs" partnerLinkType="pltWhs" .. bufsize="0" />
21    </partnerLinks>
23    <variables>
24      <variable name="vPO" messageType="mtPO" />
25      <variable name="vNvc" messageType="mtNvc" />
26      <variable name="vShl" messageType="mtShl" />
27      <variable name="vAvl" messageType="mtAvl" />
28    </variables>
30    <sequence>
31      <receive partnerLink="pOP" operation="placeOrder" variable="vPO"
32        createInstance="yes" .. > .. </receive>
36      <flow>
41        <links> <link name="xStl" /> </links>
43        <sequence>
44          <assign> .. </assign>
50          <invoke partnerLink="pWhs" inputVariable="vPO" outputVariable="vAvl" ..
51            atomic="no">
55            <sources> <source linkName="xStl" /> </sources>
57          </invoke>
58        </sequence>
59        <sequence>
60          <invoke partnerLink="pPay" inputVariable="vPO" .. >
64            <targets> <target linkName="xStl" /> </targets>
66          </invoke>
67          <invoke partnerLink="pWhs" inputVariable="vPO" outputVariable="vShl"
68            .. > .. </invoke>
72          <receive partnerLink="pPay" variable="vNvc" .. />
74        </sequence>
75      </flow>
76      <reply partnerLink="pOP" operation="placeOrder" variable="vNvc" .. > ..
77      </reply>
80    </sequence>
81  </process>

```

Table 4.4: BPEL (abbreviated version) of the Purchase Order Process

```

BPEL2PROMELA (BPEL)
01  GENERATEDECLARATIONS (BPEL)
02  GENERATESERVICES (BPEL)
03  GENERATEORCHESTRATION (BPEL)

```

Table 4.5: Generate Promela code from a BPEL artifact

```

GENERATEDECLARATIONS (BPEL)
01  ▷ Generate declarations from BPEL variables:
02  actId = LOCATENEXTELEMENT(BPEL, 'variable', 0)
03  while actId
04      PRINT(GENVARDECL(BPEL, actId))
05      actId = LOCATENEXTELEMENT(BPEL, 'variable', actId)
06  ▷ Generate declarations from partnerLinks:
07  actId = LOCATENEXTELEMENT(BPEL, 'partnerLink', 0)
08  while actId
09      PRINT(GENCHANDCL(BPEL, actId))
10      actId = LOCATENEXTELEMENT(BPEL, 'partnerLink', actId)
11  ▷ Generate declarations from links:
12  actId = LOCATENEXTELEMENT(BPEL, 'link', 0)
13  while actId
14      PRINT(GENLINKDCL(BPEL, actId))
15      actId = LOCATENEXTELEMENT(BPEL, 'link', actId)
16  ▷ Generate declarations from flows:
17  actId = LOCATENEXTELEMENT(BPEL, 'flow', 0)
18  while actId
19      PRINT(GENFLOWMTYPES(BPEL, actId))
20      children = CHILDRENOF(BPEL, actId)
21      while children
22          childId = FIRSTOF(children)
23          PRINT(GENFLOWDECL(BPEL, childId))
24          children = RESTOF(children)
25      actId = LOCATENEXTELEMENT(BPEL, 'flow', actId)

```

Table 4.6: Generate Promela Declarations

Rule 1: $\exists (partnerLink, plType) \Rightarrow$
`chan (partnerLink, plType) = [(assume, buffSize)] of {mtype}; /* actId */`
`chan pltWhs = [0] of {mtype}; /* 17 */`

```

001  /* Process Name: PurchaseOrderProcess */
002
003  /* based on BPEL partnerLinks: */
004  chan pltOP = [1] of {mtype}; /* 13 */
005  chan pltPay = [1] of {mtype, byte};
006  chan pltWhs = [0] of {mtype}; /* 17 */
007
008  /* based on BPEL variables: */
009  mtype = {mtPO}; /* 24 */
010  mtype = {mtNvc}; /* 25 */
011  mtype = {mtShI}; /* 26 */
012  mtype = {mtAvl}; /* 27 */
013
014  /* based on BPEL links: */
015  chan ltxStI = [0] of {mtype, byte};
016  mtype = {xStI}; /* 41 */
017  byte vxStI; /* 41 */
018
019  /* based on targets of links: */
020  byte sjf60 = 0; /* 60 */
021  byte jc60; /* 60 */
022
023  /* based on BPEL flow activity: */
024  mtype = {awake36}; /* 36 */
025  mtype = {done36}; /* 36 */
026  chan c43 = [1] of {mtype}; /* 43 */
027  chan c59 = [1] of {mtype}; /* 59 */
028

```

Table 4.7: Promela declarations for the Purchase Order Process

```

GENERATESERVICES (BPEL)
01  actId = LOCATENEXTELEMENT(BPEL, 'partnerLinks', 0)
02  agenda = CHILDRENOF(BPEL, actId)
03  serviceId = FIRSTOF(agenda)
04  while agenda
05      ▷ Model each web service as a separate PROMELA process:
06      PRINT(GENPROCESSHEADER(BPEL, serviceId))
07      actId = LOCATENEXTBASIC(BPEL, actId)
08      while actId
09          ▷ Generate body of service
10          PRINT(GENBASICSVCSTMTS(BPEL, actId))
11          actId = LOCATENEXTBASIC(BPEL, actId)
12      PRINT(GENPROCESSTRAILER(BPEL, serviceId))
13      agenda = RESTOF(agenda)
14      serviceId = FIRSTOF(agenda)

```

Table 4.8: Generate a model of services from a BPEL artifact

Rule 2: $((partnerLink, plName) \equiv (basic, pLink)) \Rightarrow$
 active proctype $(partnerLink\ plName)() \{ do :: \{$
 active proctype pWhs() { do ::{

Rule 2': $(partnerLink, plName) \not\equiv (basic, pLink) \Rightarrow \} od \}$
 } od }

Rule 3: $\exists ((assume, basic, atomic) \wedge$
 $(partnerLink, plName) \equiv (basic, pLink) \wedge$
 $((link, lDecl) \equiv (basic, target) \vee$
 $(variable, vName) \equiv (basic, inVar) \vee$
 $(variable, vName) \equiv (basic, outVar) \vee$
 $(link, lDecl) \equiv (basic, source)$
 $)) \Rightarrow$
 atomic {
 atomic {

Rule 4: $((partnerLink, plName) \equiv (basic, pLink) \wedge$
 $(variable, vName) \equiv (basic, inVar)) \Rightarrow$
 $(partnerLink, plType)?(variable, msgType); /* actId */$
 pltWhs?mtP0; $/* 50 */$

```

029  /* ----- */
030  /* ----- S E R V I C E S ----- */
031
032  active proctype pOP()    /* 13 */
033  {
034  do
035  :: {
036      pltOP ! mtP0;    /* 31 */
037      pltOP ? mtNvc;    /* 76 */
038      progress: skip;
039  }
040  od
041  }
042
043  active proctype pPay()    /* 15 */
044  {
045  do
046  :: {
047      /* receive result of join cond. -- */
048      pltPay ? mtP0(jc60);    /* 60 */
049      printf("60 jc60=%d\n", jc60);
050      /* propagate jc60 if sjf60 != 1 -- */
051      if
052      :: (sjf60 == 1) ->
053          pltPay ! mtNvc(1);    /* 72 */
054      :: (sjf60 != 1) ->
055          pltPay ! mtNvc(jc60);    /* 72 */
056      fi;
057  }
058  od
059  }
060
061  active proctype pWhs()    /* 17 */
062  {
063  do
064  :: {
065      atomic
066      {
067          pltWhs ? mtP0;    /* 50 */
068          pltWhs ! mtAvl;    /* 50 */
069          /* 50 origin of link */
070          if
071          :: (1) -> ltxStI ! xStI(0);
072              /* .. 50 link failure */
073          :: (1) -> ltxStI ! xStI(1);
074              /* .. 50 link success */
075          fi;
076      }
077      pltWhs ! mtShI;    /* 67 */
078  }
079  od
080  }
081

```

Table 4.9: Promela environment for the Purchase Order Process

```

GENERATEORCHESTRATION (BPEL)
01  ▷ Generate opening portion of first orchestration process:
02  scopeId = LOCATENEXTELEMENT(BPEL, 'process', 0)
03  actId = LOCATECHILD(BPEL, scopeId)
04  while ACTCLASS(BPEL, actId) ≠ 'structOpen'
05      actId = LOCATENEXTSIBLING(BPEL, actId)
06  scopeContents = ELEMENTCONTENTS(BPEL, actId)
07  PRINT(GENPROCESSHEADER(scopeContents, scopeId))
08  agenda = actId
09  while agenda
10      ▷ Generate body of orchestration process in PROMELA
11      if ACTTYPE(scopeContents, actId) ≡ 'flow'
12          children = CHILDRENOF(scopeContents, actId)
13          PRINT(GENCALLERCODE(scopeContents, children))
14          agenda += children
15          actId = LOCATENEXTSIBLING(scopeContents, actId)
16      else if ACTTYPE(scopeContents, actId) ≡ 'pick'
17          choices = CHILDRENOF(scopeContents, actId)
18          while choices
19              choiceId = FIRSTOF(choices)
20              PRINT(GENCHOICECODE(scopeContents, children))
21              choices = RESTOF(choices)
22              actId = LOCATENEXTSIBLING(scopeContents, actId)
23      else if ACTCLASS(scopeContents, actId) ≡ 'structOpen'
24          PRINT(GENOPENSTRUCT(scopeContents, actId))
25          actId = LOCATECHILD(scopeContents, actId)
26      else if ACTCLASS(scopeContents, actId) ≡ 'basic'
27          PRINT(GENBASICORCHSTMTS(scopeContents, actId))
28          actId = LOCATENEXTSIBLING(scopeContents, actId)
29      else if actClass(scopeContents, actId) ≡ 'structClose'
30          PRINT(GENCLOSEDSTRUCT(scopeContents, actId))
31          actId = LOCATENEXTSIBLING(scopeContents, actId)
32      if NULL(actId)
33          ▷ Generate closing portion of orchestration process:
34          PRINT(GENPROCESSTRAILER(scopeContents, scopeId))
35          agenda = RESTOF(agenda)
36          scopeId = FIRSTOF(agenda)
37      if scopeId
38          ▷ Generate opening portion of next orchestration process:
39          scopeContents = ELEMENTCONTENTS(BPEL, scopeId)
40          PRINT(GENPROCESSHEADER(scopeContents, scopeId))
41          actId = scopeId

```

Table 4.10: Generate a model of orchestration from a BPEL artifact

```

080  /* ----- */
081  /* ----- ORCHESTRATION ----- */
082
083  active proctype sequence30()
084  {
085  do
086  :: {
087      pltOP ? mtP0;    /* 31 */
088
089      /* flow36 start */
090      c43 ! awake36;
091      c59 ! awake36;
092      /* -- exec. flow36 in parallel -- */
093      c43 ? done36;
094      c59 ? done36;
095      /* flow36 end */
096
097      pltOP ! mtNvc;    /* 76 */
098
099      /* provide snapshot -- */
100      printf("vxStI:%d\n", vxStI);
101      printf("sjf60:%d\n", sjf60);
102      printf("jc60:%d\n", jc60);
103
104      /* destroy Instance -- */
105      end: skip;
106  }
107  od
108  }
109

```

Table 4.11: Promela orchestration for the Purchase Order Process

```

110 active proctype sequence43()
111 {
112 do
113 :: {
114     c43 ? awake36;
115     /* - assign at line 44 goes here - */
116     atomic
117     {
118         pltWhs ! mtP0;    /* 50 */
119         pltWhs ? mtAvl;   /* 50 */
120     }
121     c43 ! done36;
122 }
123 od
124 }
125
126 active proctype sequence59()
127 {
128 do
129 :: {
130     c59 ? awake36;
131     ltxStI ? xStI(vxStI); /* 60 */
132     printf("50 vxStI=%d\n", vxStI);
133     /* send result of join condition: */
134     pltPay ! mtP0(vxStI && 1); /* 60 */
135     pltWhs ? mtShI;    /* 67 */
136     pltPay ? mtNvc(jc60); /* 72 */
137     c59 ! done36;
138 }
139 od
140 }

```

Table 4.12: Promela orchestration for the Purchase Order Process (contd)

CHAPTER 5

AUTOMATING MODEL PRESENTATION

This chapter is based on the paper titled: *From Web Service Artifact to a Readable and Verifiable Model* [121]. Here we enhance Petri nets that have already been generated from a BPEL artifact according to a set of requirements. The result is a colored Petri net (CPN) having desired properties.

Models of web service compositions that are both readable and verifiable will benefit organizations that integrate purportedly reusable web services. CPN's are at once verifiable and visually expressive, capable of presenting subtle flaws in service composition. Constructing CPN models from Business Process Execution Language (BPEL) artifacts had been a manual process requiring human judgment. Building on results from the Workflow Community, we automate the mapping of artifacts written in BPEL to models used by CPN Tools – a formal verification environment for development, simulation, and model checking of colored Petri nets. We extend related work that already converts BPEL to Petri nets, to reflect hierarchy and data type (*color* in CPN terminology), while improving model layout. We present a prototype implementation that mines both a BPEL artifact, and the Petri net generated from

it by an existing tool. The prototype partitions the Petri net into subnets, lays them out, colors them, and generates their XML file for import into CPN Tools. Our results include depictions of subnets produced and initial simulation results for a well-known case study.

5.1 Chapter Introduction

Service oriented architectures (SOA) are commonly implemented as compositions of loosely coupled autonomous web services. Among other benefits, SOA's free developers from issues involving platform, implementation, and versioning.

These freedoms, however, render traditional testing techniques ineffective. Without access to source code of individual services that may not behave as advertised, unforeseen usage scenarios and implicit assumptions can cause race conditions or other undesirable forms of interaction. Machine verification helps guard against inclusion of badly designed, underspecified or incorrectly specified services, particularly services having incomplete or ambiguous descriptions. Nonetheless, otherwise properly designed and well-described services can unexpectedly fail when composed with other services.

For machine verification to be practical, architectural mismatches between source artifacts and models, and dependence on human proficiency in model capture must be minimized. These mismatches manifest themselves as (i) agglomerating the entire model onto one dense diagram, (ii) ignoring notions of *type* specified in

BPEL as partner links and variables, (iii) presenting models that violate a number of desirable properties of graph drawings. High assurance systems need models that are not only machine-verifiable, but also human-readable, since readable ones are easier to troubleshoot.

5.1.1 Context

This chapter builds on research reported in [100, 114] that resulted in tool BPEL2PNML that converts BPEL artifacts to place/transition type Petri nets. These nets were encoded in the Petri Net Markup Language (PNML) [15, 92] according to a formal semantics of control flows appearing in [100]. Starting with the results generated by this tool, we further developed a prototype that converts the PNML artifact to an XML file suitable for import into CPN Tools. The prototype preserves BPEL partner links and variables, notions lost by the BPEL2PNML tool. We then hierarchically partitioned and laid out these models for readability. The layout applies results from graph drawing research including the closely related topics of planar embeddings involving palm trees [68], visibility representations [10], and book embeddings [29].

We first describe the modeling formalism we chose as the destination for our conversion facility. CPN Tools is a formal verification environment based on the colored Petri net formalism that supports the modeling, analysis, and simulation of distributed systems, with an introduction [73] and tutorial [107] available. CPN Tools enhances conventional Petri net tool support through hierarchical construction, an

extended notion of data type or *color*, simulation monitoring, analysis techniques, and model checking.

5.1.2 Contributions

We automate model construction so that practitioners can concentrate on validating and verifying BPEL compositions rather than expending effort on model capture. Even modeling control flows among and between basic activities, structured activities, and crosscutting concerns like execution and remediation, can result in large and intricate models. To manage this scale and intricacy, we automate the process by which a Petri net is partitioned into subnets. Some subnets, like those for structured activities and crosscutting concerns, may remain large but must be laid out to facilitate human comprehension. Even smaller nets, like those for basic activities and simple applications of control links, would benefit from a simple and consistent layout. Much of the information specific to a BPEL process like notions of partner links and variables, need to be carried over to the model. We automate the encoding of partner links and variables into CPN Tools as *colors* and *color sets*, respectively.

5.1.3 Organization

The rest of this chapter is organized as follows: Section 5.2 reviews related work that emphasizes tool support for conversion from BPEL artifacts to Petri nets, introduces CPN Tools, and summarizes pertinent results from graph drawing research.

Section 5.3 bridges the gap between PNML artifact and the input used to compute the embedding. Section 5.4 describes the layout algorithm that produces the embedding. Section 5.5 describes the generation of the CPN Tools XML artifact from the embedding. Section 5.6 summarizes simulation results. Finally, Section 5.7 provides a summary and description of future work.

5.2 Related Work

This section focuses on related work that resulted in tool support for conversion from BPEL artifacts to machine verifiable models. Of particular interest is an assessment of tool support for conversion to Petri nets. Such an overview requires a brief look at verification tools and pertinent research into graph drawing.

To motivate this discussion, we introduce CPN Tools and describe why it provides a suitable destination for conversion from a BPEL artifact. Obstacles that prevent wider-spread adoption of CPN Tools to formal verification of web service orchestrations appear not to center on limitations of the tool. Rather, the obstacle is simply its lack of conversion tool support. This stands in contrast to other more limited verification environments that feature more mature conversion support.

This chapter builds on research reported in [100, 114] that describe and provide tool support for conversion of BPEL artifacts to ordinary place/transition type Petri nets. Others propose but provide no tool support for conversion to CPN, where [150] proposes an optimistic activity-level conversion without concern for the more

intricate control flows needed for a robust behavioral model. Finally, [148] uses an algebraic representation of activity-level constraints as a means of deriving a CPN, but detailed discussion of conversion tool support is lacking. A final obstacle concerns the geometry of the visible model. Graph drawing research into planar embeddings using the palm tree construction [68], and visibility representations [10], and book embeddings [29] will assist in overcoming this particular obstacle.

5.2.1 Verification Tools

In addition to being the leading research tool for colored Petri nets, we chose CPN Tools for its freely available Windows binaries, Java implementation, visual design formalism, and XML format for its modeling artifacts. At least two other verification environments provide attractive alternatives, including the Labelled Transition System Analyser (LTSA) [90] and UPPAAL [13] (named after the collaboration between Uppsala University in Sweden and Aalborg University in Denmark). The leading model checker, Spin, was not appropriate since, among other things, it did not offer a visual modeling formalism, nor was its freeform artifacts self-describing. The SMV model checker presents similar limitations, but seems to be intended more for embedding into other applications.

LTSA already offers an Eclipse plug-in that converts BPEL code to the process algebraic formalism that underlies its verification capabilities. Unlike a process algebraic formalism, colored Petri nets provide a more visual presentation that involve

nesting and lexical ordering of activities in BPEL, along with constructs like partner links and variables.

UPPAAL, a tool based on timed automata, has as yet no conversion tool support, however it poses mapping problems in addition to those posed by CPN Tools. Its models demand greater parsimony and abstraction due to its use of real-valued time constraints and invariants that increase state space size. The timed automata modeling formalism of UPPAAL complements modeling activities using CPN Tools as the authors had recently shown [126]. Solutions to mapping problems for CPN Tools can eventually be adapted to mappings from BPEL to UPPAAL, perhaps via CPN.

Before reviewing conversion tools we describe the modeling environment we chose as the destination for the conversion facility. CPN Tools is a formal verification environment based on the Colored Petri net formalism that supports the modeling, analysis, and simulation of distributed systems, with an introduction [73] and tutorial [107] available. It was developed by the CPN group at the University of Aarhus in Denmark.

Like LTSA and UPPAAL, CPN Tools supports model-driven development including syntax checking and code generation from the model under construction. It supports both timed and untimed Petri nets, and can generate and analyze either full or partial state spaces. It enhances conventional Petri net tool support through an extended notion of data type or color, hierarchical construction, and simulation monitoring. Unlike model checkers like Spin and LTSA it simulates directly with

its model rather than indirectly via a message sequence chart. Enforcement of color compatibility is done at edit time. Connecting arcs of different colors to the same transition causes the editor to highlight the transition and display an error. During simulation, instant feedback takes the form of the highlighting of successive markings. This lends insight into the actual behavior of the web service artifact. Figures 5.2 through 5.6 are screen shots from CPN Tools, resulting from running the prototype. CPN Tools has a variety of toolbox controls and binders for editing, viewing, simulation, monitoring (breakpoints), and state space analysis. The remaining tools support editing the CPN.

CPN Tools provides model checking capabilities for ordinary place/transition nets having uniquely labeled places and transitions through their state space generation/analysis tool. This tool enables one to verify properties expressed in computation tree logic. The state space explosion problem associated with exhaustive verification techniques like model checking have been extensively studied by [51] and [65], with their results incorporated into CPN Tools. Simulation and partial state space construction each provide confidence about this boundedness property, prior to construction of the full state space. Nonetheless, applications with intractably large state spaces need to use Petri net analysis techniques like *transition invariants*. One such workflow analysis tool is WofBPEL, which does not require generating a state space [100]. Since WofBPEL uses the identical PNML artifact as our prototype, future work may involve closer integration with this tool.

Finally, the programming language interface, CPN ML, enables reading or writing to a file, calculating some value, or sending/receiving information to/from external processes. This programmatic interface to the outside world may enable a CPN Tools model to become a central artifact in the instrumentation of ongoing processes.

The gap between the BPEL artifact and the XML format for CPN Tools is addressed by conversion tools, the state-of-the-art of which we discuss in the next paragraphs.

5.2.2 Conversion Tools

Bridging application and model, conversion tools must preserve enough behavior of web service artifacts to identify undesirable interactions. LTSA uses an Eclipse plug-in to do this conversion into their internal representation that can thence be compared to that generated from user-specified message sequence charts [49]. To date, the most mature conversion tool support is offered by the WSAT utility [50] for conversion to modeling languages for the Spin and SMV model checkers. Nonetheless, for reasons stated earlier, we chose instead to convert to CPN.

Tool support for conversion from BPEL to CPN is otherwise lacking. BPEL to CPN conversion described in [74, 150] provides neither tool support nor algorithm. Rather, they provide a set of rules for translating textual representations of each BPEL activity into a corresponding visual representation in CPN Tools. Although

helpful in conceptualizing BPEL activity-level constructs in CPN Tools, it does not algorithmically describe this conversion.

Conversion of BPEL structured activity types to Petri nets must account for execution concerns like cancellation, correlation set match-up, message-triggered instantiation, and boundedness [114]. Problems with converting BPEL to PNML, concern differing ways of modeling control flows. Seeing this as part of a larger problem known as *schema integration*, behavior inherent in BPEL artifacts must be carried over to other formalisms like Petri nets for them to be proven correct with respect to some specified set of properties. This translation requires a formal semantics of BPEL control links as specified in [100]. That paper also lists additional concerns involving dead path elimination, join conditions, suppressing join failures, and skipping behaviors. In addition to a formal semantics for BPEL control flows, it specifies local replacement rules for all activity types and control links. The BPEL2PNML tool translates a BPEL artifact into the place/transition nets encoded in PNML. It implements these concerns, which is why a seemingly simple BPEL artifact tends to result an unexpectedly large Petri net.

Using this tool,⁹ we first translated the BPEL artifact of the case study into PNML, providing the starting point for the subsequent conversion to CPN Tools. On the top side of Table 5.1 is BPEL code for an abbreviated version of the Purchase Order Process. Below it is a silhouette of its Petri net generated by BPEL2PNML

⁹ <http://www.bpm.fit.qut.edu.au/projects/babel/tools/BPEL2PNML.jar>

and rendered through the visualization tool PIPE. Even this modest sized web service composition resulted in a PNML artifact containing 94 places, 89 transitions, and 246 arcs. During the model reduction step, we were able to minimize it by about thirty percent. This example motivates the need for factoring into subnets, arranging each subnet to facilitate human understanding, and coloring each with partner links and variables.

It is worth noting that a prototype version of the translation from PNML to CPN is available as a web service ¹⁰. The service appears to be intended for general-purpose translation from PNML, and not intended to exploit modeling opportunities posed by web service artifacts. Thus, using this service we noticed that the resulting CPN supports neither hierarchy nor color, nor does it attempt to improve layout. We use hierarchy for improved understanding and encapsulation of otherwise repetitive subnets, improving the layout of each. We use color to express more of the content of the BPEL artifact, namely partner links that highlight data-related control flows and variables that strongly type places using color sets.

Figure 5.1, which we will be discussing in greater detail later, provides silhouettes of the results of running the prototype conversion to CPN Tools. They contrast to the PNML silhouette in Table 5.1, with the most obvious one being layout which we discuss in the following paragraphs.

¹⁰ <http://wwwis.win.tue.nl/~jmw/pnml>

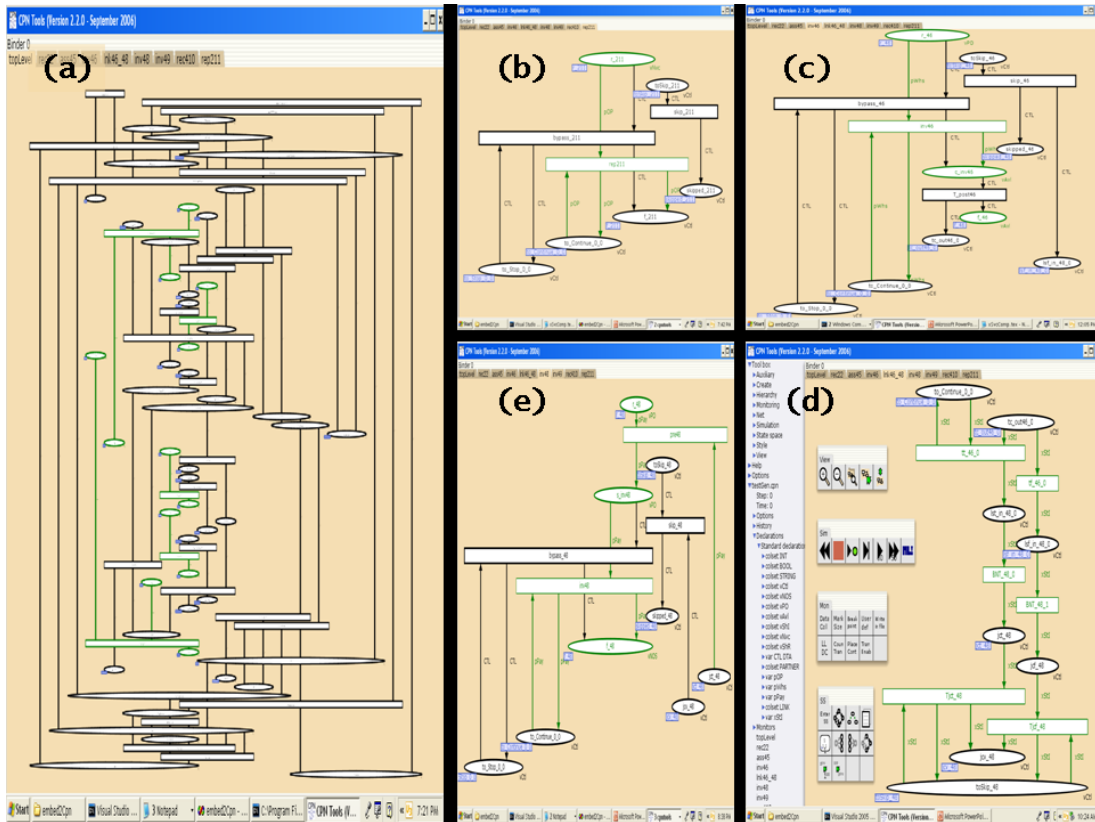


Figure 5.1: Silhouettes of generated CPN's

5.2.3 Graph Drawing

A major obstacle to an understandable model centers on the geometry and partitioning of the visible model. Graph drawing research into planar embeddings using the palm tree construction [68], visibility representations described in [10], and book embeddings [29] will assist in overcoming this obstacle.

To date this problem has been ignored, with layout decisions depending instead on human judgment, used for example by [150] in the following three intertwined areas: (i) where to embed places and transitions (nodes), (ii) how to render each arc

connecting place and transition, and (iii) what portions of a colored Petri net should be placed into a hierarchy of subnets. We wish to automate this otherwise subjective process.

Area (i) involving how to embed nodes, requires rendering them in an order corresponding to how they appear in the source artifact. The y coordinate in each node in the PNML artifact preserves this ordering for activity-related nodes, but not for nodes associated with crosscutting concerns. Refining this embedding for these concerns is reserved for future work.

Area (ii) involving the rendering of each arc connecting each place and transition, requires use of a *visibility representation*, which is a means of eliminating arc bends [10]. Strictly speaking, visibility representations are defined for only planar graphs, since by definition, visibility representations are free of any edge crossings. We relax this definition to allow for some edge crossings, while entirely eliminating arc bends.

Addressing area (iii) as to what portions should be placed in a hierarchy, requires partitioning the Petri net by activity type (i.e., structured or basic). Each occurrence of basic activities and control links in Figures 5.2 through 5.5 are adequately represented by a two-page *book embedding* described in [29] for the special case in which the order of vertices is fixed. The subnet for structured activities, the silhouette for which appears in Figure 5.1(a), often requires more than a two-page book embedding. Alternatively, we may tolerate a larger number of edge crossings.

For the class of hierarchical Petri nets used by CPN Tools, this involves extending existing graph drawing algorithms to bipartite graphs subject to the constraint that entire transition nodes must appear on exactly one page pair or subnet. Places, however, may be distributed across multiple subnets as *fusion places*. This extension is left for future work.

5.2.4 Starting Point

We used the BPEL2PNML tool to convert the BPEL artifact for the abbreviated version of the Purchase Order Process to PNML. As such, we leveraged all the syntax and semantics embedded in the output of BPEL2PNML for our conversion utility. In this section we describe the case study in terms of the figures we generated.

The left of Table 5.1 shows an excerpt of a BPEL artifact. To its right is the PNML file which we rendered using the PIPE visualization tool, highlighting each arc and node. In this rendering, we did not distinguish between places and transitions, since we intend only to convey its intricacy. We also highlighted in purple the seven basic activities and one control link, each appearing in the same order as in the BPEL code to its left.

Each of these basic activities, including their places, transitions, and arcs adopt the naming conventions from [100] and are shown in Table 5.2. Terminal symbols in the **Name** column are in normal typeface. The **S** column indicates node type, be it (P)lace, (T)ransition, or (B)oth. The **Role** column describes how the so-named

node functions within its subnet. Control links crosscut the nested syntax of BPEL artifacts. As such, they required special treatment by the BPEL2PNML utility. Thus, Table 5.3 outlines the naming conventions for nodes involved in control links.

In later steps, the prototype exploits these naming conventions when partitioning the large PNML artifact into a set of smaller subnets. These conventions will also be used to assign colors to transitions and color sets to places as the prototype mines the BPEL document for values of attributes not captured by the BPEL2PNML utility. Given the naming conventions and the bipartite graph comprising the Petri net, we partitioned, colored, and laid it out as shown in Figure 5.1. By way of setting a context, the next paragraphs describe the control logic, reflected in each figure generated by the prototype.

5.2.4.1 Infrastructure

Figure 5.1(a) shows the silhouette that includes both the infrastructure and the structured activities in a BPEL process. A BPEL process can be imagined as having a green light in the place marked `to_Continue` and another place with a red button marked `to_Stop`. The control logic defined in [100] and preserved by the prototype, can model the cancellation of an entire BPEL process by placing a token into `to_Stop`. The control flows were designed to fulfill the requirement in [114] of not leaving unwanted tokens throughout the net after process cancellation by returning the net to some known marking. This involves activation of a number of skipping

activities that increase the size of the net. The availability of a snapshot of the state of the BPEL process is also indicated by the place named **snapshot**. The role of these and other process-level places and transitions appear in the lower half of Table 5.2.

5.2.4.2 Basic Activities

Figure 5.1(b) provides a silhouette of Figure 5.2 showing a typical basic activity. Of the seven basic activities in this case study, five of them follow this identical pattern. For activity occurrence *n*, place **r_n** receives a message enabling either the **bypass_n** or **activity_n** transitions, depending on whether a token appears in **to_Stop** or **to_Continue**. If the **activity_n** transition fires, it will place a token into the **f_n** place indicating the activity finished. There is also a **skip_n** transition to support the skipping behavior when either (i) a branch in a BPEL **<pick>** activity is not taken, or (ii) a token appears in the **to_Stop** place prior to reaching this subnet, or (iii) a join failure occurred on a control link prior to reaching this subnet.

5.2.4.3 Link-related activities

Figure 5.1(c) provides a silhouette of Figure 5.3 showing how a basic activity would behave if it were the source of a control link. Table 5.3 describes the behavior of transitions **T_post_m**, and places **tc_out_m** and **lsf_in_m**. These added places provide an interface down into the control link.

Figure 5.1(d) provides a silhouette of Figure 5.4 showing the structure and behavior of a BPEL control link. If link status evaluates to true, a token will appear

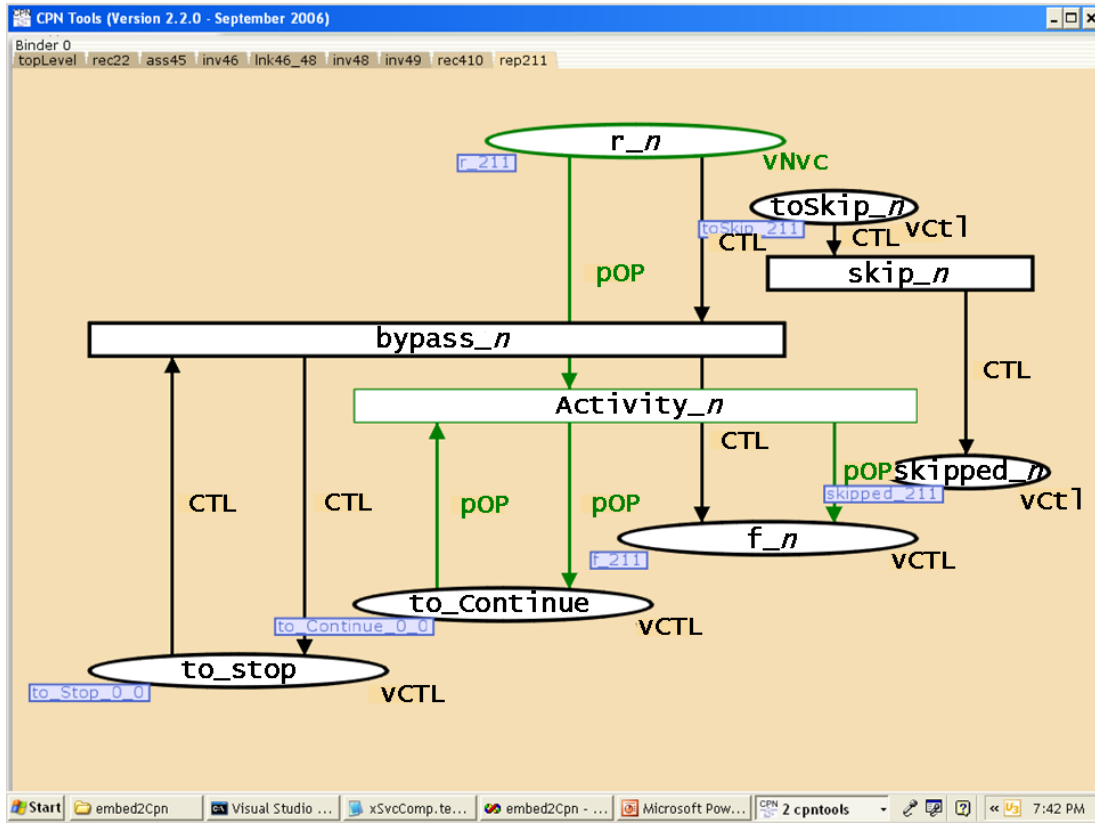


Figure 5.2: A typical BPEL basic activity in CPN

in place `1st_in_n`, which only happens if the activity at the source of the link were to complete successfully, namely a token appears in places `tc_out_m` and `toContinue`. If the Boolean net, shown here as transition `BNT_n_0`, evaluates to true a token will appear in `jct_n` indicating join condition evaluates to true. In general, nets involving more than one control link will have a fan into `BNT_n_0` in excess of one. Absent a token in `toContinue`, the transition condition fails and the right hand branch is taken. Notice the error in Figure 5.4 where the marking will make no further progress, since a token is also required in place `toSkip_n`. This was the result of the original

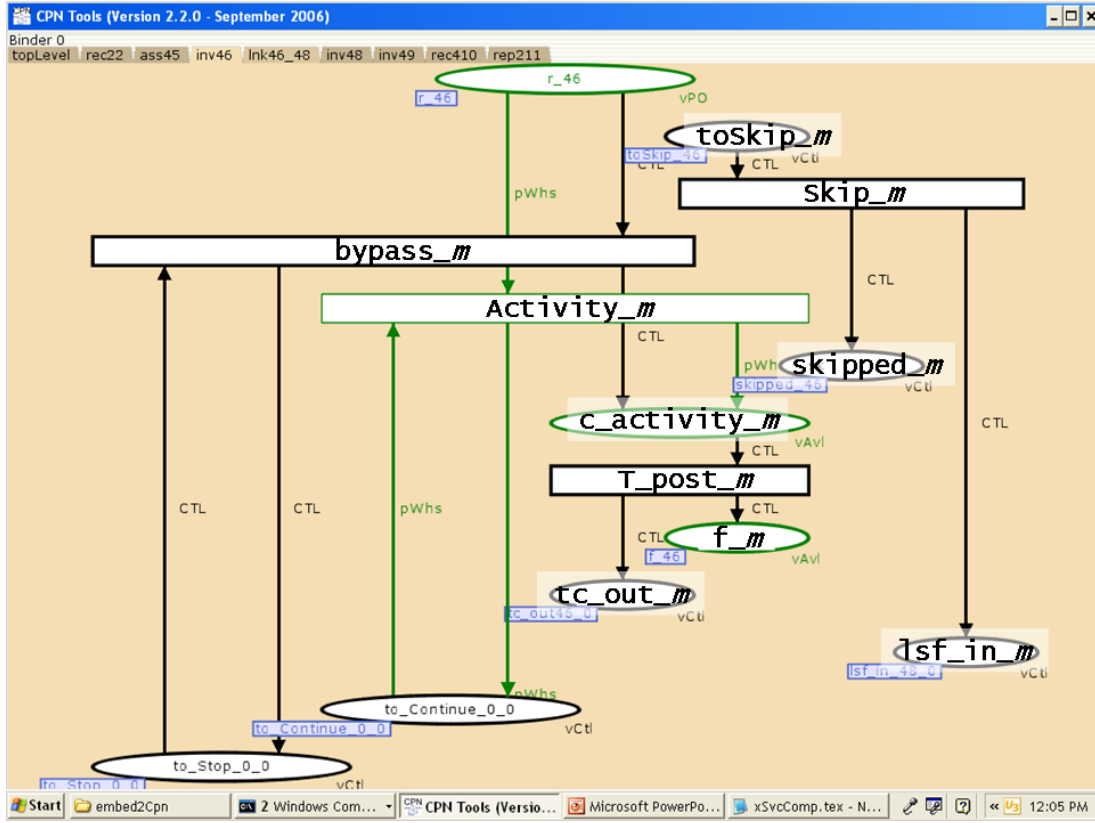


Figure 5.3: A basic activity as an origin of a link

BPEL artifact failing to specify whether or not to `suppressJoinFailure`. Although the value of this attribute defaults to `no` [4], a verification tool that deprecates such defaults is enforcing a best practice in coding BPEL artifacts.

Finally, Figure 5.1(e) provides a silhouette of Figure 5.5 showing how a basic activity behaves if it were the destination of a control link. Table 5.3 describes the behavior of transition `pre_n` that functions as a precondition for initiating the basic activity only if the join condition evaluates to `true` evidenced by a token in place `jct_n`. The activity can be skipped only if the join condition variable is present

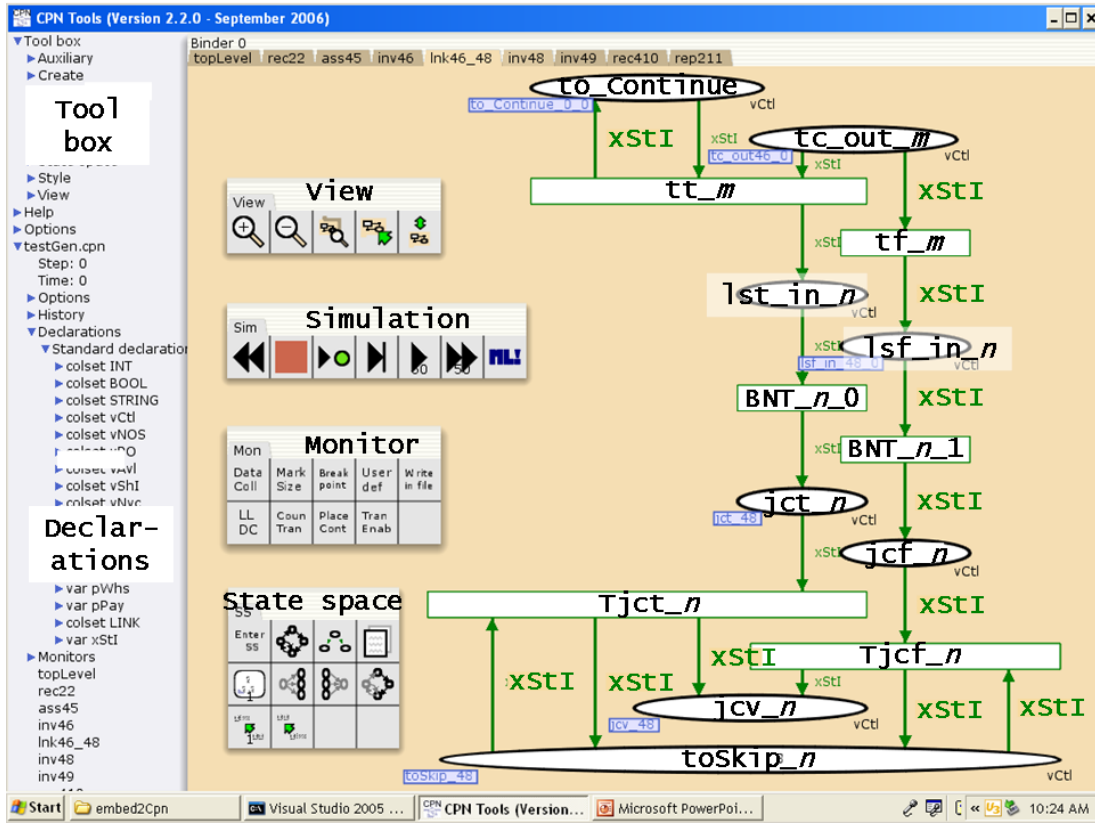


Figure 5.4: A link construct

(either true or false) by a token in place `jcv_n`.

The next section describes how we mined both the BPEL and PNML artifacts for information particular to the CPN.

5.3 Mining Source Artifacts

We used an abbreviated version of the BPEL artifact for the Purchase Order Process ¹¹ to illustrate the conversion to CPN, via the PNML artifact generated by

¹¹ The code for this version is available at: <http://www.osoa.org/display/Main/Relationship+between+SCA+and+BPEL>

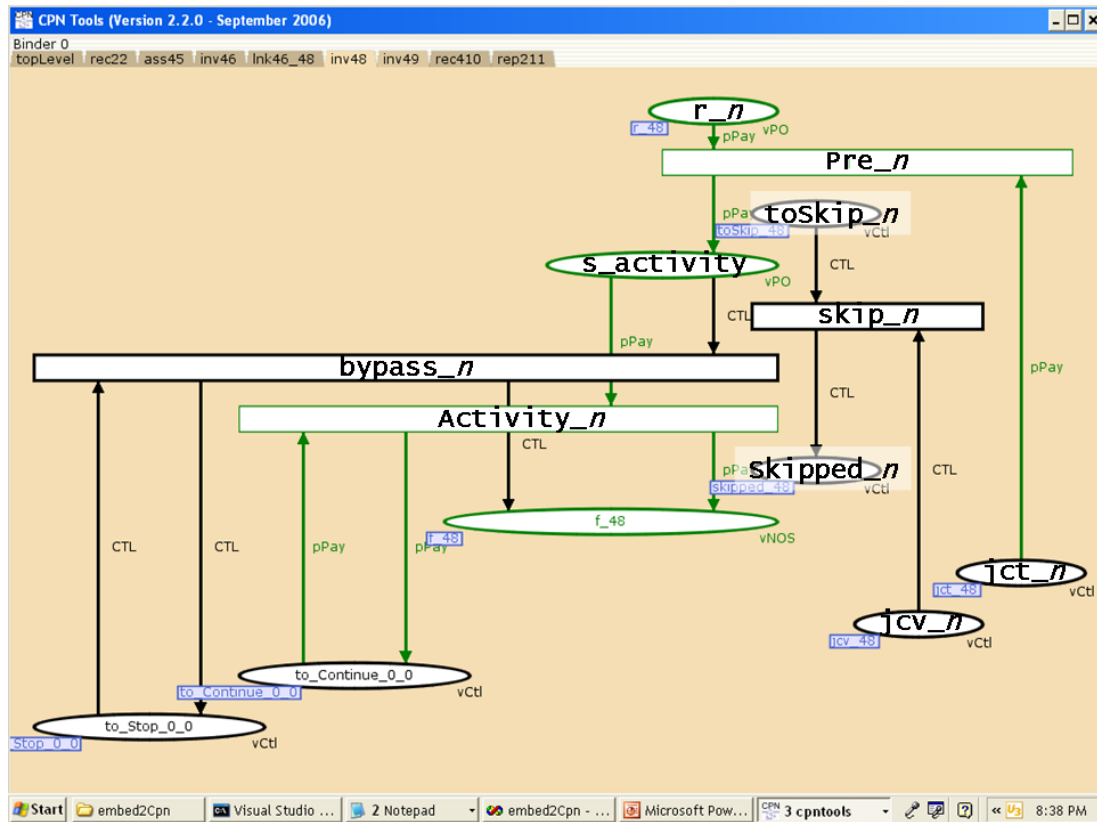


Figure 5.5: A basic activity as a link’s destination

the BPEL2PNML utility. The Petri net from the PNML artifact in Table 5.1 appears tangled, not broken out into subnets, and provides no insight into data type or color. Rather than using the (non-visual) Petri net analysis tool WofBPEL as was done in [100], we chose instead to produce a machine-verifiable model that is also human-readable. This conversion requires a number of preprocessing steps, which we describe in the following paragraphs.

Using structured document mining techniques, we mined the PNML file for `<arc>` elements to produce a list of arcs and nodes, the latter of which include either

places or transitions. Mining for `<place>` and `<transition>` elements, it classified each node and recorded their x and y coordinates which we later use for embedding the Petri net. Based on the naming conventions in [100], we automatically identified the type of subnet into which we wish to place each node, be they *basic*, *structured*, or in the case of places, a third category of *fusion*. Fusion places serve as the interface between top-level nets comprised of structured type nodes and subnets each comprised of basic type nodes. The top-level nets that we will construct control overall execution of the BPEL process, which at minimum include structured activities. Top-level nets may also control remediation, including exception and compensation handling, and interaction with its environment. When partitioned into subnets, each lower-level Petri net realizes either a basic activity, a BPEL *link* construct, or a basic activity that serves as either an origin or destination of a link. The silhouettes of the top-level net and its four types of subnet appear in Figure 5.1.

Given the naming conventions for places and transitions in [100] and summarized in Tables 5.2 and 5.3, the tool assigned a color to each transition and a color set to each place according to the rules appearing in the following paragraphs:

5.3.1 Partner Links

All arcs originating from the fusion place designated to receive the message to the core transition for that basic activity are assigned a partner link name as a color from color set PARTNER. Likewise, all arcs originating from the core transition to

the fusion place designated to send the message are also assigned this color. the tool generated results shown in Figure 5.2 that highlight these data flows in green. The arc originates from the fusion place `r_n` to the core transition which we manually highlighted as `activity_n` to make this diagram generic enough to express any basic activity.

Likewise, the arcs from `activity_n` to fusion place `f_n` are also labeled with the color for this partner link, as is the fusion place `to_Continue`. In this example, these arcs are annotated with the color `pOP` denoting the partner link for Order Processing.

5.3.2 Data-related Flows

Portions of the top level net that connect the fusion places for successive basic activities are also automatically highlighted in green but are assigned the color `DTA`. This color assignment indicates that these are not data flows *per se*, rather they activate data flow events inside each basic activity.

For example, Figure 5.6 connects the output of the `<receive>` activity through fusion place `f_22`, to the input to assign activity `r_45`, via the forking portion of the BPEL `<flow>` construct represented by transition `T_sFlowflow`. This assign activity happens to be the first activity in the first `<sequence>` construct. The other outbound arc from transition `T_sFlowflow` ultimately connects to fusion place `r_48` representing the input to the first basic activity in the second occurrence of

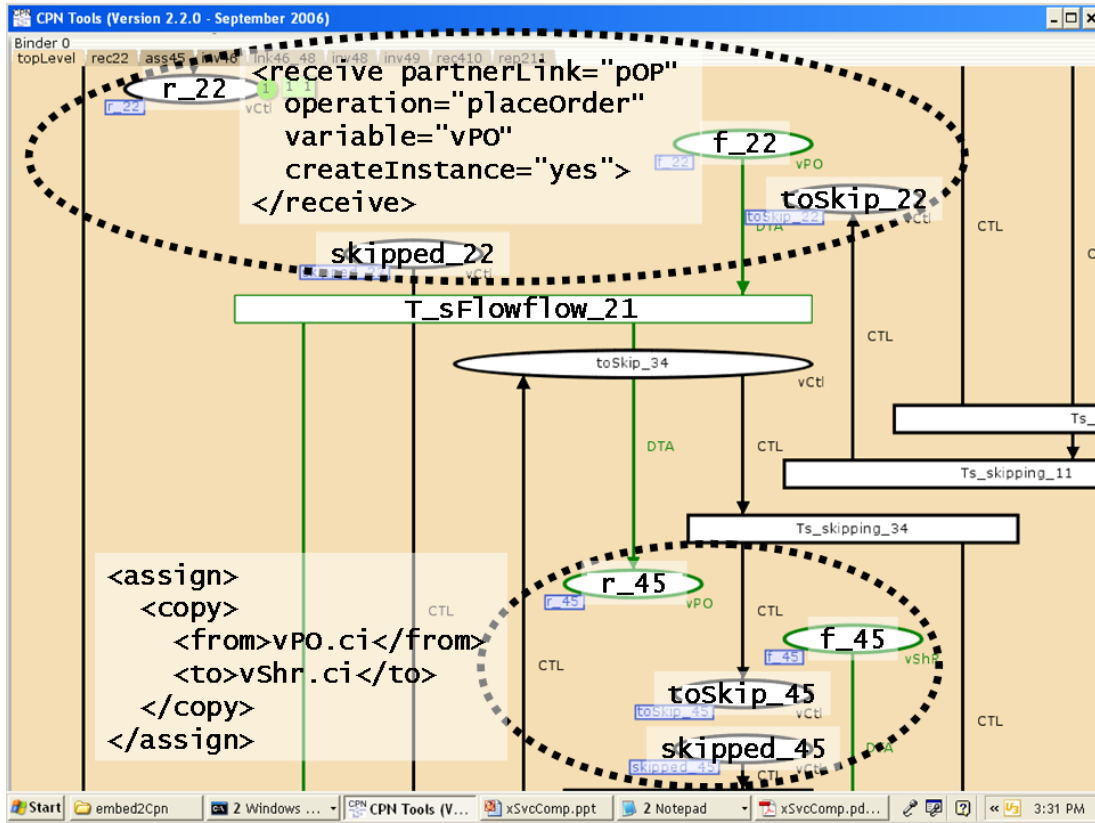


Figure 5.6: Closeup of top-level CPN

the `<sequence>` construct, located within the same BPEL `<flow>` construct. Hence, Petri net transition `T_sFlowflow` naturally models a parallel split workflow.

5.3.3 BPEL Variables

The tool defines a color set for each BPEL variable. For each input variable, the tool assigns the variable's color set to the basic activity's input fusion place. The result in Figure 5.2 shows variable `vNvc` as the color set associated with place `r_n`, which happens to stand for the invoice variable in the final `<receive>` activity in the

case study. Likewise, each output variable is associated with the output fusion place of a basic activity.

5.3.4 BPEL Links and Defaults

The tool defines a color set for each BPEL link, assigning it to all places inside the link type subnet as the result shown in Figure 5.4 reveals. In this example, the color set contains only one element, namely **xStI**, which stands for "ship-to-invoice".

All arcs and nodes shown in black manage control-related flows. The layout algorithm tends to cluster data-related activities in the top-level net of Figure 5.1(a) toward the center of the drawing, with control-related flows arrayed toward the periphery. Control flows located farthest from the center tend to involve interaction with the environment, or address crosscutting concerns like event, fault, and compensation handling.

5.3.5 CPN Subnet Names

Since the PNML file follows the naming conventions in [100] and summarized in Table 5.2, we mined the PNML document for subnet names, each of which will occupy a *page* in a CPN binder. (A CPN *page* or subnet is, in graph drawing terms, a *two page* book embedding – think of the pages of a book open to you). Recall from Table 5.2 how the core transition of each basic activity is named after the first three letters of the BPEL basic activity type followed by two or more digits. Each of these serve as a subnet name in CPN Tools, where a "binder" in CPN Tools can contain

one or more such named subnets. In this experiment, we allowed the layout algorithm to construct one large top-level subnet at the expense of edge crossings.

5.3.6 Data Cleansing

The end result of the previous steps are three sorted tables or lexically ordered relations which we name after their attributes: $ASXYTC$, A_sA_t , and U_pA_p . All remaining steps, except for the graph embedding step, are computed using ordinary relational operators. Information in these three tables are sufficient to compute an embedding and generate a CPN Tools artifact. First, we briefly summarize each of their contents, and describe any supervised data cleansing required.

Node set $ASXYTC$ is comprised of (i) abbreviated name A that uniquely identifies the node per Tables 5.2 and 5.3, (ii) node type S whether it be (P)lace or (T)ransition, (iii) X coordinate and (iv) Y coordinate of node from its PNML embedding, (v) subnet type T to which the node belongs which assumes one of three values including (S)tructured, (B)asic, and (F)usion, and (vi) a transition’s color or place’s color set C . The differing meanings ascribed to C are because a place can hold many tokens, while a transition is enabled and fires with exactly one token to/from each adjacent place. Arc set A_sA_t is a simple binary relation mined directly from the PNML file and made up of source node name A_s and target node name A_t . Finally, the set of subnet designators U_pA_p is made up of subnet number U_p and subnet name A_p . The PNML artifact uniquely numbers each occurrence of an activity, which we

simply index as U_p . Subnet name A_p assumes the name of the core transition within that activity.

Given arc set A_sA_t the tool identified nodes that do not have inbound or outbound arcs. In our experiment, we remedied this by manually inserting transitions and arcs to simulate the interaction of the entire BPEL process with its environment.

Augmenting arc set A_sA_t with information in node set $ASXYTC$ results in the relation named $T_sT_tS_sS_tA_sA_tX_sY_sX_tY_tC_yC_z$. This one table captures all of the information present in both the arc and node tables. The meaning of each attribute remains unchanged, with the lower case s and t referring to source and target nodes, respectively. Attribute C_y refers to the transition's color, while C_z refers to the color set of the place end of the arc. The tool partitioned set $T_sT_tS_sS_tA_sA_tX_sY_sX_tY_tC_yC_z$ into basic activities subnet $_bT_sT_tS_sS_tA_sA_tX_sY_sX_tY_tC_yC_z$, infrastructure and structured activities subnet $_sT_sT_tS_sS_tA_sA_tX_sY_sX_tY_tC_yC_z$, and finally the link-related subnet $_lT_sT_tS_sS_tA_sA_tX_sY_sX_tY_tC_yC_z$ based on activity type T . Activity pairs in which T_s and T_t differ will prompt the creation of fusion places. The tool further partitioned the basic type subnet $_bT_sT_tS_sS_tA_sA_tX_sY_sX_tY_tC_yC_z$ into a subnet for each occurrence of a basic activity, resulting in seven such basic subnets. The tool then invoked the embedding algorithm on each of the subnets for both the orientation provided by the BPEL2PNML tool, and for a 90 degree rotation. Rotating the subnets for the basic activities 90 degrees resulted in fewer edge crossings, so from here on out we worked with the rotated versions of these subnets, with the remaining subnets assuming their

original orientation.

Since these subnets were originally constructed in [114] using a local replacement strategy and later refined in [100], their PNML artifact was not yet minimized. We found that applying just one minimization rule involving collapsing degree two nodes while retaining the bipartite nature of places and transitions, and retaining all fusion places resulted in a smaller model. Originally, the PNML artifact contained 94 places, 89 transitions, and 246 arcs. These were reduced to 63, 57, and 180, respectively. Recently, [87] described a tool named BPEL2oWFN for constructing a type of Petri net known as an open workflow net from a BPEL artifact. That tool performs model reduction prior to generating its artifact. Adapting our prototype to this tool is left for future work. For each subnet, the tool invoked the embedding algorithm, which we describe in the next section.

5.4 Embedding Subnets

Placement and curvature of arcs in the figures in [150] is the result of human judgment, which we wish to remove. Minimizing the number of arc bends is widely recognized as a desirable property of graph drawings [10]. A visibility representation is a means of rendering graphs in a manner free of arc bends. Recent improvements in computing layout and efficiency are typified by [19, 60]. The approach we use is easy to implement and explain, but far from optimal. Intuitively, it involves a topological deformation of nodes.

Figures 5.7(a) and (b) appeared in the celebrated paper on planarity testing by Hopcroft and Tarjan [68] who used the graph in Figure 5.7(a) as input to their palm tree construction that resulted in Figure 5.7(b). This construction was thence used to determine planarity.

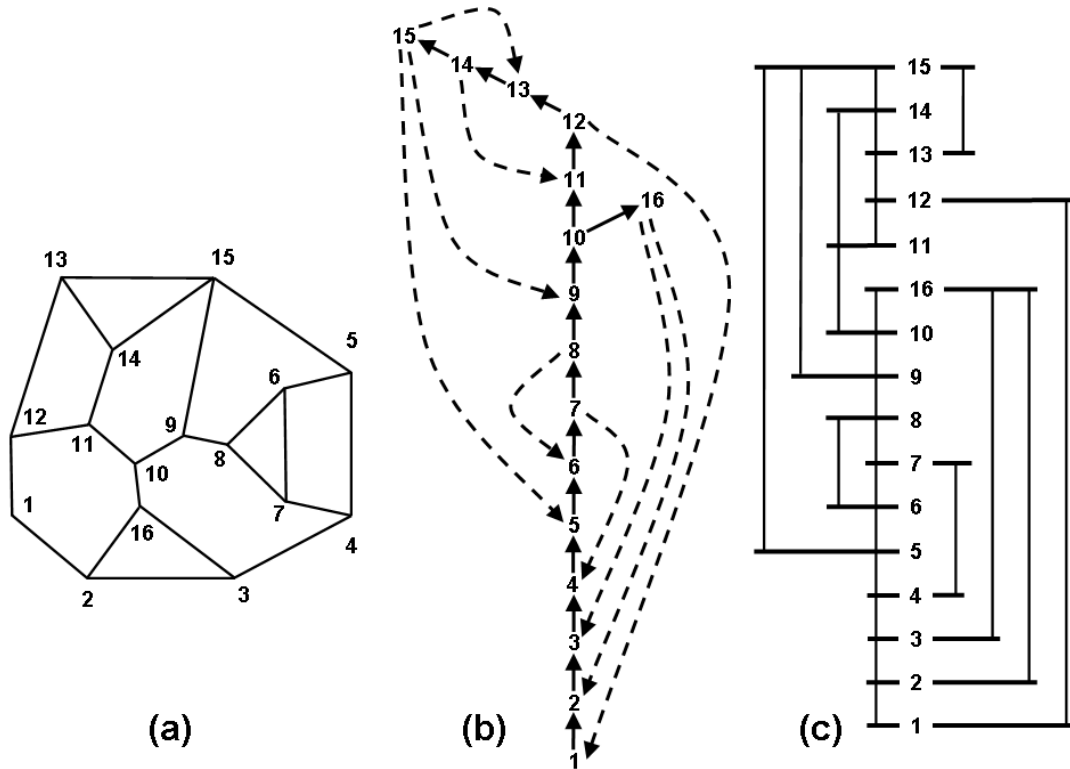


Figure 5.7: Equivalent graphs: (a) original, (b) palm tree and (c) visibility representation

5.4.1 Strategy

Deforming each vertex of the *palm tree* in Figure 5.7(b) into a horizontal line, results in the visibility representation in Figure 5.7(c). To assure planarity, their algorithm reordered vertices (i.e., by placing vertex 16 to appear before 11). Although their embedding is planar, the vertices ceased to be lexically ordered. We wish instead to retain this lexical ordering at the expense of arc crossings, since reordering of vertices results in a visual artifact that would require reorienting eye movement whereas arc crossings require only a measure of visual discernment. To accomplish this, we need to modify the palm tree construction of [68].

We devised an algorithm for expressing each subnet in terms of a pair of *pages* approximating a *book embedding*. A graph representing a page in a *book* has its vertices arranged in a line along the *spine* of the book, with arcs arranged to avoid crossings [29]. This algorithm relaxes the requirement that there be no crossings. Knowing that minimizing the number of crossings for an arbitrary graph is NP Hard, we instead chose to define a simple greedy strategy that produces a correct albeit suboptimal embedding. We extend this simple algorithm to account for two mutually conflicting constraints: (i) minimize number of arc crossings, and (ii) balance the aggregate arc length between even and odd pages in the embedding. Introducing a third constraint would allow us to minimize the number of crossings at the expense of an increased number of page pairs. Such a constraint must place entire transition type nodes, including all inbound and outbound arcs and places, onto the same page pair

(or CPN subnet) to produce a valid bipartite graph acceptable for import into CPN Tools. Incorporating this constraint into an optimized state-of-the-art algorithm is left for future work.

5.4.2 Processing

Before solving for layout attributes in the CPN Tools XML artifact under construction, we substitute each (x, y) coordinate in $T_s T_t S_s S_t A_s A_t X_s Y_s X_t Y_t C_y C_z$ with counting numbers (i_s, i_t) for $i_s \in I_s$ and $i_t \in I_t$ determined primarily by the value of its y coordinate. From the resulting relation $I_t I_s T_s T_t S_s S_t A_s A_t C_y C_z$, the tool computed the length L of each arc, forming the input relation LI_s to algorithm BOOKEMBED-SUBNET. For each arc in LI_s , the algorithm computes subnets U and width specifiers W for each source node I_s and target node I_t , outputting relation $UWI_s I_t$.

After computing the direction of each arc, we included the remaining attributes from the input relation, resulting in relation $UWI_s I_t T_s T_t S_s S_t A_s A_t C_y C_z$. At this point, the tool has enough information to place each directed arc onto the proper page and at the correct ordinal distance from the spine of its two-page book, which we will be doing in Section 5.5. That final phase first performs the required topological deformation of each node, and then generates a valid XML file for importing into CPN Tools. Before doing so, we first discuss the algorithm.

5.4.3 Implementation

We use a simple greedy strategy, by which arcs are first sorted from shortest to longest to form graph G for input to algorithm BOOKEMBEDSUBNET. The algorithm performs an arc-wise insertion that first fills in the spine of the book. Successively longer arcs get arranged around either the left or right hand sides depending on which side results in fewer edge crossings. The result is a 2-page book embedded into a single CPN subnet. The following paragraphs describe this algorithm in greater detail.

Given graph G , we allocate and initialize tables M and W , and vector b in lines 2-4 using built-in functions ORDEROF, INITIALIZETO, HEADOF, FIRSTOF, AND SECONDOF, all of which are self-explanatory. The size of these tables is proportional to the order of the graph, namely its number of vertices. Crossings table M is a two dimensional array indexed by page i and vertex j . Thus, entry $M_{i,j}$ represents the number of arcs situated between the spine of the book, at page i and vertex j , and the outer side of the page where the arc may be inserted. Widths table W has the same dimensions as M but each of its entries $W_{i,j}$ denotes the *geometric width* (not cut width) at page i and vertex j . It is used to specify the horizontal dimension along which the arc is to be inserted. To insert a new arc we compute the maximum geometric width along the extent of the new arc as the width at which it will be inserted. Vector b denotes for each page i the aggregate arc length for non-spine arcs and is used to assure balance around the spine of the graph.

The **while** loop that occupies the remainder of the algorithm processes each

successive arc in G , identifying the page in the pair best suited for insertion. First it parses out arc variables, including source and terminating arcs s and t in lines 8-11. Next, it identifies which page in the pair where the arc should be inserted using the decision structure on lines 13-20, which entails two decisions. Lines 14 and 15 inserts the arc into the page in the pair that minimizes arc crossings. If both pages are otherwise equally satisfactory, lines 19 and 20 select the page having the lower aggregate arc length for insertion. This helps to assure a balanced appearance of the page pair around its spine.

The portion of the algorithm in lines 21-33 updates crossings table M , geometric widths table W , and balance vector b . Finally results of the form $(pageNumber, width, startVertex, terminatingVertex)$ are output as relation UWI_sI_t described earlier.

These 4-tuples provide sufficient information to draw the visibility representation. With regard to the algorithm's complexity, one may initially guess it to be $\mathcal{O}|\mathcal{E}|$ for graph size $|E|$. However, during preprocessing, the arc set needed to be sorted, so the complexity is actually $\mathcal{O}|\mathcal{E}|\log|E|$. The algorithm seeks not to minimize edge crossings as much as place nodes and edges that are involved in longer distance dependencies toward the periphery of the diagram. Presumably these elements represent the execution concerns that change little between applications. Conversely, application-specific concerns tend to be clustered toward the diagram's center as shown in Figure 5.1(a).

5.5 Generating the CPN

After some post-processing, the result of the algorithm described in the previous section is relation $UWI_sItTsTtSsStAsAtCtCp$. It preserves all meaning and values of attributes from input relation $TsTtSsStAsAtXsYsXtYtCtCp$ but with x and y coordinates X_s , X_t , Y_s , and Y_t replaced by subnet U , width specifier W , source node index I_s , and target node index I_t . This one relation provides sufficient information to generate the XML model for CPN Tools. We use it directly to produce CPN `<arc>` type elements, and apply further processing to generate CPN `<place>`, `<trans>` and `<fusion>` type elements. As mentioned earlier, this final phase first performs the required topological deformation of each node, and then generates a valid XML file for importing into CPN Tools.

To topologically deform each node, the tool must first produce minimum and maximum widths for each page and node index in $UWI_sItTsTtSsStAsAtCyCz$. This deformation is required to generate `<place>` and `<trans>` type CPN elements. The tool generated set $UIW_lW_hTSAC_yC_z$, where W_l and W_h represent the low and high values of the width specifier for each node I in subnet U .

Since a book embedding involves page pairs that share a spine, using relation $UIW_lW_hTSAC_yC_z$, the tool produced pair wise page statistics so that each subnet can be made up of a page pair from a book embedding. The resulting relation $UIP_lW_lP_hW_hTSAC_yC_z$ provides enough information to produce these mirrored page pairs. Nodes straddling the spine will have different values for attributes P_l and P_h ,

whereas nodes to the left or to the right will have the same even or odd values for P_l and P_h , respectively.

A final step before generating the XML file for a CPN model involves gluing together corresponding fusion places so that tokens may travel between subnets during simulation and for construction of the state space. The tool generated relation $A_f U_f I_f$ from $UIP_l W_l P_h W_h TSAC_y C_z$. A_f is the name of the fusion place, while subnet U_f and node index I_f together locate the fusion place. Thus, any given name A_f of a fusion place will have two or more values of (U_f, I_f) that effectively bind two or more subnets.

Finally, the tool generated the CPN model using the following four relations: (i) nodes $UIP_l W_l P_h W_h TSAC_y C_z$, (ii) arcs $UW I_s I_t T_s T_t S_s S_t A_s A_t C_y C_z$, (iii) fusions $A_f U_f I_f$, and (iv) pages $U_p A_p$. Excerpts from these results appear in figures throughout this chapter. An explanation of each figure appears in Section 5.2. The algorithm produces all elements required by CPN Tools, namely subnets in a binder, each subnet comprised of node elements, followed by arc elements. The geometry of each node and arc is determined by the nodes and arcs relations respectively.

Finally, we manually inserted initial markings and simulated the resulting system of nets.

5.6 Simulation Results

Once built and imported into CPN Tools, the simulator detected a number of interaction faults manifested mostly as deadlocks. Partitioning into subnets provided a clearer context of where these faults lie. The layout of control flows from top to bottom, with some exceptions, was easy to follow. Color assignment to places and transitions provided business-level intuition of what type of data flows were being orchestrated. Color coding the distinction between data-related control flows and control flows for cross-cutting concerns provides insights into what control flows are application-related (i.e. basic and structured activities), and what are implied by the BPEL execution model. Together, these provide more intuition into an orchestration's structure and behavior than that provided by PNML. More specific results appear in the following paragraph.

Running the simulation revealed either deadlocks or unreachable subnets, examples of which we describe here. The workflow net produced by BPEL2PNML does not automatically generate an environment with which the BPEL process interacts. At minimum, this required including an arc between a process-level finished place and the process's initiating transition. This arc can be replaced by an environment exhibiting arbitrarily intricate behavior. We also encountered an unreachable portion of the net that did not completely specify the non-deterministic choice between a faulty and normal execution. We traced that fault back to a failure to properly specify in the BPEL artifact if an instance was created. Finally, the simulation deadlocked in

the subnet for the control link, due to a failure to specify whether or not to suppress join failure. Our contributions involving partitioning, layout, and automatic generation of CPN artifacts, enabled us to efficiently locate and troubleshoot problems like these.

5.7 Chapter Summary

We described an end-to-end process by which an executable BPEL artifact is automatically translated into a colored Petri net suitable for formal verification by CPN Tools. Using a tool that already converts a BPEL artifact to a workflow net, we generated a colored Petri net with equivalent topology and labeling, but imbued with notions of hierarchy and color and enhanced with a simple and consistent layout. Perhaps the biggest contribution was development of the prototype that automates this process thereby minimizing human judgment in model capture. We presented a means of inducing the hierarchy of subnets while preserving the notion of type or color through a stepwise description of a recently developed prototype. Included in this prototype conversion tool is a layout algorithm which renders the resulting CPN artifact as a visibility representation.

The layout algorithm preserves the sequencing of basic activities and the nesting of structured activities found in the BPEL artifact. However, portions of the BPEL artifact crosscuts activities (raising exceptions) or are implied but not written

(overall control of process execution). Although the workflow net we use as input correctly reflects these concerns, crosscutting concerns may require reordering of nodes prior to computing the embedding.

Enhancing the layout algorithm to reduce the number of edge crossings at the expense of an increased number of page pairs is left as future work. Such a constraint must place entire transitions, including all inbound and outbound arcs and places, onto the same page pair (or CPN subnet) to produce a valid bipartite graph acceptable for import into CPN Tools. Incorporating these features and constraints into a state-of-the-art algorithm is also left for future work.

Applications with intractably large state spaces must instead be verified using Petri net analysis tools like WofBPEL that implement techniques like transition invariants that do not require generation of a state space. Since WofBPEL uses the identical PNML artifact as our prototype, future work may involve more seamlessly integrating this tool.

We already highlight arcs in green for data-related control flows representing structured activities in the top-level net. An obvious enhancement would be to factor these data-related control flows out into their own subnet. Doing so would result in the activity-level flows resembling the hand-drawn ones in [150]. Further discussion and implementation of such an enhancement is reserved for future work.

```

01 <process name="purchaseOrderProcess"
02   ..
03   <sequence>
04     <receive partnerLink="pOP" operation="placeOrder"
05       variable="vPO" createInstance="yes">
06     </receive>
07     <flow>
08       ..
09       <sequence>
10         <assign>
11           <copy> <from>vPO.ci</from> <to>vShr.ci</to> </copy>
12         </assign>
13         <invoke partnerLink="pWhs" operation="checkInventory"
14           inputVariable="vPO" outputVariable="vAvl">
15           <sources> <source linkName="xStI" /> </sources>
16         </invoke>
17       </sequence>
18       <sequence>
19         <invoke partnerLink="pPay" operation="orderPayment"
20           inputVariable="vPO">
21           <targets> <target linkName="xStI" /> </targets>
22         </invoke>
23         <invoke partnerLink="pWhs" operation="shipOrder"
24           inputVariable="vPO" outputVariable="vShI">
25         </invoke>
26         <receive partnerLink="pPay" operation="sendInvoice"
27           variable="vNvc"/>
28       </sequence>
29     </flow>
30     <reply partnerLink="pOP" operation="placeOrder"
31       variable="vNvc">
32     </reply>
33   </sequence>
34 </process>

```

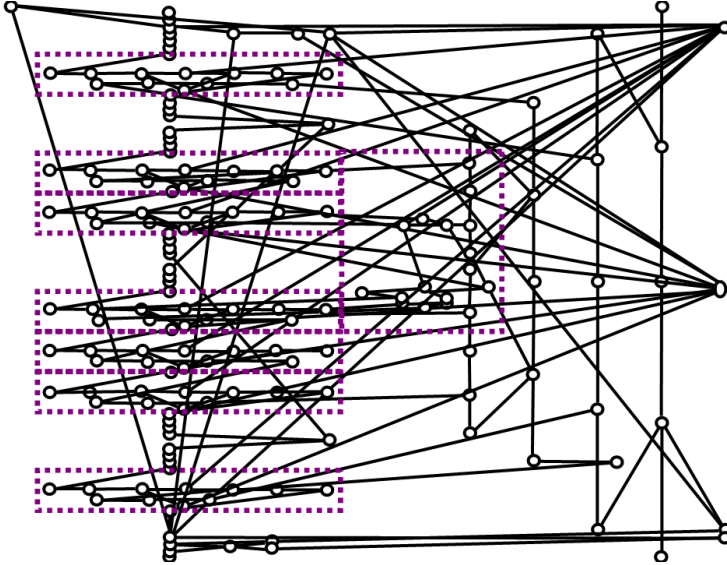


Table 5.1: Excerpt of purchase order process (top) and its PNML silhouette (bottom)

Name:	S	Role:
<i>activity</i>	T	<i>act mnemonic</i> + <i>act number</i>
<i>act mnemonic</i>	T	(ass)ign (inv)oke (rec)ieve (rep)ly .. for core activity in basic subnet
<i>act number</i>	B	2 or 3 digit number <i>n</i> uniquely identifying occurrence of an activity
<i>r_n</i>	P	activity number <i>n</i> is ready
<i>f_n</i>	P	activity number <i>n</i> is finished
<i>bypass_n</i>	T	bypass basic core activity <i>n</i>
<i>toSkip_n</i>	P	activity <i>n</i> to be skipped
<i>skip_n</i>	T	skip activity <i>n</i>
<i>skipped_n</i>	P	activity <i>n</i> was skipped
<i>to_Continue</i>	P	to continue entire BPEL process
<i>to_Stop</i>	P	to stop entire BPEL process
<i>toInvoke</i>	P	invoke entire BPEL process
<i>eFault</i>	T	register a fault within a scope of a BPEL process
<i>beginScope</i>	T	begin a scope inside a BPEL process
<i>endScope</i>	T	end a scope inside a BPEL process
<i>snapshot</i>	P	snapshot of BPEL process state is available at end of scope

Table 5.2: Activity and infrastructure namings

Name:	S	Role:
<i>m</i>		identifies the origin activity of a link
<i>c_activity</i>	P	to stage result of either <i>activity</i> or of bypass transitions for evaluation of the post condition that feeds one or more links
<i>T_post_m</i>	T	initiate evaluation of post condition .. used for source end of links
<i>tc_out_m</i>	P	transition condition that activates a control link
<i>lsf_in_m</i>	P	link status false when skipping an activity that feeds a control link
<i>lst_in_m</i>	P	link status true if BPEL process is to be continued
<i>tt_m</i>	T	link transition producing a true result
<i>tf_m</i>	T	link transition producing a false result
<i>n</i>		identifies the destination activity of a link
<i>BNT_n_0</i>	T	a Boolean Petri net abstracted to one transition for one set of link status paths
<i>BNT_n_1</i>	T	a Boolean Petri net abstracted to one transition for another set of link status paths
<i>jct_n</i>	P	join condition evaluating to true
<i>jcf_n</i>	P	join condition evaluating to false
<i>Tjct_n</i>	T	transition activated by jct
<i>Tjcf_n</i>	T	transition activated by jcf
<i>jcv_n</i>	P	join condition has been evaluated
<i>pre_n</i>	T	precondition requiring basic activity to be both ready and have its join condition evaluate to true.
<i>s_activity</i>	P	place holding the result of the precondition

Table 5.3: Link-related node namings

```

BOOKEMBEDSUBNET ( $G$ )
01  ▷ Allocate and initialize working variables
02   $M \leftarrow \text{ALLOCATE ORDEROF}(G)/2$  by  $\text{ORDEROF}(G)$  integers,  $\text{INITIALIZETo}(0)$ 
03   $W \leftarrow \text{ALLOCATE ORDEROF}(G)/2$  by  $\text{ORDEROF}(G)$  integers,  $\text{INITIALIZETo}(0)$ 
04   $b \leftarrow \text{ALLOCATE ORDEROF}(G)$  reals,  $\text{INITIALIZETo}(0)$ 
05  ▷ Process each arc in graph  $G$ :
06  while  $\text{NONEMPTY}(G)$ 
07      ▷ Parse out arc variables
08       $ls \leftarrow \text{HEADOF}(G)$           ▷ input arc of the form:  $(l \ s)$ 
09       $l \leftarrow \text{FIRSTOF}(ls)$         ▷ length of arc
10       $s \leftarrow \text{SECONDOF}(ls)$     ▷ starting vertex of arc
11       $t \leftarrow l + s$               ▷ terminating vertex of arc
12       $i \leftarrow 0$                   ▷ page number 0==left, 1==right
13      ▷ Identify which page in pair minimizes crossings
14      if  $(M[1, s] + M[1, t]) < (M[0, s] + M[0, t])$ 
15          then  $i \leftarrow i + 1$ 
16      else if  $(M[1, s] + M[1, t]) > (M[0, s] + M[0, t])$ 
17          then do nothing
18      ▷ Or else identify which page can lend more balance
19      else if  $b[1] < b[0]$ 
20          then  $i \leftarrow i + 1$ 
21      ▷ Find maximum geometric width for verticies spanned by the arc
22      for  $j$  from  $s$  through  $t - 1$ , with width  $w$  initially 0, do
23          if  $w < W[i, j]$ 
24              then  $w \leftarrow W[i, j]$ 
25      ▷ Update geometric widths of each spanned vertex with maximum in range
26      for  $j$  from  $s$  through  $t - 1$  do
27           $W[i, j] \leftarrow w + 1$ 
28          ▷ .. and update the crossing number for each spanned vertex
29          if  $j > s$ 
30              then  $M[i, j] \leftarrow M[i, j] + 1$ 
31      ▷ Update balance vector
32      if  $(t - s) > 1$ 
33          then  $b[i] \leftarrow b[i] + t - s$ 
34      ▷ Output arc with book embedding coordinates
35       $\text{OUTPUT}(i, w + 1, s, t)$ 
36      ▷ Enable processing of the next arc
37       $G \leftarrow \text{TAILOF}(G)$ 
38  end while

```

CHAPTER 6

ASSURING TIMELINESS

This chapter is based on the article titled: *Beyond Correctness Assuring Timeliness of Volunteer e-Science SOA's* [126]. It introduces an application of colored Petri nets (CPN) and timed automata to an e-Science service oriented architecture (SOA).

Public-resource computing, known also as Volunteer Supercomputing, does not implement the typical SOA. Its service providers operate on myriads of otherwise idle PC's, each working in isolation beyond the unreliable fringes of the Internet ¹². A typical provider at any one time, can be temporarily put out of service for a variety of reasons. Such events are often of little consequence since each provides but one service – computing power. So when providers *register* with an e-Science portal, the only types of questions asked concern platform and scheduling preferences. On the other hand, their few customers hail from the comparatively tiny research community who *request* the help of the many service providers via that same e-Science portal. That portal dynamically *discovers* available service providers, each of which may be fully capable of *supplying* the solution to any one key piece of some vast

¹² Known as the *Last Mile Problem*, this refers to the performance and reliability degradation associated with delivering broadband services over its final leg to the home.

computational puzzle. In this uncertain setting, timing provides the key to whether a portal must reassign a work unit it had previously assigned to some otherwise non-responsive provider. This chapter models this decision making process, first by presenting a simplified model of the Berkeley Open Infrastructure for Network Computing (BOINC). We then translate this model into the kind of timed automata used by UPPAAL, a model checking and simulation environment for verifying real time systems, while extending the model to cover additional time-sensitive use cases. Finally, we share our experiences and results from building and extending this model.

6.1 Chapter Introduction

As the deployment platform for such e-Science portals as **Einstein@Home**, **climateprediction.net**, and the ever-popular **SETI@home**, BOINC sets out to solve the type of problem that can be partitioned into myriads of subproblems, each of which can be solved on an Internet-enabled PC [6]. For BOINC, timely interaction between portal and provider is an issue separate from correctness or accuracy of results as was addressed in [134].

SOA focuses instead on coordination and deployment issues that can involve increasingly intricate e-Science workflows, more I/O-bound processes, and participation by less sophisticated providers. Concrete problems arise from unreliable network connections, new human users with different schedules, or software bugs that can freeze up a provider's PC. Most of these problems exhibit the same symptom – a

provider takes substantially more (or less) time to execute its work unit than the portal expects. These may be addressed by an SOA deployment that uses machine-verified code artifacts imbued with appropriate real-valued time constraints.

Ultimately, the portal's decision to reassign or withdraw a work unit will be based on measurements of the provider's past performance and on the expected time required by a work unit. Such measurements, however, must be taken with respect to some reference model that can reflect both sound interaction and appropriate timing between the portal and its providers. This chapter describes a four-stage process by which such a model can be derived. We first model a deployment platform like BOINC using an untimed formalism like a Petri net. Secondly, we identify the time-critical places in the Petri net and construct a timed automata from these places and their transitions. Thirdly, we specify properties and model check the resulting timed automata to these properties. Finally, we simulate the timed interactions between the portal and multiple providers.

Model checking is a popular formal verification technique with extensive tool support. It detects interactions that are in some unexpected order – an order that violates pre-specified sequencing constraints. As the most frequently referenced model checking tool in the SOA testing literature, *SPIN* [66] exhaustively lists each permissible sequence of interactions to detect deadlock, lack of progress, or other such violations. Model checking involving real-valued time constraints, however, requires a different model checker.

UPPAAL is an integrated verification environment for model checking systems that are subject to real-valued time constraints [13]. It enables the practitioner to graphically specify timed automata from its Editor tab, simulate it through its Simulator tab, and to model check for temporal logic properties through its Verifier tab (see Figure 6.4). Model checking to temporal properties, however, results in a greatly enlarged state space [79]. Thus, practitioners using timed model checkers like UPPAAL will need to make simplifying assumptions on their models that they would otherwise not need to make when using untimed model checkers like SPIN, or LTSA [49].

In Section 3, we will create a timed automata from the model defined in Section 2, extending it to model e-Science applications that are I/O-bound. Finally in Section 4, we will be sharing some of our experiences, conclusions, and forecasts.

6.2 Modeling BOINC

Requestors from the Research Community interact with a BOINC-enabled portal through its front door, like any other user process accessing a web service over the Internet. A BOINC-enabled portal, however, interacts with each of its service providers through its back door in three phases.

The first phase schedules work units based on preferences expressed by the human user who administers that PC. Here, the user acts on behalf of the service provider. The user may donate machine cycles to more than one project within a portal, and may even specify what share of otherwise unused cycles shall go to which

project. In this phase, a scheduler server at the portal also matches a requestor's requirements with the provider's resources [6].

As the input to a computation, a work unit could point to or include an executable file, input data, parameter values, or deadlines. It is downloaded in the second phase by a portal's download manager, executed by the service provider (reified as a BOINC *core client*); with the result uploaded in the third phase to the portal's upload manager. The second and third phases repeat until the human user changes scheduling preferences, adds or removes a portal, or brings the provider PC down for any extended period of time. In the following paragraphs, we model only the download and upload phases of a portal's interaction with a typical service provider and with a typical requestor from the research community.

The Petri net extended with inhibitor arcs [101] shown in Figure 6.1 describes how a portal might operate. Starting from an initial marking at places (2), (3), and (10) we assume that a human user is already using the provider PC (2) while a request from the research community (10) is pending, and the service provider is blocked in its initialized state (3). As long as the user is logged in (1) to a provider PC, the inhibitor arc prevents any further progress. The service provider remains initialized (3) but blocked (6) until the user logs out (4) ¹³.

Logging out causes the provider PC to become idle (5), unblocking the service

¹³ Since most home PC's run with Administrator privilege, without requiring the user to login or logout, BOINC waits until the machine had become idle with the screen saver enabled before unblocking.

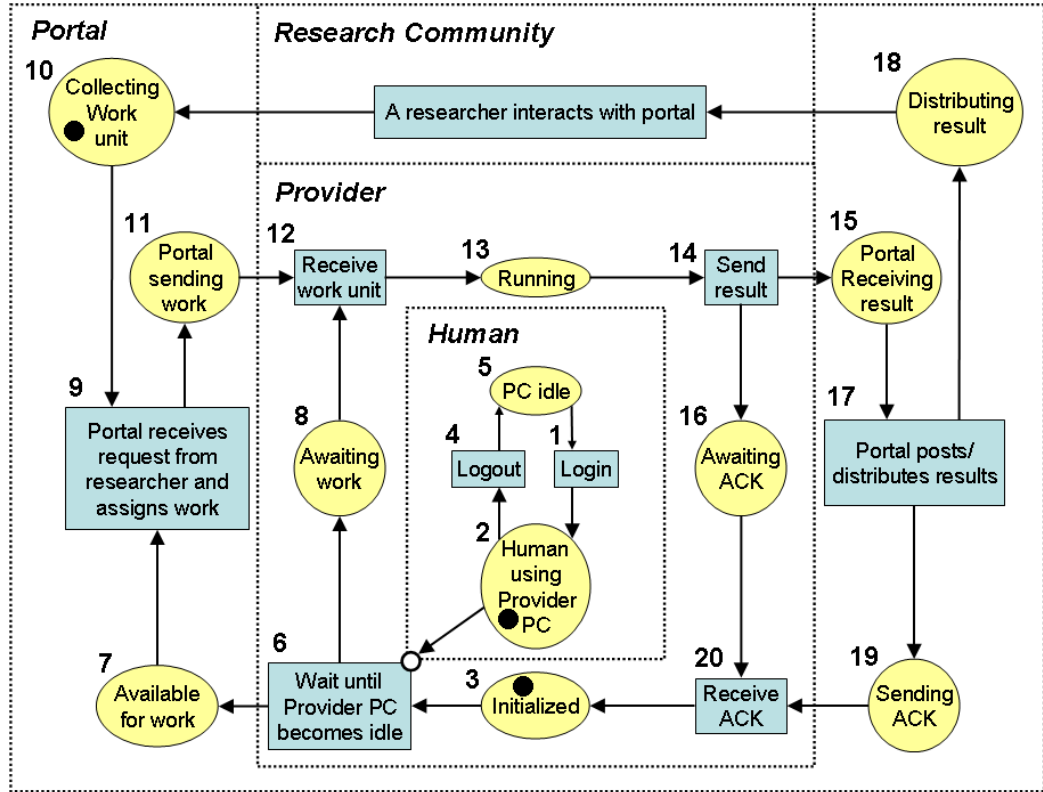


Figure 6.1: A Petri net model of BOINC.

provider (6), causing the service provider to inform the portal of its availability (7), and to await a work unit (8). In the meantime, the portal has been collecting requests (10) from the research community. Given a request, the portal assigns a work unit (9), sending it (11) to an awaiting (8) service provider. Upon receiving a work unit (12), the provider PC executes it (13), sending the result (14) to the portal (15), and awaits acknowledgement (16). Receiving the results (15), the portal posts and distributes them (17) back to the appropriate member of the research community (18), sending an acknowledgement back to the provider PC (19). Awaiting an acknowledgement

(16), and upon receiving it (20), the service provider returns to its initialized state (3). From here on out, steps (6) through (20) will repeat until the human user logs back in (1), enabling the inhibitor arc, to use the provider PC (2).

6.3 Modeling BOINC with UPPAAL

Expressing the untimed model in Figure 6.1 using a Petri net tool that supports inhibitor arcs (i.e., CPN Tools [73, 96]) would reveal three distinct problems. Firstly, in an untimed net, the user would be logging in (1) and out (4) as frequently as the firing of any other transition. This is not realistic, since a service provider typically completes multiple work units between user logout and login. Secondly, Figure 6.1 is optimistic, since it does not model network errors that result in lost work units. Modeling such an error with an untimed net would result in a deadlock that could have otherwise been resolved with the expiration of a timer. Lastly, the provider spends the vast majority of its time in one of only three places, or *dwell points*, in Figure 6.1. Thus, it makes sense to abstract a timed model from only these portions of the Petri net.

6.3.1 Mapping the Petri Net to Timed automata

The timed automata in Figures 6.2 and 6.3 were transcribed from the UPPAAL application we developed and show in Figure 6.4. They abstract away details not directly pertinent to timing. Dwell points include places 3 (initialized but blocked), 5 (provider PC idle), and 13 (provider PC running) of the Petri net of Figure 6.1. They

are the only places that directly correspond to locations ¹⁴ in the timed automata for the service provider in Figure 6.3.

Figure 6.2 depicting the portal, has its location 1 (paused) corresponding to location 1 (available) of Figure 6.3 that depicts a provider. These locations in Figures 6.2 and 6.3 together correspond to place 3 (initialized but blocked) in Figure 6.1. Location 3 (idle) of the provider in Figure 6.3 corresponds to place 5 (PC idle) in the Petri net of Figure 6.1. Focusing on the time-constrained behavior of a provider, we chose to model place 13 (provider PC running) in Figure 6.1 in greater detail. We mapped this place to locations 6 (Loaded), 8 (Running), and 11 (Done) of Figure 6.3. Before describing the timing behavior for the portal in Figure 6.2, and the provider in figure 6.3, we wish to make three observations that will further motivate extensions to these figures.

6.3.2 Extending the Timed Automata

The Petri net leaves out three important things that we wish to simulate in our UPPAAL case study. First, the human user can pre-empt the process only at one point, unfairly requiring the user to wait for completion of any long-running work units before having full use of the machine. Here, we wish to explore some policy alternatives allowing pre-emption half way through a run. Second, no distinction is made between predictably long-running work units, and those requiring predictably less time. Default configurations of infrastructures like BOINC are currently tuned to

¹⁴ Nodes in timed automata are known as *locations* while their arcs are known as *edges*.

compute-bound processes involving little data traffic. Not all e-Science applications, however, fit this mold. For example, deploying a facility like BLAST [3] for querying gene sequences in this massively parallel fashion requires first downloading a portion of a gene sequence database to a provider PC prior to querying it. The download will be a predictably long work unit, while an individual query will be a predictably short one. This case study models the typical situation in which a batch of 50 or so queries are processed in order to defray the initial cost of downloading a portion of the database [16]. Thirdly, Figure 6.1 ignores any notion of time or time constraint, considering only sequences of events. Much can go wrong, so often the best strategy, is to simply abort the timed out work unit.

The portal depicted in Figure 6.2 abstracts to a generic queue manager that eagerly sends tokens (i.e., work units) subject to time constraints on the provider in Figure 6.3. Thus, to understand the timing behavior of the portal, it is best to explain it from the perspective of the provider.

Initially in the Available location (1), the provider sends a signal to the portal once it becomes idle (2) with the *sending* synchronization construct $\text{idle[id]}!$, setting timer X to 0 units. The portal will, in turn, be listening for this message with its corresponding *listening* synchronization construct $\text{idle[e]}?$ at either of its transitions (2) or (4). Once the provider is in the Idle location (3), the portal has up to 20 time units to respond with a database download as specified by the *location invariant* ¹⁵

¹⁵ A location invariant is a time constraint associated with a location in a timed automata.

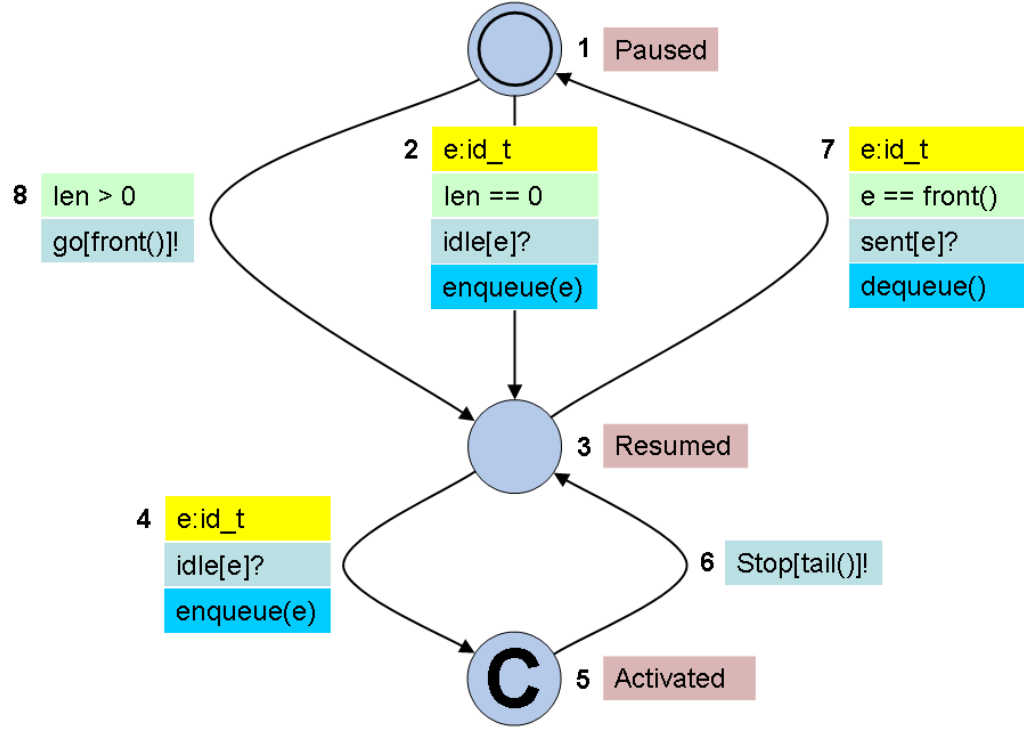


Figure 6.2: UPPAAL model of service portal.

$X \leq 20$. If the download takes more than the expected number of time units (i.e., 10) the provider will abort it and reset timer X (4). From there, it is done (11) and will send an error message back to the portal (12).

On the other hand, if the download completes when expected (5), then the provider transitions to the Loaded location (6). Once loaded, the provider listens for a batch of queries on the *go* channel with the construct `go[id]?` and resets the clock in (7). The portal will send a corresponding `go[front()]!` signal from (8) only if it has at least one work unit queued. From time to time in our simulations, we

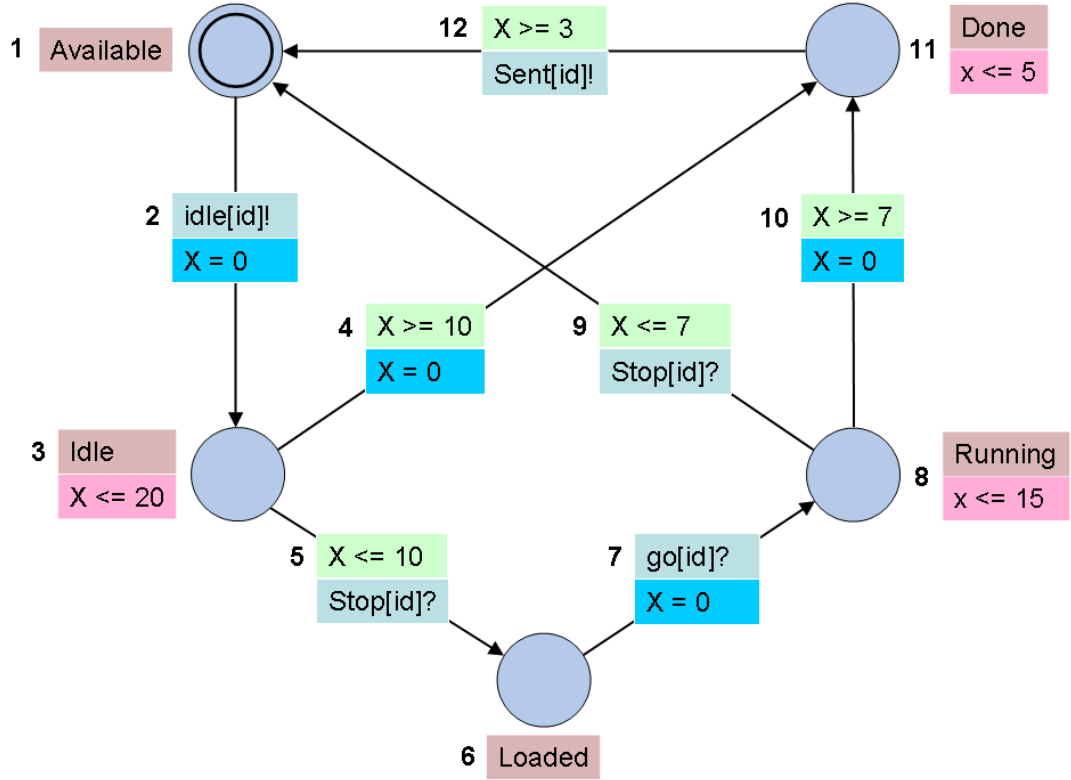


Figure 6.3: UPPAAL model of service provider.

observed the portal having no work to send a provider, which corresponds to the real world situation in which a provider momentarily waits for a work unit from a BOINC application. Once the portal sends a batch of queries for the provider to process, the provider transitions to the Running location (8).

In the Running location, the provider has up to 15 time units to process this batch, trickling the results back to the portal. If a human user pre-empts the provider PC before 7 time units, then it will take transition (9) and simply abort the job. On the other hand, if pre-empted after 7 time units, we modeled the batch as running

to completion (10) to reach the Done location (11). In the meanwhile, both the service provider and the human user will be using the machine, each experiencing some performance degradation. From the provider's Done location (11) it has up to 5 time units to upload results via the `Sent[id]!` synchronization construct (12) corresponding to the portal's `Sent[e]?` construct on its transition (7). A screen shot of this simulation in the UPPAAL environment appears in Figure 6.4.

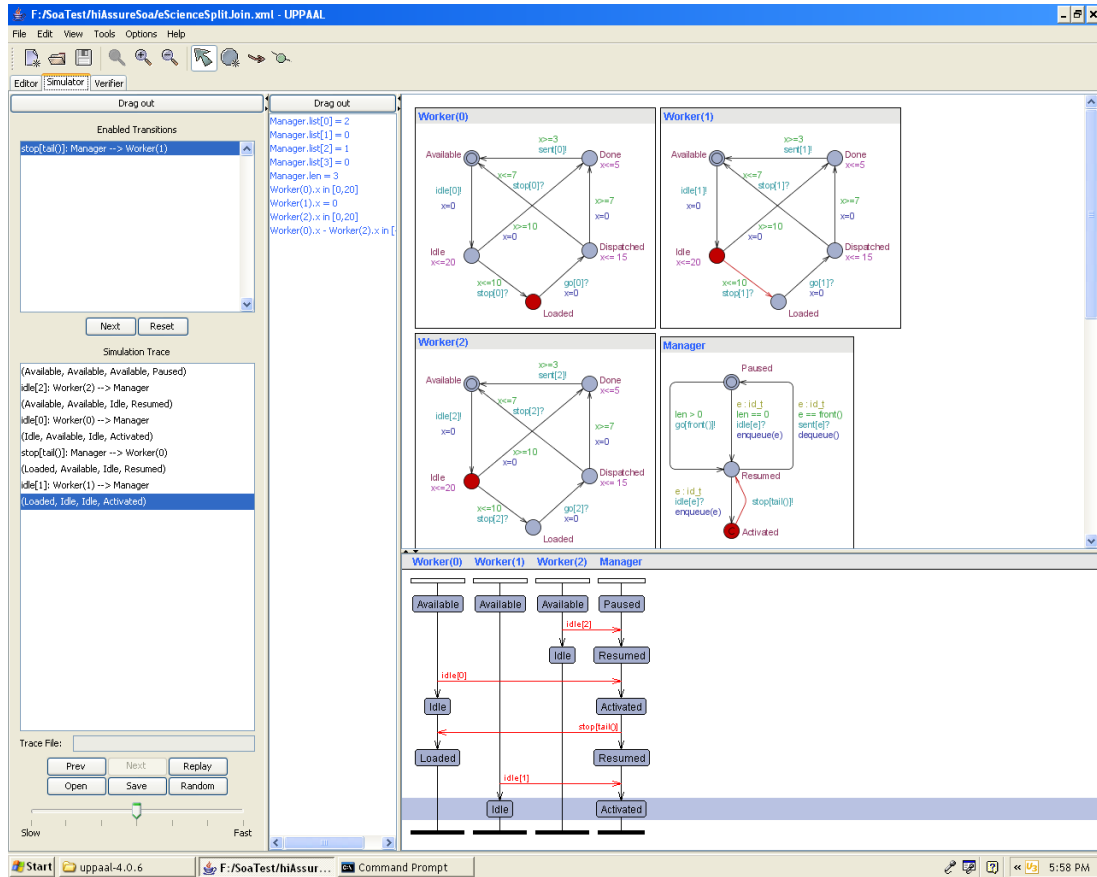


Figure 6.4: UPPAAL simulation environment showing one portal and three providers.

6.3.3 Model Check Timed Automata

We specified and successfully model checked nine properties expressed in computational tree logic (CTL) [13] in Table 1. Validation properties (1) through (4) provide sanity checks. For example, in (1) the portal can *eventually* receive and enqueue work units from waiting providers. In (2) provider 0 can *eventually* reach the Done location, while in (3) provider 0 can be finishing a work unit while provider 1 is Loaded and awaiting a query type work unit. Some of these properties can get quite sophisticated as in (4) in which provider 0 can be finishing a work unit while all other providers are Loaded and awaiting query type work units. One safety property, known as a *boundedness condition* is (5) where there can never be N work units in the queue, where N is greater than the number of providers. Liveness properties (6), (7), and (8) assure that whenever a work unit is queued up for processing, it will eventually be processed. Finally, (9) specifies that the system must be deadlock-free. Note how most locations have invariants that force a timeout if a work unit remains there for too long. This is how UPPAAL uses timers to break possible deadlocks. Negating the properties in Table 1 causes UPPAAL to generate counterexamples that can be used as a test suite. This and other novel uses of model checkers are discussed in [119].

6.4 Experiences and Conclusions

This chapter is an outgrowth of a comparison of three model checkers over three case studies [103], prompting us to notice certain patterns. Little change was

Table 6.1: Model Checked Properties

	Validation Properties:
1	$E \langle \rangle \text{Portal.Resumed}$
2	$E \langle \rangle \text{Provider}(0).\text{Done}$
3	$E \langle \rangle \text{Provider}(0).\text{Done}$ and $\text{Provider}(1).\text{Loaded}$
4	$E \langle \rangle \text{Provider}(0).\text{Done}$ and $(\text{forall } (i : \text{id_t}) i \neq 0 \text{ imply } \text{Provider}(i).\text{Loaded})$
	Safety Properties:
5	$A[] (\text{Portal.list}[N] == 0)$
	Liveness Properties:
6	$\text{Provider}(0).\text{Idle} \dashrightarrow \text{Provider}(0).\text{Done}$
7	$\text{Provider}(1).\text{Idle} \dashrightarrow \text{Provider}(1).\text{Done}$
8	$\text{Provider}(2).\text{Idle} \dashrightarrow \text{Provider}(2).\text{Done}$
	Deadlock freedom:
9	$A[] \text{ not deadlock}$

required in the UPPAAL design of the portal side between seemingly disparate case studies, since Figure 6.2 implements a queuing manager interaction pattern with portal behavior subject to provider-side time constraints. Additionally, all three model checkers (i.e., SPIN, LTSA, and UPPAAL) support modeling from a *process-oriented* perspective.

UPPAAL exhibited some superficial idiosyncrasies concerning time bounds, which ranged from a low of 3 to a high of 20. This narrow range avoids state space explosion due to wide variations in time constraints. A more realistic interpretation of these bounds involves scaling them in powers of 2. Thus 3 time units may correspond to 2^3 or 8 seconds – a maximum request/response latency over the Internet. On the other hand, 20 time units may correspond to 2^{20} or just over a million seconds or two weeks – the amount of time a human user may be on vacation, rendering the provider

inactive. Interpreting Figure 6.3 using these scaled values can provide more realistic timing constraints, although caution must be exercised when reasoning in terms of these scaled values. To model e-Science workflows that each operate within tight (i.e., [3..20]) time bounds at each of two distinct time scales (i.e., milliseconds and hours), requires introduction of two distinct *unscaled* clock variables. In a broader sense, we foresee increased interest in time as a first-class concept.

Contrasting e-Science Volunteer Supercomputing from e-Commerce Web Services applications, we note that the former has a larger and possibly more volatile set of providers, where each provider furnishes a narrower range of services (i.e., compute power). These characteristics may pose challenges to applying existing Web Services approaches that use standards like WS-BPEL [7].

We foresee the incorporation of I/O-bound processing into Volunteer Supercomputing. To this end, we modeled this proposed extension to BOINC's functionality. In our case study of BLAST, batching may inconvenience the requestor, who must wait for other requestors to submit the required sized batch. Modeling smaller batches will not change the topology of the timed automata, only the time invariant for location (8) and for edges (9) and (10).

We foresee an externalizing of certain operating systems functionality out to the Internet. Implementing a *demand paging* scheme where each provider holds a distinct portion of a database over some period of time, can support the processing of individual independent queries. Modeling this extension involves bypassing the

loading phase by placing an additional edge between locations 3 and 8 of Figure 6.3.

Process-oriented modeling, treatment of time as a first-class concept, burdens placed on existing standards, the incorporation of I/O-bound processing, and the externalization of operating systems functionality – all promise to shape the future of e-Science SOA's.

CHAPTER 7

STATE COMPRESSION AND ABSTRACTION

This chapter is based on the paper titled: *Formalizing Fault Trees for Remote Ocean Systems* [125], which provides a set-theoretic formulation for fault trees. Evaluating a fault tree given some system state, results in an abstraction on that state containing a list of failure types and metrics for severity.

Real time condition-based evaluation of system health must not only be efficient, but also produce usable and expressive results. To this end, this chapter presents a set-theoretic formulation of fault trees. Such a formulation provides a usable scaffolding on which ensembles of machine health measurement techniques may eventually operate. Initially, we present a negation-free formulation of a forest of fault trees as a set of 2-level sum-of-products expressions. Given this formulation, we express measures for *certainty* and *specificity*, both of which further qualify the various well-studied measures for *severity*. This formulation is subsequently refined to represent multi-state systems, non-coherent valuations, and node sharing – all necessary for practical monitoring solutions. Finally, we present an evaluation rule embodying these refinements and analyze its complexity. Examples pertaining to unattended ocean systems illustrate these concepts.

7.1 Chapter Introduction

Fault tree analysis uses individual state snapshots emanating from the machines being monitored for condition-based reliability assessment. This analysis requires well-conditioned and de-noised data from a machine’s data acquisition / manipulation system. As a type of *logical fault model*, fault trees require data captured at some point in time and bundled into a vector V^n of n state variables indexed by $\{i : 0 < i \leq n\}$. Such data initially¹⁶ comprises the state snapshot. Prior to conducting fault tree analysis, each state variable $v_i \in V^n$ had been coarsely discretized based on a variety of state detection algorithms. Furthermore, each variable constitutes the head element of its own timed data stream, where all such streams are subject to some form of barrier synchronization [137].

In this work, we will assume that each snapshot V^n had been made *temporally coherent*. That is, all values in V^n have been acquired within some acceptable time window using data fusion strategies surveyed in [41]. Later work will parameterize each state variable v_i with time (i.e., v_{ti}) to represent some state variable in a snapshot at time t . *Transition fault models*, like Markov chains or Petri nets, can then be applied to sequences of these time-parameterized snapshots.

This chapter focuses on fault trees used for continuous monitoring and diagnosis of unattended ocean machinery. For an overview of reliability issues associated with such machinery, see [123]. Since responding to false alarms (i.e., false positives)

¹⁶ In a later section we will be augmenting V^n with events derived from combinations in V^n .

for such machinery incurs a high expeditionary cost, this chapter considers logical fault models that are sound.

A unifying formalism is required to express the capabilities of successively more robust classes of fault trees. Section 7.2 defines an initial class of fault trees based on simple sum-of-products expressions. Section 7.3 augments this basic structure for multi-state systems, (non-)coherent valuation functions, and shared nodes. Section 7.4 presents a real time fault tree evaluation rule and examines its complexity, distinguishing this result from complexity results for various aspects of fault tree construction. Section 7.5 presents a summary and lists future work.

7.2 Mathematical Structure

A physical machine produces a *valuation* of vector V^n of n state variables using Boolean function $\Phi : V^n \rightarrow \{0, 1\}^n$, where for state variable v_i a fault is indicated by its value $[v_i] = 1$. The collection of all such faults, or more generally *events*, are referred to as the *extension* of V^n , namely $V^+ = \{v_i : [v_i] = 1\}$. Extension V^+ provides the input instance to an analysis involving fault trees described in the next paragraph, while Table 7.1 summarizes parameters and properties of the state snapshot.

Fault trees are used to assess the machine's health given V^+ . Consider a forest of p 2-level fault trees for p possible event types. Each tree represents a logical sum-of-products expression over $v_i \in V^n$ mapping to some event type x_l . In particular,

a fault (sub)tree l is a mapping $\Psi_l : E^q \rightarrow X^1$ defining the disjunction of q_l *cut sets* corresponding to exactly one event type $x_l \in X^p$ among p possible fault types. The resulting event type x_l may be interpreted as a type of low-level failure or higher-level fault.

Each cut set $e_j \in E^q$ is itself a conjunction or product over r_j state variables wherein each state variable is indexed by k . Each such variable comprising e_j is thence denoted as u_{jk} . Cut sets are manually specified by a domain expert, with additional cut sets generated using a variety of imputation techniques. The resulting m cut sets indexed by $\{j : 0 < j \leq m\}$ spanning all p fault trees are nonetheless many orders of magnitude smaller than the cardinality the power set over V^n . Already known at run time, the total number of cut sets m is expected not to exceed several million. Table 7.2 summarizes parameters and properties of cut sets. Formula 7.1 computes the value $[x_l]$ as a Boolean function in disjunctive normal form (DNF).

$$[x_l] = \bigvee_j^{q_l} \bigwedge_k^{r_j} u_{jk} \quad (7.1)$$

Formula 7.1 can be made concrete by the following *evaluation* condition:

$$[x_l] = 1 \Leftrightarrow \{k : (\exists j)(u_{jk} \in e_j, x_l = \Psi(e_j), e_j \subseteq V^+)\} \quad (7.2)$$

Formula 7.2 stipulates that a fault tree rooted in x_l evaluates to true (i.e., $[x_l] = 1$) iff every variable u_{jk} in at least one cut set e_j corresponding to x_l is wholly

contained in extension V^+ . This firing condition provides a scaffold on which diagnostic measures for *certainty*, *severity*, and *specificity* can be computed. First, consider the certainty measure. For some fault, two cut sets evaluating to true will result in a higher certainty measure for an event than if only one cut set were to fire. Likewise, if the same number of two or more cut sets were to fire for one event as it did for some other event, then the event having the more dissimilar cut sets would have the higher certainty score. Next, consider the severity measure. An extensive literature exists for the computation of such scores based on the severities of individual events making up the cut set and on the combination of events comprising the cut set. Finally, consider the specificity measure. Given two distinct firings $x_{l_0} = \Psi(e_{j_0})$ and $x_{l_1} = \Psi(e_{j_1})$, if $e_{j_0} \subset e_{j_1}$, then x_{l_1} will have a higher specificity value than would x_{l_0} .

Although Formula 7.2 may be satisfactory for a single 2-level fault tree, real time fault tree evaluation requires listing *all* such fault trees evaluating to true given V^+ . This entails propagating extension V^+ , including its associated metrics, to extension E^+ for cut sets and thence to extension X^+ for fault trees in the forest. Extension $E^+ = \{e_j : e_j \subseteq V^+\}$ comprises all cut sets e_j that are wholly contained in V^+ . Based on E^+ , the list of all fault trees evaluating to true entails computing X^+ described in Table 7.3, which also summarizes the parameters and properties of fault trees. Composing extension properties from Tables 7.1 - 7.3 results in Formula 7.3.

$$X^+ = \{(x_l, j) : (\exists e_j)(x_l = \Psi(e_j), e_j \subseteq V^+)\} \quad (7.3)$$

Among other things, the following section augments this 2-level structure into a hierarchy so that X^+ and its metrics may thence be propagated up to some Top-Level Event (TLE).

7.3 Refinements

The structure in Section 7.2 can neither represent multi-valued state variables, nor composition of fault trees into a hierarchy ultimately rooted in some Top-Level Event (TLE). The negation-free notion of extension developed thus far has limitations, which we extend to multi-state systems in Section 7.3.1. Section 7.3.2 introduces a means of specifying hierarchies of events using the notion of shared nodes. Shared nodes reduce the need to replicate fault trees for multiple types of events common to specific combinations of lower level events. As alluded to earlier, shared nodes provide placeholders for certainty, severity, and specificity measures, as well as a framework for propagating these values to the TLE for overall machine health assessment. These values will furthermore depend on whether the valuation of the system's state is coherent. Closely related to multi-state systems, the coherence property is examined in Section 7.3.3.

7.3.1 Multistate Systems

The extension expressed in Formula 7.3 can express the *presence* but not the *absence* of events. That is, negation is not supported. Unfortunately, the inability to affirmatively test for the *absence* of events prevents use of diagnostic procedures

involving the *ruling out* of certain other lower-level events. Modeling explicitly binary-valued variables will enable *differential diagnosis* – a process by which one event can be distinguished from some other event based on the absence of certain other events. A fault tree that supports negation can make explanation of an event easier by stipulating inside the contents of cut sets which events are expressly absent.

Suppose by V^{n_1} , we mean vector V^n of negation-free states defined in Section 7.2. Introducing the notion of negation entails augmenting V^n with a set V^{n_2} of binary-valued state variables. Applying the technique used to reduce the Satisfiability Problem (SAT) to Monotone SAT referred to in Appendix A9 of [52], we can augment V^n with distinct and indivisible state variables $\neg v_i$ subject to the *separation* condition in Formula 7.4.

$$\{i, j : v_i \in e_j, \neg v_i \in e_j\} = \emptyset \quad (7.4)$$

A parsimonious state V^{n_2} would include only $\neg v_i$ variables for which there exists at least one cut set. Subjecting V^{n_2} to this *relevance* condition – necessary for *coherence* – results in Formula 7.5.

$$V^{n_2} = \{\neg v_i : \exists(e_j)(\neg v_i \in e_j)\} \quad (7.5)$$

Three-valued variables, like operating temperature, are useful for detecting departures from some *interior* optimum. Notions like: 'depressed', 'normal', and 'elevated' can be expressed in a manner similar to the negation case by including

state variables like $\{-v_i, \neg v_i, +v_i\}$ respectively into V^{n_3} . By a similar construction one may define the four-valued notion: 'normal', 'alert', 'warning', and 'emergency' into V^{n_4} . Hence, the refinement for multi-state systems involves computing the state vector in Formula 7.6, and redefining superscript n accordingly.

$$V^n = V^{n_1} \cup V^{n_2} \cup V^{n_3} \cup V^{n_4} \quad (7.6)$$

Fault trees for multi-state systems entail more intricate computation of severity scores like those described in [25, 149]. Such fault trees require construction of a larger number of cut sets, or alternatively, incorporating a notion of ordering when computing extension X^+ . Subjecting their construction to separation and relevance conditions partially mitigates these tractability problems.

7.3.2 Node Commonality

As a two-level sum of products expression, Formula 7.1 cannot represent an arbitrary Boolean expression without replicating a potentially large number of sub-trees. The same event x_l may be in common with more than one fault sub-tree. Furthermore, x_l may be a member of more than one cut set within a fault tree. Finally, fault trees are rarely balanced so that a (sub)tree rooted in event x_a may include cut sets comprised of faults v_i and child events x_l . By a technique similar to that in Section 7.3.1 we may further augment V^n with $x_l \in X^p$ to result in W^{n+p} . Inclusion of x_l into W^{n+p} is subject to a separation condition like that in Formula 7.4.

If all events $x_l \in X^p$ are *relevant* to the TLE – by a condition similar to Formula 7.5 – then the augmented state is computed by Formula 7.7.

$$W^{n+p} = V^n \cup X^p \quad (7.7)$$

Defining W^+ in a manner similar to V^+ , extension X^+ can be redefined by Formula 7.8.

$$X^+ = \{(x_l, j) : (\exists e_j)(x_l = \Psi(e_j), e_j \subseteq W^+)\} \quad (7.8)$$

7.3.3 State Coherence

The hazard scoring and propagation techniques described in [25, 75, 149] assume the multi-state system being modeled is *coherent*, namely the valuation of its state does not improve with an increasing number of component faults. We adapt the definition of coherence in [31] to specifically highlight the problem of transient effects causing false positives.

Valuation function $\Phi : W^{n+p} \rightarrow B^{n+p}$ of augmented state snapshot W^{n+p} is said to be *coherent* if (i) all variables are relevant, and (ii) $\Phi(W^{n+p})$ is monotonically non-decreasing.¹⁷ For condition (i), every state variable $v_i \in W^{n+p}$ and failure type $x_l \in W^{n+p}$ are part of at least one cut set, so all variables in W^{n+p} are *relevant*. For

¹⁷ $\Phi()$ is said to be *strictly coherent* if it is monotonically increasing.

condition (ii), consider two successive state snapshots, $W_{t_0}^{n+p}$ and $W_{t_1}^{n+p}$ at times t_0 and t_1 respectively. If $W_{t_0}^{n+p} \subseteq W_{t_1}^{n+p}$ then valuation function Φ is coherent.

The succession of states in the previous paragraph indicates either the presence of a spreading fault or transient effects. To rule out transient effects, suppose the only difference between the two states is the change of one or more variables w_i from zero to one. If at some later time t' , $W_{t'}^{n+p} = W_{t_0}^{n+p}$ and there were no repair actions, then intermediate state $W_{t_1}^{n+p}$ reflects transient effects.

The notion of what constitutes *transient effects* is open to interpretation. Few would argue that a machine experienced transient effects when it broke down but was subsequently repaired. Considering events requiring repair as *stuck-at* faults, excludes these situations from the notion of transient effects.

Less obvious are transient effects stemming from routine operation and automated control. Such effects are expected to occur often as ocean turbines right themselves in the presence of turbulent waters. Data streaming from an attitude sensor known as an Inertial Measurement Unit (IMU) at 10 Hz may be used for self-righting. The fault tree, however, must depend on these pitch yaw and roll measurements conditioned over longer time intervals. The intention is to detect failures in the control system, rather than registering an abundance of false positives that have been automatically corrected by movement of any rudders or fins. Other sensor types noted for displaying transient effects include the five vibration sensors located at various points on the turbine's drive train.

A recent study [31] surveyed other scenarios in which non-coherence apparently arises, and how machine health assessment procedures can be adapted. One such scenario in [31] anticipated our need to operate ocean turbines at reduced output as its state gradually degrades. For example, operating the turbine closer to the surface maximizes momentum flux, and hence output, but at the expense of both turbulence and accelerated rates of fouling. As its state degrades, due either to bad weather or biofilm formation, the turbine can operate at reduced output further down the water column. Such degraded operation comes at the expense of increased bathymetric pressure on seals. Hence, formulating an appropriate response to degraded states due to one set of variables like turbulence and fouling, will often depend on other variables like bathymetric pressure.

7.4 Algorithm Design and Analysis

Fault tree evaluation ultimately entails evaluation of Boolean functions. A variety of implementations of Boolean function manipulation and evaluation involve binary decision diagrams (BDD)’s, hypergraphs, and SAT solvers. In addition to work already cited, implementations of fault trees using BDD’s or their variants also appear in [8, 18, 95, 97, 109]. Visualizing Satisfiability (SAT) instances and Boolean functions that include the use of hypergraphs were reported in [118]. Use of hypergraphs that relate vertices to state predicates w_i and hyperedges to clauses e_j were reported in [59], providing a context for the complexity analysis in the following

paragraphs.

7.4.1 The Evaluation Problem

Recall that to solve the Satisfiability Problem (SAT) requires finding a satisfiable truth assignment, given some Boolean formula [53]. That entails first *guessing* a truth assignment, then *checking* if that assignment is indeed satisfied. Although both operations can be done in low-order polynomial time, what makes SAT NP-Complete (NP-C) in the number of variables $n + p$ is the intractably large number of guesses required to identify a satisfying truth assignment.

The *checking* phase of SAT trivially reduces to computing extension X^+ in Formula 7.8. To see this, negate the problem instance for SAT originally expressed by Cooke in Conjunctive Normal Form (CNF) to obtain the fault tree in DNF shown in Formula 7.1, and then reverse the sense of the truth assignment.

In condition-based monitoring, we are already given truth assignment V^+ so *checking* whether X^+ is non-empty can be done in polynomial time. Event explanation and possibly localization, however, requires listing *all* event types generated ultimately from extension V^+ . Known as the SAT Evaluation Problem, implementations of this checking procedure are defined for most variants of BDD's surveyed in [38]. One state-of-the-art data structure known as Zero-suppressed Decision Diagrams (ZBDD) is adapted to efficient identification of sets of subsets [93]. Run time comparison of these and other data structures and techniques for computation of

X^+ is left for future work. As a reification of Formula 7.8, Formula 7.9 provides an evaluation rule for computing extension set X^+ .

$$\begin{aligned} & \{i, j : [[w_i \in W^+, w_i \in e_j] \Rightarrow c_j \leftarrow c_j + 1]; c_j = |e_j| \Rightarrow \\ & X_{t+1}^+ \leftarrow X_t^+ \cdot (\Psi(e_j), j); W_{t+1}^+ \leftarrow \{\Psi(e_j)\} \cup W_t^+; c_j \leftarrow 0\} \end{aligned} \quad (7.9)$$

This rule supposes that for each cut set e_j , we maintain a count c_j , initially 0, that gets incremented for each $w_i \in W^+$ whenever $w_i \in e_j$. Once count c_j equals $|e_j|$, clause e_j is fully satisfied, so we append the event associated with e_j and cut set identifier j to extension X^+ . Including j in the solution facilitates fault localization and explanation. Finally, we include the event associated with e_j into set W^+ and re-initialize c_j to zero. Implied in this rule is some stepwise algorithm, the description of which is forthcoming.

Formula 7.9 solves the Fault Tree Evaluation Problem with a worst case complexity of $\mathcal{O}(m^2)$. To see this, contrive a problem instance comprised of m cut sets with $n + p = m$ elements in W^+ . Such pathological cases are easy to spot at the time of fault tree construction by detecting violations of the bounds in Formula 7.10.

$$\{i, j : |W^+| \ll (n + p), |e_j| \ll (n + p), |Q_i| \ll m\} \quad (7.10)$$

Formula 7.10 asserts that the fault tree rooted in the TLE is comprised of a sparse set of cut sets wherein each cut set is itself sparse in the number of variables. Note that set Q_i represents the set of cut sets in which w_i is a member. The actual

number of steps can be expressed by Formula 7.11 and can be predicted in $\mathcal{O}(|\mathcal{W}^+|)$ steps.

$$\sum_i^{|\mathcal{W}^+|} |Q_i| \quad (7.11)$$

7.4.2 Construction Problems

Although this chapter focuses on SAT Evaluation, a number of problems associated with the *construction* of fault trees remain NP-C. Unless $P = NP$, fault trees for machinery like ocean turbines will always remain not fully specified with imperfect fault coverage. This is unfortunate, since silent failures (i.e., false negatives) disrupt maintenance schedules, which are driven by the high expeditionary cost incurred prior to ocean equipment maintenance and repair. Silent failures too often mask unexpectedly severe damage, while providing insufficient or misleading diagnostic information to maintenance personnel.

An NP-C problem known as the Automatic Test Pattern Generation Problem (ATPG), from the field of electronic design automation, can be reduced to a corresponding problem in fault tree construction. ATPG requires the listing of all test patterns (cut sets) that can lead to failure. Not only is ATPG NP-C, but it is also PSPACE-Complete – requiring intractably large storage. Under a similar guise, any imputation technique by which one must infer a complete set of cut sets given some

'starter' set, also appears to be PSPACE-Complete. Still another intractable problem reducible from SAT involves the computation of prime implicants in a Boolean expression – important for identifying the cut sets having the least number of conditions that can prompt some given event. Additional NP-C problems associated with BDD's and hence fault trees were identified in [38]. Fault trees generated by imputation procedures will have its number of cut sets vastly exceeding the number of variables. Partially mitigating this, we observed cut sets rarely exceeding four terms when formulating the starter set for ocean turbines.

7.5 Chapter Summary

The set-theoretic perspective on condition-based evaluation of fault trees enabled us to define a class of fault trees useful for health assessment of remote ocean machinery. This class can represent multi-state systems, non-coherent valuations, and node sharing – all of which are pre-requisites for current research into logical fault models like fault trees. Due to high expeditionary cost to service this machinery, we sought to minimize the number of false positives by formally characterizing one class of transients in terms of a state coherence condition.

Each successive refinement exposes variables to which we may attach health assessment indicators. In addition to the well-studied phenomenological measures for severity, the compositional framework also characterized epistemological measures of certainty and specificity. The proposed framework provides an effective fault tree

evaluation rule having complexity bounds known at fault tree construction time.

Future work involves identifying fault tree evaluation tools included with SAT solvers and BDD packages. Run time and usability comparison of these tools to our software implementation of the SAT evaluation rule is anticipated. Technical documentation currently underway will be posted, along with the current version of the executable code for the SAT evaluator in Formula 7.9, its supporting tools, and sample fault trees.

Property:	Description:
indexing	$\{i : 0 < i \leq n\}$
membership	$v_i \in V^n$
valuation	$\Phi : V^n \rightarrow B^n$
extension	$V^+ = \{v_i : [v_i] = 1\}$

Table 7.1: Properties of state snapshots

Property:	Description:
indexing	$\{j : 0 < j \leq m\}$
membership	$e_j \in E^m$
valuation	$E^m = \{e_j : e_j \subseteq V^n\}$
extension	$E^+ = \{e_j : e_j \subseteq V^+\}$

Table 7.2: Properties of cut sets

Property:	Description:
indexing	$\{l : 0 < l \leq p\}$
membership	$x_l \in X^p$
valuation	$\Psi : E^m \rightarrow X^p$
extension	$X^+ = \{(x_l, j) : x_l = \Psi(e_j), e_j \in E^+\}$

Table 7.3: Properties of fault trees

CHAPTER 8

CASE STUDY – OCEAN TURBINES

This chapter is based on the paper titled: *Ocean Turbines – a Reliability Assessment* [123], which identifies factors that impact reliability and safety of ocean turbines. We describe how physical and environmental factors will impact the design of its machine condition monitoring (MCM) system. Environmental factors like fouling, corrosion, and inaccessibility of equipment sets this MCM problem apart from those encountered by wind turbines, hydroelectric plants, or even ship hulls and propellers. Fouling constitutes the primary and most persistent source of failure. In addition to compromising turbine efficiency and reliability, fouling reduces sensor data quality – masking faults that will ultimately lead to failure. Unmitigated fouling triggers a form of biological succession known as *flocculation* that may eventually attract threatened species of tortoises and cetaceans to this rotating machinery. We review and suggest refinements to a class of non-toxic biologically-inspired anti-fouling techniques known as *engineered topographies*. Advances in this area will enable turbines to operate in portions of the water column that maximize momentum flux while minimizing retrieval cost.

8.1 Chapter Introduction

Ocean turbines for generating electricity from the Gulf Stream had been proposed since the 1970's. Although no such equipment had been deployed in the Gulf Stream, an experimental turbine had been deployed in the turbulent waters of the East River in New York City by Verdant Power, LLC. Since 2007, the authors from the Center for Ocean Energy and Technology (COET) at Florida Atlantic University¹⁸ have been measuring the Gulf Stream's potential as an energy source using acoustic Doppler current profilers (ADCP) [40]. They concluded that the ocean current at the deployment site is suitable for base power generation at some minimum guarantee [39] – fetching a higher rate per kilowatt hour than intermittently available sources like wind. One minimum guarantee may promise a minimum output from a minimum current velocity for a guaranteed uptime. An example of the former is a water current velocity of 1.2 meters per second, and the latter of eighty-five percent.

Concurrent with the ADCP effort, COET is developing a small scale ocean current turbine and mechanical test bed. This turbine and test bed will provide baseline technical, environmental, and ecological data to help guide the commercial and policy development of open ocean hydrokinetic resources.

Recent advancements in ocean and marine technology introduced complexity in MCM systems, while long-standing problems remain. A problem spanning both worlds – the world of the measurement and the world of the observation – is *fouling*.

¹⁸ <http://coet.fau.edu/>

Fouling compromises accuracy of data emanating from various sensors, which consequently reduces effectiveness of MCM systems. Fouling constitutes the initial stage of flocculation. As a process in which successively more complex species are attracted to the surfaces of submerged structures, flocculation encounters some state transition in which whole clumps peel off from the big colony into individual *flocs*. Each floc can sustain life until the floc fastens itself to some surface farther downstream. Ultimately, this process can attract threatened species of tortoises and cetacean mammals to (albeit) slowly rotating machinery.

This chapter examines risk factors associated with deployment of a proposed fleet of ocean turbines that will be harnessing the energy from the Gulf stream located about forty miles off the East coast of Fort Lauderdale Florida. Monitoring the rate of fouling and mitigating it, presents the single biggest and most unique reliability challenge for this particular application. Any MCM solution will demand dedicated functionality for assessing degree of biofouling. Data from these automated assessments need to be fused with data emanating from the other types of sensors (i.e., vibration) to compensate for the masking of higher modes due to biofouling. Barring significant advances in anti-fouling technology, retrieval and maintenance schedules will continue to be dominated by rates of fouling. Responding to this critical need, we examine the effectiveness of biologically-inspired or *biomimetic* surface topographies. We extend a definition for engineered roughness to account for the effectiveness of self-similar surfaces. Finally, we suggest refinements to the design and fabrication of

such surfaces.

The rest of this chapter is organized as follows: Section 8.2 describes the physical design of the turbine and its moorings. Section 8.3 elaborates on reliability concerns that are unique to this application. Section 8.4 surveys related work. Section 8.5 takes aim at biofouling, particularly for propellers, by proposing refinements to engineered surfaces. Finally, Section 8.6 provides a summary and description of future work.

8.2 Physical Design

A 20-kW turbine prototype [39] being developed at COET along with its moorings are shown in Figure 8.1. This figure denotes (a) Northbound direction of the ocean current as seen from shore, (b) observation control and deployment platform (OCDP), (c) mooring and telemetry buoy (MTB), (d) a tether to the ocean floor, and (e) a *nacelle* that includes the turbine, buoyancy chambers, and connecting structures. In short, the turbine is housed in a buoyancy controlled vessel tethered to a barge, with electrical and communication cabling to a buoy, and anchored to the seafloor via a towline.

Figure 8.2 provides a close up look at the turbine. A three-blade propeller (a), is connected to pressurized enclosure or nacelle (b). The nacelle contains an asynchronous electric motor/generator connected by a shaft supported by bearings and thence connected to a gear reduction box. Pitch, yaw, and roll of the nacelle are

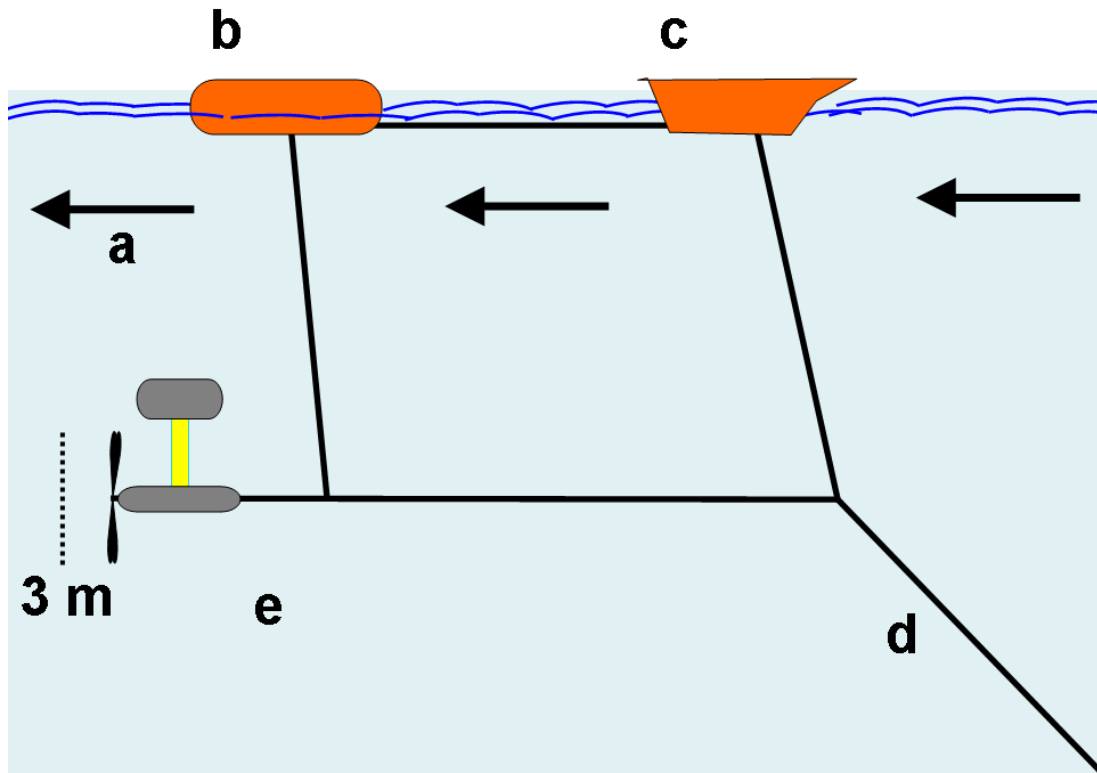


Figure 8.1: Moorings for an ocean turbine

partially controlled by pressure buoy (c). The propeller occupies the most downstream portion of this design, where the nacelle and its propeller are passively pulled by the Gulf Stream current.

Table 8.1 lists the physical characteristics of the turbine and its moorings. The critical components to be monitored include the turbine nacelle pressure vessel, motor/gearbox, propeller, and electrical system. Temperature, position, roll, pitch, yaw, and bilge water level of the turbine nacelle pressure vessel will be monitored using thermometers, a 6-axis inertial measurement unit (IMU), and water sensors.

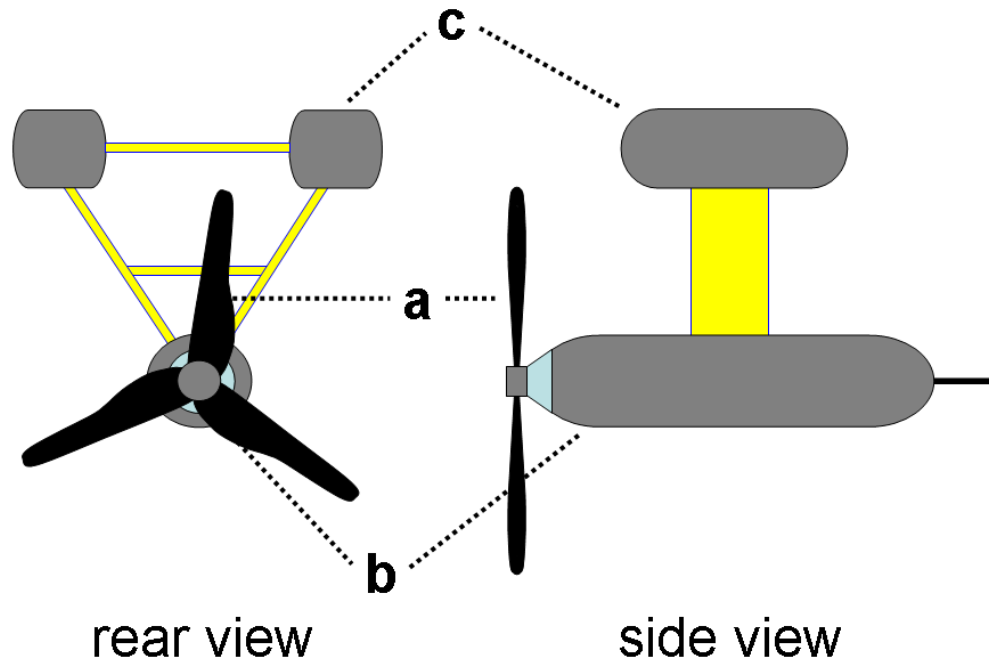


Figure 8.2: Nacelle and adjoining structures

Vibrations in the transmission shaft, gearbox, and motor will be monitored using low and high-frequency accelerometers, indicating any imbalance or wear on bearings or gears. The torque, strain/vibration, water flow, turbulence, and immediate environmental surrounding of the propeller during operation will be monitored respectively by a load cell, strain gauge, flow meter, ADCP, and video cameras. This will indicate the loss of a portion of the propeller, excessive strain thereto, or significant unbalance due to biofouling. Finally, a ground fault interrupter will detect and protect the system from ground faults in the electric motor or electric cable.

The safety system operates independently from its monitoring system and can halt turbine operation automatically in the event of a number of top-level faults. A non-exhaustive list includes signs of overheating, dynamic instability, leak, excessive voltage and current across the power plant, cable failure or dramatic change in propeller angular velocity (i.e., sudden stop). This safety system enables an external operator to properly shutdown the power plant.

The machine condition monitoring (MCM) system records and processes a series of measurements, producing a timed data stream for real time health assessment, prognostics, and advisory generation. Some of these measurements include mechanical vibration, propeller angular velocity, heat, cable tension, voltage and current of the power plant, lubricant quality, and video imaging. Some of the sensors are shared between the safety system and the MCM system. Due to its steep downside risk, the safety system samples data at a substantially higher rate than the one for MCM.

Both safety and MCM systems are distributed between the wet-side (i.e., submerged nacelle) and the topside (i.e., pontoon and shore). Initially the user control and display is located on the pontoon, however, this functionality will eventually be migrated onto shore. The safety system uses lower latency but slower throughput serial channels. The machine monitoring unit uses Ethernet due to its higher bandwidth. Data is relayed between wet-side and topside via fiber-optic cables that can support a large number of sensors. Since this design presents a single point of failure, future designs might involve redundant channels implemented perhaps by acoustic

modems to transmit safety-related signals.

The wet-side portion of the safety and MCM systems will be packaged as a self-contained unit approximating one cubic foot in volume. The topside portion will include a user panel that indicates the safety status through a simple LED-type display and contains a switch for emergency shutdown. Initially housed on the pontoon, this topside portion will be migrated to an on-shore control center, with the topside component remaining unmanned during routine operation. The topside portion will initially be centralized to a single microprocessor, collecting all the information onto a single drive and displaying the health diagnostic and prognostic results using a single display. Future generations will add in spatial redundancy for these computing and networking resources.

8.3 Reliability Concerns

This section discusses several classes of concerns, some of which are highly complex and interrelated. Some aspects of these concerns can be addressed by MCM, while others require advances in materials science, while still others remain inherent to this application. MCM applications can address seasonally changing water currents, surface conditions that range from calm to rough, turbidity due to biological activity and suspended debris. Advances in materials science can retard fouling due to suspended biota and corrosion due to salinity. Concerns that cannot be addressed include distance from medical facilities in the event of an at-sea mishap, travel time

required to the site for maintenance, and retrieval cost of each turbine for servicing. Along the margin, retrieval cost may be reduced with advances in anti-fouling technologies.

Table 8.2 lists broad classes of reliability concerns from specific to general, the top ranked of which are specific to ocean turbines. Somewhat less specific concerns follow and pertain to ocean systems in general. Finally, the most general concerns should pertain to any human-made system. Although some of these concerns are shared with wind turbines, hydroelectric plants, and ship hulls and propellers, the top concerns predominate for ocean turbines. In the following paragraphs, we discuss each class of concerns, starting with the most chronic concern of fouling.

Fouling presents the single biggest most unique and persistent challenge to annualized reliability of ocean turbines. Fouling of sensors reduce effectiveness of MCM applications, while triggering flocculation. Unstable biofilms during early stages of fouling exert unstable loads on the propeller, accelerating wear. Turbid waters can contain human-made debris, gelatinous species like jellyfish, and sessile organisms like barnacles and whelks. For biofouling, factors in addition to depth include water temperature and time of year, which also influences velocity of water flow. COET will be assessing these rates and types of fouling at their deployment location, with the development of a turbidity model left for future work.

The presence of human-made structures (i.e., ocean turbines) and suspended debris (i.e., plastic bags) provide an ecological niche that attracts these and other

fouling organisms. Bacteria, diatoms, spores, and other single celled organisms initially settle on the surface of a structure forming a slime layer that sets the stage for flocculation. Left unchecked, this process attracts marine life at successively higher levels on the food web, eventually including turtles and cetacean mammals. Flocculation both accelerates turbine failure and may ultimately expose threatened species to increased concentrations of floating plastic flocs. Additionally, hazards associated with machinery rotating at less than one revolution per second cannot be overlooked. Assessments of these environmental impacts is left for future work.

Salinity can corrode all human-made structures with network cabling presenting a single point of failure. Corrosion of network cabling disrupts identification of machine status. Backup/redundant communication channels using acoustic signaling has low bandwidth and high propagation delays [5], which confines their usefulness to short safety-critical status/shutdown signals. This communication problem for ocean turbines is a subproblem of that for autonomous underwater vehicles (AUV). As such, ocean turbines can assume three things that AUV's can't: (i) centralized topside master, (ii) tethering that simplifies routing and acoustic relay of data, and (iii) ability to detect degree of biofouling of network cabling by percussion of the tether to measure degree of dampened oscillation.

Turbulence places stress on propellers and connecting structures and is inversely proportional to depth of submersion. Starting with a pilot deployment in 2007, with more extensive profiling since March of 2009, COET had submerged acoustic

Doppler current profilers (ADCP) at the deployment site to measure currents, turbulence, and momentum flux at a variety of depths. Empirical analysis established that both turbulence and momentum flux decrease with submersion depth [39]. Increased depth, however, increases bathymetric pressure. To minimize fabrication cost and pressure on seals, the initial deployment depth for the 20-kW prototype will be ten meters.

Based on the previous discussion, we wish to maximize turbine efficiency while minimizing costs related to depth of submersion. To minimize costs associated with turbidity, fouling, and turbulence suggests greater submersion depths. To minimize retrieval cost and bathymetric pressure suggests just the opposite. Maximizing momentum flux requires operation closer to the ocean's surface. Advances in anti-fouling technology pushes the optimum depth closer to the ocean's surface. MCM technology will do likewise if it can both adjust orientation to turbulence and safely halt with sudden incidence of waste fouling (i.e., plastic bags, monofilament line, or crude oil). Development of a depth optimization model is reserved for future work.

The remaining reliability concerns are not specific to ocean turbines, nor even rotating machinery. Since ocean turbines are a relatively new engineering application, COET anticipates creating more generations of turbine prototypes. Each generation will not only produce more electricity, but will also demand a simpler design. What constitutes simplicity is not always clear. For example, the first generation 20-kW

turbine uses a fixed pitch propeller, which requires an electric motor to initiate rotation to its operating speed of between 40 and 55 revolutions per minute. A variable pitch propeller will not require such a motor. However, varying propeller pitch could require more control logic, while exposing more moving parts to the harsh and turbid ocean environment. Each generation turbine would require modifications to the MCM, with some revision of equipment health indicators and failure modes.

8.4 Related Work

To date, few scientific papers concern reliable ocean turbine design. This is surprising considering the ocean’s immense potential as a source for base power generation. Reliability issues that set ocean turbines apart from other power generation systems, stem from their harsh yet fragile ocean environment. Here we survey related literatures concerning its potential, its environment, MCM, and wind turbines.

8.4.1 Potential

The potential for generating energy from ocean currents, including benefits and promising technologies were discussed in [94]. As early as the late 1970’s the Coriolus Program [83] proposed construction of an array of large ducted catenary turbines moored about 30 km east of Miami. They provided early estimates of power available and listed environmental issues that needed to be addressed. They cited earlier studies that concluded, based on simulating the hydroelastic behavior of submerged components, that rotors will be free of adverse vibrations. Neither an ocean-deployed

prototype nor any use of an MCM system was subsequently reported for that project. From this we surmise that reliability problems due to impact from the environment may have been overlooked.

8.4.2 Environment

Fouling of submerged components by gelatinous zooplankton and sessile organisms has been widely studied in conjunction with ship's propellers and hulls. For ocean turbines, formation of unstable biofilms on moving members will induce imbalance that reduces efficiency and accelerates wear on rotor bearings. Fouling due to the increasing prevalence of human-made debris, mostly plastics, was assessed in one meta-study for the Caribbean Basin [37]. The following paragraphs focus on the impact of biofouling on submerged sensors and structures.

Sensors need to be the last of the submerged components to fail, since they report failures to the MCM on all other types of components. In the UCSD Spray Project ¹⁹, sensors mounted on their AUV stopped functioning after three to four weeks of deployment in the highly productive waters of the Monterey Canyon [116]. Although turbidity of Gulf Stream waters is substantially lower, achieving the reliability goals of a one-year trouble-free deployment with a minimum of an 85 percent availability ²⁰ requires a variety of on-board mitigation techniques. One such technique is to induce mechanical surface vibrations using piezo-polymer transducers to

¹⁹ <http://spray.ucsd.edu/>

²⁰ This availability is required for the electricity generated to qualify as *base load*, which typically fetches a higher price per kilowatt hour.

prevent the adhesion of fouling species on immersed structures, particularly glass components like sensors and lenses [80].

Submerged structures, particularly rotating members, should be as free as possible from biofouling. Due to high cost per surface unit of such piezo-polymers, coatings have traditionally been considered for rotating and structural members. A variety of coatings have been developed to retard fouling, typified by use of toxic metals like tin, copper, and zinc. One mitigation strategy employs less toxic alternatives like methyl caproate [104]. Still another uses biomimetically designed phosphorylcholine-based polymers as a substrate [81]. A protein-rich coating reported in [82] depletes the oxygen within 0.2mm of the surface. Since it is effective in still waters, it would be more appropriate for docked watercraft than for ocean turbines. Other mitigation strategies involve superhydrophobic surfaces [54] and *engineered topographies* [26, 45, 88, 115, 130].

Examining the impact of feature size, geometry and roughness of engineered surface topographies, [115] identified factors that influence the rate of bioadhesion. These factors include surface chemistry, topography, and bulk properties of the substrate. The authors devised non-toxic antifouling techniques that involve manipulating the surface topography of a polymer surface by embossing it with periodic patterns. Feature sizes of these patterns approximated 1 to 2 μm , corresponding to that of common zoospores and marine bacteria. Using polydimethylsiloxane (PDMS) elastomer, or silicone, they fabricated a variety of textured surfaces, each with its own

Engineered Roughness Index r_E . As a dimensionless ratio, the r_E in Equation 8.1 is a function of rugosity r (i.e., Wenzel's Roughness Factor), depressed surface area fraction ϕ , and degrees of freedom ν of cell movement.

$$r_E = (r * \nu) / \phi \quad (8.1)$$

As the ratio of actual surface area to projected planar surface area, r alone did not sufficiently explain affinity of cells to walls of some types of textured surfaces. Hence, [115] extends rugosity r with parameters ϕ and ν . As the ratio of recessed area to projected area, a decrease in ϕ increases r_E , assuming r remains constant. This has the effect of reducing period λ_{x0} of Figure 8.3(a) to λ_x in Figure 8.3(b). From a biological perspective, smaller values of ϕ discourage cell attachment, since recessed areas need to be narrower (Δ_x) than cell diameter and deeper (Δ_z) than the cell's elongated actin cytoskeleton (i.e. foot). From the perspective of *macro-scale* fluid dynamics, such channels maximize hydraulic radius, and hence efficiency of drainage. Effects at the micro- and nano-scales warrant future examination.

Thus far, ϕ was defined for surfaces having exactly two elevations or values of z – an artifact of their fabrication technique. Generalizing ϕ to more than two elevations, [33] introduces the notion of *topographic aspect ratio*, Δ_z/Δ_x which they interpreted according to the theory of contact guidance in microbiology. They found that aspect ratios as low as 5 percent caused cells to align to a grooved topography.

Degrees of freedom ν refers to the number of possible paths along a surface a

cell can take at any given time. For example, patterns involving a series of parallel grooves drained by gutters orthogonal to these grooves have one degree of freedom when subject to unidirectional water flow. By contrast, grooved patterns drained by a diagonal rectangular lattice of gutters have two degrees of freedom. Increasing the number of possible directions of drainage, decreases the likelihood of a cell adhering to the wall of an engineered surface. This diagonal lattice pattern in Figure 8.3(c) can be seen at the $10\mu\text{m}$ scale with the placoid skin of sharks. This figure depicts features on two scales – finer grained grooves with period $\lambda_x \approx 2\mu\text{m}$, and coarser grained diagonal gutters with period $\lambda_y \approx 10\mu\text{m}$. The coarser gutter width Δ_{xy} needs to be much larger than the finer groove width Δ_x , without somehow being too large. In [115], these relative sizes were empirically derived from observation of the placoid scales of sharks and incorporated into the design of the Sharklet²¹ anti-fouling surface.

Culturing the *cobetia marina* bacterium in an artificial seawater medium containing the *Ulva linza* alga, [115] measured rate of colonization for each type of surface using the smooth surface as the control. They established a correlation between r_E and colonization rate for five candidate surfaces. The surface that minimized colonization rate was Sharklet. Contrary to the intuition that smooth surfaces discourage bio-adhesion, the smooth sample exhibited the fastest rate of fouling. Nonetheless, pitted surfaces with $\nu = 0$, typifying basalt rock, exhibit even faster rates of fouling.

²¹ Sharklet is a Registered Trademark of the University of Florida Research Foundation.

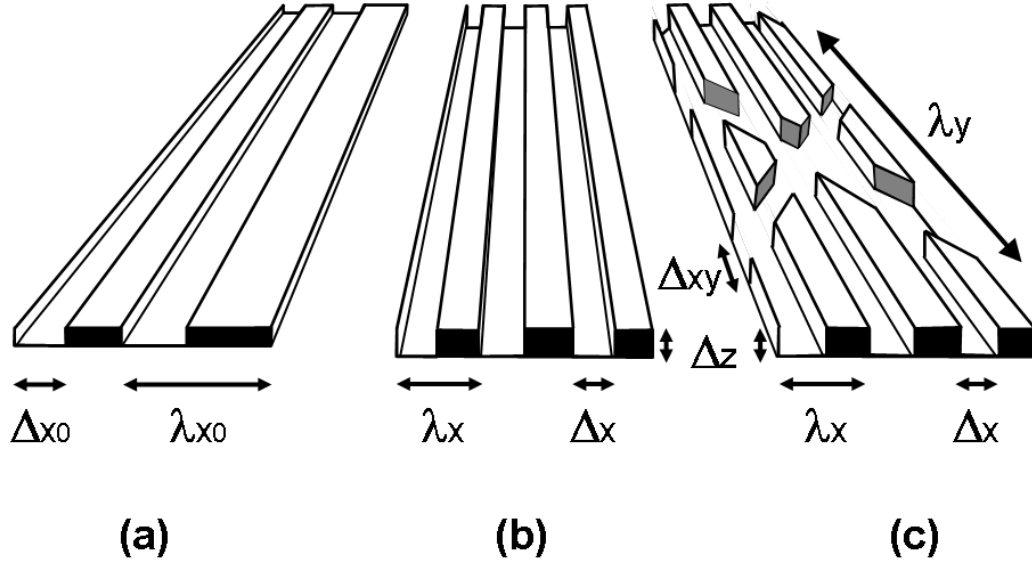


Figure 8.3: Surfaces ranked by r_E : (a) shallow and wide, (b) deep and narrow, (c) lattice drained

A word of caution is needed with regard to biomimetic surfaces. Such surfaces have been optimized on the survival behaviors of the species being mimicked. With sharks, these surfaces needed to remain pliant to support all six degrees of freedom associated with underwater navigation. Propellers or other submerged structures do not need these high degrees of pliancy. In general, designs having fewer constraints will be easier to design and fabricate. Relaxing this particular constraint will require further optimization of widths Δ_x and Δ_{xy} , which we leave for future work.

In [88], the authors extended the assay protocol in [115] by fabricating and evaluating a *library* of surfaces, each having a different feature size or scale λ . In addition to [33] which introduced the notion of topographic aspect ratio, another study positively related surface elasticity to integrin protein mediated cell adhesion [27]. This suggests fabrication of surfaces having a low modulus of elasticity, consequently requiring the relaxation of the biomimetic constraint of surface pliancy.

Although [88] and [33] studied *promotion* of bio-adhesion in a bio-medical context, their investigation into the influence of feature scale on bio-adhesion provided a conceptual bridge to surfaces in which any given surface can have multiple levels of nesting. These nested or *self-similar* surfaces were fabricated as Hierarchically Wrinkled Topographies (HWT) in [45]. This involved stretching a PDMS elastomer outward by about thirty percent, and then performing five iterations of hardening and relaxation. The first iteration $i = 1$ produced wrinkles having periods $\lambda_1 \approx 50\text{nm}$. Each subsequent iteration, produced wrinkles having $\lambda_i \approx 10\lambda_{i-1}$. After the fifth and final iteration, $\lambda_5 \approx 500\mu\text{m}$.

The design outperformed smooth surfaces in ocean emersion tests lasting from fifteen weeks to sixteen months. Three reliability problems were identified in decreasing order of severity, including surface cracking, cell agglomeration, and delamination from the substrate. Due to the material's positive Poisson ratio, cracks developed during each release phase of fabrication. These faults did not become evident until

the presence of diatoms (silica-rich organisms) were observed after 15 weeks of submersion. Secondly, agglomeration of *Ulva* spores into grooves approximating their size was observed in accelerated testing in a nutrient-rich artificial seawater medium. This agglomeration was also consistently observed on samples after eighteen months of ocean emersion. Since portions of wrinkled surfaces had no gutters, degrees of freedom ν and hence r_E would have locally vanished to zero, while other portions drained into either one or two higher-scaled channels. Precise determination of ϕ for this surface type would have required a microscopic examination followed by a statistical analysis of drainage patterns. Since ν and ϕ were defined in [115] for surfaces at only two discrete elevations, they could not be directly applied to experiments involving the more *analog* type surfaces in [45]. Section 8.5 further refines r_E and applies it to a proposed fabrication process for multi-scale self-similar surfaces that may overcome cracking and agglomeration. The problem of delamination can be a topic for future study.

8.4.3 Machine Condition Monitoring

Sensor technologies and inexpensive microprocessors birthed a vast literature on MCM. ISO Standard 13374 promulgates a six-layer architecture spanning Data Acquisition up through the Advisory Generation layers. Our brief survey follows this architecture, focusing on three aspects: (i) physical layer issues involving vibration, (ii) interface layer issues like data or sensor fusion, and (iii) logical layer issues

involving remote monitoring and control of turbine farms.

On the physical layer, ISO Standard 13373 promulgates use of well-accepted practices for acquiring and evaluating vibration measurements over extended periods of time, emphasizing changes in vibration behavior rather than any particular behavior taken in isolation. This standard recommends procedures for processing and presenting vibration data and analyzing vibration signatures, particularly for rotating machinery [35, 36]. Expert systems (i.e., [44, 147]) have been proposed for vibration analysis for fault detection, where [147] applies adaptive order-tracking techniques to rotating machinery.

Middle or interface layer issues like data or sensor fusion has been studied for rotating machinery using machine learning techniques. Collecting signals that may indicate rotational imbalance vibration from an array of sensors, [84] extracts characteristic features of each vibration signal using an auto-regressive model, then implements data fusion with a cascade-correlation neural network.

For fault diagnosis of induction motors, signals emanating from multiple sensors are preprocessed and then put through a discrete wavelet transform for decomposition into different frequency ranges of products, followed by a feature extraction step. Finally, an ensemble of two decision-level fusion strategies are employed, including a form of Bayesian belief fusion and a fusion technique involving multiple agents. In this machine learning technique, fault features are classified using several classifiers with generated decisions in turn fused using a specific fusion algorithm [99].

Field balancing of rotors was addressed in [85] to reduce turbine vibration in power plants. Using a unidirectional sensor mounted on one bearing section does not capture complex spatial motions. Instead, the authors propose a field balancing technique involving multiple sensors situated at various bearing sections along with a data fusion technique. They applied a holospectral principle and a genetic algorithm to simulate and minimize rotor vibration, empirically validating results by field balancing several 300 MW turbo-generator units.

Remote monitoring and control of equipment health, specifically the role of information and communication technologies, was surveyed in [22]. The authors identified emergent roles of web and agent technologies for remote MCM and control, and traced their origins to efforts in distributed artificial intelligence. Their survey was organized in terms of the OSA-CBM (Open System Architecture Condition-Based Maintenance) framework²², an implementation of ISO 13374. They conclude that only limited consistent and systematic efforts have been made, in an isolated manner, to apply web service and agent techniques to MCM.

A case study in remote monitoring and diagnosis of an electro-mechanical system was presented in [110]. Of interest, is their development of a virtual (software) instrument using LabView. This may provide an approach to generating output from the Data Manipulation Layer up to the higher levels of the OSA-CBM (ISO 13374) framework. Messages from the top layers, particularly the Advisory Generation layer,

²² <http://www.mimosa.org/downloads/43/specifications/index.aspx>

can be implemented as a remote monitoring and control center using web services standards. Unfortunately, such implementations may involve many middleware layers that will impede timeliness, posing steep downside risks for safety-critical systems. Addressing this problem, [106] suggests an architecture for predictable and interactive control, with a case study involving remote laboratory experiments.

8.4.4 Wind Turbines

Sharing many commonalities, experience from the wind energy sector can be applied to ocean turbines. IEC Standard 61400-25 stipulates how wind power plants should be integrated into the power grid to assure power system stability. A paper describes how this standard seeks to promulgate vendor-neutral messaging protocols [102].

Other standards like IEC 61850 and IEC 61499 specify automation of distributed power systems. Based on these standards, [63] proposes a means of combining functionality of IEC 61850-compliant devices with IEC 61499-compliant "glue logic" via the communication services of IEC 61850-7-2. The result is the ability to customize control automation logic, particularly important for developing power generation systems. IEC 61850-compliant devices are abstractions of system components. On its bottom-layer, each (virtual) device corresponds to the set of all sensors responsible for a given portion of the turbine. For rotating machinery, a bearing assembly may be represented as several virtual devices. Each such device can be thought of as

being located along the circumference of the assembly. The actual physical sensors, however, may be located near the bearing assembly in such a way as to minimize noise. The state of each virtual device becomes the collection of fused calibrated and de-noised signals emanating from its set of physical sensors.

Like wind turbines, ocean turbines may require optimizing orientation to maximize laminar flow over the surface of the propeller, reducing vibration and improving reliability. One proposal [111] describes a smart sensor that is insensitive to turbulent air flux, enabling measurement of incident wind direction and energetic transformation efficiency. This recently patented sensor extracts suitable information from the structural deformation of rotating members, where deformation is a function of incident wind direction, velocity and vibration modes. Adapting this sensor to submerged environments will pose additional technological challenges, which we leave for future work.

8.5 Anti-fouling Topographies

The Gulf current is always moving, placing kinetic energy on turbine propellers. Ocean turbulence will require the nacelle to orient itself via all six degrees of freedom. To fully exploit the momentum flux of ocean current and to assure equipment accessibility, turbines will need to operate close to the surface in the more turbid portions of the water column [39]. Turbid waters, however, accelerate biofouling. This section describes the role of surface topographies in mitigating the effects of biofouling,

providing refinements to biologically-inspired *biomimetic* solutions.

Sharks are always moving, efficiently exercising all six degrees of freedom, while remaining completely free of the biofilms commonly found on tortoises and some cetaceans. Taking inspiration from nature, precision biomimetic topographies like Sharklet have been found to be effective against organisms in the 1 to $10\mu\text{m}$ range – a range occupied by many common species of bacterium and algae spores. On the other hand, HWT is effective against biofouling at the broader 50nm to $500\mu\text{m}$ range, with impressive performance in ocean emersion tests. Both end products have merit and, as we will see, their properties are compatible.

An engineered roughness index r_E was derived in [115] to explain the effectiveness of certain biomimetically designed surfaces. In addition to rugosity r , it also factored in the number of possible drainage paths ν , and depressed area fraction ϕ . Although ϕ hints at the need for features smaller than cell diameters, [115] considered surfaces having only two scales. In that study, the most effective surface, Sharklet, had the highest r_E followed by grooved surfaces.

HWT having a *self-similar* topography performed surprisingly well against a wide range of fouling species of wide-ranging sizes despite a varying ν , difficult to measure ϕ , and fabrication problems [45]. These surfaces exhibited cracking due to deformation of the silicone substrate, consequently creating a habitat for silica-rich organisms. The deformation required to produce these topographies cannot be precisely generated. Consequently, cells tended to agglomerate wherever wrinkles

abruptly ended in *cul-de-sacs*. Furthermore, this wrinkling cannot be precisely reproduced, and hence cannot be improved upon. Nevertheless, these results highlight the importance of self-similar topographies in counteracting biofouling.

8.5.1 Self-similar Surfaces

We apply the notion of *engineered roughness* r_E to corrugated self-similar surfaces, proposing the use of one such surface known as the *Koch Curve* of Figure 8.4. We chose this curve for its known properties and consistent structure – one that maximizes r_E . Please be aware that fractal surfaces do not automatically confer anti-fouling properties. To the contrary, surfaces like those found on basalt rock – a material commonly used in the construction of ocean jetties – actually *facilitate* bioadhesion. Pits on such surfaces cause ν to locally vanish to zero, providing ecological niches for successive species of fouling organisms.

Figure 8.4 depicts how the Koch curve is generated, along with the size of representative fouling organisms. For this curve, rugosity tends toward infinity as cross-sectional feature size approaches zero. Feature sizes coming closest to those reported in [45], range from 75nm to 500 μ m. Achieving this will require $i = 8$ generations resulting in $r_8 \approx 10$. At each generation i , the curve has $n_i = 4^i$ features each of size $\lambda_i = 3^{-i}$ units. Since $\lambda_0 = 500\mu$ m, $\lambda_8 = 75$ nm with $n_8 = 65,536$ features and $r_8 = (4/3)^8 \approx 10$. Without surface defects or further modifications, this corrugated surface will have one degree of freedom, $\nu = 1$ – the same as for the

grooved topography reported in [115]. Depressed area fraction ϕ can be deduced by inspection from the unit generator in the upper left corner of Figure 8.4. By similar triangles, *elevated* area fraction is $1/6$, making *depressed* area fraction $\phi = 5/6$. Since the same generator is used at successively smaller scales, their depressed area fractions will also be the same. Derivation and usefulness of the notion of a *composite* depressed area fraction is reserved for future work.

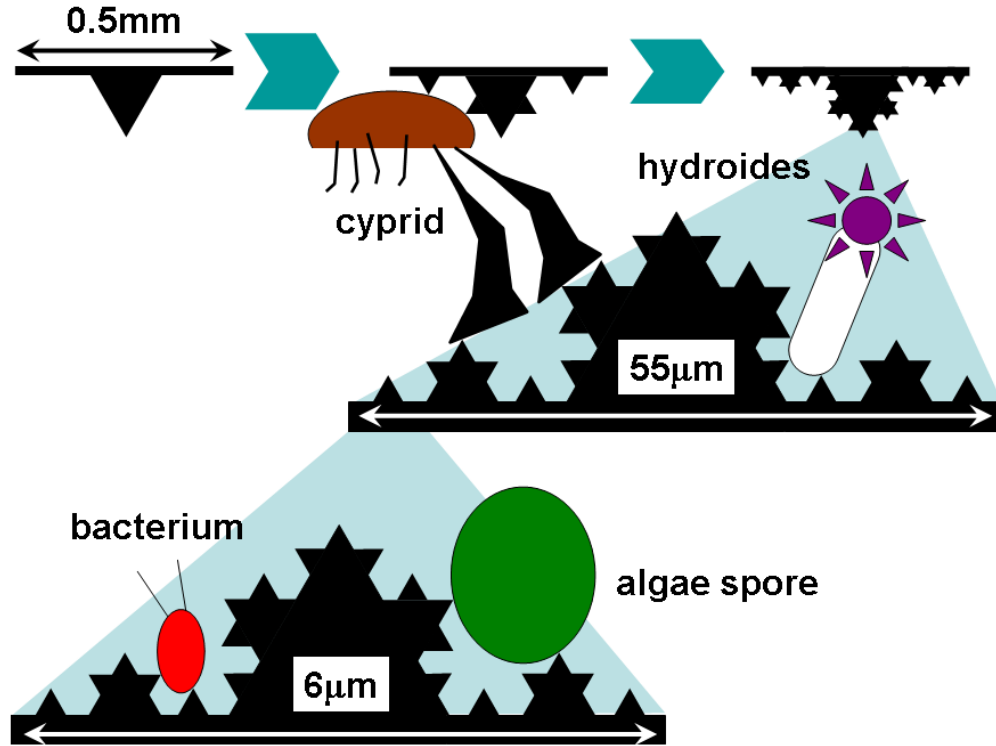


Figure 8.4: Generation of Koch curve

Table 8.3 shows roughness parameters by surface type, with the row labeled *Koch 60°* pertaining to an *ideal* fractal surface that spans the range of scales reported

in [45]. Such an ideal surface may not be possible to fabricate to eight generations with present technology. Later paragraphs provide initial ideas as to how such a surface might be fabricated. Nonetheless, if r_E were an accurate predictor, then embossing the Koch surface with a diagonal lattice at the $6\mu\text{m}$ scale should double its effectiveness, possibly requiring fewer generations. Embossing a lattice at each scale may also require fewer generations. However, effectiveness of the wrinkled topography may stem from the very fact that its finest wrinkles are at the nanoscale, halting the attachment of nanoscale bacteria and preventing biological succession. Evaluating this nanoscale fabrication requirement will involve future statistical estimation of ν and ϕ based on microscopic examination of the wrinkled surfaces in [45].

Surfaces based on the standard Koch curve will be difficult to fabricate due to its angle of incidence $\theta = 60^\circ$. After just the second generation, Figure 8.4 shows surfaces that are hidden along the (vertical) z-axis. After eight generations, access to such surfaces would require as much as a 480° curl, precluding ejection of molded elastomer along the z-axis. A more feasible approach to fabricating these corrugated surfaces involve smaller angles of incidence. An eight-generation Koch curve having no hidden surfaces will require $\theta < 90^\circ/8$. This constrained Koch curve has the same number of features $n = 65,536$, but rather than the generator being of length $l = 4/3$, it is $l' = 2\sqrt{(1/2)^2 + (\sin(\theta)/2)^2}$. Thus, for a $\theta = 11^\circ$, $l' = 3.0182/3$, making $r \approx 1.05$ after eight generations. Since there are no hidden surfaces, the constrained Koch surface may also be more amenable to embossing with diagonal lattices on multiple

scales. Despite the low rugosity, topographic aspect ratio $\Delta_z/\Delta_x = \sin(11^\circ)/2$ approximates 9 percent. Based on evidence reported in [33], this may be sufficient to align these organisms along various drainage paths. The row in Table 8.3 labeled *Koch* 11° predicts a *worse* performance than the smooth surface. Assessing the accuracy of this prediction is left for future work.

8.5.2 Fabrication Issues

To fabricate the Koch surface involves techniques described in [26, 130, 33, 12]. These include fine-tuning the properties of the polymer [26], tool path optimization for fractal shapes [130], and nanoscale lithography techniques [33, 12]. The following paragraphs review these techniques.

Most experiments have used the multi-purpose PDMS formulation known as Sylgard 184 from Dow Corning. The authors of [26] propose a formulation more appropriate for nanofabrication that (i) enhances photo-curing, (ii) exhibits a higher modulus of elasticity, (iii) increased physical toughness, (iv) decreased curing-induced shrinkage, and (v) a decreased thermal expansion coefficient.

Traditional tool-path generation techniques based on Euclidean geometry are unsuitable for laser scanning of multi-scaled structures like fractals. In addition to defects from both jerk and having to approximate durations of electron pulses, [130] cited the excessive time required for prototyping fractal curves. For their case study involving the Koch curve, any given scale has exactly one feature length and two

angles – a 'gradual' concave one and a 'pointy' convex one. Introducing the Radial Annular Tree data structure and implementing it for a variety of fractals and CAD systems, tool-paths optimized to each fractal were reported in [130].

The process of nanoscale lithography starts with scanning a beam of electrons across a surface coated with a *resist*, selectively removing it, and chemically etching through the exposed surface. These techniques have been intensively studied in connection with fabrication of semiconductor devices, details of which are provided in [12, 33, 77].

Fabrication involves three steps. First, design and fabricate a micro-die suitable for nanoscale lithography. The result will have a cross-section resembling the Koch Curve. Second, lithograph n 'perfectly' flat billets for n features. This large number of billets are needed, since the ratio of billet thickness to minimum (generation 8) feature size cannot exceed 2 without producing surface defects. Third, assemble the billets to approximate the corrugation of the Koch surface, avoiding misalignment that will vanish ν to zero. The result is a $500\mu\text{m}$ by $500\mu\text{m}$ stamp for poured polymer replication, with assembly of such replicates forming a tessellated surface suitable for ocean emersion testing.

For angle of incidence $\theta = 60^\circ$, deformation of the partially cured polymer during ejection from its stamp becomes a problem. Such an ejection may not be possible, considering the high rugosity and the abundance of hidden surfaces. Reducing θ reduces rugosity but makes fabrication possible. Considering the high modulus for

PDMS, some hidden surfaces may be tolerated placing $11^\circ < \theta < 60^\circ$.

Direct embossing of the diagonal lattice onto the PDMS substrate would require three dimensional microscale movement of a laser probe. An easier way may involve ablating billets with slots of width $\sqrt{2}\Delta_{xyi}$ at each scale i , with each successive billet having its slots transposed Δ_{xyi} units. Billet flatness along the x -dimension, groove alignment along the y -dimension, and cross-sectional uniformity along the z -dimension may introduce constraints that make the fabrication of such a surface infeasible. Nonetheless, the standard Koch surface may serve as an upper bound on engineered roughness, while lower values of θ may make less expensive techniques like embossing, extrusion, or calendarization more feasible.

8.6 Chapter Summary

We examined risk factors associated with deployment of a proposed fleet of turbines that harness the hydrokinetic energy of ocean currents. After describing a 20-kW turbine prototype, we identified a number of factors that impact its reliability and safety. Specific reliability concerns include fouling, salinity, and inaccessibility of equipment. We focused on the most critical source of faults, namely fouling. This focus led to our refinement of a measurement for engineered roughness to account for the effectiveness of self-similar surfaces. We then proposed an ideal anti-fouling topography based on a deterministic fractal known as the Koch curve. Finally, we outlined obstacles to its fabrication, and proposed a variant that may be easier to

fabricate.

Future generations of the physical design need to add spatial redundancy for computing and networking resources. This includes possibly redundant communication channels for safety-related signals, and the migration of the MCM application from topside onto shore. We described non-biocidal countermeasures to bio-fouling, while preserving more complex marine species by delaying flocculation. Added macro-level assessment of impacts *on* the environment, along with more specific descriptions of our MCM implementation are left for future work.

A turbidity model needs to be developed to gauge fouling rates on turbine propellers. An assay at the deployment site should gauge fouling rates as a function of depth, water temperature, time of year, and water velocity. Such a model needs to distinguish fouling due to biota from fouling due to human-made polymers.

A depth optimization model based on turbidity, fouling, turbulence, retrieval cost, bathymetric pressure, and momentum flux also needs to be developed for the deployment site. Simulation and subsequent empirical verification will be needed to gauge how optimum submersion depth is affected by advances in anti-fouling technologies.

The field of biomimetic anti-fouling surfaces presents more open engineering problems than immediate solutions. A reproducible means of fabricating HWT that admits to accurate measurement of r_E , will be needed prior to its direct comparison to Sharklet. Ocean emersion tests will then provide one way of affecting a direct

comparison between the two surfaces. Since both surfaces have exhibited significant anti-fouling properties, accelerated testing may be required. Such testing will require development of a uniform bioassay protocol to test three hypotheses. The first hypothesis asserts that HWT provides a defense against a broader range of fouling organisms. The second asserts that two-elevation surfaces like Sharklet, are sufficient in preventing succession to more complex organisms. The third hypothesis asserts that topographies are more effective than coatings that use tin, zinc, or copper.

Further refinements to and empirical validation of the Engineered Roughness Index r_E will be needed, particularly of the notions of submerged area fraction ϕ and degrees of freedom ν . This validation needs to be conducted from both biological and physical perspectives. Relaxing certain biomimetic constraints in the design of engineered topographies may result in human-made structures that in some respects outperform their natural counterparts. Identifying what constraints can be relaxed and designing accordingly is also left for future work.

MTB	
weight	2750 kg
length	5.25 m
H_2O displacement	12,730 kg
wall thickness	1 cm
wall material	steel
number of compartments	3
dry compartments reqd.	2
winged compartments	2
Identification beacon	yes
time between battery charges	7 days
OCDP	
weight	5051 kg
length	4.8 m
H_2O displacement	14,645 kg
hull type	twin
hull thickness	1 cm
hull material	steel
hull diameter	1 m
number of compartments	5
hydraulic jack	24 hp
winch capacity	2000 kg
Nacelle	
weight	to be determined
length	to be determined
H_2O displacement	to be determined
casing material	stainless steel
connections material	stainless steel
fastener material	copper alloy
maximum pitch	2°
maximum roll	30°
buoyancy	+10-20 kg
submersion depth	10 m
turbine output	20 kW
propeller diameter	3 m
drive shaft diameter	2.5 cm
gearbox step-up ratio	25:1
maximum torque	4,000 joules
vessel pressure above ambient	1 atm.
cooling	contact conduction
acoustic locator beacon	yes

Table 8.1: Physical features of turbine and moorings

- 1 fouling of propeller and other mechanical components
- 2 impact on marine life including species affected
- 3 salinity that causes corrosion of network cabling
- 4 underwater turbulence affecting turbine operation
- 5 high cost to retrieve equipment for servicing
- 6 equipment heterogeneity
- 7 presence/absence of technological maturity
- 8 eventual impact on global climate

Table 8.2: Reliability concerns

surface:	r :	ν :	ϕ :	r_E :	ref
smooth	1.00	2	1.00	2.0	[115]
grooved	2.50	1	0.50	5.0	[115]
shark	2.50	2	0.53	9.5	[115]
wrinkled	1.30	0-2	—	—	[45]
Koch 60°	9.97	1	0.83	12.0	proposed
Koch 11°	1.05	2	0.83	1.8	proposed

Table 8.3: Roughness parameters by surface type

CHAPTER 9

CONCLUSION AND FUTURE WORK

My research has been focusing on verifying the safety of distributed high-assurance systems, be they a collection of web services or a fleet of ocean machinery. We examined both mission type and technology type tradeoffs in determining whether to non-exhaustively test versus exhaustively verify. Mission type tradeoffs identify if formal and exhaustive verification is worth doing based on the mission of a system, be it for engineering, commerce, or entertainment. The safety-critical portions of an engineered system, like the portion of an oil well located kilometers beneath the ocean's surface, or the fiscally-critical portions of a financial system, like online credit card payments, all require formal and exhaustive verification.

Using a variety of software tools for implementing transition fault models including model checkers, Petri nets, and timed automata, I described a number of practical engineering methods to support the logistics underlying exhaustive verification. With these methods I developed and presented software tools that require tractable state spaces. This led me to formally describe a class logical fault models known as fault trees to provide suitably abstracted states for these transition fault

models. I then described how these techniques may be incorporated into engineering projects involving the monitoring of novel ocean machinery. In my reliability assessment of such machinery, I also identified how various marine life were able to combat persistent reliability problems associated with biofouling. This suggested finite models for biomimetic surfaces that may also impede biofouling.

In the sections that follow, I briefly state conclusions and describe future work.

9.1 Conclusions

Internal and external sets of tradeoffs for assuring the quality of high-assurance systems, are influenced by the coupling between critical and non-critical portions of a system. Considering external tradeoffs between assurance on the one hand and flexibility and performance on the other, we drew one obvious conclusion: Complete decoupling is not possible in many legacy systems, since critical and non-critical portions often share resources. The remainder of this section draws more specific conclusions concerning transition fault models and logical fault models.

9.1.1 Transition fault models

When applying this class of model to web service compositions, not only must the internals of each web service be individually verified, but verification must be performed with respect to each remaining service prior to verifying the top-level web service composition. This can be accomplished by representing the behavior of each web service by its own process algebraic expression. Verification of the composition of

web services therefore reduces to the verification of the composition of these process algebraic expressions.

The state space resulting from these expressions has long been observed to become intractably large, and a number of state space reduction strategies have been proposed elsewhere. This dissertation proposes structuring this interaction using a form of assume-guarantee reasoning known as the two-phase commit. The two-phase commit structures interactions between web services so that certain state space reduction strategies like linearization and removal of non-determinism can be made computationally feasible.

Even after application of these techniques, state spaces may still remain intractably large, so this dissertation further proposes a non-exhaustive agent-oriented testing framework appropriate for lower-assurance systems. In such a framework, the behavior of each agent is guided by its process algebraic expression, while the behavior of the society of agents is guided by the process algebraic expression that represents the two-phase commit. We conclude that the advantage of such an approach supports massive parallelism, wherein cyclic dependencies (i.e., deadlock) can be more efficiently detected than other forms of automated testing.

Not all legacy systems lack compositionality. Indeed, we conclude that the BOINC execution framework is compositional. Under BOINC, an individual worker process interacts only with the manager process that delegated work to it, without sharing resources with other workers. Hence, massively (and embarrassingly) parallel

processes can be independently spawned, executed, and completed. The Petri net for BOINC manager and workers is therefore sufficiently small for process interactions to be exhaustively verified. Furthermore, we conclude that more state-space intensive verification techniques like timed automata can be feasibly applied to assess the timeliness of such a system's behavior.

BPEL provides an execution framework for implementation of systems that are not necessarily compositional. This dissertation describes how fault transition models may be generated from BPEL artifacts. Although a number of such generators appear in the literature, this dissertation enables solution providers to stipulate what specific portions may be regarded as compositional. We conclude that newer portions cannot be assumed to be compositional, since they risk introduction of hidden dependencies. Portions deemed reliable, however, can be abstracted to models having smaller state spaces while newer, less reliable portions can be modeled in greater detail. By enabling solution providers to confine assumptions concerning synchrony, atomicity, and parallelism to specified portions of a web service composition, we conclude that engineering practices that encourage incremental changes to artifacts under test can be made more effective.

When automating model capture and presentation, we introduced notions of hierarchy layout and typing along with a prototype for translating a flat classical Petri net in the Petri Net Modeling Language to one implemented as a hierarchical colored Petri net in CPN Tools. The contributions include an algorithm for improving layout

through construction of a visibility representation for a Petri net, and partitioning that net into sub-nets of similar structure, while carrying over notions of partner links and variables in BPEL to the coloring of places and arcs in a colored Petri net.

9.1.2 Logical fault models

The bulk of this dissertation thus far concerns transition fault models for web service compositions – models that span a succession of points in time. Logical fault models, by contrast, concern exactly one point in time. Logical models may be distinguished from transition models based on notions of *state signature* and *state space*. In Petri net parlance, a state signature is a specific marking at some given time, while the state space is the set of all reachable markings for all times. By reducing the cardinality of each state signature, we can hence reduce the cardinality of the overall state space. Again in Petri net parlance, reducing the state signature is tantamount to reducing the number of tokens in a Petri net. Such reduction strategies, however, need to robustly characterize the phenomena being modeled, predicted, and evaluated.

Toward the construction of predictive models, this dissertation considers a specific class of logical fault models that can be decomposed into state signatures and amplitude signatures, each of which are stochastic. Fault trees can provide an implementation of such state/amplitude stochastic logical fault models. The goal of fault tree analysis in prognostic assessment of machine health involves predicting states

and amplitudes given present and previous states and amplitudes. These predictions may be evaluated by some fitness function that recalibrates these signatures to predict values for successive signatures. Evaluation of specific fitness functions lies outside the scope of this dissertation.

This dissertation instead sought to control level of abstraction by controlling the level of detail expressed in the underlying state space. It did so by reducing the number of variables from which the state space is generated. Using a purely set-theoretic formulation of fault trees and fault tree evaluation, we described how our formulation supports epistemological measures like well-studied *event certainty* measures and the less studied *event specificity* measures. These measures can be used to further predicate any severity measure associated with each fault signature or event.

9.1.3 Case study

Finally, we described a tangible system – a proposed fleet of ocean turbines – to which techniques in this dissertation may be applied. We conclude that monitoring the rate of fouling and mitigating it will postpone or counteract secular drifts in and masking of baseline state and amplitude signatures. Since fouling poses a challenge unique to any other type of energy production system, this dissertation examined the role of engineered topographies in postponing fouling. From that investigation, we conclude that self-similarity of topographies at the nanoscale may be more important

than initially realized.

9.2 Future Work

Future work will formally seek a nexus between transition fault models and logical fault models. In particular, real time evaluation of a fault tree given a system state, results in a marking for a fault transition model like a Petri net. Future work will compose logical fault models into transition fault models. Work currently under way broadens the notion of *single point in time* to include a fixed (and small) number of points in time. Although these points indicate a sequence of states within some fixed window, collectively treating them as a set can avoid problems of overfitting when training and evaluating fault trees. Nonetheless, there remains an abundance of work within each class of model described in the following sections.

9.2.1 Transition Fault Models

This dissertation examines only safety properties. Liveness properties like fairness and efficiency are best investigated in the context of quality of service. Timed automata could play a role in such investigations. Indeed, we may opt to implement a portion of the Advisory Generation Layer in an MCM/PHM framework for ocean turbines using timed automata.

State space reduction strategies can be a topic for future investigation. This may include deriving an automated means of separating critical portions of a system

from its non-critical portions. Tools like WofBPEL sidestep the state space generation problem. Yet, future work can provide such tools with improved graphics support.

The ocean turbine case study revealed a need to represent each of its physical components as its own state transition system. For this we may leverage our work on colored Petri nets so that each component has its own subnet. For CPN Tools, this will entail extending existing graph drawing algorithms to bipartite graphs subject to the constraint that entire transition type nodes must appear on exactly one page pair or subnet. Place type nodes, however, may be distributed across multiple subnets as *fusion places* in CPN Tools parlance. For CPN Tools, this dissertation already highlights data flows in one color and control flows in another. An obvious enhancement would be to partition data flows from control flows into their own subnets.

Assumptions that confine fault proneness to portions of a web service composition, and their incorporation into a model capture mechanism could also be investigated. Likewise, modeling exception or compensation handling can be further investigated. Although these efforts have already been done elsewhere on an all-or-nothing basis, to my knowledge, confining assumptions of fault proneness and handling to specified portions of interacting services has yet to be done. One approach may involve translation of workflow patterns supported in BPEL into some CSP-like process algebra like that implemented by PROMELA.

9.2.2 Logical Fault Models

Using the formalization of fault trees in this dissertation as a reference, a run time comparison of various fault tree evaluation techniques can be performed. such comparisons can identify implementations capable of supporting real time evaluation of fault trees for data streaming from a potentially large collection of sensors. Such implementations must efficiently identify sets of subsets, with Zero-suppressed Decision Diagrams (ZBDD) as one implementation of interest.

The fault tree evaluation rule stated in this dissertation requires development of suitable algorithms and data structures. Work is under way to do just that, along with its integration into a larger MCM/PHM framework currently under development. In addition to the run time comparison alluded to earlier, a usability comparison between an implementation of the evaluation rule and competing implementations packaged with SAT solvers and BDD packages may be needed.

9.2.3 Case study

As we incorporate the techniques described in this dissertation into the design of ocean turbines, we identified a number of topics for future work. These include: (i) baseline calculation, (ii) timing display, (iii) status display, (iv) fault trees, and (v) fouling countermeasures.

In the first area, finite models of operating behavior constituting a *coordination baseline* will provide operations personnel with expected sequences of events on

startup, shutdown, and during fault-free and fault-prone operation. This, along with the *computation baseline* comprised of healthy vibration signatures that may vary with operating conditions, occupy the State Detection Layer in an ISO-defined machine condition monitoring (MCM) hierarchy. These two classes of baselines still need to be computed.

The second area concerns a *timing display* at the Advisory Generation Layer of the MCM hierarchy. At that layer, we can develop transition fault models that visually represent state transitions driven in real time and presented to shore-side operations staff. We can use the simulator for a timed automata to concretely represent propeller motion for any given turbine unit. The simulation, however, will be driven in real time by incoming data from the topside buoy. In this way, an UPPAAL *virtual* propeller can depict propeller motion using only a fraction of the bandwidth required of real time video. This is a particularly salient advantage given the bandwidth constraints on ocean buoy-to-shore telemetry. This data can also drive generation and pinpoint alerts on radar-like radial sweep diagrams rendered for easy recognition by operations personnel. In these cases, a form of semantic compression can be achieved by exchanging only the markings of the timed automata.

The third area concerns *status display*, wherein a succession of operating states can be reflected as a succession of markings through the rendering of CPN models described herein. Hence, data on a turbine stuck in some undesired state can be presented to shore-side operations staff. Being hierarchical, CPN models support yet

another form of semantic compression in which common cause failures shared by more than one turbine will prompt display of a CPN at a higher fleet or sub-fleet layer of abstraction. Often these common cause failures are due not to any one turbine's behavior, but to operating conditions shared by more than one unit. Conversely, a semi-automated process can drill down to the subnet for a component of some faulty turbine unit. Semantic compression can be achieved by exchanging only the markings of the CPN.

The fourth area of future work concerns logical fault models like *fault trees*. Given a comparatively small collection of higher level events for a state snapshot at some given time, future work entails inducing fault transition model instances like Petri nets from successions of state snapshots. Future attempts to unify fault trees and Petri nets will lend a perspective on both classes of fault models. This unification effort will improve the usefulness of advisories generated for operations personnel of turbine farms.

The fifth and final area of future work addresses reliability, but not in terms of fault models. Instead, this area expands the definition of *finite model* to include a notion of self-similarity over a finite and countable number of scales. Waves, be they tangible and made of water, or less tangible and made of vibrations observe a self-similar structure that can be characterized by wavelet transforms. Turbulent waters that maximize momentum flux also stress machinery, yet they discourage biofilm formation known to clog moving elements. Future work will examine how biofouling

can be mitigated through use of non-toxic biomimetic surfaces, like those on mollusks and sharks. These include examining the mechanism underlying biomimetic surfaces and why these marine species are so effective at doing what the human race had failed to do since the time of the Phoenicians, namely staving off biofouling. Further examination of issues related to the nano-fabrication of biomimetic topographies will be needed, as well as participation in projects examining impacts of human-made waste on marine species, but from a Computational (e-)Science standpoint.

BIBLIOGRAPHY

- [1] M. Abadi and L. Lamport. Composing specifications. *ACM Transactions on Programming Languages and Systems*, 15(1):73–132, 1993.
- [2] R. Akkiraju, J. Farrell, J. Miller, M. Nagarajan, M.-T. Schmidt, A. Sheth, and K. Verma. Web service semantics – WSDL-S, November 2005.
<http://www.w3.org/Submission/WSDL-S/>.
- [3] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410, October 1990.
- [4] A. Alves, A. Arkin, S. Askary, C. Barreto, B. Bloch, F. Curbera, M. Ford, Y. Goland, A. Guzar, N. Kartha, C. K. Liu, R. Khalaf, D. Knig, M. Marin, V. Mehta, S. Thatte, D. van der Rijn, P. Yendluri, and A. Yiu. Web Services Business Process Execution Language Version 2.0, April 2007.
<http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html>.
- [5] E. An, P.-P. Beaujean, B. Baud, T. Carlson, A. Folleco, and T. J. Tarn. Multiple communicating autonomous underwater vehicles. In *Proceedings of the 2004 IEEE International Conference on Robotics and Automation (ICRA'04)*, pages 4461–4464. IEEE, 2004.
- [6] D. Anderson. BOINC: a system for public-resource computing and storage. In *Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing, (GRID'04)*, pages 4–10, November 8 2004.
- [7] T. Andrews, F. Cubera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana. Business process execution language for web services, BPEL4WS v1.1 specification, May 2003.
<http://download.boulder.ibm.com/ibmdl/pub/software/dw/specs/ws-bpel/ws-bpel.pdf>.

- [8] T. Assaf and J. Dugan. Diagnosis based on reliability analysis using monitors and sensors. *Reliability Engineering & System Safety*, 93(4):509 – 521, 2008.
- [9] A. Barros, M. Dumas, and A. H. ter Hofstede. Service interaction patterns. In W. van der Aalst et. al., editor, *Business Process Management*, volume 3649 of *Lecture Notes in Computer Science*, pages 302–318, Berlin/Heidelberg, Germany, 2005. Springer.
- [10] G. D. Battista, P. Eades, R. Tamassia, and I. G. Tollis. *Graph Drawing – Algorithms for the Visualization of Graphs*. Prentice Hall, Upper Saddle River, New Jersey, 1999.
- [11] P.-P. Beaujean, T. M. Khoshgoftaar, J. C. Sloan, N. Xiros, and D. Vendittis. Monitoring ocean turbines: a reliability assessment. In *Proceedings of the 15th ISSAT International Reliability and Quality in Design Conference*, pages 367–371. ISSAT, August 2009.
- [12] H. Becker and C. Gärtner. Polymer microfabrication technologies for microfluidic systems. *Journal Analytical and Bioanalytical Chemistry*, 390(1):89–111, January 2008.
- [13] G. Behrmann, A. David, and K. G. Larsen. A tutorial on UPPAAL. In M. Bernardo and F. Corradini, editors, *Formal Methods for the Design of Real-Time Systems*, volume 3185 of *Lecture Notes in Computer Science*, pages 200–236. Springer, 2004.
- [14] A. Bertolino and A. Polini. The audition framework for testing web services interoperability. In *EUROMICRO-SEAA*, pages 134–142. IEEE Computer Society, 2005.
- [15] J. Billington, S. Christensen, K. M. van Hee, E. Kindler, O. Kummer, L. Petrucci, R. Post, C. Stehno, and M. Weber. The Petri Net Markup Language: Concepts, Technology, and Tools. In van der Aalst and Best [138], pages 483–505.
- [16] R. D. Bjornson, A. H. Sherman, S. B. Weston, N. Willard, and J. Wing. TurboBLAST: a parallel implementation of BLAST built on the Turbohub. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'02)*, pages 183–190, 2002.

- [17] S. Blom, W. Fokkink, J. F. Groote, I. van Langevelde, B. Lisser, and J. van de Pol. μ CRL: A toolset for analysing algebraic specifications. In G. Berry, H. Comon, and A. Finkel, editors, *Computer Aided Verification*, volume 2102 of *Lecture Notes in Computer Science*, pages 250–254. Springer, 2001.
- [18] A. Bobbio, D. Codetta-Raiteri, M. D. Pierro, and G. Franceschinis. Efficient analysis algorithms for parametric fault trees. *Techniques, Methodologies and Tools for Performance Evaluation of Complex Systems, Workshop on*, 0:91–105, 2005.
- [19] J. M. Boyer. A new method for efficiently generating planar graph visibility representations. In P. Healy and N. S. Nikolov, editors, *Graph Drawing*, volume 3843 of *Lecture Notes in Computer Science*, pages 508–511. Springer, 2005.
- [20] A. Bucchiarone, A. Polini, P. Pelliccione, and M. Tivoli. Towards an architectural approach for the dynamic and automatic composition of software components. In *Proceedings of the ISSTA workshop on Role of software architecture for testing and analysis, (RoSATEA'06)*, pages 12–21, New York, NY, USA, 2006. ACM.
- [21] T. Bultan, J. Su, and X. Fu. Analyzing conversations of web services. *IEEE Internet Computing*, 10(1):18–25, Jan-Feb 2006.
- [22] J. Campos. Survey paper: Development in the application of ICT in condition monitoring and maintenance. *Computers in Industry*, 60(1):1–20, 2009.
- [23] G. Canfora and M. Di Penta. Testing services and service-centric systems: challenges and opportunities. *IT Professional*, 8(2):10–17, March-April 2006.
- [24] G. Canfora and M. D. Penta. SOA testing and self-checking. In *Proceedings of the International Workshop on Web Services, Modeling and Testing, (WS-MaTe'06)*, pages 3–12, 2006.
- [25] Y.-R. Chang, S. Amari, and S.-Y. Kuo. Obdd-based evaluation of reliability and importance measures for multistate systems subject to imperfect fault coverage. *Dependable and Secure Computing, IEEE Transactions on*, 2(4):336–347, Oct.-Dec. 2005.

- [26] K. M. Choi and J. A. Rogers. A photocurable poly(dimethylsiloxane) chemistry designed for soft lithographic molding and printing in the nanometer regime. *Journal of the American Chemical Society*, 125(14):4060–4061, 2003.
- [27] S.-Y. Chou, C.-M. Cheng, and P. R. LeDuc. Composite polymer systems with control of local substrate elasticity and their effect on cytoskeletal and morphological characteristics of adherent cells. *Biomaterials*, 30(18):3136 – 3142, 2009.
- [28] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web services description language (WSDL) version 1.1, March 2001.
<http://www.w3.org/TR/wsdl>.
- [29] F. R. K. Chung, F. T. Leighton, and A. L. Rosenberg. Embedding graphs in books: A layout problem with applications to VLSI design. In Y. Alavi, G. Chartrand, L. Lesniak, D. R. Lick, and C. E. Wall, editors, *Graph Theory with Applications to Algorithms and Computer Science*, pages 175–188. John Wiley & Sons, New York, USA, 1985.
- [30] E. M. Clarke, J. M. Wing, R. Alur, R. Cleaveland, D. Dill, A. Emerson, S. Garland, S. German, J. Gutttag, A. Hall, T. Henzinger, G. Holzmann, C. Jones, R. Kurshan, N. Leveson, K. McMillan, J. Moore, D. Peled, A. Pnueli, J. Rushby, N. Shankar, J. Sifakis, P. Sistla, B. Steffen, P. Wolper, J. Woodcock, and P. Zave. Formal methods: state of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, 1996.
- [31] S. Contini, G. Cojazzi, and G. Renda. On the use of non-coherent fault trees in safety and security studies. *Reliability Engineering & System Safety*, 93(12):1886 – 1895, 2008. 17th European Safety and Reliability Conference.
- [32] M. S. Coyne and J. A. Thompson. Soil texture and surface area. In *Math for Soil Scientists*, chapter 5. Thomson Delmar Learning, Clifton Park, New York, U.S.A., 2006.
- [33] A. S. Crouch, D. Miller, K. J. Luebke, and W. Hu. Correlation of anisotropic cell behaviors with topographic aspect ratio. *Biomaterials*, 30(8):1560 – 1567, 2009.

- [34] A. K. A. de Medeiros. *Genetic Process Mining*. PhD thesis, Technische Universiteit Eindhoven, November 2006. Advised by W.M.P. van der Aalst.
- [35] Dieter Hansen and Alf H. Olsson. ISO Standard 13373-1:2002 - Condition monitoring and diagnostics of machines – Vibration condition monitoring – Part 1: General procedures. International Standards Organization, February 2002.
- [36] Dieter Hansen and Alf H. Olsson. ISO Standard 13373-2:2005: Condition monitoring and diagnostics of machines – Vibration condition monitoring – Part 2: Processing, analysis and presentation of vibration data. International Standards Organization, January 2009.
- [37] J. I. do Sul and M. Costa. Marine debris review for Latin America and the wider Caribbean Region: From the 1970s until now, and where do we go from here? *Marine Pollution Bulletin*, 58(8):1087–1104, August 2007.
- [38] R. Drechsler and D. Sieling. Binary decision diagrams in theory and practice. *International Journal on Software Tools for Technology Transfer (STTT)*, 3(2):112–136, May 2001.
- [39] F. R. Driscoll, G. M. Alsenas, P. P. Beaujean, S. Ravenna, J. Raveling, E. Bussold, and C. Slezycki. A 20 kW open ocean current test turbine. In *Proceedings of the MTS/IEEE Oceans '08*, Quebec City, Quebec Canada, Sep 15-18 2008.
- [40] F. R. Driscoll, S. H. Skemp, G. M. Alsenas, C. J. Coley, and A. Leland. Florida's Center for Ocean Energy Technology. In *Proceedings of the MTS/IEEE Oceans '08*, Quebec City, Quebec Canada, Sep 15-18 2008.
- [41] J. Duhaney, T. M. Khoshgoftaar, A. Agarwal, and J. C. Sloan. Mining and storing data streams for reliability analysis. [72].
- [42] J. Duhaney, T. M. Khoshgoftaar, I. Cardei, B. Alhalabi, and J. C. Sloan. Applications of data fusion in monitoring inaccessible ocean machinery. [72].
- [43] S. Dustdar and S. Haslinger. Testing of service-oriented architectures - a practical approach. In M. Weske and P. Liggesmeyer, editors, *Object-Oriented*

and Internet-Based Technologies, volume 3263 of *Lecture Notes in Computer Science*, pages 97–109. Springer, 2004.

- [44] S. Ebersbach and Z. Peng. Expert system development for vibration analysis in machine condition monitoring. *Expert Systems with Applications*, 34(1):291–299, 2008.
- [45] K. Efimenko, J. Finlay, M. E. Callow, J. A. Callow, and J. Genzer. Development and testing of hierarchically wrinkled coatings for marine antifouling. *ACS Applied Materials & Interfaces*, 1(5):1031–1040, 2009.
- [46] A. Engels, L. M. G. Feijs, and S. Mauw. Test generation for intelligent networks using model checking. In E. Brinksma, editor, *Tools and Algorithms for Construction and Analysis of Systems*, volume 1217 of *Lecture Notes in Computer Science*, pages 384–398. Springer, 1997.
- [47] H. Foster, W. Emmerich, J. Kramer, J. Magee, D. S. Rosenblum, and S. Uchitel. Model checking service compositions under resource constraints. In I. Crnkovic and A. Bertolino, editors, *In Proceedings of the 6th ESEC/SIGSOFT Symposium on Foundations of Software Engineering*, pages 225–234. ACM, 2007.
- [48] H. Foster, S. Uchitel, J. Magee, and J. Kramer. Tool support for model-based engineering of web service compositions. *In Proceedings of the IEEE International Conference on Web Services, (ICWS’05)*, pages 95–102 vol.1, 11–15 July 2005.
- [49] H. Foster, S. Uchitel, J. Magee, and J. Kramer. LTSA-WS: a tool for model-based verification of web service compositions and choreography. In L. J. Osterweil, H. D. Rombach, and M. L. Soffa, editors, *Proceedings of the 28th International Conference on Software Engineering (ICSE’06)*, pages 771–774. ACM, 2006.
- [50] X. Fu, T. Bultan, and J. Su. WSAT: A tool for formal analysis of web services. In R. Alur and D. Peled, editors, *Computer Aided Verification*, volume 3114 of *Lecture Notes in Computer Science*, pages 510–514. Springer, 2004.

- [51] X. Fu, T. Bultan, and J. Su. Synchronizability of conversations among web services. *IEEE Transactions on Software Engineering*, 31(12):1042–1055, Dec. 2005.
- [52] M. R. Garey and D. S. Johnson. *Computers and Intractability: A guide to the Theory of NP-Completeness*. W.H. Freeman and Company, New York City, 1979.
- [53] M. R. Garey and D. S. Johnson. *Computers and Intractability: A guide to the Theory of NP-Completeness*, chapter 2.6, pages 38–44. W.H. Freeman and Company, New York City, 1979.
- [54] J. Genzer and K. Efimenko. Recent developments in superhydrophobic surfaces and their relevance to marine fouling: a review. *Biofouling: The Journal of Bioadhesion and Biofilm Research*, 22(5):339–360, 2006.
- [55] A. Gravel, X. Fu, and J. Su. An analysis tool for execution of BPEL services. In *CEC/EEE*, pages 429–432. IEEE Computer Society, 2007.
- [56] J. F. Groote and F. van Ham. Interactive visualization of large state spaces. *International Journal on Software Tools for Technology Transfer, STTT*, 8(1):77–91, 2006.
- [57] O. Grumberg and S. Katz. VeriTech: a framework for translating among model description notations. *International Journal on Software Tools for Technology Transfer*, 9(2):119–132, 2007.
- [58] F. Guerin and J. Pitt. Verification and compliance testing. In M.-P. Huget, editor, *Communication in Multiagent Systems*, volume 2650 of *Lecture Notes in Computer Science*, pages 98–112. Springer, 2003.
- [59] D. Habet, L. Paris, and C. Terrioux. A tree decomposition based approach to solve structured sat instances. In *Proceedings of the 2009 21st IEEE International Conference on Tools with Artificial Intelligence (ICTAI’09)*, pages 115–122, Washington, DC, USA, 2009. IEEE Computer Society.
- [60] X. He and H. Zhang. Nearly optimal visibility representations of plane graphs. *SIAM Journal of Discrete Mathematics*, 22(4):1364–1380, 2008.

- [61] R. Heckel and H. Voigt. Model-based development of executable business processes for web services. *Lectures on Concurrency and Petri Nets 2003*, 3098:559–584, April 2004.
- [62] M. Hekking, J. Lindemans, and E. S. Gelsema. Design and representation of multivariate patient-based reference regions for arterial pH, Pco2 and base excess values. *Clinical Biochemistry*, 25(6):581–585, December 1995.
- [63] N. Higgins, V. Vyatkin, N.-K. C. Nair, and K. Schwarz. Concept for intelligent distributed power system automation with IEC 61850 and IEC 61499. In *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics (SMC'08)*, pages 36–41, October 2008.
- [64] J. Himmelspach, M. Röhl, and A. M. Uhrmacher. Next generation modeling III - agents: simulation for testing software agents - an exploration based on james. In *Proceedings of the 35th Winter Simulation Conference, (WSC'03)*, pages 799–807. Winter Simulation Conference, 2003.
- [65] G. J. Holzmann. The logic of bugs. In *In Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering (SIGSOFT '02/FSE-10)*, pages 81–87, New York, NY, USA, 2002. ACM.
- [66] G. J. Holzmann. *The Spin Model Checker, Primer and Reference Manual*. Addison-Wesley, Reading, Massachusetts, U.S.A., 2003.
- [67] H. S. Hong, S. D. Cha, I. Lee, O. Sokolsky, and H. Ural. Data flow testing as model checking. In *International Conference on Software Engineering*, pages 232–243. IEEE Computer Society, 2003.
- [68] J. E. Hopcroft and R. E. Tarjan. Efficient planarity testing. *Journal of the ACM*, 21(4):549–568, 1974.
- [69] C.-L. Huang, C.-C. Lo, Y. Li, K.-M. Chao, J.-Y. Chung, and Y. Huang. Service discovery through multi-agent consensus. In *Proceedings of the IEEE International Workshop on Service-Oriented System Engineering, (SOSE'05)*, pages 37–44, 20-21 Oct. 2005.

- [70] H. Huang, W.-T. Tsai, R. Paul, and Y. Chen. Automated model checking and testing for composite web services. In *Proceedings of the Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, (ISORC'05)*, pages 300–307, Washington, DC, USA, 2005. IEEE Computer Society.
- [71] R. Hull and J. Su. Tools for design of composite web services. In G. Weikum, A. C. König, and S. Deßloch, editors, *SIGMOD Conference*, pages 958–961. ACM, 2004.
- [72] ISSAT. *Proceedings of the 16th ISSAT Reliability and Quality in Design Conference, Washington DC, USA*. ISSAT, August 5-7 2010.
- [73] K. Jensen, L. M. Kristensen, and L. Wells. Coloured Petri nets and CPN Tools for modelling and validation of concurrent systems. *International Journal on Software Tools for Technology Transfer (STTT)*, 9(3-4):213–254, 2007.
- [74] W. jun Li, X. jun Liang, H. mei Song, and X. cong Zhou. QoS-driven service composition modeling with extended hierarchical CPN. In *Proceedings of the First Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering, (TASE'07)*, pages 483–492, Washington, DC, USA, 2007. IEEE Computer Society.
- [75] W. S. Jung, S. H. Han, and J. Ha. A fast bdd algorithm for large coherent fault trees analysis. *Reliability Engineering & System Safety*, 83(3):369 – 374, 2004.
- [76] R. Kazhamiakin, M. Pistore, and L. Santuari. Analysis of communication models in web service compositions. In *Proceedings of the 15th international conference on World Wide Web (WWW'06)*, pages 267–276, New York, NY, USA, 2006. ACM.
- [77] R. W. Kelsall, I. W. Hamley, and M. Geoghegan. *Nanoscale Science and Technology*, chapter 1, pages 36–37. John Wiley & Sons, Ltd, Chichester, West Sussex, England, 2004.
- [78] M. Koshkina and F. van Breugel. Modelling and verifying web service orchestration by means of the concurrency workbench. *SIGSOFT Software Engineering Notes*, 29(5):1–10, 2004.

- [79] K. Larsen, F. Larsson, P. Pettersson, and W. Yi. Efficient verification of real-time systems: compact data structure and state-space reduction. In *Proceedings of the Eighteenth IEEE Symposium on Real-Time Systems*, pages 14–24, 2–5 Dec 1997.
- [80] M. Latour. Mechanical vibrations induced on elastic structures by piezopolymer transducers. *IEEE Transactions on Dielectrics and Electrical Insulation*, 5(1):40–44, Feb 1998.
- [81] A. L. Lewis. Phosphorylcholine-based polymers and their use in the prevention of biofouling. *Colloids and Surfaces B: Biointerfaces*, 18(3-4):261 – 275, 2000.
- [82] F. J. Lindgren, M. Haeffner, C. T. Ericsson, and P. R. Jonsson. Oxygen-depleted surfaces: a new antifouling technology. *Biofouling: The Journal of Bioadhesion and Biofilm Research*, 25(5):455–461, 2009.
- [83] P. Lissamen and R. Radkey. Coriolis program: A review of the status of the ocean turbine energy system. In *OCEANS*, volume 11, pages 559–565, Sep 1979.
- [84] Q. Liu and H.-P. Wang. A case study on multisensor data fusion for imbalance diagnosis of rotating machinery. *Artificial Intelligence in Engineering Design Analysis and Manufacturing*, 15(3):203–210, 2001.
- [85] S. Liu and L. Qu. A new field balancing method of rotor systems based on holospectrum and genetic algorithm. *Applied Soft Computing*, 8(1):446–455, 2008.
- [86] M. Lohmann, L. Mariani, and R. Heckel. A model-driven approach to discovery, testing and monitoring of web services. In L. Baresi and E. D. Nitto, editors, *Test and Analysis of Web Services*, pages 173–204. Springer, 2007.
- [87] N. Lohmann, P. Massuthe, C. Stahl, and D. Weinberg. Analyzing interacting WS-BPEL processes using flexible model generation. *Data & Knowledge Engineering*, 64(1):38–54, 2008.
- [88] J. Lovmand, J. Justesen, M. Foss, R. H. Lauridsen, M. Lovmand, C. Modin, F. Besenbacher, F. S. Pedersen, and M. Duch. The use of combinatorial

topographical libraries for the screening of enhanced osteogenic expression and mineralization. *Biomaterials*, 30(11):2015 – 2022, 2009.

- [89] C. K. Low, T. Y. Chen, and R. Rönquist. Automated test case generation for BDI agents. *Autonomous Agents and Multi-Agent Systems*, 2(4):311–332, 1999.
- [90] J. Magee and J. Kramer. *Concurrency: State Models & Java Programs*. Wiley, Chichester, England, 2nd edition, 2006.
- [91] D. Martin, M. Burstein, J. Hobbs, O. Lassila, D. McDermott, S. McIlraith, S. Narayanan, M. Paolucci, B. Parsia, T. Payne, E. Sirin, N. Srinivasan, and K. Sycara. OWL-S: Semantic markup for web services, November 2004. <http://www.w3.org/Submission/OWL-S>.
- [92] J. Mendling, C. P. de Laborda, and U. Zdun. Towards an integrated BPM schema: Control flow heterogeneity of PNML and BPEL4WS. In K.-D. Althoff, A. Dengel, R. Bergmann, M. Nick, and T. Roth-Berghofer, editors, *Wissensmanagement*, volume 3782 of *Lecture Notes in Computer Science*, pages 570–579. Springer, 2005.
- [93] S.-I. Minato. Zero-suppressed bdds and their applications. *International Journal on Software Tools for Technology Transfer (STTT)*, 3(2):156–170, May 2001.
- [94] Minerals Management Service – U.S. Dept. of the Interior. Ocean current energy potential on the U.S. Outer Continental Shelf. Technology White Paper, May 2006. <http://oscenergy.anl.gov>; accessed March 16, 2009.
- [95] Y. Mo. Variable ordering to improve bdd analysis of phased-mission systems with multimode failures. *Reliability, IEEE Transactions on*, 58(1):53–57, March 2009.
- [96] N. A. Mulyar and W. M. P. van der Aalst. Patterns in colored Petri nets. BETA Working Paper Series WP-139, Eindhoven University of Technology, Eindhoven, Netherlands, April 2005.

- [97] A. Myers and A. Rauzy. Efficient reliability assessment of redundant systems subject to imperfect fault coverage using binary decision diagrams. *IEEE Transactions on Reliability*, 57(2):336–348, June 2008.
- [98] S. Nakajima. Verification of web service flows with model-checking techniques. In *First International Symposium on Cyber Worlds*, pages 378–385. IEEE Computer Society, 2002.
- [99] G. Niu, A. Widodo, J.-D. Son, B.-S. Yang, D.-H. Hwang, and D.-S. Kang. Decision-level fusion based on wavelet decomposition for induction motor fault diagnosis using transient current signal. *Expert Systems with Applications*, 35(3):918–928, 2008.
- [100] C. Ouyang, E. Verbeek, W. M. P. van der Aalst, S. Breutel, M. Dumas, and A. H. M. ter Hofstede. Formal semantics and analysis of control flow in WS-BPEL. *Science of Computer Programming*, 67(2-3):162–198, 2007.
- [101] J. L. Peterson. Petri nets. *ACM Computing Surveys*, 9(3):223–252, 1977.
- [102] E. S. T. Quecedo, I. Canales, J. L. Villate, E. Robles, and S. Apiñaniz. The use of IEC 61400-25 standard to integrate wind power plants into the control of power system stability. In *Proceedings of the European Wind Energy Conference and Exhibition*, pages 1–4, May 7–10 2007.
- [103] V. Raghava. A comparison of model checking tools for service oriented architectures. Master’s thesis, Florida Atlantic University, Boca Raton, Florida USA, December 2007. Advised by T. M. Khoshgoftaar.
- [104] H. B. Rai, S. M. Jung, M. Sidharthan, J. H. Lee, C. Y. Lim, Y.-K. Kang, C. Yeon, N. S. Park, and H. W. Shin. Chemotactic antifouling properties of methyl caproate: its implication for ship hull coatings. In *Proceedings of the 6th WSEAS International Conference on Applied Informatics and Communications (AIC’06)*, pages 474–480, Stevens Point, Wisconsin, USA, 2006. World Scientific and Engineering Academy and Society (WSEAS).
- [105] P. Ramsokul and S. Ramesh. A test bed for web services protocols. In *Second International Conference on Internet and Web Applications and Services (ICIW’07)*, pages 16–22, 2007.

- [106] A. Rasche, F. Feinbube, P. Tröger, B. Rabe, and A. Polze. Predictable interactive control of experiments in a service-based remote laboratory. In *Proceedings of the 1st International Conference on Prvasive Technologies Related to Assistive Environments (PETRA'08)*, pages 1–7, New York, NY, USA, 2008. ACM.
- [107] A. V. Ratzer, L. Wells, H. M. Lassen, M. Laursen, J. F. Qvortrup, M. S. Stissing, M. Westergaard, S. Christensen, and K. Jensen. CPN Tools for editing, simulating, and analysing coloured Petri nets. In van der Aalst and Best [138], pages 450–462.
- [108] S. Rayadurgam and M. P. E. Heimdahl. Coverage based test-case generation using model checkers. In *Engineering of Computer-Based Systems*, pages 83–. IEEE Computer Society, 2001.
- [109] R. Remenyte-Prescott and J. Andrews. An enhanced component connection method for conversion of fault trees to binary decision diagrams. *Reliability Engineering & System Safety*, 93(10):1543 – 1550, 2008.
- [110] I. Resceanu, M. Niculescu, N. G. Bizdoaca, and C. Pana. Remote monitoring and diagnosis of a mechatronic system. In *Proceedings of the 8th conference on Applied informatics and communications (AIC'08)*, pages 234–239, Stevens Point, Wisconsin, USA, 2008. World Scientific and Engineering Academy and Society (WSEAS).
- [111] J. Rodriguez and C. Perez. Advanced sensor for optimal orientation and predictive maintenance of high power wind generators. In *Proceedings of the IEEE 28th Annual Conference of the Industrial Electronics Society (IECON'02)*, volume 3, pages 2167–2172, Nov. 2002.
- [112] M. Rohl, F. Marquardt, and A. M. Uhrmacher. Exploiting web service techniques for composing simulation models. In *Proceedings of the Winter Simulation Conference (WSC'07)*, pages 833–841, December 9-12 2007.
- [113] N. Russell, A. H. M. ter Hofstede, W. M. P. van der Aalst, and N. Mulyar. Workflow control-flow patterns, a revised view. Technical Report BPM Center Report BPM-06-22, Queensland University of Technology, Brisbane, QLD, Australia, January 2006.

- [114] B.-H. Schlingloff, A. Martens, and K. Schmidt. Modeling and model checking web services. *Electronic Notes Theoretical Computer Science*, 126:3–26, 2005.
- [115] J. F. Schumacher, M. L. Carman, T. G. Estes, A. W. Feinberg, L. H. Wilson, M. E. Callow, J. A. Callow, J. A. Finlay, and A. B. Brennan. Engineered antifouling microtopographies effect of feature size, geometry, and roughness on settlement of zoospores of the green alga ulva. *Biofouling: The Journal of Bioadhesion and Biofilm Research*, 23(1):55–62, 2007.
- [116] J. Sherman, R. Davis, W. Owens, and J. Valdes. The autonomous underwater glider “Spray”. *IEEE Journal of Oceanic Engineering*, 26(4):437–446, Oct 2001.
- [117] M. P. Singh and M. N. Huhns. *Service-Oriented Computing: Semantics, Processes, Agents*. John Wiley & Sons, Ltd., Hoboken, New Jersey, U.S.A., 2005.
- [118] C. Sinz. Visualizing sat instances and runs of the dpll algorithm. *Journal of Automated Reasoning*, 39(2):219–243, August 2007.
- [119] J. C. Sloan and T. M. Khoshgoftaar. Toward model checking web services over the web. In *Proceedings of the Twentieth International Software Engineering and Knowledge Engineering Conference, (SEKE’08), San Francisco, California*, pages 519–524. Knowledge Systems Institute Graduate School, July 1-3 2008.
- [120] J. C. Sloan and T. M. Khoshgoftaar. Tradeoffs in testing service oriented architectures. In *Proceedings of the 14th ISSAT International Reliability and Quality in Design Conference, Orlando, Florida*, pages 141–145. ISSAT, August 7-9 2008.
- [121] J. C. Sloan and T. M. Khoshgoftaar. From web service artifact to a readable and verifiable model. *IEEE Transactions on Services Computing*, 2(4):277–288, October 2009.
- [122] J. C. Sloan and T. M. Khoshgoftaar. Testing and formal verification of service oriented architectures. *International Journal of Reliability, Quality & Safety Engineering*, 16(2):137–162, April 2009.

- [123] J. C. Sloan, T. M. Khoshgoftaar, P.-P. Beaujean, and F. Driscoll. Ocean turbines – a reliability assessment. *International Journal of Reliability, Quality and Safety Engineering*, 16(5):413–433, 2009.
- [124] J. C. Sloan, T. M. Khoshgoftaar, and A. Folleco. Testing web services as agents. In *Proceedings of the 14th ISSAT International Reliability and Quality in Design Conference*, pages 151–155. ISSAT, August 7-9 2008.
- [125] J. C. Sloan, T. M. Khoshgoftaar, and H. Hanson. Formalizing fault trees for remote ocean systems. [72].
- [126] J. C. Sloan, T. M. Khoshgoftaar, and V. Raghav. Assuring timeliness in an e-science service-oriented architecture. *Computer*, 41(8):56–62, August 2008. IEEE Computer Society.
- [127] J. C. Sloan, T. M. Khoshgoftaar, and A. Varas. An extendible translation of bpm to a machine-verifiable model. In *Proceedings of the Twenty-first International Software Engineering and Knowledge Engineering Conference, (SEKE'09), Boston, Massachusetts*, pages 344–349. Knowledge Systems Institute Graduate School, July 1-3 2009.
- [128] O. V. Sokolsky and S. A. Smolka. Incremental model checking in the modal mu-calculus. In *Proceedings of the Sixth International Conference on Computer-Aided Verification (CAV'94), Lecture Notes in Computer Science 818*, pages 351–363, 1994.
- [129] M. Solanki, A. Cau, and H. Zedan. Introducing compositionality in web service descriptions. In *Proceedings of the Tenth IEEE International Workshop on Future Trends of Distributed Computing Systems, (FTDCS'04)*, pages 14–20, May 26-28 2004.
- [130] S. C. Soo and K. M. Yu. Tool-path generation for fractal curve making. *International Journal of Advanced Manufacturing Technology*, 19(1):32–48, 2002.
- [131] Y. Sun, S. He, and J. Y. Leu. Syndicating web services: A qos and user-driven approach. *Decision Support Systems*, 43(1):243–255, 2007.

- [132] K. Sycara, M. Paolucci, J. Soudry, and N. Srinivasan. Dynamic discovery and coordination of agent-based semantic web services. *IEEE Internet Computing*, 8(3):66–73, May-June 2004.
- [133] E. Tarjan, Robert. Depth first search and linear graph algorithms. *SIAM Journal of Computing*, 1(2):146–160, 1972.
- [134] M. Taufer, D. Anderson, P. Cicotti, and C. Brooks. Homogeneous redundancy: a technique to ensure integrity of molecular simulation results using public computing. In *Proceedings of the 19th IEEE International Symposium on Parallel and Distributed Processing*, pages 119a–119a, April 2005.
- [135] W.-T. Tsai, Y. Chen, and R. Paul. Specification-based verification and validation of web services and service-oriented operating systems. In *Proceedings of the Tenth IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, pages 139–147, 2-4 Feb. 2005.
- [136] W.-T. Tsai, X. Wei, Y. Chen, and R. Paul. A robust testing framework for verifying web services by completeness and consistency analysis. In *Proceedings of the IEEE International Workshop on Service-Oriented System Engineering, (SOSE'05)*, pages 151–158, 20-21 Oct. 2005.
- [137] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.
- [138] W. M. P. van der Aalst and E. Best, editors. In *Proceedings of the Twenty-fourth International Conference on Applications and Theory of Petri Nets, (ICATPN'03), Eindhoven, The Netherlands, June 23-27, 2003*, volume 2679 of *Lecture Notes in Computer Science*. Springer, 2003.
- [139] W. M. P. van der Aalst, J. B. Jørgensen, and K. B. Lassen. Let's go all the way: From requirements via colored workflow nets to a BPEL implementation of a new bank system. In R. Meersman, Z. Tari, M.-S. Hacid, J. Mylopoulos, B. Pernici, Ö. Babaoglu, H.-A. Jacobsen, J. P. Loyall, M. Kifer, and S. Spaccapietra, editors, *Proceedings of the OTM Conferences (1)*, volume 3760 of *Lecture Notes in Computer Science*, pages 22–39. Springer, October 31 - November 4 2005.

- [140] A. Varas. Toward push-button verification of web service compositions. Master's thesis, Florida Atlantic University, Boca Raton, Florida USA, December 2009. Advised by T. M. Khoshgoftaar.
- [141] A. Vreze, B. Vlaovic, and Z. Brezocnik. Sdl2pml – tool for automated generation of promela model from sdl specification. *Computer Standards & Interfaces*, 31(4):779 – 786, 2009.
- [142] R. R. Wald, T. M. Khoshgoftaar, P.-P. Beaujean, and J. C. Sloan. Combining wavelet and fourier transforms in reliability analysis of ocean systems. [72].
- [143] R. R. Wald, T. M. Khoshgoftaar, P.-P. Beaujean, and J. C. Sloan. A review of prognostics and health monitoring techniques for autonomous ocean systems. [72].
- [144] Y. Wang, X. Bai, J. Li, and R. Huang. Ontology-based test case generation for testing web services. In *Proceedings of the Eighth International Symposium on Autonomous Decentralized Systems, (ISADS'07)*, pages 43–50, Washington, DC, USA, 2007. IEEE Computer Society.
- [145] G. Weiß. Agent orientation in software engineering. *Knowledge Engineering Review*, 16(4):349–373, 2001.
- [146] M. Wooldridge. *An Introduction to Multiagent Systems*. John Wiley & Sons, Ltd., Hoboken, New Jersey, U.S.A., 2002.
- [147] J.-D. Wu, M. R. Bai, F.-C. Su, and C.-W. Huang. An expert system for the diagnosis of faults in rotating machinery using adaptive order-tracking algorithm. *Expert Systems with Applications*, 36(3):5424–5431, April 2009.
- [148] Q. Wu, C. Pu, and A. Sahai. DAG synchronization constraint language for business processes. In *Proceedings of the 3rd IEEE International Conference on E-Commerce Technology*, pages 10–17, 2006.
- [149] L. Xing and Y. Dai. A new decision-diagram-based method for efficient analysis on multistate systems. *IEEE Transactions on Dependable and Secure Computing*, 6(3):161–174, 2009.

- [150] Y. Yang, Q. Tan, J. Yu, and F. Liu. Transformation BPEL to CP-nets for verifying web services composition. In *Proceedings of the International Conference on Next Generation Web Services Practices, NWESP'05*, pages 137–142, Washington, DC, USA, 2005. IEEE Computer Society.
- [151] M. Younas, Y. Li, and C.-C. Lo. An efficient transaction commit protocol for composite web services. In *Proceedings of the Twentieth International Conference on Advanced Information Networking and Applications, (AINA'06)*, pages 591–596, April 18-20 2006.
- [152] Y. Zheng, J. Zhou, and P. Krause. Analysis of BPEL data dependencies. In *EUROMICRO-SEAA*, pages 351–358. IEEE Computer Society, 2007.
- [153] Y. Zheng, J. Zhou, and P. Krause. A model checking based test case generation framework for web services. In *Information Technology: New Generations*, pages 715–722. IEEE Computer Society, 2007.