

9-2009

# Resource Management in Complex and Dynamic Environments

Mohammad Salimullah Raunak  
*University of Massachusetts Amherst, raunak@cs.umass.edu*

Follow this and additional works at: [https://scholarworks.umass.edu/open\\_access\\_dissertations](https://scholarworks.umass.edu/open_access_dissertations)



Part of the [Computer Sciences Commons](#)

---

## Recommended Citation

Raunak, Mohammad Salimullah, "Resource Management in Complex and Dynamic Environments" (2009). *Open Access Dissertations*. 141.  
[https://scholarworks.umass.edu/open\\_access\\_dissertations/141](https://scholarworks.umass.edu/open_access_dissertations/141)

This Open Access Dissertation is brought to you for free and open access by ScholarWorks@UMass Amherst. It has been accepted for inclusion in Open Access Dissertations by an authorized administrator of ScholarWorks@UMass Amherst. For more information, please contact [scholarworks@library.umass.edu](mailto:scholarworks@library.umass.edu).

**RESOURCE MANAGEMENT IN COMPLEX AND  
DYNAMIC ENVIRONMENTS**

A Dissertation Presented

by

MOHAMMAD SALIMULLAH RAUNAK

Submitted to the Graduate School of the  
University of Massachusetts Amherst in partial fulfillment  
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

September 2009

Department of Computer Science

© Copyright by Mohammad Salimullah Raunak 2009

All Rights Reserved

# RESOURCE MANAGEMENT IN COMPLEX AND DYNAMIC ENVIRONMENTS

A Dissertation Presented

by

MOHAMMAD SALIMULLAH RAUNAK

Approved as to style and content by:

---

Leon J. Osterweil, Chair

---

Lori A. Clarke, Member

---

George S. Avrunin, Member

---

Prashant J. Shenoy, Member

---

Ian R. Grosse, Member

---

Andrew Barto, Department Chair  
Department of Computer Science

To my parents, my wife and my son

## ACKNOWLEDGMENTS

First and fore most, I would like to thank the Almighty for giving me the courage and patience to work on this challenging problem (Alhamdulillah).

I would like to sincerely thank my advisor, Leon J. Osterweil, for his continuous encouragements and guidance throughout my graduate study and the dissertation research. I have found a great mentor in him for both this work and for academic life in general.

I would like to thank my other committee members for their guidance. Lori Clarke provided thoughtful feedback on the architecture. George Avruning helped me closely with the notations and formalism of the concepts. Both Prashant Shenoy and Ian Grosse have provided me with useful suggestions for the research.

I would like to thank all the members of Laboratory for Advanced Software Engineering Research (LASER) during my time. I would like to particularly thank Sandy Wise for many thought provoking discussions. He has been instrumental in many detail part of the work.

My colleagues and friends at the lab have been very supportive all along. I would like to thank Guillaume Viguiere, Bobby Simidchieva, Matt Marzilli, and Tiffany Chao for helping me with the experiments. I would also like to thank Heather Conboy, Stefan Christov, and Junchao Xiao for reading and providing feedback on different sections of the draft. I would like to thank Bin Chen, Jianbin Tan, Zongfang Lin, and Amr Elsamadisy for their friendly words and encouragements.

I would like to especially thank my domain expert, Phil Henneman, for his many hours of dedicated time toward this work. His input has shaped the research in many

ways. I would also like to thank Hari Balasubramanian for lending his expertise on simulation and guiding the work.

And last, but not least, I thank my family for their unconditional support. My parents, Mohammad Abdus Subhan and Khaleda Akhter, have encouraged me all my life to pursue my dreams. My wife, Anjuman, was the person who provided me with consistent motivation and sacrificed the most during my graduate student life. And finally, there is Raeid, my son, a bundle of joy and a source of unlimited inspiration. Thank you.

## ABSTRACT

# RESOURCE MANAGEMENT IN COMPLEX AND DYNAMIC ENVIRONMENTS

SEPTEMBER 2009

MOHAMMAD SALIMULLAH RAUNAK

B.S., NORTH SOUTH UNIVERSITY

M.S., UNIVERSITY OF MASSACHUSETTS AMHERST

Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Leon J. Osterweil

Resource management is at the heart of many diverse science and engineering research areas. Although the general notion of what constitutes a resource entity seems similar in different research areas, their types, characteristics, and constraints governing their behavior are vastly different depending on the particular domain of research and the nature of the research itself. Often research related to resource modeling and management focus on largely homogeneous resources in a relatively simplified model of the real world. The problem becomes much more challenging to deal with when working with a complex real life domain with many heterogeneous resource types and intricate constraints. In this dissertation, we have looked at the modeling and management of resource instances and tried to develop a better sense of what makes them different from other objects in a system. As part of this work, We formally define the general resource management problem, identify its major sub



problem areas and their associated complexities, and look at the problem in the context of a particularly complex and dynamic environment, namely the emergency department (ED) of a hospital. We propose an approach to the problem and some of its complexities by presenting an overall unifying view, as well as tools and methods for dealing with, this pervasive, yet surprisingly under examined, type of entity, i.e. resources.

We have discovered that one of the discerning characteristics of resource instances in complex and dynamic environments seem to be their dynamic capability profile that may changes depending on system context. This, in turn, often results in complex substitutability relationship amongst resource instances.

We have identified four major sub-problem areas that can provide a holistic view of any resource management service. These separate, yet interconnected, areas of concerns include resource modeling, resource request specification, resource constraint management, and resource allocation. Resource modeling involves capturing of resource characteristics and their potentially dynamic behavior. Request definitions describe how resource users specify requirements for resources in a particular domain. In most domains, there are constraints that need to be satisfied while serving resources to fulfill specific requests. The fourth area of concerns, the allocation of resources, is a complex component with multiple subcomponents that closely interact with each other. In this thesis, we have described an architecture for a flexible resource management service based on the above described separation of concerns. We have proposed some simple, yet effective, techniques for modeling resource instances, specifying resource requests, specifying and managing resource constraints, and allocating resource instances to meet a resource demand characterized by a continuous stream of requests. Using our proposed design, we have developed ROMEO, a resource management service and customized it to serve a task coordination framework based on Little-JIL process definition language. Our work then concentrated

on evaluating the effectiveness of ROMEO in supporting simulations and executions of complex processes. For this evaluation purpose, we developed a simulation infrastructure named JSim on top of Juliette, Little-JIL’s execution environment. We ran a variety of simulations of patient care processes in EDs using our ROMEO-JSim infrastructure. We also used ROMEO to support the actual execution (rather than just the simulation) of a large mediation process.

A central premise, hypothesized and explored in this thesis, is a novel way of thinking about resource instances in dynamic domains, namely defining them with a set of *guarded capabilities*, some of which may be dependent on the execution state of the system. This has led us to think about how to represent execution states of a running system and what types of system state information might be important for representing the *guard functions* on the *capabilities* of a resource instance that define the resource instance’s ability to satisfy a request at a given execution state of the system. We have also identified a small set of common types of attributes of resource instances that seem able to support specification of a large variety of resource instances in complex domains. We believe that our research supports our hypothesis that specifying resource instances as having sets of *guarded capabilities* provides a useful abstraction for modeling many of the complex dynamic behaviors of resource instances in such domains as hospital EDs.

# TABLE OF CONTENTS

	Page
<b>ACKNOWLEDGMENTS</b> .....	<b>v</b>
<b>ABSTRACT</b> .....	<b>vii</b>
<b>LIST OF TABLES</b> .....	<b>xiii</b>
<b>LIST OF FIGURES</b> .....	<b>xiv</b>
 <b>CHAPTER</b>	
<b>1. INTRODUCTION</b> .....	<b>1</b>
1.1 Motivation .....	1
1.2 Research Contributions .....	6
<b>2. RELATED WORK</b> .....	<b>8</b>
2.1 Resource Management in Hospital Emergency Department .....	8
2.2 Resource Management in Networking and Operating Systems .....	12
2.3 Resource Management in Distributed Computing .....	13
2.4 Resource Management in Workflow and Process Languages .....	18
2.5 Resource Scheduling in Artificial Intelligence .....	22
2.6 Resource Allocation in Operations Research .....	22
2.7 Resource Management in Knowledge Based Systems .....	23
2.7.1 Related Work in Ontology .....	24
2.8 Programming language support for dynamic objects .....	28
<b>3. APPROACH</b> .....	<b>31</b>
3.1 Specification of the General Problem .....	32
3.2 Resource Model .....	37
3.3 Request Model .....	45

3.3.1	Specification of Required Characteristics .....	45
3.3.2	Specification of Required Capacity .....	46
3.3.3	Specification of the Protocol While Satisfying the Request .....	47
3.3.3.1	Blocking and Nonblocking Requests .....	47
3.3.3.2	Atomic Assignments .....	47
3.4	Constraint Model .....	48
3.4.1	Constraints as Requests .....	48
3.4.2	Constraints as Part of the Resource Model .....	50
3.5	Allocation Decision .....	52
<b>4.</b>	<b>RESOURCE MANAGER ARCHITECTURE .....</b>	<b>55</b>
4.1	Overall Architecture .....	55
4.2	Resource Manager Modules .....	61
4.2.1	Resource Model .....	62
4.2.2	Request Model .....	64
4.2.2.1	Resource Client .....	65
4.2.2.2	Specification of the Required Resource .....	66
4.2.3	Repository Management .....	68
4.2.3.1	Managing Multiple Repositories .....	73
4.2.4	Allocation Decision .....	73
4.2.5	Constraint Management .....	75
4.3	Resource Manager API to support a task coordination framework .....	79
4.3.1	Agent and Non-agent resources .....	79
4.3.2	Request Structure to Support a Task Coordination Framework .....	88
4.3.3	Support for Blocking Reservation and Acquisition .....	89
<b>5.</b>	<b>EVALUATION SETUP .....</b>	<b>93</b>
5.1	Little-JIL Process Programming Language .....	94
5.2	Modeling an ED Process using Little-JIL .....	99
5.3	Resource Request Specification in Little-JIL .....	102
5.3.1	Resource Acquisition and Resource Use .....	102
5.3.2	Request Constraints .....	104

5.3.2.1	Resource Collection Constraint . . . . .	105
5.3.2.2	Resource Iterator Constraint . . . . .	106
5.3.3	Resource Exceptions . . . . .	106
5.4	Juliette: the Little-JIL Process Execution Environment . . . . .	107
5.5	JSim: The Simulation Environment . . . . .	109
5.5.1	JSim Agent Behavior Specification (JABS) . . . . .	112
5.5.2	Simulation Outputs . . . . .	114
<b>6.</b>	<b>CASE STUDIES AND EXPERIENCES . . . . .</b>	<b>115</b>
6.1	Validating Simulation Results . . . . .	115
6.1.1	Impact of Varying a Bottleneck Resource . . . . .	116
6.1.2	Little's Law . . . . .	120
6.1.3	Comparing with a Commercial Simulation Product . . . . .	122
6.2	Capturing ED Domain Policies . . . . .	123
6.2.1	Impact of Same-Doctor Constraint . . . . .	125
6.2.2	Dynamic Substitution . . . . .	127
6.2.3	Dynamically Changing Process based on Resource Availability . . . . .	131
6.2.4	Impact of Request Priority . . . . .	136
6.3	Resource Sharing in a Multi-department ED . . . . .	138
6.4	Experiences with Processes in Other Domains . . . . .	144
<b>7.</b>	<b>CONCLUSIONS AND FUTURE WORK . . . . .</b>	<b>149</b>
7.1	Summary of the Research . . . . .	149
7.2	Future Directions for the Research . . . . .	152
7.2.1	More validation of ROMEO-JSim . . . . .	153
7.2.2	Infrastructure Improvement . . . . .	154
7.2.3	Experimenting with Intelligent Scheduling . . . . .	154
	<b>BIBLIOGRAPHY . . . . .</b>	<b>156</b>

## LIST OF TABLES

Table	Page
6.1 Optimum resource mix for different patient arrivals .....	125
6.2 Guard function defining services offered by resource instances.....	128
6.3 Elaboration of substitution condition for triange nurses .....	130
6.4 Task groups for relative priority experiment .....	136
6.5 Triangular distribution of the step execution times in VerySimpleED .....	137
6.6 Impact on LOS based on different priority combination .....	138
6.7 Resource mix for running simulations with ‘EDCare2’ process .....	143

## LIST OF FIGURES

Figure	Page
2.1 Example of Condor classads . . . . .	14
2.2 Example of Request Description . . . . .	15
2.3 Syntax of an RSL Request . . . . .	18
4.1 Resource Manager Architecture . . . . .	57
4.2 Class diagram of a simple resource instance . . . . .	62
4.3 Resource Client API . . . . .	65
4.4 Required resource specification class diagram . . . . .	67
4.5 Resource Repository Manager Class Diagram . . . . .	69
4.6 Multiple Repository Manager . . . . .	74
4.7 Resource Selector API . . . . .	74
4.8 Resource Collection Constraint Specification . . . . .	75
4.9 Specification of required resource with additional constraint . . . . .	78
4.10 Resource Manager API for a Task Coordination Framework . . . . .	81
4.11 Life cycle of a Task instance . . . . .	82
4.12 Life cycle for the usage of an agent resource instance . . . . .	83
4.13 Life cycle for the usage of a non-agent resource instance . . . . .	84
4.14 Combined state diagram of a task instance and an agent resource instance . . . . .	84

4.15	Message Sequence Chart defining the protocol between Task Instance and the Resource Manager .....	86
4.16	Resource Request Class and its subclasses .....	88
4.17	Resource Request Class and its subclasses .....	90
5.1	Little-JIL iconography .....	95
5.2	A Little-JIL definition of a very simple ED process .....	101
5.3	Resource acquisition syntax .....	103
5.4	Resource use syntax .....	104
5.5	Architecture of Juliette .....	108
5.6	Architecture of JSim .....	111
5.7	Example of Agent Behavior Specification .....	113
6.1	A Simple ED process in Little-JIL .....	117
6.2	Discharge part of ‘SimpleED’ process .....	118
6.3	Validating Simulation Results .....	119
6.4	Comparing Little-JIL based simulations with Arena .....	122
6.5	Declaration of the same-doctor constraint .....	124
6.6	The impact of adding doctor constraint .....	126
6.7	The impact of dynamic substitution in the waiting room. ....	129
6.8	Impact on LOS with dynamic substitutions. ....	131
6.9	The root diagram of ‘EDCare’ process .....	132
6.10	The patient care process inside the treatment area in EDCare .....	133
6.11	The impact of blocking and nonblocking bed acquisition .....	134
6.12	Root diagram of EDCare2 process .....	139



6.13 Patient care inside main-ED in EDCare2 process .....	140
6.14 Elaboration of step PerformTests .....	141
6.15 Elaboration of step BedsideProcedureMainED .....	142
6.16 Resource requirement specification for the MDAssessmentMainED step .....	142
6.17 The impact of allowing fast-track doctor in main-ED .....	143
6.18 High level view of the mediation process .....	146
6.19 Elaboration of a part of the mediation process .....	146

# CHAPTER 1

## INTRODUCTION

### 1.1 Motivation

Computer Scientists, as well as researchers in other domains as diverse as engineering, management, and the natural sciences, frequently require an (often simplified) model of the world as part of their research. Some important yet common elements in these models are entities that have often been referred to as resources. Many studies in diverse research areas have focused on the effective utilization of these resources in different domains. Although the general notion of what constitutes a resource seems similar in different research areas, their types, characteristics, and constraints governing their behavior are vastly different depending on the particular domain of research and the nature of the research itself. This dissertation aims to subsume the many different and fragmentary models of resources and present an overall, unifying view of, as well as tools and methods for dealing with, this pervasive, yet surprisingly underexamined, type of entity.

To get a better sense of the diversity of the projects that have dealt with resources, consider a representative sample of studies from a few different domains. Urgaonkar and Shenoy [80] studied “resource overbooking and application profiling in a shared hosting platform”. This study focused on optimizing only two resources: CPU and network interface bandwidth of shared hosting servers in the Internet. The resource structure was primarily static and the requests for resources were kernel level system calls specified by the operating system of the hosting platform. Mailler et al. [52] studied “cooperative negotiation for soft real-time distributed resource allocation”.

This multi-agent system study looked at the problem of allocating only one resource, namely a set of sensors for tracking targets in a real time environment. Kulkarni et al. [48] studied a system in which the resources were networked cameras. These resources were structured in tiers. While they had varying power and functionality, all of these resources were cameras and were thus relatively homogeneous.

In the manufacturing domain, Monch et al. [60] studied the modeling and allocation requirements for resources required to control functionality on a shop floor. Resources in this study were far more heterogeneous, and were structured hierarchically. The main goal of this model of resources was to support optimization through simulation. But the simulations assumed that the resources had statically defined sets of capabilities, although considerable flexibility in capabilities typically characterizes actual shop floor activities. Du and Shan [28] presents a resource management system whose aim is the efficient use of resources by a workflow management system. This work incorporates some of the generality and rigor that we aim to achieve in our own proposed research, as it includes a resource definition language (RDL) to describe the structure of resources, a resource query language (RQL) for specifying resource requirements, and a resource policy language (RPL) for specifying additional constraints. This study, however does not address the need for resource allocation capabilities, which we regard as a primary reason for modeling resources in many of the applications we are encountering in our own work. This dissertation goes beyond these works, using the need for resource allocation as a driving requirement in creating a holistic approach to resource specification and management, and the architecture of a resource management service to support resource specification and management.

The work presented in this dissertation is focused on identifying the separation of concerns in a resource management architecture that addresses resource specification and management needs in complex dynamic environments encompassing a wide variety of resource types and complex domain policies. The dissertation provides a

basis for suggesting how resource specification and management needs might be met in other domains as well. Although the applicability of this work to a few different domains is presented here, the driving and motivating example in this research has been resource management in the emergency department (ED) of a hospital. According to a recent survey from the National Center for Health Statistics [54], the average amount of time that a patient spends in the ED is 3.3 hours. For larger hospitals, such as the Bay State Hospital in Western Massachusetts, this time can be significantly higher (close to 7 hours). A large part of an ED visit involves waiting for patient care services. Depending on the acuity level of the patient, 60% to 90% of the time of an ED visit is spent waiting [54]. A primary reason for this considerable amount of waiting seems to be the suboptimal use of scarce resources. We believe that it is not hard to find other important applications domains in which suboptimal or inefficient use of resources is a core problem, and in which the inherent complexity of the resources themselves hampers the successful approach to that problem. We believe successful approaches to the problems in specification and management of resources in the hospital emergency department domain could effect improvements in the situation in this domain. It could then also serve as an example of how to approach similar problems in other domains. The following elaboration of some of the complicating issues in dealing with resources in the hospital ED domain should help to further motivate the value of careful examination of this domain.

A hospital ED is an extremely complex domain in which resources are usually scarce and their utilization is typically constrained by a wide range of relations that arise from such considerations as the patient care processes as well as the inherent nature of the resources themselves. Emergency departments contain many types of resources. Many of the most important types of resources are humans such as doctors, nurses, registration clerks, orderlies, and patients. Equipment, such as X-ray or CT-scan machines, constitutes another important type of resources. Other types

of resources, such as beds, blood, and medicines are also crucially important. In contrast to the situation in most of the studies of resources described earlier in this section, these ED resources are very heterogeneous. The broad types just enumerated are very different from each other, and yet have important relations to each other. The inherent complexity of the resources in this domain is further complicated by the considerable use of aliasing in describing these resources, with many different names being attached to the same ED resource. For example, the names ‘doctor’, ‘pediatrician’, ‘surgeon’, ‘attending MD’, ‘director’, and ‘primary care giver’ may all be attached to the same individual. Further complicating the situation, we note that the use of these different names is often in response to different circumstances, with the specific circumstances sometimes influencing either the name used or the functionality required or both. Thus, the dynamism of this application domain makes such tasks as capturing the structure of its resources and disambiguating requests for these resources quite complex.

Indeed, there are additional ways in which dynamism makes resource management in this domain very complex. We note, for example, that in an emergency situation an ED resource may provide services or perform functions that it would ordinarily not perform. For example, a physician assistant may perform an activity such as writing an order (i.e. prescribing medication) for a patient with chest pain in an extraordinary situation, whereas this is a task that would only be performed by a doctor under ordinary circumstances. Such possible changes in resource capabilities require modeling and managing them in special ways that bear investigation. In addition, ED resources often have complex substitution relationships amongst themselves. For example, in most hospitals, a patient usually waits until a bed becomes available inside the main ED where treatment is performed. However, under extraordinary conditions in some EDs, a bed located in a hallway can sometimes be treated as though it were a bed in the main ED. On the other hand, in many EDs a trauma bed can never be

allocated to a non-trauma patient even if the ED is badly overloaded with patients. Requests for ED resources are highly dependent on many different parameters. For example, incoming patients get an acuity level attached to them during initial triage. This acuity level dictates the type of priority resource requests for this patient should receive. But, there are times in the ED care-giving process when the patient's acuity level can change and the patient's priority for resource allocation would thus need to change accordingly. Further complicating ED resource allocation is the fact that resources in EDs are frequently preempted. Thus, for example, a doctor who is supervising the treatment of a patient having a low acuity level, may be preempted in order to supervise the treatment of a patient who is acutely ill. This, in turn, might then entail preemption of a bed or a place on the queue of patients waiting for a device such as an X-ray machine. Thus, resource priority and preemption are issues that must be addressed in order to cope with the hospital ED domain. Resource allocation decisions in an ED are often subject to multiple constraints, and may indeed create others. For example, many EDs are divided into multiple sections, such as the main ED, the ED fast track, and ED pediatric care. Pediatric patients are ordinarily constrained to be assigned to a bed in ED pediatric care. Under emergency circumstances, however, a pediatric patient may be assigned to a bed in the main ED. A patient in the main ED, however, may never be allowed to be treated in the pediatric section under the policies in force at some hospitals.

Still another challenging aspect of resource management is the need to allocate resources in response to streams of requests while meeting constraints resulting from domain policies. This dissertation has studied the usefulness of flexibility in modeling such complex domain policies that drive the allocation decision in a resource management framework.

The above discussion illustrates just some of the dimensions of complexity and special needs for resource management in hospital EDs. We argue that a flexible

resource management framework that can support the specification and management of ED resources, taking into account many of these sorts of complexities is likely to be quite useful in dealing with resources in other, less demanding, domains. Thus, we expect that this research will lead us to some fundamental understandings regarding how to model and manage resources in these sorts of highly complex and dynamic environments. There are many software engineering challenges in developing such a resource management framework. Thus, this work should also lead us to develop architectures, tools, and techniques that are of general use for dealing with resource issues in other domains.

## 1.2 Research Contributions

This research was aimed at improving some fundamental understandings about what resources are and how to deal with them in a consistent way across different domains. In particular, there seems to be an intuitive sense that resources are different in some basic way from such other sorts of entities as the objects that are found in systems modeled in traditional programming languages. But attempts to define precisely what this difference is have been frustrating and largely unsuccessful. In this dissertation, we have looked at resources and their management in a particularly complex and dynamic environment. This has forced us to address resource management issues that have often tended to get oversimplified in many other applications, thereby complicating efforts to understand the basic nature of what might make them different. As part of this dissertation, we present an examination of what is needed in order to be effective in specifying resources in complex and dynamic domains. Based on the insights we have gained from this investigation, we have proposed and demonstrated an engineering approach and a generic architecture for developing systems that can provide complex management services for resource instances. Although the primary motivating domain for this research has been the hospital ED, we have eval-

uated the effectiveness of the approach in such diverse domains as computer-aided negotiation and composition of web-service resources to accomplish a complex task online.

To study the larger problem of resource management we have developed ROMEO, a prototype resource management service. We have evaluated ROMEO and the resource management architectures and approaches that it implements by using ROMEO as a key component of a larger system for supporting discrete event simulation. In using ROMEO to support resource management issues posed by the various domains whose demands we have simulated, we have gained insights into some of the larger issues of resource management. It is important to note here that we do not represent that ROMEO is a ‘one size fits all’ solution to all problems of resource specification and management. Rather ROMEO represents the results of exploring different engineering approaches to dealing with resource management issues. This research suggests a promising approach to an overall resource management framework and interactions between such a framework and other systems (e.g., a simulation engine or a process guided execution environment) that would use it.



## CHAPTER 2

### RELATED WORK

Resource management is a very broad and pervasive research area. There have been numerous studies looking at different aspects of managing resource entities in a wide variety of different domains. In this section, we present the works that are relevant to our proposed approach and the application areas we have focused on.

#### **2.1 Resource Management in Hospital Emergency Department**

Since our motivating domain is the hospital EDs, we first look at the resource management related works researchers have done in this domain. Connelly and Bair [22] presents development and use of a discrete event simulation (DES) platform named EDSim to investigate the ability to predict actual patient care times using simulation. They also looked at the effect of two different triage methods on patient service time in the ED. The authors collected patient data from a five day period of an academic ED and ran simulation of their modeled ED activities using those data. They modeled the ED activities by defining patient paths, the series of activities that had to be done on a patient while she was in the ED. These activities include history and physical examination, nursing activity, imaging studies, laboratory studies, consulting and bedside procedures such as suturing, casting, and intubation. In addition to individual patient care paths, the EDSim model of this study also considered continually updated job queue prioritization and mid-task preemption capabilities of ED

staff activity. All staff activities were prioritized according to patient acuity. According to the study, this model was able to predict average patient service time within 10% of actual values. However, for individual patient paths, only 28% of individual patient treatment times had an absolute error of less than one hour. According to the paper, one of the reasons for their results not accurately producing the values of the real events was because their model did not include the changes in the staffing level at different times of the day. They also had to make many generalizations, which could have contributed to the inaccuracies.

From the discussion of the study, we understood that their model did not include the substitutability information of resources. It also was not clear whether the model accurately captured the constraints of different resource utilizations. The study also did not include any discussion about possible allocation optimization or its effect in their simulation results.

There have been many simulation studies on staff scheduling in hospitals in general and ED in particular. Kumar and Kapur [49] used simulation to analyze nurse-scheduling alternatives for ED services at Georgetown University Hospital. Draeger [26] developed simulation modeling for three EDs at Bethesda Hospitals to assess present nurse staffing concerns and to assess alternatives for improvements. McGuire [57] discusses the use of simulation to test process improvement alternatives to select an alternative to reduce the length of stay (LOS) for ED patients. Rossetti [66] looks at the use of computer simulation to test alternative ED attending physician-staffing schedules and to analyze the corresponding impacts on patient throughput and resource utilization.

As one would expect, many of these DES based ED studies have focused on some basic ‘what-if’ scenarios that relate a particular resource mix to its impact on average LOS and waiting time for patients in the EDs [58, 68, 69, 78, 7, 29, 72]. Samaha et. al. [68] shows the use of ED simulation studies to perform ‘what-if’ analysis regarding

the effect of process change and staff level change on patient LOS. Their study found that based on the ED model and patient flow they created, there was no significant impact of additional beds and space on average LOS. Similarly, Duguay [29] showed, based on their simulation model of a Canadian ED, that there was no real impact of adding more examination rooms in the ED. However, the study did report improved LOS with increased care providers. Saunders [69] showed the impact of triage acuity level of incoming patients along with the number of nurses and doctors available for care on average LOS. Takakuwa et. al [78] presented a mechanism for stepwise adjustment of available resource instances (doctors, nurses etc.) based on the places where patients were waiting the most. Khare et. al [46] used DES to study the impact of boarding time of admitted patients (the time it takes for the admitted patients to actually leave ED) on overall LOS in ED. They identified boarding delay to be one of the major reasons for ED overcrowding. Storrow [77] studied the impact of lab turnaround times on the patient flow in an ED.

Some ED simulation studies have been aimed at predicting required resource level for an estimated demand. Wiinamki et. al.[82] used DES to project bed requirements for the proposed extension of an existing ED. Baesler [7] used simulation to estimate the maximum demand an existing ED can handle. Like other studies mentioned above, they also investigated the impact of increasing different types of resource instances on patient LOS. There are other ED studies using DES, where the researchers have focused on the impact of certain process changes. One such study is by Garcia et. al [36], where the authors study the impact of introducing a fast-track lane for patients and reported favorable results. Similar results were reported in [68], and [57].

All these simulation studies have taken a factory view of the ED, where patients come in like orders on factory floor with fixed priority and drive the process by requesting resources. Many of these studies were concerned with only one type of

resource, i.e. either the attending physician or nurse and focused on only one issue of resource management, such as scheduling. Hay et. al [?] identifies this issue in their study and proposes a different way of modeling and studying ED processes. They argue that with only a factory view of the world, low acuity patients will continually starve and many will not receive treatment. They also argued in favor of modeling the skill hierarchy of ED staff, skill based request specification, and the ability for an ED agent to decide on what task to deal with next. By modeling the changing request priorities and resource mapping based on requested skill sets, they were able to produce simulation results that align more closely with how resources in a real ED operation get utilized. They were also able to reduce the maximum length of stay (LOS) for patients using their modeling approach. We should note that our proposed approach includes all these modeling techniques and more.

Another area of research that has received attention lately is that of scheduling ED staff under different constraints. Chun et al. [20] describes a "Staff Rostering System (SRS)" for creating nurse rosters for The Hong Kong Hospital Authority that manages over forty (40) public hospitals in Hong Kong. The system defines different constraints to be satisfied while creating the roster. For example, it ensures that an adequate number and mixture of skilled staff is present all the time to maintain a committed level of service quality. On the other hand, the constraints are used to make sure that each staff member is assigned an appropriate number of working hours in accordance with their terms of appointment i.e. they are not overworked or underutilized. The SRS generates the rosters using constraint programming. This type of scheduling is done at macro level of shift assignment, where as we are primarily concerned with more micro level task based resource assignment in our work.

## 2.2 Resource Management in Networking and Operating Systems

Managing resources is at the heart of many operating systems and networking research. Most of these studies, however, are concerned with primarily the allocation strategies and resource scheduling. The modeling of resource entities, their constraints, and the request language for resources do not get much attention. This is due to the fact that most of the resource type objects in systems research are relatively static in nature and the systems are built on top of fixed protocols that define the request language for that particular system. In this section, we present a sample of the system's area studies concerning resource management.

Systems' area researchers are mostly concerned with hardware resources like processor time, memory/disk space, network bandwidth, Internet hosting servers etc. Shenoy [74] describes a disk scheduling framework, cello, that studies algorithms for supporting applications with different requirements, e.g., real-time applications like audio stream and best effort application like a file transfer. Cello proposes an application class based scheduling and servicing of the resource requests on two different timescale. The proposed mechanism in this study can be mapped to our approach by grouping resource requests into classes and attaching different levels of priorities to the different classes of requests. The Resource Allocation Component (RAC) will take into account the priorities associated with different classes and apply different allocation algorithms to achieve response time constraints associated with the requests.

As described in chapter 1, Urgaonkar, Shenoy et al. [80] presents techniques for provisioning CPU and network resources in shared hosting platforms using controlled overbooking of resources. With a similar approach, [19] has studied the effectiveness of dynamic resource allocation for handling Internet flash crowds. These studies have focused on coming up with intelligent allocation mechanism and assumed fixed resource structure with no dynamism in their behavior. The resource requests were also

fixed and often based on well defined system calls to a fixed application programming interface (API) of the operating systems.

Banga, Druschel, and Mogul [8] looked at one of the shortcomings of performing resource management based on a fixed operating system API. The authors argue such an assumption restricts server scaling and effective control over resource consumption. This is because existing APIs do not allow applications to directly control resource consumption throughout the host (e.g. a web server) system. The paper presents a new operating system abstraction called resource containers and its use in fine grained resource management in monolithic kernels. The resource container represents all the resources required or used by a particular independent activity, such as servicing a client connection in a web server system. Aron, Druschel, and Zwaenepoel [5] extends this work to apply into a cluster of web servers by enhancing the concept of resource containers to cluster reserves, which can be thought of as a cluster wise resource container. These approaches validate our claim that a flexible resource management service needs to provide abstraction for bundled requests and transactional allocation.

### **2.3 Resource Management in Distributed Computing**

Distributed computational platforms like grid computing and server clusters are primarily concerned with managing distributed and possibly heterogeneous types of resources. Managing such resources can be very challenging due to many factors. In this section, we present relevant studies aimed at providing effective resource management for applications running on such platforms.

Raman, Livny and Solomon [64] argues that conventional resource management systems use a system model to describe resources and a centralized scheduler to control their allocation, which does not adapt to distributed high throughput systems. The authors point out the issues associated with heterogeneity of resources and their distributed ownership, which make it difficult to formulate uniform allocation algo-

```

Request ClassAd:
[Type = Job; Owner = user1;
Constraint = other.Type == Machine && Arch == INTEL
  && OpSys == Solaris251 && Disk >= 1000;
Rank  other.Memory]

Resource ClassAd:
[Type = Machine; Name = m1; Disk = 30000; Arch = INTEL;
OpSys  Solaris251; ResearchGrp = user1, user2;
Constraint = member(other.Owner.ResearchGrp) && DayTime > 18*60*60;
Rank  member(other.Owner.ResearchGrp)]

```

Figure 2.1: Example of Condor classads

rithm for varying allocation policies. They present the design and implementation of a resource management framework based on Matchmaking of classified advertisements (ClassAds). The framework was used in deploying a high throughput computing system named Condor. Matchmaking uses a semi-structured data model the classified advertisements data model to represent the resource objects of a system. The query (requests for resources) language is folded into the data model. This way both resource users and resource entities can specify their requests and offered services in the same way through ClassAds. The framework also distinguished between matching and claiming (assignment) as two distinct operations of the resource manager.

The type of resources that were modeled using the matchmaking framework included workstations, tape drives, network links, application instances, and software licenses. The ClassAd description of resources used a semi-structured data model, meaning that no fixed predefined schema was used by the matchmaker. Constraints (queries in case of their system) were part of the ClassAd description of resource. This way the resources could describe both their attributes and policies in one data structure. When multiple resource entities match a request, the resource manager uses a ranking function to return the highest ranked resource. Raman, Livny et al. [65] extended the Condor Matchmaking framework to include gang matching where Clas-

```

Request RedLine:
[user = globus-user;
group = dsl-uc;
Computation ISA SET [type=computation]
Storage ISA [type=storage; space >100]
Foall x in computation;
  x.cpuspeed > 150;
  x.bandwidth[storage.hn] > 30;
  x.accesstime > 18;
Sum (computation.memory) > 300;
Storage.space > 80;
Storage.accesstime > 18]

Resource RedLine:
R1 = [Type = computation;hn=ucsd1;cpuspeed=200;
Bandwidth=DICTIONARY[{s1,20},{s2,40}];accesstime>17];

```

Figure 2.2: Example of Request Description

sAds could specify multiple resources. Liu and Foster [51] extended this concept of matching semi structured data even further by modeling it as a generalized constraint satisfaction problem. The paper identified limitations of ClassAd system and designed a symmetric description language called RedLine for both resource entities and requests. The RedLine grammar allows specification of requests for resource sets with aggregate characteristics (e.g., a set of nodes with more than 10GB of combined memory) and provided way to specify preferences to guide the matching outcome. Condor ClassAds were designed to perform only exact matches on properties. RedLine enhanced it to include resource descriptions with varying levels of generality and complexity. RedLine was designed to also match advertisements based on policies as well as properties. This allowed RedLine to specify and match requests like “find all machines that allow access between 7:00 pm and 9:00 pm”. Figure 2.2 shows an example of request and resource description using RedLine.



Decker, Tangmunarunkit, and Kesselman [25] extended the matchmaking line of work even further by designing an ontology-based resource matching in the grid environment. The authors of this study argued that Condor Matchmaker type of symmetric, attribute-based matching of resources to requests is highly constrained as it requires resource providers and consumers to agree upon attribute names and values. Such system, they also argue, is inflexible and is difficult to extend to new characteristics. To reduce the coupling between resource and request description, instead of exact syntax matching, they opted for semantic matching based on ontologies. An ontology is a structured representation of knowledge about the concepts of a domain. It describes the concepts in a domain and the relationship amongst those concepts [39]. Most ontological frameworks use some sort of logic languages like first order logic or description logic [6] to express these concepts and their relationships and more importantly, to infer knowledge from the structured information captured within the ontologies. [25] developed three different ontologies for their matchmaker: a resource ontology, a resource request ontology and a policy ontology. The ontologies were developed using Resource Description Framework (RDF) schema, an XML based World Wide Web Consortium (W3C) standard for describing web resources. They also used TRIPLE [24], a rule system based on deductive database techniques to represent background knowledge in a domain. The rules were used to add additional axioms on the ontological concepts, which could not be expressed by the Ontology language.

We argue the service type structure in our modeling approach will work as the semantic layer for resource objects of a domain. The substitution and composition relationships are exactly the type of domain knowledge Decker et al. [25] tried to capture by using TRIPLE. We believe our approach is more flexible as it allows for specification of predicates on the relationships.

Kee, Yocum, and Chien [45] argues against approaches that, like the ones described above, separate the resource selection (discovery) and resource binding (acquisition) tasks in the resource management architecture. The authors point out that resource binding may fail in the real world due to inaccurate resource information, authentication failure, and contention amongst applications for resources. They argue a separate resource selection and binding approach cannot deal with binding failures efficiently. In this study, they propose an integrated selection and binding approach by grouping resource requests into independent resource allocation components.

Foster and Kesselman [32] identifies five challenging issues of resource management for metacomputing environment and presents the design and implementation of a resource management architecture addressing these concerns. Metacomputing systems have been defined as platforms that allow applications to assemble and use collection of computational resources on an as needed basis, without regard to physical location. The five challenging problems described in this paper are: site autonomy, heterogeneous substrate, policy extensibility, co-allocation and online control. Site autonomy refers to the problem of managing resources that are owned by multiple institutions. The heterogeneous substrate problem is caused by site autonomy and different sites using different local resource management systems. The problem of policy extensibility refers to the fact that metacomputing applications are drawn from diverse domains and a resource management architecture supporting them need to frequently adapt to new domain specific management structure. Many applications may need to use resources simultaneously that are located at several sites. This gives rise to the problem of co-allocation of resources. Finally, the problem of online control refers to the need for real time negotiation between application requirements and resource availability, especially when the characteristics of resource or requirements are dynamic. [32] presents a resource management architecture built around components like resource brokers, resource co-allocators and resource managers. It also defines

```

specification := request
request := multirequest|conjunction|disjunction|parameter
multirequest := + request-list
conjunction := & request-list
disjunction := | request-list
request-list := (request) request-list | (request)
parameter := parameter-name op value
op := = | > | < | >= | <= | !=
value := ([a..Z][0..9][_])+

```

Figure 2.3: Syntax of an RSL Request

a resource specification language, RSL, which the components use to communicate amongst themselves. Information about resource characteristics and their availability, on the other hand, is obtained from a directory service based on LDAP (lightweight directory access protocol). Figure 2.3 shows the syntax of RSL:

An RSL specification defines requests for resources and it gets refined by multiple resource brokers and co-allocators before ending up at appropriate resource managers of a particular site. Transformations effected by resource brokers generate a specification in which the locations of the required resources are completely specified. There is a global Metacomputing Directory Service (MDS), which is just an ensemble of all the resource servers.

## 2.4 Resource Management in Workflow and Process Languages

Workflow languages are primarily used to define business workflows that describe task coordination, flow of documents and responsibilities of who is doing what [4, 63]. Process languages are more general in nature, which are designed to define a variety of processes including business, software and other processes. Both these sets of languages provide mechanism for resource utilization in a coordinated task structure

to accomplish some goal or higher level task. Naturally managing resources is an important part of workflow and process based systems.

Different software process programming languages like APEL [23], MVP-L [13], ALF [15], Statemate [41], Little-JIL [84]) and Process Weaver [12] have developed some resource management capabilities to facilitate process execution. However, the modeling capabilities in these languages are restrictive and the support for describing resource relationships, constraints, request specification and resource allocation are minimal. Amongst these process languages, Little-JIL and APEL have taken a more detailed and explicit look at modeling and managing resource objects. We shall discuss Little-JIL in detail at a later section. Here we only provide a very high level overview of APEL's resource management.

APEL is a process language designed to model software processes. It has a graphical process representation for intuitive understanding of the process. The primary static concepts of APEL include *agent*, *activity* and *product*. There are also two other concepts named measure and version. An activity is represented by a rectangle in a workflow like diagram. There are inputs and outputs to the activity. And there is also an agent with role constraints associated with every activity. The dynamic aspect of the process is described using control flow, data flow, and the state diagram. APEL's resource management only concerns the human resources of an organization. This is modeled in terms of teams, positions, roles and agents. A team consists of positions that are required for the team tasks. A position represents a place holder for a person (e.g., project manager, software engineer etc.) in the organization structure. A role defines the function or responsibility of a resource in the realization of a specific task. Performers holding the same position (e.g., software engineer) may play different roles (e.g. coder or tester). Finally agents are human individuals who are actually performing activities during the process execution. An agent has a single position in an organization; but can play many roles. This concept of role associated with an

activity and agents binding to those roles according to their capabilities address some of our concerns related to managing dynamic behavior of resource objects. APEL does not consider automated agents or non-agent resources in their model. Moreover, APEL does not provide any support for specifying constraints on resources and avoids addressing anything related to resource request specification or resource allocation.

There are many workflow languages and infrastructures that have been developed over the years. Some prominent ones include Business Process Execution Language for Web Services (BPEL4WS), Business Process Modeling Notation (BPMN) [1, 2] and Yet Another Workflow Language (YAWL) [3]. Vasko and Dustdar [81] provides a comparative study of these languages. BPEL4WS is an XML based standard notation for describing business processes involving web services. The web services are described with an XML notation standard known as WSDL (Web Services Description Language). The goal of BPEL4WS is to provide inter-operation of loosely coupled system primarily over the web. It is layered on top of several XML specifications like SML Schema 1.0, SPath 1.0, WSDL 1.1 and BPEL. Business Process Execution Language (BPEL), as its name suggests, provides process notation to specify common business protocols. BPMN is also another such standard language. However, the focus of BPMN has been to provide user-friendly notations readily understandable by all business users. BPMN supports an internal model that enables the generation of executable BPEL4WS. YAWL, on the other hand, is a petri-net based modeling notation that provides support for formal analysis of workflows. All these languages have the implicit understanding of resources and the need to specify them. However, like many of the process languages discussed earlier, BPMN is primarily concerned with organizational human resources. On the other hand, in BPEL4WS, one can consider the web services to be resource objects and WSDL provides an XML based description language to specify the capability, location and communication protocol to these services. However, human user interaction is not covered in BPEL4WS. It is

primarily designed to support automated business processes based on Web services. To include the user interactions in business processes, extensions have been proposed under the standard BPEL4People [53].

All of the above mentioned languages seem to have taken an assumption of infinite supply (no contention) when modeling resource objects. YAWL is a work in progress where the authors have been actually working on providing a rich support for specifying resource objects and requests for these objects in the modeled workflow. Yet, like many other workflow languages, they do not discuss anything about non-active or non-agent resources. There is also no mention of issues related to allocation of resources.

Russell, Aalst et. al. [67] presents a nice work on categorizing resource specification in workflow languages. The authors have termed them as “workflow resource patterns”. They have categorized resource specification and management issues related to workflow languages into seven pattern groups namely creation patterns, push patterns, pull patterns, detour patterns, auto start patterns, visibility patterns and multiple resource patterns. The characterization of resource modeling specified at the beginning of this paper is akin to our notion of resources with a subtle difference. They define resources to be something capable of carrying out work. Like YAWL, this notion restricts the definition of resources within the boundary of what we call active or agent resources. The work mentions a lot of the characterization like human and non-human resources, consumable and reusable resources etc. It also discusses the issue of resources playing multiple roles in different contexts. We believe that this work is most relevant to our resource request specification issues we have discussed in section 3.3. Some of the patterns specified in [67] might be useful in evaluating the expressiveness of our resource specification language.

## 2.5 Resource Scheduling in Artificial Intelligence

A large area of artificial intelligence is concerned with scheduling of scarce entities, i.e., resource objects. Prof. Stephen F. Smith is one of the leading researchers on scheduling in AI. In his ‘research direction’ paper [76], Prof. Smith argues that although significant milestones have been achieved in scheduling research, there is still a lot to be done in this area. One major problem identified in [76] is that all the existing solution techniques so far seem to define scheduling as a static, well-defined optimization task like some sort of puzzle solving activity. In real life, scheduling is rarely a static, well-defined activity. It is typically an ongoing iterative process and there seems to be considerable room for improvement in the heuristics and other scheduling techniques developed to approximate solutions for this set of NP-hard problems. As part of the major challenges, the author identifies the need for generating schedules under complex constraints, objectives and preferences directly mapped to some practical domain. A second major challenge area is identified as adapting to changes in schedules due to the dynamic nature of the system that the scheduling algorithm is supporting. We feel that our techniques of resource modeling, request specification and allocation of resources should provide a flexible framework to develop systems to perform systematic studies in these directions.

Another area of AI that is focused on resource scheduling is that of multi-agent systems. Monch [60] presents simulation studies on modeling for dynamic resource allocation problems in a manufacturing setting. Mailler, Vincent et al. [52] describes a cooperative negotiation technique for solving distributed resource allocation problem.

## 2.6 Resource Allocation in Operations Research

Resource management, specifically allocation or scheduling of resources, has received a lot of attention over the last four decades in operations research (OR) community. The type of resources OR researchers are primarily concerned with are machines,

equipments etc. in a factory setting and humans in an organizational environment. The problem that has received the most attention in this research area is that of task scheduling under limited resource availability. Litsios [50] presents the early work where the author discusses the problem of task sequencing with reusable and consumable resources under discrete and overlapping time periods. The paper provides a formulation of the problem as a resource allocation problem and provides solutions using a combination of dynamic programming and combinatorial approach. This problem of task scheduling under constraints and scarce resources is broadly known as job-shop scheduling problem. Mellor [59] advocated the use of heuristics to find centralized solutions to this problem. Numerous centralized algorithms with many different heuristics have been tried on this resource allocation problem since Mellor's work. Panwalkar and Iskander [61] and Gere [37] provides a survey of all the different scheduling rules studied with this problem.

Our interest in these resource allocation works in operations research area is to primarily identify the different strategies and algorithms our modeling and management approach should be able to support. We are also quite interested in studying some additional allocation algorithms to find out the impact of novel scheduling on resource objects.

## **2.7 Resource Management in Knowledge Based Systems**

The research area that works with knowledge representation and development of ontologies is quite relevant to our proposed work. The primary goal of knowledge representation work has been to organize concepts or objects of a domain into categories. There are usually class-subclass relations amongst categories that organize them into a taxonomy or taxonomic hierarchy. Knowledge Based Systems (KBS) use such taxonomies to infer information about objects that are not directly associated with the objects. Over the years, researchers in KBS area have come up with



many modeling frameworks and approaches. One prominent knowledge engineering approach is KADS [70] and its further development to CommonKADS [71]. In CommonKADS, there are five different distinct models that are developed namely organizational model, task model, agent model, communication model, expertise model and design model. The task model provides a hierarchical description of the tasks which are performed in the organizational unit in which a Knowledge Based System will be used. The task model also holds specification as to which agents are assigned to different tasks. The agent model, on the other hand, specifies the capabilities of each agent involved in the execution of tasks at hand. In general, an agent can be a human or some kind of software system like a KBS. Agents, as defined here, are commonly found as resource objects in many systems. In commonKADS agents are primarily modeled according to organizational hierarchy and they perform some well defined functional roles associated with the task model.

### **2.7.1 Related Work in Ontology**

The primary approach KBS researchers have taken to model agents and tasks are through developing ontologies. An ontology is a structured representation of knowledge about the concepts of a domain. It describes the concepts in a domain and the relationship amongst those concepts [39]. Most ontological frameworks use some sort of logic languages like first order logic or description logic [6] to express these concepts and their relationships and more importantly, to infer knowledge from the structured information captured within an ontology. This structured information has become the standard for sharing and reuse of components in knowledge based systems.

Another research area that is increasingly using ontologies heavily is the semantic web [10]. The objective of the semantic web work is to provide meaning and structure to the vast information available through web pages to facilitate communication and

inference for web based applications. Many languages like DAML (DARPA Agent Markup Language) [42], DAML+OIL (Ontology Inference Layer) [9] and OWL (web ontology language) [56] have been developed progressively one extending another to achieve this goal. All of these languages have originated on top of the the eXtensible Markup Language (XML) and the Resource Description Framework (RDF) [47] standards proposed by the World Wide Web Consortium (W3C). RDF is a framework for representing information in the web about web resources. Resources in RDF have a narrow scope that includes entities such as web page meta-information like content rating, capability descriptions, privacy preferences, etc. DAML, OIL and finally OWL have taken the work gradually further and provided us with a language for describing not only web resources but also web based applications (web services), organizational processes, annotation of web resources etc. More importantly OWL provides rich vocabulary for describing properties and classes of resource entities. It describes relations between classes (e.g. disjointness), cardinality (e.g. exactly one), equality, characteristics of properties (e.g. symmetry), enumerated classes and so on as one would expect from a standard ontological language.

An ontology language like OWL can be useful in describing some of the structures and relationships of the resource objects we are concerned with. However, they fall short while describing constraints associated with the resources' characteristics. For example, OWL is not suitable for specifying substitutability relationships amongst resource classes. It is also difficult to capture other dynamic characteristics or behavior of resource objects using just the existing syntax of an ontology language.

Hobbs, Lassila et al. [43] presents an informal study on creating an ontology of resources. This unpublished work from DAML+OIL initiative is very relevant to our approach. The authors here propose an ontology with high enough abstraction to cover physical, temporal, computational and other sorts of resources. The paper describes the principal classes of properties resource objects can have. It approaches

resource objects with the differentiation of 'resource types' and 'resource tokens' (instances). Resource objects have been categorized as either consumable or reusable type. Resources have also been categorized based on their capacity (discrete capacity vs. continuous capacity) and composition type (atomic resources vs. aggregate resources). The work was only concerned with modeling resource objects and did not look into other parts of resource management like request specification and allocation.

We shall take close look at Fadel and Fox's work [31, 34], which presents a very relevant work on developing generic enterprise resource ontology for a manufacturing enterprise environment. The work also includes a first order logic implementation of the resource definitions and constraints as axioms in prolog. The ontology developed in this study could deductively answer common sense questions about the enterprise knowledge. The focus has been to reason about how properties of resources change as the result of activities, and also to reason about allocation of resources in a scheduling task through capacity recognition. The ontology was developed as part of the TOVE (Toronto Virtual Enterprise 1992) [33] project and it included the following competency questions for evaluating the effectiveness of the ontology:

- *Divisibility*: Can the resource be divided and still be usable?
- *Quantity*: What is the stock level at time t?
- *Location*: Where is resource R?
- *Consumption*: Is the resource consumed by the activity? If so, how much?
- *Commitment*: What activities is the resource committed to at time t?
- *Structure*: Can the resource be shared with other activities?
- *Trend*: What is the capacity trend of a resource based on the machine usage history? In TOVE model, a resource has always a role with respect to an activity. The roles specified in the ontology are: raw material, product, facility, tool,

and operator. There are three types of divisibility defined on resource entities: physical, functional and temporal. The ontology also defines the following terms and predicates to specify different aspects of resource entities: Unit of measurement: Specifies a default measurement unit for a resource, when associated with activity.

- *Component of*: Specifies a resource being composed of one or more sub resources. A resource can be a physical or functional component of another resource with respect to an activity and each does not share the same role with the original resource.
- *Quantity*: A resource point (rp) specifies a resources quantity at a point in time.
- *Continuous vs. discrete resource*: A continuous resource is uncountable whereas a discrete resource is countable.
- *Capacity*: Defines the maximum set of activities that can simultaneously use/-consume a resource at a specific time.
- *Simultaneous use restriction*: Prohibits use/consumption of a resource by two activities simultaneously.
- *Activity history*: This predicate specifies the history of usage or consumption of a resource before a specified time point.
- *Set up constraint*: Specifies the duration required to set-up a resource for usage by an activity.
- *Alternative resource*: Specifies an alternative resource(s) to be used or consumed by an activity.

[31] showed how the developed ontology coded in first order logic could be used to answer the competence questions about resource they had defined earlier. This

work provides a nice approach to model resource objects. However, like the other works described in this section, the study is confined only within the modeling part of resource management and does not discuss issues with request specification and allocation.

## 2.8 Programming language support for dynamic objects

It has been our claim in this dissertation that resource type objects manifest some special dynamic characteristics while responding to requests for them. The interplay of resource requests and actual assignment of resources to requests brings forth these special features of resource objects. We have also claimed that existing object oriented design and programming languages don't provide any direct way of supporting the special characteristics of these objects. It is thus important to first look at what support is available in existing programming languages to model and manage objects that seem to have system state dependent dynamic behavior.

There have been studies to incorporate some of the dynamic features of objects in languages like Smalltalk [38], Common Lisp Object System (CLOS) [11], Objective-C [62] etc. These languages are commonly known as dynamic languages. There are also specialized languages like Dylan [73] and SELF [79] focused on some specific dynamic aspects of objects. Some dynamic features that are found in these languages include:

- Variables can be of unrestricted type; e.g. Smalltalk and Dylan.
- Objects can be used interchangeably if they implement compatible methods. In smalltalk-80 [38] and other dynamic languages, this is known as “structural conformance”.
- Methods on instances can be wrapped by other methods defined by programmers to get executed before and/or after a method of the target instance. CLOS [11] provides ‘hooks’ to achieve this capability.

- Instances can be created of classes that “mix” multiple parent classes dynamically allowing for multiple-inheritance.

In addition to the above features, some languages have even done away with class based specification. SELF [18] is such a language. It is similar to Smalltalk in terms of its syntax, typelessness, blocks (objects which behave like procedures) and incremental exploratory environment. However, there are no classes; rather SELF is a prototype based language and there is object level inheritance. One of the limitations of class based languages is that if one needs to change an object’s behavior or structure, one needs to change the class. Changing the class dynamically may have far reaching implications as there could be many instances of that class. To address this issue, SELF has introduced prototypes. To create a new object, the programmer finds an existing similar object and copies it. The programmer then can safely change the copy without affecting anything else. Almost all object oriented languages provide some level of polymorphism through method overloading and dynamic dispatching of methods. Some of the dynamic languages mentioned above also provide multiple-inheritance. Dylan (Shalit) and CLOS [11], for example, allow classes to be direct subclass of multiple super classes. The conflicts of inherited variables or values are disambiguated through an additional mechanism called class precedence list. This precedence list also allows for correct method dispatch in the case of multiple-inheritance. Some dynamic languages like CLOS also provide what is known as generic functions. An ordinary Lisp function has a single body of code that is always executed when the function is called. A generic function, on the other hand, has a set of bodies of code of which a subset is selected for execution. The selected bodies of code and the manner of their combination are determined by the classes or identities of one or more of the arguments to the generic function and by its methods combination type. This is also similar to method overloading. CLOS also provides some added feature like dynamic invocation of wrapper functions before and

after a function call. There have also been efforts to add some dynamic features like multiple inheritance and structural conformance to popular object oriented languages like Java using dynamic code generation [14].

Some of the language features discussed above will certainly be helpful in describing and manipulating the dynamic nature of resource type objects. The ability to define multiple-inheritance is needed to describe resource objects that need to provide services apparently inherited from different ancestry. As we have discussed above, disambiguating behavior of an object that inherits from multiple classes is usually done by keeping a precedence list of classes in the class hierarchy. This may not be sufficient in describing some resource objects' behavior. For example, a natural class hierarchy of resource objects may have two subclasses of the root class resource namely *active* resource and *passive* resource. In a task coordination framework, the former may represent resources that can be made responsible for carrying out a task (often known as *agents*) and the latter may represent resources that an *agent* resource instance requires to carry out the task assigned to it. In a hospital ED simulation application, for example, the patient resource can be a subclass of both active resource and passive resource. A patient is an active resource (or agent) when it needs to perform active tasks like giving consent or confirming his or her identity. The patient is a passive resource when he or she is required for the task of performing some lab tests. There is not necessarily a fixed precedence class list that can be applied for disambiguating the methods of this class. More importantly, none of the dynamic language features are directly applicable to describe an object that has methods that becomes active or inactive under different situations.

## CHAPTER 3

### APPROACH

Chapter 1 has provided some examples that should lead to intuitions about the causes and nature of researchers' interest in resource management. These examples have also pointed out some specific challenges that can be expected to arise in dealing with resources in highly complex environments. In chapter 2, we have presented some of the most relevant work from widely different domains that have investigated different aspects of the resource management problem. In this dissertation, we have tried to take a systemic and holistic approach to the resource modeling, request specification and resource allocation problems and have developed techniques capable of supporting studies of complex real world scenarios. We then explored how successfully our approach can be applied to the overall handling of resource-related issues in the highly complex domain of ED resource management. We were also interested in exploring the applicability of our work to other domains with very different resource and request models as well as domain policies. This work reemphasizes that this problem space is very large, and that care must therefore be taken to be precise about the problem subspaces that we are approaching. Accordingly we suggest that the overall holistic resource specification and management problem can be decomposed into four related sub-problems, namely:

1. How can we represent resources? Are there characteristics of resources that are dynamic, i.e., dependent on the state of system in which resources are to be used? What effect do dynamic characteristics have on the eligibility of a resource instance to fulfill a request?



2. How should we specify requests for resources? What information should be captured as part of a resource request? How should resource requests represent any constraints that might be relevant in making allocation decisions?
3. How can we represent and deal with constraints arising from such considerations as domain policies? How do we model the state of a running system so that it can be effective in supporting the computation of applicable dynamic constraints related to resource allocation? How should the constraints and system state be modeled in order for them to best be taken into consideration in the assignment of resource instances in that domain?
4. How should the matchmaking between resource requests and resource entities be performed in order to determine which resources are eligible to fulfill a request based on resource suitability and availability, and yet subject to constraints? Once a set of eligible resources is found, how can we identify which is the *best* match to the request?

### 3.1 Specification of the General Problem

In this section, we define some useful notation and then use the notation to state some of the central resource management issues and problem precisely. We also use the notations to describe how we approach to some of these problems. Let  $D$  be some domain of interest. Let  $T$  represent the set of all of the different activities that are carried out in performing any of the processes in  $D$ . Let  $\Sigma$  be a specific application process in  $D$ . Now let  $R = \{r_1, r_2, \dots, r_k\}$  be the set of all resource instances that are available for use by  $\Sigma$ .

**Definition 1** *A task instance  $t \in T$  is an atomic activity in  $D$  that requires a set of resource instances, all of which are needed in order for the task instance to be performed.*

Thus, every task instance  $t \in T$  has associated with it a corresponding collection of  $n_t$  resource requests:  $Q(t) = [q_1^t, q_2^t, \dots, q_{n_t}^t]$ , where it should be noted that  $Q(t)$  is a collection, rather than a set. This is because a task instance  $t \in T$  may require more than one instance of a requested resource, in which case there would be two or more identical requests, whose satisfaction would then require two or more resource instances each capable of satisfying a different one of the requests. Let  $S$  be the set of all possible execution states that application  $\Sigma$  in domain  $D$  can assume. We assume that any particular execution of the system can be discretized by time units and time is one of the components that characterizes a specific system state  $s \in S$ . Other components that may be used to define a specific  $s \in S$  include states of various artifacts being manipulated by  $\Sigma$  and the configuration of active tasks. One component of particular interest to us is the set of bindings of resource instances to active activities. The following definitions are useful in defining this more carefully.

In a given system state  $s \in S$ , there is a subset of  $T$ ,  $T_s$ , consisting of  $m_s$  task instances  $T_s = \{t_1, t_2, \dots, t_{m_s}\}$  that are ready and waiting to get started. Thus, we define the total demand for requests at system  $s$ ,  $Q(s)$  or  $Q(T_s)$ , by the following collection of requests.

$$\begin{aligned}
Q(s) &\equiv Q(T_s) \\
&\equiv [[Q(t_1)], [Q(t_2)], \dots, [Q(t_{m_s})]] \\
&\equiv [q_1^{t_1}, q_2^{t_1}, \dots, q_{n_{t_1}}^{t_1} \\
&\quad q_1^{t_2}, q_2^{t_2}, \dots, q_{n_{t_2}}^{t_2} \\
&\quad \vdots \\
&\quad q_1^{t_{m_s}}, q_2^{t_{m_s}}, \dots, q_{n_{t_{m_s}}}^{t_{m_s}}]
\end{aligned}$$

We use the notation  $[ ]$  to represent the request collection. Each element,  $q_j^{t_i}$ , inside the collection represents a request for a resource instance. The request  $q_j^{t_i} \in Q(s)$  is associated with the task instance  $t_i$  at some state  $s$  and  $1 \leq j \leq n_{t_i}$ . Let us denote  $Q(\Sigma)$  to represent the set of all possible requests the application  $\Sigma$  may produce.

We define that each resource request,  $q \in Q(\Sigma)$ , specifies some requirements on the resource instances that can be used to fulfill the request. Later we shall discuss how requests may define such requirements by using required resources' *attributes* and *capabilities* and additional constraints on allocation. For any request  $q \in Q(\Sigma)$  at some specific system state  $s$ , we define  $R(q, s) \subseteq R$  to be the set of resource entities eligible to fulfill the request at that state. This includes resources that fulfill the requirement and are available for assignment. Given the requirements specified in the requests, we are interested in the problem of finding *allocations of resources* to fulfill the requests. This entails mapping requests to eligible resource entities.

**Definition 2** A resource allocation at some state  $s$ ,  $ALLOC$ , is a set of ordered pairs,  $(q, r)$ , where  $q \in Q(s)$  and  $r \in R(q, s)$ . We define  $ALLOCS(s)$  to be the set of all possible such allocations in state  $s$ .

We note that for some system state  $s$ , it may be the case that there is no possible allocation; i.e.  $ALLOCS(s) = \{\}$ . This would happen if there is no  $r \in R$  that can satisfy any of the  $q \in Q(s)$ . We also note that according to the above definition, an allocation  $ALLOC(s) \in ALLOCS(s)$ , may not map all the requests defined in  $Q(s)$ .

In some domain  $D$ , there may be constraints upon resource allocations in  $\Sigma$  resulting from domain policies or from the inherent nature of resources in  $D$ . Let  $C(D)$  represent the set of all constraints on resource assignments in the domain. At any system state  $s$ , we define a subset of constraints,  $C(D, s) \subseteq C(D)$ , as the set of active constraints. We consider each constraint  $c \in C(D, s)$  at some state  $s$  to be a predicate defined on  $ALLOC(s)$ . The purpose of each  $c \in C(D, s)$  is to determine whether or not the entire set of all ordered pairs defined by an allocation  $ALLOC(s) \in ALLOCS(s)$  conforms to this active constraint. If it does, then  $c(ALLOC(s)) = True$ , otherwise  $c(ALLOC(s)) = False$ .

**Definition 3** A resource allocation at system state  $s$ ,  $ALLOC(s) \in ALLOCS(s)$ , is a valid resource allocation if and only if

$$\forall c \in C(D, s); \quad c(ALLOC(s)) = True$$

Even after filtering out allocations that are not valid, there will often be multiple possible *valid resource allocations* for a collection of requests  $Q(s)$  and a set of constraints  $C(D, s)$ . The objective of many resource allocation problems is to find a valid allocation that optimizes some function, often known as a *utility* function.

**Definition 4** A utility function,  $UTIL$ , maps a set of resource allocations at a particular state,  $s$ , into the set of real numbers,  $\mathbb{R}$ . Thus,

$$UTIL : ALLOCS(s) \rightarrow \mathbb{R}$$

**Definition 5** Given a utility function  $UTIL$  defined over a set of allocations,  $ALLOCS(s)$ , a specific allocation  $ALLOC(s) \in ALLOCS(s)$  is an optimal allocation if  $UTIL(ALLOC(s))$  produces a value that is optimal (maximal or minimal) compared to other allocations in  $ALLOCS(s)$ . We say that  $ALLOC(s)$  is optimal with respect to  $UTIL$  at state  $s$ .

An example of a utility function could be the number of tasks that can get started with a resource allocation,  $ALLOC(s)$  (i.e. the number of tasks all of whose requests are met). The optimal resource allocation would be the one that maximizes this function; i.e., enables the maximum number of tasks to get started. It is conceivable that in some state  $s$ , there will be multiple allocations producing the optimum utility value. If each  $ALLOC(s) \in ALLOCS(s)$  finds assignments for all requests,  $Q(s)$ , then, according to the above defined utility function, all valid allocations in state  $s$  will evaluate to the same possible optimum value. In such scenarios, the objective could be just to find or choose one optimal allocation.

Many other optimality metrics are possible. Thus, for example, let us assume that there is a cost associated with the failure of a task  $t \in T(s)$  to get started at state  $s$  due to resource unavailability. Under this assumption, another utility function could be the aggregate cost of the tasks whose resource requirements remain unfulfilled in some state  $s$ . An optimum resource allocation in this example would be one that minimizes this utility function. Again, it is possible to have multiple allocations that optimize this utility function by producing the minimum value of the function. Zhang [85] has looked at this problem with the assumption that each task-resource binding is exclusive. The problem, termed by the author as *bundled exclusive resource allocation problem or BERAP*, has been shown to be NP-Hard when there are more than two requests per task.

The above definitions have helped provide the basis for being rigorous in defining some classes of resource allocation problems. In order to define any specific resource allocation problem, a number of additional details and complicating factors must also be specified. Thus, to specify a specific resource allocation problem, one typically needs to provide:

1. A clear definition of  $R$ , the set of resource instances available in a domain and information about the circumstances (e.g. the specific system execution states) under which each will be available.
2. A precise definition of  $Q(s)$ , the resource requests that will be issued at all of the different system execution states, as these are the requests against which resources will have to be allocated.
3. An articulation of all of the constraints  $C(D, s)$  of a particular domain  $D$  that will have to be satisfied when the system execution is in each of its different specific state  $s$ , and

4. Algorithms for finding or approximating the optimum values with respect to a well defined utility function *UTIL*.

This brings us back to the four sub-problem areas that we identified earlier as key parts of a resource management service. These are the separate concerns based on which we develop our solution approaches and the overall **Resource Manager** architecture. Next we look more closely at each of these concerns.

### 3.2 Resource Model

We noted in chapter 1 that in some domains, such as operating systems, networking, and multi-agent systems, the resources that are managed are usually homogeneous, and their characteristics are often static. The specification of the task instance by itself is often sufficient in some of those domains to identify and assign resources. Modeling such resources can be relatively straightforward. However, resources in a highly complex environment like a hospital ED are usually more difficult to model, as they can be highly heterogeneous, they are often constrained by various constraints and their composition can change dynamically. It is crucially important to model such complex resources effectively; as such a resource model will be a key basis for subsequent efforts to identify and assign resource entities in response to resource requests. The requests in such complex environments specify the need for resource instances by specifying some static and/or dynamic characteristics, possibly defined by the *attributes*, *capabilities* and even capability associated *service-quality* of the required resource. We shall shortly provide definitions of these terms. The specified requirements in the resource request may sometimes uniquely identify the resource being asked for. A more common case, however, is the situation where the required characteristics identify more than one resource instance, any one of which can fulfill the request at a certain system state. Thus, the identification and assignment of resources to requests are dependent on the *attributes*, *capabilities* along with their

*service-quality* measures of the required resource entities specified in the requests, as well as the context provided by the application system’s state, which determines their eligibility and availability for satisfying a request. Often, but not always, *attributes* specify static characteristics of a resource instance that remain unchanged in all system states. *Job-Title* or *Degree* can be examples of attributes for resource instances that are human in a hospital ED domain. Typically the values of these attributes do not change during the execution of a process. In some cases, however, (e.g. a promotion) the values might indeed change. *Capabilities* are dynamic characteristics of resource instances that specify a resource instance’s ability to satisfy a resource request at a specific system state. Examples of *capabilities* for resources in the ED domain may include ability to *perform ECG* or ability to *perform suturing* for laceration repair or the ability to perform *triage* for incoming patients. We suggest that often a request may adequately specify requirement simply by means of the specification of a required *capability* or *service*, while in other cases an *attribute-name, attribute-value* pair may be more appropriate way of requesting a resource instance. *Service-quality* is basically the quality information associated with each of the *capabilities* of resource instance. In the following discussion, we provide rigorous definitions of these terms.

**Definition 6** Resource Attribute: *We define a resource attribute,  $ATT$ , to be a function that maps the set of resource entities  $R$  of some domain  $D$  to some predefined set of elements, denoted as  $V_{ATT}$ . We assume that the value  $null$  is an element of  $V_{ATT}$ . Any attribute function  $ATT$  that is not meaningful for a resource  $r \in R$  evaluates to  $null$  for that resource. Thus,*

$$ATT: R \rightarrow V_{ATT} \text{ where } \forall r \in R, ATT(r) \in V_{ATT}$$

**Definition 7** *Let us suppose that  $ATTRS(D)$  represent the set of all  $ATT$  functions associated with resources of domain  $D$ . We define  $ATTRS(r)$  as the set of  $ATT$  func-*

tions that are meaningful for resource instance  $r$ . We say that an attribute function  $ATT$  is meaningful for a resource instance  $r$  if  $ATT(r)$  returns a non-null value.

**Definition 8** *Capability and Capability Set:* We define a capability,  $CAP$  to be a service that may be required, and assume that each service in a domain will be identified by a name that will be known and mutually understood by both the resource instances and resource clients. Examples of service names might be *triage-patient*, *assess-patient* etc. We say that a resource instance  $r \in R$  possesses a capability,  $CAP$ , if it is able to satisfy a request seeking that service. Let  $CAPS(D)$  be all such service names that can be included in requests for resources in a domain  $D$ . We define  $CAPS(r, s)$ , a subset of  $CAPS(D)$  to be the set of services the resource entity  $r$  is capable of providing at some execution state  $s$  in  $D$ , and  $CAPS(r)$  to be the union of  $CAPS(r, s)$  across all possible execution states  $s$  in  $D$ :

$$CAPS(r) = \bigcup_{s \in S} CAPS(r, s)$$

As the definition suggests, while a request may ask for a resource instance through the specification of required *attribute-names* and corresponding *attribute-values* in some cases, we assume that there are other cases in which a request may instead specify a *capability*. This seems important, for example in the ED domain, because we note that a given *capability* may be offered by many different resource instances that potentially may have combinations of *attribute-names* and *attribute-values* that may vary considerably from each other, and indeed may vary over time as a process executes. Thus supporting only requests for specified resource instances defined by their attributes (which are relatively static) may be too restrictive. We note that it must be possible for tasks to request resource *capabilities* as well. In this case, the resource management service must match resources offering the requested *capability* to the request seeking it. As an example from the ED domain, all *nurse* resource



instances may have the capability to perform tasks like ‘Perform ECG’, ‘Draw Blood’, ‘Triage Patient’. However other resource instances (e.g. a medical technician) may also offer some of these capabilities. In some cases, a request (or requester) specifying one of these capabilities may care which category of resource provides it, and would thus specify an attribute-name attribute-value pair that characterizes this category. In other cases, the request (or requester) specifying one of these capabilities may not care (or may not even know, or want to know) which category of resource offers a required capability. In these cases the requesting entity should be able to simply request the capability. We note that in domains such as the hospital ED, *not* all of the capabilities defined by the set,  $CAPS(r)$ , of a resource entity  $r$ , can be accessed or utilized under every system state to provide the services specified by resource requests. This is because access to some capabilities of resource instances may be determined dynamically, with some resource instances making some capabilities available only under unusual circumstances, represented by a subset of system states. Depending on the current execution state,  $s$ , resource instances will always possess a set of *active capabilities*. These are the capabilities of  $r$  that can be used to fulfill a request under the current execution state of the system. Thus we define:

**Definition 9** Active Capabilities: *At any specific system state  $s \in S$ , a resource instance  $r \in R$  has a set of capabilities,  $ACTCAPS(r, s) \subseteq CAPS(r)$ , that can be used to fulfill requests. We denote  $ACTCAPS(r, s)$  to be the set of active capabilities of resource  $r$  at execution state  $s$ .*

Resource instances that possess the same *capability* may not be equally suitable for fulfilling a request seeking that *capability*. For example, many resource instances in the ED domain may be capable of providing the service of patient *triage*. However, there may be different levels of service-quality (e.g. low, medium or high) associated with the *triage* capability of different resource instances. We call this measurement

information the *service-quality* of a resource instance with respect to a specific capability in its capability set.

Sometimes requests may not only specify a capability but also the required level of *service-quality* as part of the request. There are many different ways one can concretely define this quality measure, a particularly straightforward example of which is presented below:

**Definition 10** Service-Quality: *The service-quality for a resource entity  $r$ , denoted as  $SRQL(r)$ , is a function that maps capabilities of  $CAPS(r)$  to the enumerated set:  $\{low, medium, high\}$ . Thus,*

$$SRQL(r) : CAPS(r) \rightarrow \{low, medium, high\}$$

We note that it is possible, and in some domains necessary, to model *service-quality* with more precision, perhaps using a numeric scale. In addition there are circumstances under which the *service-quality* value might be computed only by taking the execution state of the system into consideration. This would mean that given a specific state  $s$ , a capability  $c$  of resource instance  $r$ , ( $c \in CAPS(r)$ ) will have different values for its associated *service-quality* depending on the state of the system  $s$ , where  $s$  includes such information as the availability and utilization level of  $r$ . This would lead to modification of the above function in the following way:

$$SRQL(r, s) : CAPS(r) \times s \rightarrow \{low, medium, high\}$$

Again note that the range of the  $SRQL$  function can be other domains such as the set of all real numbers ( $\mathbb{R}$ ) instead of the simplified enumerated set ( $low, medium, high$ ) that we have used in the example definition given above.

For the purpose of allocating resources, there are three attributes that we explicitly specify in our resource model: *consumability, capacity* and *cost*.

- *Consumability*: This is a Boolean valued attribute that defines if a resource instance's ability to satisfy a request is consumable or not. If a resource instance  $r$  is not consumable (i.e. it is reusable), it can satisfy multiple requests one after another.
  
- *Capacity*: This attribute is used to quantify the amount of effort that can be provided by a resource instance in fulfilling requests. For *consumable* resources, this attribute is simply a measure of available quantity of a resource. For *reusable* resource instances (i.e. the ones that are *not consumable*), this attribute specifies the number of requests a resource instance can simultaneously satisfy. In the most straightforward case, every capability would require the resource instance to use the same quantity of capacity to fulfill a request for that capability. But it is also possible that some capabilities may require greater quantities of capacity than others. In such domains, each capability will require a specification of the quantity of capacity it requires. This quantity may indeed even vary depending upon the state of the execution of the system. For some reusable resource instances in some domains, it might be necessary to elaborate *capacity* with two separate attributes, namely *reservation-capacity* and *allocation-capacity*. For example, in a task coordination system, an important category of resource instances are the ones that are made responsible for carrying out the tasks. In such systems, it is often required to make a distinction between the *reservation-capacity* and the *allocation-capacity* of a resource instance that is assigned to carry out some task.
  - *Reservation Capacity*: This quantity specifies the amount of capacity that a resource instance has available to fulfill requests.

- *Allocation Capacity*: This quantity specifies the maximum amount of capacity that a resource instance can make available to support simultaneous execution of assigned tasks.
- *Cost*: There is usually a *cost* associated with the usage of resource instances. Even though *cost* may not be an important attribute for systems in all different domains, it is nevertheless a characteristic of resource instances that is ubiquitously present all the domains we have looked at.

In addition to capturing the resource characteristics (*attributes, capabilities, service-quality, capacity, cost* etc.), we suggest that it is useful for a resource model to also classify resources into different groups like *doctors, nurses, beds* etc. In particular, we note that this may facilitate the specification of resource requests (to be discussed shortly). For example, it might be most convenient to allow a request for a *doctor*, which is a label for a specific group of resources rather than requiring the request to specify an instance of a resource such as *resource-instance-21* or *resource with intubation capability*, which, according to an ED resource model, may refer to a particular *doctor* or a group of *doctors* and other such resources in the ED. Thus, we note that it would be useful for a resource model to also have a number of named groups  $G = \{g_1, g_2, \dots, g_z\}$  and a membership criterion function  $MEM_{g_y}$  associated with each group that defines which resources are members of which group. For a group name  $g_y$ ,  $MEM_{g_y}(r)$  defines whether a resource entity  $r$  belongs to this group or not:

$$MEM_{g_y}(r) \rightarrow \{true, false\}$$

It is important to note here that some of these group membership functions may define de facto type structures. However, we intentionally leave out the definition and discussion of possible resource type structures in our research. We argue that this notion of groups and group-membership seems to provide a more flexible way

to model resource entities than a more rigidly defined type structure. This research looks into the use of various mechanisms for grouping resources.

Based upon the above discussion, we suggest that the purpose of resource modeling is to capture the resource instances' characteristics, some of which can be context dependent, and perhaps some categorization (group-membership) information. We suggest that a resource instance in the context of a specific domain can be described as an object defined by a set of *attribute* functions, and a set of *capabilities* (or *services*) each of which may have associated with it *service-quality* specification. Thus, a resource instance  $r$  in some domain  $D$  can be defined as:

$$r = \{ATTRS(r), CAPS(r), SRQL(r)\}$$

At any specific system state,  $s$ , a resource entity can be defined as the following:

$$r(s) = \{ATTRS(r), ACTCAPS(r, s), SRQL(r, s)\}$$

We also propose some attributes, namely *consumability*, *capacity*, and *cost* that are common to almost all resource instances:

$$\forall r \in R \quad \{CONS, CAPACITY, COST\} \in ATTRS(r)$$

As one can immediately see that this definition of a resource instance is dependent on the execution state of a system i.e. a specific context. This particular notion has important consequences in understanding one of our fundamental research questions: 'what is a resource?' As this proposed definition would suggest, many resource instances can only be defined according to a given context. It also provide at least a partial explanation of why the specification of a resource model can be very challenging.

### 3.3 Request Model

We indicated in section 3.1 that we propose to model requests as predicates defined over a resource model. Thus we model requests that are defined using required *attribute-name*, *attribute-value* relations, or required *capability* and *service-quality*. In our model, an important part of a request specification is a specification of the required capacity. These specifications allow for specifying the required characteristics of resource instances. In addition to the required characteristics, a request may need to specify some required protocol for fulfilling the request. Also, a request may often point to constraints that resource instances must satisfy in fulfilling the request. We shall discuss the constraints specification separately in our discussion about modeling constraints in section 3.4. In our model, a request for a resource instance is composed of four types of information:

- Specification of required attributes or capabilities
- Specification of required capacity or quantity
- Specification of protocol to be followed in satisfying the request
- Specification of constraints based on runtime information

Next we elaborate each of these modeling constructs.

#### 3.3.1 Specification of Required Characteristics

The most common criteria specified in a request are a set of requirements regarding the characteristics of resource instances that can be used to fulfill the request. Formally, this set of required characteristics in some request  $q$  is a predicate  $P_q$  over the set of *attributes* ( $ATTRS(D)$ ) or *capabilities* ( $CAPS(D)$ ) as has been defined earlier in definition 7 and 8. The set of resources that would be eligible to satisfy the request  $q$  is specified by  $P_q(r)$  or  $\{r|P_q(r)\}$ .

Often such a request may specify a list of preferences in addition to also specifying required characteristics. In this case, a request  $q$  would specify multiple predicates,  $P_q^1, P_q^2, \dots, P_q^n$  in some preferential order. Let us denote  $PREF(P_q^1, P_q^2, \dots, P_q^n)$  to be the function that provides a specific order to the predicates. Each predicate  $P_q^i$  represents a set of resources such that

$$\{r | P_q^i(r)\} \subseteq R$$

The Resource Manager is expected to satisfy the request using these predicates in turn starting from the beginning of the preferential order. Thus, if no satisfiable and available resource instance is found using the first predicate in the preferential list, Resource Manager will use the second predicate in identifying resource instances that can be used to satisfy the request.

### 3.3.2 Specification of Required Capacity

This part of the request model can be very simple depending on the domain for which requests are being modeled. This specification of amount is primarily dependent on the nature of resource instances in terms of their *consumability*. If a resource instance is consumable, the specified amount is going to refer to the quantity that is being requested. The definition of quantity can be different for different types of consumable resource instances. On the other hand, for reusable resources this measure would refer to the *capacity* that is being requested. Also depending on the type of request (for example, request for reserving a resource or request for allocating a resource), this required capacity might indicate whether the request is for a specific category of capacity such as *reservation capacity* or *allocation capacity*. We allow such detail in the specification of a request in order to enable a high degree of flexibility in modeling assignment of resource instances to requests in some domains. For example, if a request seeks to reserve a resource instance and if it requires its full reservable

capacity, that allows us to model the scenario where a resource instance is completely reserved and thus cannot be reserved by anyone else.

### 3.3.3 Specification of the Protocol While Satisfying the Request

In addition to specifying the required *attributes* and *capacity*, a request for resource instances often needs to specify protocol-related requirements that would dictate the way a resource management service needs to satisfy the request. In many domains and applications, the protocol for satisfying all the requests is same and may be well established as part of the application specification. However, in our approach we allow the protocol to be modeled explicitly as part of the request.

#### 3.3.3.1 Blocking and Nonblocking Requests

Although the specification of protocols for satisfying requests can be quite complex, we use the example of the need to specify whether or not a resource request is *blocking* or *non-blocking* to show that some of these protocols may actually be relatively simple to specify.

**Definition 11** *A non-blocking request  $q^{nb}$  requires that the resource management service immediately notifies the requesting entity if a request cannot be satisfied at that time.*

**Definition 12** *A blocking request  $q^b$  is a request that does not require an immediate response in case the Resource Manager cannot satisfy it right away. Under such circumstances, the Resource Manager places the request in a queue of pending requests and tries to satisfy it once resource instances that meet the requirements in  $q^b$  becomes available.*

#### 3.3.3.2 Atomic Assignments

In addition to the *blocking* and *non-blocking* protocol requirements for satisfying resource requests, we also model grouping of resource requests that need to be satisfied



atomically. If acquisition of a set of requests is specified to be atomic, the **Resource Manager** is expected to acquire the resource instances only when each of the requests within an atomic group is satisfiable. In our model, as defined at the beginning of this chapter, a request group  $Q(t) \equiv [q_1^t, q_2^t, \dots, q_{n_t}^t]$  associated with some task instance  $t$  is considered to be atomic for the purpose of getting  $t$  activated and completed. In such a case, the **Resource Manager** needs to wait until it can satisfy each of the requests  $q_i^t \in Q(t)$  before making an allocation. Chapter 4 discusses how our resource management service supports this protocol requirement.

### 3.4 Constraint Model

Our research indicates that constraints on resource allocation primarily originate from domain policies. Consequently, we have modeled this as an important separate concern in our approach. Different application domains may have different policies that dictate or at least influence which resource instances can satisfy which request. Often such restrictions are dependent on the execution state of a running system. Even if a resource instance  $r_i$  is eligible for satisfying a request  $q_j$ , i.e.,  $r_i \in P_{q_j}(r)$ , there may be additional constraints that make  $r_i$  ineligible for satisfying  $q_j$ . We approach the modeling of these constraints in two primary ways: as additional requirements on top of the requests and as part of the resource model.

#### 3.4.1 Constraints as Requests

At the beginning of section 3.3, we discussed the different components of a request in our model. In some domains, required resource instances can have additional requirements that are not dependent on the mostly static *attributes* or often dynamic *capabilities* of a resource instance, but instead are based on run time assignments, i.e. allocation of resource instances at some specific execution state of the system. As presented in definition 2, this would mean that any such constraint specifies some

predicate  $P$  over the allocation of resource instances,  $ALLOC(s)$ , at system state  $s$ . To specify some of these constraints, we create a special type of request for identifying a group of resource instances. In our model, we denote such requests as  $q^{gc}$ , where  $gc$  indicates that this is a request for *group-constraint*. Like a regular request for a resource instance, a request specifying a *group-constraint* also describes a set of resource instances that satisfy some required criteria.

**Definition 13** *A group-constraint request,  $q^{gc}$ , is a predicate  $P_{q^{gc}}$  over all resource instances  $R$  of a domain  $D$ .*

To put it simply, a *group-constraint* just specifies a group of resource instances. Next we augment the group-constraint definition by limiting the size of the group that can be instantiated to satisfy a specific *group-constraint* request.

In our model, we call such a constraints a *restricted-group-constraint*. Like a *group-constraint* request, a *restricted-group-constraint* specifies which resource instances can be part of an instantiated group that can satisfy the constraint; and it also specifies the maximum allowed cardinality of such a group.

**Definition 14** *Let  $P_{q^{gc}}$  define a set of resource instances where  $P_{q^{gc}}$  is a predicate defined over all resource instances  $R$  of a specific domain. A restricted-group-constraint request,  $q_n^{gc}$ , specifies a set of resource-instance sets  $R_{q^{gc}} = \{R_1, R_2, R_3 \dots R_n\}$  where each  $R_i$  represents a set of resource instances and  $R_i$  is an element of the power set  $P_{q^{gc}}$  and cardinality of  $R_i$  is less than or equal to  $n$ . More concisely:*

$$q_n^{gc} = \{R_i\} \text{ where } R_i \in 2^{P_{q^{gc}}} \text{ and } |R_i| \leq n$$

In our resource request model, we allow specification of a request  $q$  optionally augmented by the specification of a *group-constraint* or a *restricted-group-constraint*. If a resource request  $q_i \in Q$  is accompanied with (i.e. constrained by) a *group-constraint*  $q^{gc}$ , the Resource Manager needs to first instantiate a group of resource

instances that satisfies  $q^{gc}$  and then limit the set of eligible resource instances that can satisfy  $q_i$  by making sure that all eligible resource instances are members of that group. Thus, a request  $q_i$  constrained by a *group-constraint*  $q_{gc}$  can only be satisfied by a resource instance  $r$  such that

$$r \in P_{q_i} \text{ and } r \in P_{q_{gc}}$$

Corresponding to the *group-constraint* and *restricted-group-constraint*, we have two more constraints: *iterator-constraint* and *restricted-iterator-constraint*. An *iterator-constraint* is exactly same as the *group-constraint* with the added restriction that each resource instance in an instantiated group that represents that *iterator-constraint* can be used only once to satisfy a request that is constrained by the *iterator-constraint*. In other words, if a resource instance from an instantiated *iterator-constraint* is used to satisfy a request, that resource instance is marked to be used and is taken out of the group (collection) that got instantiated from the *iterator-constraint*. *Restricted-iterator-constraint* augments the *iterator-constraint* by placing a maximum cardinality for any group or collection of resource instances instantiated from the *iterator-constraint* specification.

In chapter 6, we present examples of how such constraints in requests can allow for flexible modeling of various complex domain policies regarding resource assignments. As an example, consider the case of two requests associated with two separate task instances both of which must be satisfied by using the same resource instance. We can specify this by using a *restricted-group-constraint* as defined above.

### 3.4.2 Constraints as Part of the Resource Model

The second way we model constraints on resource assignments is by using the specification of guard functions on active capabilities of resource instances  $r$  in system state  $s$ ,  $ACTCAPS(r, s)$ . In this case, we model a set of system states  $S_{CAP}^r$  such

that a capability  $CAP$  of resource instance  $r$  ( $CAP \in CAPS(r)$ ) is active if and only if system state  $s \in S_{CAP}^r$ . Thus we have the following definition:

**Definition 15** *A guard function  $G_{CAP}^r$  is a mapping from the domain of all resource instances  $R$  and all capabilities  $CAPS(R)$  (i.e.  $R \times CAPS(R)$ ) to a set of systems states  $S_{CAP}^r$  such that*

$$CAP \in ACTCAPS(r, s) \Leftrightarrow s \in S_{CAP}^r$$

To define the set of system states  $S_{CAP}^r$  in which *capability*  $CAP$  is active for resource instance  $r$ , we may need a number of pieces of dynamic state information from a running system. It is worthwhile to specify some examples of the types of information we model and use as part of defining the guard functions.

- *Available Capacity Information:* A guard function can use the available capacity information of a resource instance  $r$  at state  $s$ . Sometimes for example, there is a need for such information as the number of resource instances that are available from a group of resource instances. This allows us to specify if none of the members of a group is available at some system state  $s$ .
- *Pending Requests Information:* When blocking requests are used, a very useful way to model the state of the system is by specifying a Boolean function over the number of pending requests. These pending requests can be for a type of task (i.e. for a set of task instances  $\{t_1, t_2, \dots, t_n\}$ ), for a specific resource instance  $r$ , or for a set of resource instances.
- *Stages of the Tasks:* Let us suppose the task instances generating resource requests go through a life-cycle that contains states such as: ready to be worked on (instantiated), getting worked on (started), getting completed (completed) etc. Under such a scenario, it is also important to model a set of system states

based on the state that a specific task instance  $t$  is in or a set of task instances  $\{t_1, t_2, \dots, t_n\}$  are in.

Our research shows that capturing system states in that ways suggested by the above examples is remarkably powerful. Chapter 6 will present some specific examples of guard functions defined using this approach.

### 3.5 Allocation Decision

This dissertation explores the resource allocation scenario where the **Resource Manager** receives a stream of requests for resource instances and tries to satisfy the requests as they are received. In section 3.3 and 3.4, we presented our approach for modeling the specification of required characteristics, capabilities, capacity, and constraints as part of a resource request. We also specified additional constraints that can be specified as part of the resource model and restrictions on the required protocol for satisfying the resource requests. The last major component of a resource management service that we model is the approach to making the decision about assigning a resource instance to fulfill a request. While describing the generic resource management problem at the beginning of this chapter, we presented the primary objective that forms the basis upon which such allocation decisions are made. Here we present the high level process we propose for making these allocation decisions.

Given a request  $q$ , to make a decision to bind a resource instance  $r$  to  $q$ , we follow a three step process: 1) Matchmaking, 2) Filtering, and 3) Selecting:

1. *Matchmaking*: This step refers to the identification of all the resource instances that can satisfy  $q$  based on only the required characteristics of resource instances in  $q$ . If required characteristics of resource instances are specified using a predicate defined over the static attributes of resource instances, then this matchmaking is a simple exercise of evaluating that predicate regardless of the

current state of system  $s$ . On the other hand, if the requirements in  $q$  are specified in terms of the dynamic characteristics of potential resource instances, this matchmaking would entail evaluation of the predicate given the current system state. At the end of the match making exercise, we shall have a set of potential resource instances that matches the required characteristics criteria in  $q$ . Let us denote this set of eligible resources as  $R_m^q \subseteq R$ , where  $R$  is the set of all resource instances of a domain under consideration. We note that  $R_m^q$  may be empty at a particular system state  $s$ . In that case the following two steps would become unnecessary and the resource manager will respond based on the protocol specification for satisfying requests.

2. *Filtering*: This is a two step process. In the first step of filtering, we remove from  $R_m^q$  the resource instances that do not have the required capacity requested by  $q$ . In the second step we apply the additional constraints (e.g. *group-constraint* or *restricted-group-constraint*) that may be associated with  $q$  and remove any resource instance that does not satisfy the additional constraint. This filtering, we shall see in chapter 6, allows us to model some interesting domain policies. Let us denote the set of resource instances at the end of this step by  $R_{mf}^q$ . We note that  $R_{mf}^q \subseteq R_m^q$
3. *Selecting*: If the cardinality of the set  $R_{mf}^q$  is more than one, this final step of allocation decision process comes into play. Our approach to the selection process is to model it as an auction. Each resource instance  $r \in R_{mf}^q$  will compute a bid based on such information as its *cost*, *service-quality* for satisfying  $q$ , *required capacity* as a fraction of *available capacity* etc. We would model the winning criteria for the bids based on the specific utility function  $UTIL$  that we are trying to optimize for any specific execution of a system for which the resource management service is being provided. In our Resource Manager

architecture this selection process is cleanly separated so that it can be easily customized for different domains and applications.

## CHAPTER 4

### RESOURCE MANAGER ARCHITECTURE

The general resource management problem incorporates a number of complex areas with challenging issues. An architecture in which the various components are assigned clear responsibilities thus would seem to provide an attractive basis upon which to develop a comprehensive and flexible resource management service. This chapter first presents the high level architecture of such a proposed resource management service. The discussion then elaborates the architecture by focusing on each of the different separate concerns. We then describe our **Resource Manager** prototype and the application programming interface (API) specifications of its different components.

#### 4.1 Overall Architecture

Figure 4.1 shows the overall architecture of our proposed resource management service. There are four major components:

- Request management component
- Repository management component
- Allocation management component, and
- Constraint management component

In addition to these four major components, there is also a *System State* component that interacts with the different components to maintain an accurate and current



representation of the execution state of the **Resource Manager** while it is meeting the needs of the currently executing system. This state information is also used by some of the other components. Figure 4.1 also shows an abstract representation of a **Resource Client** component, which is not part of the core **Resource Manager**, but is shown to help represent the nature of the interactions between the resource manager and a client whose resource needs are being met. Additionally, there are two important sets of data objects in the described system architecture:

- Resource Requests, and
- Resource Instances

In figure 4.1, dashed-line boxes represent the major components, while solid-line boxes inside the components represent different sub-components of the resource management service. The directed edges represent calls from one sub-component to another. The edges have been labeled with phrases to describe the purposes of the calls. Component names are shown in bold faced and sub-component names have been italicised. Both component and sub-component names have been capitalized.

A **Resource Client** sends requests to the **Request Management** component for resource related services. We view the resource client shown here as an abstract representation of a task for which resources are required. Thus the **Resource Client** component represents the user of the resource management service, and as such is not part of the **Resource Manager** itself. In our abstract architecture we assume that the client may request such services as identification, reservation, acquisition, or release of resources. Details of this component are discussed in section 4.2.2.1

The *Request Processor* is one of the major sub-components of **Request Management** component. It receives requests from clients, processes these raw requests into meaningful resource queries, and places the queries into a pool of *Outstanding Requests*. The *Request Scheduler* sub-component decides which request or set of re-

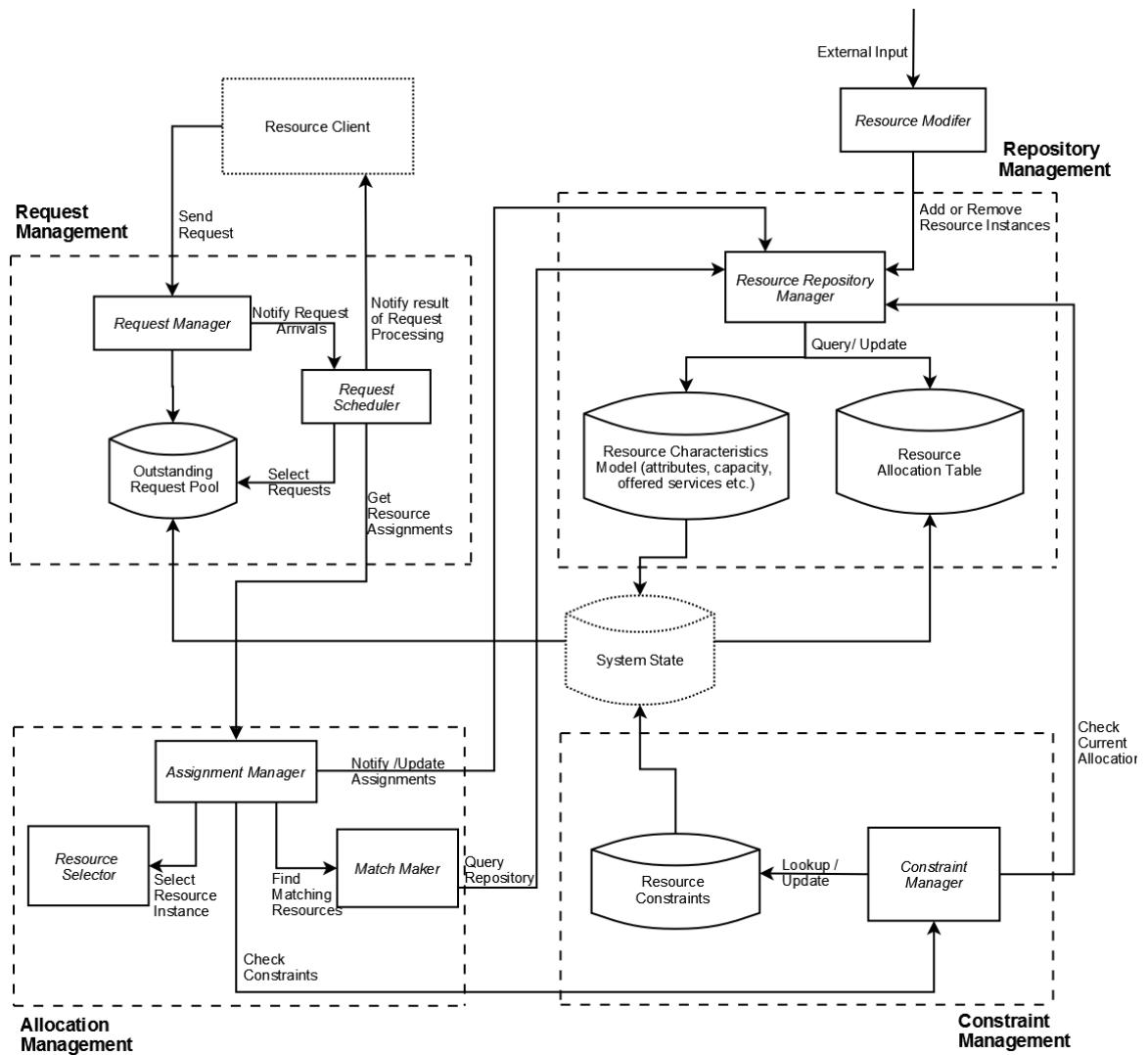


Figure 4.1: Resource Manager Architecture

quests to satisfy next and picks them up from the pool of *Outstanding Requests*. Requests from this pool are selected based on the requests' priority information. The priority of a request can be dependent on a number of factors including the priority of the requesting entity, i.e. the specific *Resource Client*, the parameters associated with the request, such as the request arrival time, how long the request is waiting in the queue etc. Note that the requests in the *Outstanding Requests* pool can be individual requests or sets of requests grouped together. Often such groups of requests would need to be fulfilled atomically to ensure that all resource requirements of a *Resource Client* are satisfied such that the task instance represented by this *Resource Client* can move forward. By deciding the order of the processing of requests, the *Request Scheduler* essentially decides on the schedule by which task instances are enabled, and thus is a major factor in deciding the overall progress of execution of the system that is issuing resource requests.

The *Allocation Management* component is composed of three sub-components: an *Assignment Manager*, a *Resource Selector* and a *Match Maker*. The *Request Scheduler* sends to the *Assignment Manager* selected requests that are to be fulfilled. The *Assignment Manager* responds by creating an *assignment*, which is a binding of a resource instance to the client's request. Requests can be for such services as finding out about the satisfiability of a request, for making an assignment, or for releasing an assignment. The *Assignment Manager* performs a number of operations in order to fulfill a request and return its responses. To fulfill requests asking for new assignments, the *Assignment Manager* passes the requests to the *Match Maker* sub-component. The *Match Maker* treats each request sent to it as a *query* against the information contained within the resource repository in order to find out which resource instances are able to satisfy the request. As discussed in chapter 3, in our resource model, the ability of a resource instance to provide a service to fulfill a request is often dependent on the dynamic state of the running system. Thus determining if

a resource instance is able to satisfy a request may require, among other considerations, evaluation of the **System State**, in order to ascertain which of the *capabilities* of a resource instance (i.e. the resource instance's *services*) is available given the current state of the system.

The **System State** consists of such information as the current allocation of resources, the different types of outstanding requests waiting to be fulfilled, past assignments during the current execution of the system etc. Once the *Assignment Manager* has identified a set of resource instances that are candidates to fulfill a request, it passes the set through the *Constraint Manager* to filter out the resource instances that do not satisfy whatever constraints on resource allocation might currently be in place. We will discuss constraints and their role in resource management in more details in section 4.2.5. Finally, the *Assignment Manager* takes the set of resource instances that match the initial requirements specified by a request, and also satisfy any additional constraints, and sends them to the *Resource Selector* sub-component. This sub-component chooses a resource instance based on some sort of 'best-fit' criteria.

The **Repository Management** component is composed of four sub-components. The *Resource Characteristics Model* defines the attributes, capacity, capabilities or offered-services, cost, etc. of resource instances in the domain being served. This component is assumed to have been specified with the help of a system modeler having knowledge of the resource management system, and especially knowledge of modeling language in use. This model describes all the resource instances of concern in the domain of interest. The second sub-component inside the **Repository Management** component, *Resource Allocation Table*, holds the complete set of all current assignments of resource instances to requests. The *Resource Allocation Table* sub-component also assists in gathering such derived information as the state of all resource instances, including their availability at any point during system execution. The third sub-component is

the *Resource Repository Manager*. This sub-component uses the information held within other sub-components and updates them as necessary. All resource allocation related communication to and from the **Repository Management** component is performed through the *Resource Repository Manager* sub-component. The *Resource Repository Manager* can be thought of as a facade over each repository that is defined for a domain that is being served by the **Resource Manager**. There is one more sub-component named *Resource Modifier* which is responsible for providing an interface to enable external inputs to be used to effect the addition, removal, or modification of resource instances in the repository. The *Resource Allocation Table* sub-component keeps two types of assignment information: *reservation* and *allocation*. Both reservation assignments and allocation assignments bind resource instances to requests. This architecture of **Repository Management** allows multiple resource repositories to work concurrently and serve requests as a single resource management service.

In order to support resource allocation, this high level component architecture considers resource assignment to be items to be auctioned. Thus, in this conceptual framework, **Resource Client's** requests for specific services are considered to be items put up for bids, and resource instances place bids. We expect that a bid placed by a resource instance is to be computed by using such information as the resource instance's suitability, availability, and cost. If a resource instance is not currently available or does not meet the criteria specified in the request, its bid for that request will be zero. Similarly, if a resource instance is able to satisfy a request, it will send a bid corresponding to its appropriateness (*service-quality, cost etc.*) for serving the request. The *Resource Selector* sub-component can thus be thought of as an auctioneer that considers different possible resource assignments (i.e. bids) for a request or set of requests and makes an assignment based on some bid-measurement metric.

Once an allocation decision is made by the Allocation Management component, the *Resource Repository Manager* is updated to reflect the new assignments. The new resource assignments are also sent back to *Request Scheduler*, which makes a call back to the *Resource Client* to report the assignment decisions. When a *Resource Client* no longer requires any reusable resources that it has been assigned, it sends notifications with usage information back to the Repository Management so that they can be released. The released resource instances' allocation and availability information are updated accordingly by Resource Repository Manager.

## 4.2 Resource Manager Modules

In earlier discussions, we established that the primary responsibilities of a resource management service include to create and manage a model of resource instances of a domain, to provide a structure and an interface for specifying resource-requests, to make decisions about which resource instances are to be bound to which requesting entities, which are abstract instances of tasks, and to keep track of resource states in a runtime environment. The binding decisions often need to satisfy some constraints and are usually intended to optimize some utility function. With these objectives, and considering the suggestions of our overall architecture presented in section 4.1, we have developed a prototype resource management service, ROMEO, centered on the following architectural components, each of which is designed to focus on a following key resource management concern:

- Resource model
- Request model
- Repository management
- Allocation decision

- Constraint management

The following sections elaborate the responsibilities of these concerns and discusses the application programming interface (API) specification of different components that are used to address each of these concerns.

#### 4.2.1 Resource Model

The Resource Model defines the static structure of the collection of resource instances available for use by processes defined in a particular domain. It captures at least a partial specification of the dynamic behavior of each resource instance. In our approach, we have defined resource instances broadly; our view is that they are uniquely identifiable objects, each of which has a set of *Attributes*. An *Attribute* is a *name-value* pair that describes some characterizing information about the resource instance. Figure 4.2 shows the class diagram of a simple resource instance and its attributes.

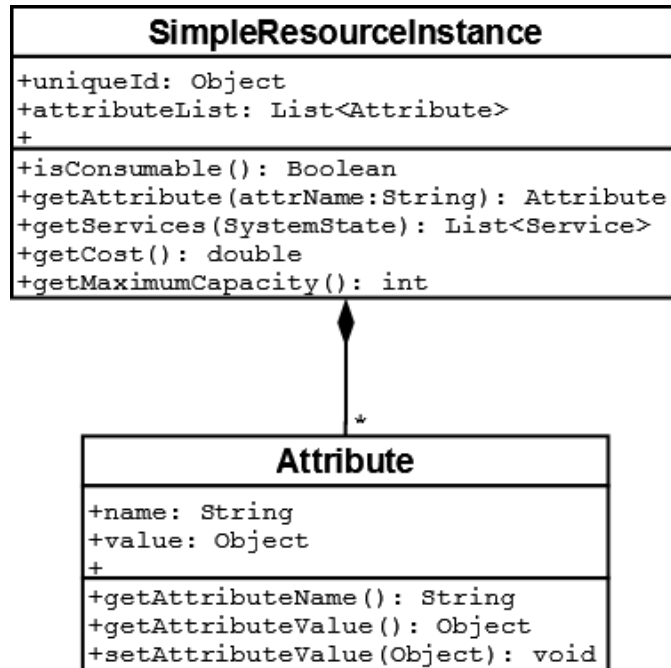


Figure 4.2: Class diagram of a simple resource instance

Note in particular that this view does not treat resource instances as comprising a type hierarchy. Our experience has indicated that simple type hierarchies are not sufficient to clearly and completely represent the complex relations among resource instances in domains such a hospital ED. Multiple inheritance schemes proved quite ungainly as well. Accordingly we have settled upon the above definition, which provides a flat structure for the resource instances in a domain. This definition also makes minimal assumptions about what attributes are most appropriate for describing different resource instances in the domain. While there seems to be a need to allow some resource instances to be described with the help of special purpose attributes, our research has also indicated that there are certain attributes that seem to be needed in common by all resource instances of many domains. We have captured these common attributes explicitly in our model. Following is a list that describes these attributes:

- *Consumability*: A resource instance can be either *reusable* or *consumable*. We have added an explicit method, *isConsumable()*, to the base *ResourceInstance* class that identifies whether a resource instance is consumable or not. The allocation and management of these two categories of resource instances differ from each other substantially.
- *Offered Services*: In our model, each resource instance is capable of offering a set of services. These services corresponds to the *resource capabilities* or *CAPS(*r*)* that we defined in chapter 3. The set of services is used to determine which resource requests a particular resource instance can satisfy. This determination may depend upon such factors as the resource client that generated the request and the dynamic state of the executing system. We have designed *getServices(*SystemState*)* to be a method that can be applied to all resource instances. This method accepts a *SystemState* object as an argument and returns the set



of services offered by the resource instance depending on the current state of the running system.

- *Cost*: The use of a resource instance usually has some cost associated with it. We have modeled cost as a common attribute of resource instances by adding a *getCost()* method that can be applied to all resource instances.
- *Capacity*: The notion of capacity represents a resource instance’s ability to satisfy a number of requests simultaneously. As discussed in chapter 3, this notion is used to specify the quantity of consumable resource that a resource instance can make available. As the resource instance is assigned to a task, the quantity of available resource declines by an amount that may depend upon the task, the system dynamic state, and the nature (e.g. the skill level) of the resource instance itself. When the capacity of the resource instance drops to zero, the resource instance is no longer available for allocation, at least until one or more of its current tasks has been completed.

There are clearly other possible attributes that are likely to be useful in supporting the specification of some resource instances in a specific domain. In the API design of our resource management service, we have tried to be explicit about only the most common methods. But we have also provided the capability to attach additional attribute types to individual resource instances, and to both set and get the values of these attributes.

#### **4.2.2 Request Model**

Modeling requests for resources is closely related to, but a separate concern from, the modeling of resource instances. The purpose of a **request model** is to support the expression of the request for (i.e. the requirements for) a resource instance. An important part of the ‘request model concern’ is the requesting entity, i.e. the resource

client. The following section describes the application programming interface (API) for resource clients.

#### 4.2.2.1 Resource Client

In the architecture presented in this dissertation, the client of a resource management service (i.e. the requesting entity) has been considered to be an abstract representation of a task instance. Resources are required for activation and completion of instances of tasks and resource instances get bound to a task instance as a result of requests communicated to the **Resource Manager**. In the following discussion, the term *resource client* has been used to represent such a requesting task instance. Figure 4.3 shows the abstract interface of such a resource client.

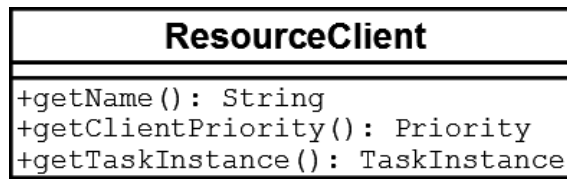


Figure 4.3: Resource Client API

- *getName()*: This method returns the name of the requesting resource client. Since we have defined a resource client to correspond to an instance of a task, this method may return the task name. Thus it might be the case that, different resource clients may have the same name (e.g. if a given task instance requires more than one resource instance).
- *getClientPriority()*: This method returns the priority associated with a resource client relative to other clients sending requests to the resource manager. This priority corresponds to the priority associated with a task type as opposed to a task instance. Note that different resource clients that corresponds to different instances of the same type of task will nevertheless have the same priority, as

this is associated with the criticality of the type of task to be done. For example, in the hospital ED domain, suppose that `Assess-Patient` and `Discharge-Patient` can both be done by the same resource instance (e.g. a doctor). In this case, the client priority may turn out to be useful in influencing the order in which requests originating from these task types are considered for assignment to resource instances. The higher priority task type will be considered first, and may result in delay in being able to assign a resource instance to the lower priority task.

- `getTaskInstance()`: This method returns the instance of the task that this resource client represents. In this framework, two resource clients are defined to be the same if they represent the same task instance.

#### 4.2.2.2 Specification of the Required Resource

To provide additional flexibility in modeling resource requests, we also provide a mechanism that can be used to allow a resource client to specify as part of a request some characteristics that are desired in the resource instance that is being sought. Thus, for example, this feature can be used to specify that a surgeon to be assigned to a task should have more than some specified number of years of experience. In addition to specifying these desired characteristics, additional requirements, in the form of constraints can be associated with a resource request. We discuss this in more detail in section 4.2.5 (constraint discussion section). Figure 4.4 shows the application programming interface we have used in our prototype implementation for supporting these specifications of a required resource instance.

There are two ways we allow specification of required resource instances:

1. Through *queries* that specify the characteristics of resource instances, and
2. Through specification of a required service

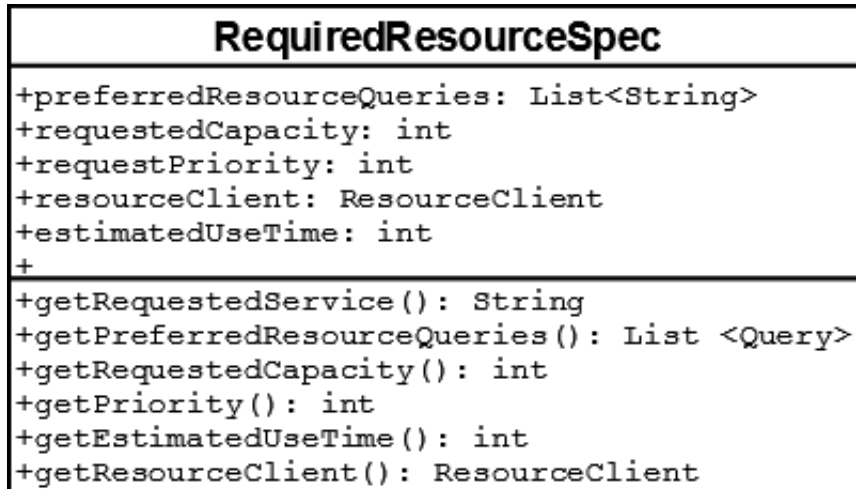


Figure 4.4: Required resource specification class diagram

For *query* based specification, we allow a resource client to specify a preferential order. When a request is processed, our resource management service attempts to satisfy the request with preferred characteristics. If it is unable to do so, any resource instance that can provide the required service is then considered. Figure 4.4 shows the API for specification of required characteristics of a resource instance that is being requested.

- *getPreferredResourceQueries()*: This method returns an ordered set of preferred *queries* specified by a resource client. Each query is a specification of a set of required characteristics that is being requested. An example of a preferred list of queries in an ED domain would look like the following: *prefer(AttendingMD, Resident, PhysicianAssistant)*.
- *getRequestedService()*: This method specifies the service or capability that is being sought. Any resource instance capable of providing this service at the time when it is being requested is eligible for satisfying this request.
- *getRequestedCapacity()*: This method specifies the available *capacity* requirement of the resource instance that is being requested.

- *getPriority()*: This method returns the priority associated with a request. This priority is associated with a specific task instance that is generating the request. Note that it is different than the *Resource Client* based priority. In chapter 5, we will discuss the utility of allowing these two different forms of priority specification. When both sorts of priority information are available, either the instance level priority (i.e. the one returned by this method) takes precedence or a weighted average of the two priorities can be used. The decision about which approach to use is determined as part of the specification of a specific resource management implementation.
- *getResourceClient()*: This method returns the resource client (task instance) that has generated this request.
- *getEstimatedUseTime()*: This method holds an estimation of the time a resource instance is likely to require in order to satisfy this request. This information is presumed to be useful in guiding the *Resource Selector* subcomponent of our resource manager architecture in making allocation decisions.

### 4.2.3 Repository Management

Section 4.1 presented the overall high-level architecture of our proposed resource management service. *Resource Repository Manager* was shown there as a primary component. In this section, we present the corresponding API for the prototype implementation of **Resource Repository Management** and other sub-components it interacts with in our prototype implementation.

Figure 4.5 shows the classes that make up the interface of a *Resource Repository Manager*. These classes represents the operations provided by *Resource Repository Manager* for adding, removing, modifying, and assigning resource instances that are located in the repository that it represents. The following list elaborates these operations:

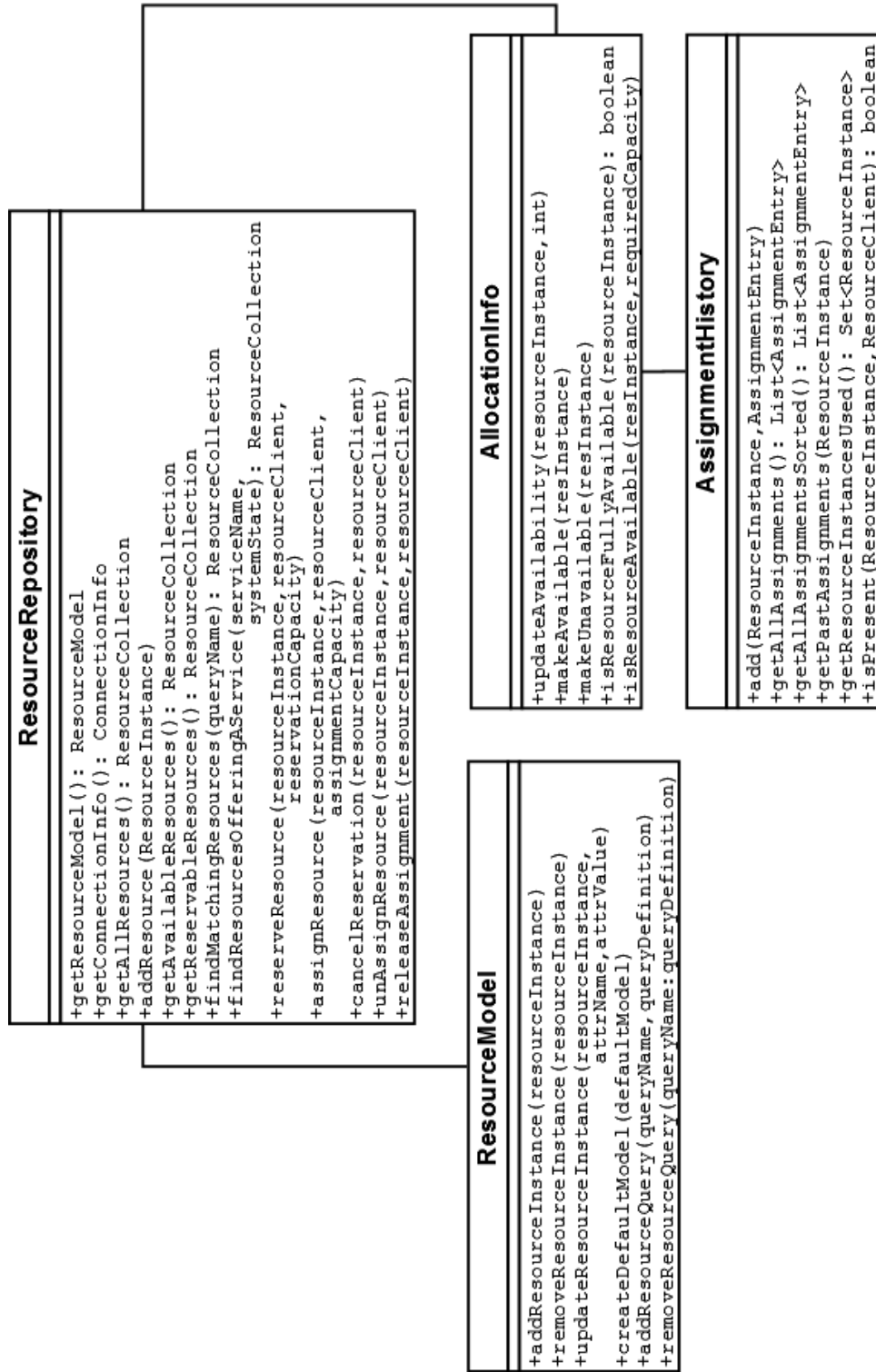


Figure 4.5: Resource Repository Manager Class Diagram

- public abstract String *getRepositoryName()*

This method returns the name of the repository it represents. In our prototype implementation, we implemented this repository as a relational database. Thus, for this implementation, this method returns the name of the database that holds the resource instance descriptions.

- public abstract AllocationInfo *getAllocationInfo()*

This method returns the AllocationInfo object, which is a complete summary of the current state of the allocation of all resource instances in the resource repository.

- public abstract ResourceCollection *getAllResources()*

This method returns all resource instances defined in this repository as a collection of SimpleResourceInstance objects.

- public abstract ResourceCollection *getAvailableResources()*

This method finds out and returns all resource instances that are currently available for allocation and returns them as a collection of ResourceInstance objects. A resource instance can be partially or fully available. This method returns the set of resources that have any available capacity.

- public abstract ResourceCollection *getReservableResources()*

This method returns all resource instances that are currently available for reservation assignments. We have designed an API that supports the reservation of a resource instance, in addition to other facilities for the allocation of a resource instance. We provide both facilities in order to support the additional flexibility of allowing a resource instance to be held out of consideration for allocation by clients other than the one that has previously reserved the resource instance. Accordingly, a resource instance can have *reservation capacity* that is different (usually higher) than its *allocation capacity*.

- public abstract ResourceCollection *findMatchingResource(Query)*  
Given a *query* that describes required characteristics of resource instances, this method returns a collection of resource instances that matches the *query* and are currently available in the repository.
- public abstract ResourceCollection *findResourcesOfferingAService(serviceName, systemState)*  
This method returns a collection of resource instances that are capable of offering a *service* under the given *system state*.
- public abstract void *reserveResource(resourceInstance, resourceClient, ReservationCapacity)*  
This method creates a binding between a given resource instance and a resource client for the purpose for reserving the resource.
- public abstract void *cancelReservation (resourceInstance, resourceClient)*  
This method breaks the reservation binding between a resource instance and a client and consequently frees up some reservation capacity of a resource instance.
- public abstract void *assignResource(resourceInstance, resourceClient, assignmentCapacity)*  
This method creates an allocation binding between a resource instance and a resource client for a given capacity.
- public abstract void *releaseAssignment(ResourceInstance)*  
This method releases an allocation binding between the given resource instance and resource client and records the binding information in the assignment history.
- public abstract void *unassignResource(resourceInstance, resourceClient)*  
This method releases an allocation binding between a given resource instance



and resource client but does not keep any record of the assignment. It is, as if the assignment never took place.

In addition to the operations listed above, the resource repository component provides some additional operations through other classes that it contains such as the *ResourceModel* and *AllocationInfo*.

- public abstract void *addResourceInstance(resourceInstance)*  
Given a *ResourceInstance* object, this method adds it to the repository.
- public abstract void *removeResourceInstance(resourceInstance)*  
Removes a resource instance from the repository.
- public abstract void *updateResourceInstance(resourceInstance, attrName, attrValue)*  
Updates some characteristics of an already defined resource instance based on the given attribute name and attribute value.
- public abstract void *addResourceQuery(queryName, queryDefinition)*  
Adds a new resource query into the repository. A request specifying the required characteristics of a resource instance needs to specify only the name of a *query*. A *query-definition* associated with this *query* is assumed to have been pre-defined and stored in the repository. An example of a *query* for ED domain could be an *AttendingMD* that is defined to be a specific collection of *attribute-names* and *attribute-values* that is considered essentially to comprise the definition of what this term means in this domain. Intuitively this capability supports the specification of something that serves some of the purposes of a type definition.
- public abstract void *removeResourceQuery(queryName, queryDefinition)*  
Removes an existing resource query and definition from the repository.
- public abstract void *updateAvailability(resourceInstance, availableCapacity)*  
Sets the availability of a resource instance to the given amount. This method

is required to update the current availability of a resource instance whenever a resource instance is assigned or released.

- public abstract void *makeAvailable(resourceInstance)*

Makes a given resource instance available with default available capacity.

- public abstract void *makeUnavailable(resourceInstance)*

Makes a given resource instance completely unavailable for any assignment.

- public abstract boolean *isResourceAvailable(ResourceInstance, requiredCapacity)*

Given a *ResourceInstance* object, this method returns a Boolean value depending on whether or not the resource instance is both available and has the required capacity for an allocation assignment.

#### 4.2.3.1 Managing Multiple Repositories

ROMEIO is architected to support managing multiple repositories of resource instances. There is a simple interface that allows requests for resource instances to be sent to a *MultipleRepositoryManager*. This class, as shown in figure 4.6, receives requests, identifies which repository a request is for, and forwards the request to the appropriate repository. Since the primary responsibility of the *MultipleRepositoryManager* class is to forward requests to the appropriate repository, its interface also defines all the operations that are specified in *ResourceRepository* class.

#### 4.2.4 Allocation Decision

As discussed in section 4.1, the final allocation decision is made by the *ResourceSelector* subcomponent. The abstract API for this class is quite simple. Figure 4.7 shows the operations we have defined in the *ResourceSelector* class of our prototype resource manager implementation.

- public abstract *SelectionPolicy getSelectionPolicy()*

This method returns the current selection policy used by the *Resource Manager*

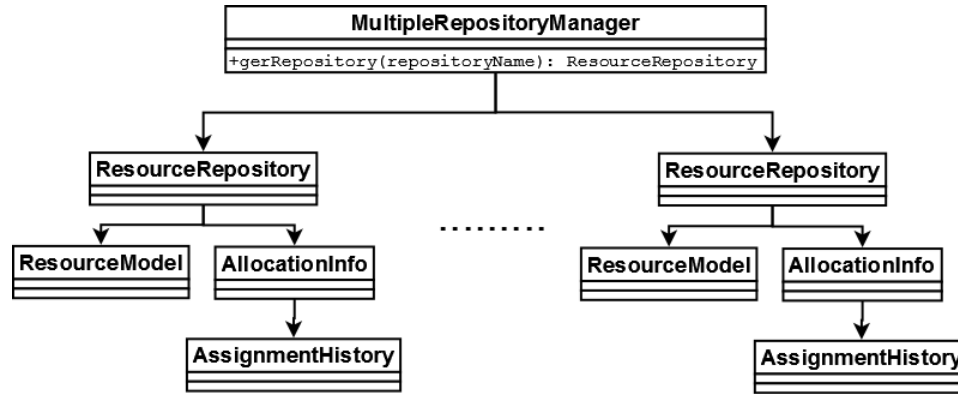


Figure 4.6: Multiple Repository Manager

to make resource selection decisions. The selection policy could be as simple as being *random* or as complex as an elaborate *auction* mechanism.

- public abstract ResourceInstance *selectOneResource* (*ResourceCollection*, *ResourceRequest*)

Given a collection of candidate resource instances (*ResourceCollection*), this method selects one resource instance to satisfy the resource request using the current *selection policy*.

- public abstract ResourceInstance *selectOneResource* (*ResourceCollection*, *ResourceRequest*, *SelectionPolicy*)

As in the case of the previous method, given a collection of candidate resource instances, this method selects one resource instance from the collection based on the given selection policy.

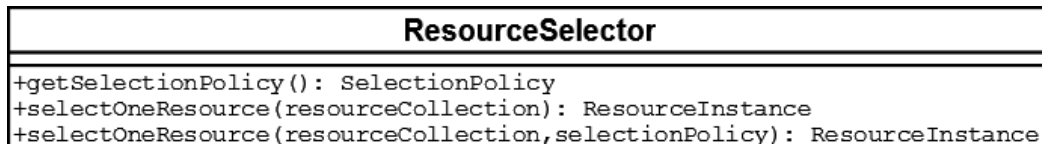


Figure 4.7: Resource Selector API

#### 4.2.5 Constraint Management

Our research indicates that many and varied types of constraints may constrain the ways in which resource instances may need to be selected in satisfying clients' requests. Some constraints may be best specified in a resource characteristics model, others might be best specified as additional requirements that are included as part of a request specification. Many constraints encountered in this dissertation work are de-facto representations of domain-specific policies that are intended to govern the overall allocation of resources in a running system.

In section 3.4, we discussed our approach to capturing constraints as additional requirements associated with requests. We introduced and defined the notion of *group-constraint* and *restricted-group-constraint*. Here we present the API for implementing those constraint specifications. Here, `ResourceCollectionSpec` represents a class for implementing both types of constraints mentioned above.

<b>ResourceCollectionSpec</b>
<code>+resQueryNames: Set&lt;String&gt;</code> <code>+innerConstraint: ResourceCollectionSpec</code> <code>+maxCardinality: int</code> <code>+</code>
<code>+getResourceQueryNames(): Set &lt;String&gt;</code> <code>+addResourceQueryName(queryName:String)</code> <code>+getRepositoryName(): String</code> <code>+getInnerConstraint(): ResourceCollectionSpec</code> <code>+setInnerConstraint(ResourceCollectionSpec)</code> <code>+getMaximumCardinality(): int</code> <code>+setMaximumCardinality(maxCard:int): void</code> <code>+getResourceCollection(): ResourceCollection</code>

Figure 4.8: Resource Collection Constraint Specification

Earlier we noted that it seems convenient to use a set of *attribute-name*, *attribute-value* pairs to define something that is somewhat similar to a type. This is done by treating this set of pairs as a set membership constraint that can then also function as a query on the resource repository. Thus, we create a set of constraints that

are predefined resource queries, name them, and store them in the repository of resources. By naming one of these resource queries in a resource request we thereby constrain the resource instances returned to all be members of the set defined by this query/constraint. There is a one-to-one mapping for each query name to its definition. Some example query names from the ED domain could be *AttendingMD*, *Resident*, *MainEDBed* etc. There is a *query* definition associated with each of these *query* names.

According to the API shown in figure 4.8, you can specify a set of such query names to define a constraining resource collection. The resource instances that make up the constraining collection when the *group-constraint* is instantiated at run time is the *union* of resource instances resulting from each of the queries. An important feature of this constraint specification is that we have designed it have a recursive structure where a `ResourceCollectionSpec` constraint can be constrained by another `ResourceCollectionSpec` constraint. The following list elaborates the methods we have defined for `ResourceCollectionSpec`.

- *getResourceQueryNames()*: This method returns a set of one or more resource query names. If there is only one resource query specified in the set (e.g. *Nurse*), the resource instances that result from running that query over the repository constitute the constraining collection when the constraint is instantiated. When more than one query is specified, the union of the query results is used. For example, if two query names  $\{AttendingMD, Resident\}$  are specified as the query names, the resulting resource collection from this constraint would include all the *AttendingMDs* and *Residents* defined in the resource repository.
- *getRepositoryName()*: This method returns the name of the repository on which this constraint is defined.

- *getInnerConstraint()*: This method returns either `null` or another `ResourceCollectionSpec`. If this constraint specification is constrained by another constraint specification, then this method returns that constraining `ResourceCollectionSpec` object. This is the way in which this method uses the recursive nature of this constraint specification.
- *getMaximumCardinality()*: This method returns the value that is used to place an upper limit on the size of the resource collection resulting from this constraint. If the resource collection is larger than the specified maximum cardinality, an implementation specific strategy is used to filter out additional resource instances from the collection.
- *getResourceCollection()*: This method, when invoked, instantiates the collection of resource instances that meet the specifications of the specified resource queries as well as other constraining `ResourceCollectionSpec` that might be present as part of this specification. Until this method is invoked, the constraint remains only a specification of *group-constraint* or *restricted-group-constraint* without any resource instances getting identified and becoming explicitly a member of the resource collection represented by this constraint.

While using constraint specification as part of the request model, we have augmented the resource requests specified using `RequiredResourceSpec` to optionally refer to a `ResourceCollectionSpec` constraint, which in turn may have reference to other `ResourceCollectionSpec` objects that are recursively defined as inner constraints. Figure 4.9 shows the class diagram illustrating this design.

An example of how this constraint can be used will make the design specification clearer to understand. Let us suppose we have two sections of an ED: a *MainED*, where all patients can be treated and a *FastTrackED* where only patients with low acuity levels are treated. Let us suppose there are two query names referring to two

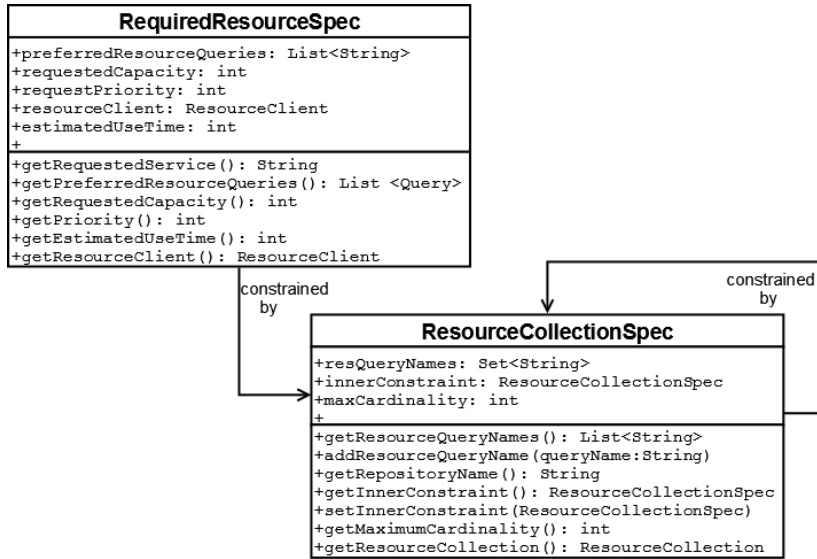


Figure 4.9: Specification of required resource with additional constraint

groups of resource instances in the resource repository: *AttendingMD*, and *Resident*. Let us also suppose that we define a resource constraint with query *FastTrackResources* which refers to a query that results in resource instances that are part of the fast track area of an ED. Let us now define a resource constraint composed of two query names: *AttendingMD* and *Resident* and a reference to *FastTrackResources* as an inner constraint specification. Let us name the constraint as *FastTrackMDConstraint*. With these definitions in place, if we specify a request for a *Doctor* constrained by *FastTrackMDConstraint*, it would specify a request for a doctor resource instance who is either an attending MD or a resident and who is currently working in the fast track section of the ED. In chapter 6, we shall present some additional examples of using such resource constraints and demonstrate their effectiveness in specifying complex domain policies.

## 4.3 Resource Manager API to support a task coordination framework

Earlier sections of this chapter have provided discussions about descriptions of the low-level components, along with their application programming interfaces(APIs), for our Resource Manager architecture. We have used these low-level APIs to design higher level resource management services that are suitable for use in supporting the execution of processes defined using a relatively generic task coordination framework. These higher level services offered by our Resource Manager make certain assumptions about the life-cycle of a task instance and the corresponding resource utilization life-cycle in such a task coordination framework. Next we discuss the notion of binding resource instances to task instances in the context of these life-cycle assumptions.

### 4.3.1 Agent and Non-agent resources

For this work we have adopted the assumption that there are two distinct ways in which a task instance may need to use a resource instance:

1. As an agent resource

**Definition 16** *We assume that every task in a task coordination framework requires one resource instance to serve as the entity that is responsible for effecting the performance of the task. This resource instance  $r_a$  that is made responsible for carrying out the task instance  $t$  is said to be bound to  $t$  as its agent. For every task instance, there is exactly one agent resource instance.*

2. As a non-agent resource

**Definition 17** *We assume that any task in our task coordination framework may require resource instances in addition to the agent resource instance. Such a non-agent resource is any resource instance  $r_i$  that is used by the agent resource*



$r_a$  in carrying out a task instance  $t$ . For every task instance  $t$ , there can be zero or more non-agent resource instances.

We assume that it is possible that there can be resource instances in a domain that cannot be made responsible for carrying out tasks and thus may not be allowed to be bound to task instances as agents. For example, in the hospital ED domain, a bed resource may never be made agent of a task instance. On the other hand, any resource instance  $r$  that can be bound to a task instance as an agent resource can also be bound to a task instance as a non-agent resource.

We also assumed that a task coordination framework would specify the temporal relationships among the tasks. In addition it is assumed that the task coordination framework would incorporate a task interpreter facility to supervise the execution of a task model, which would entail, among other things, assigning tasks to agent resources. In this model, an agent resource would, in turn, be responsible for notifying the task interpreter when it starts executing a task, when it completes the task, or when it terminates unsuccessfully its attempts to execute the task. For this arrangement to work, participating agent resources are expected to register themselves with the **Resource Manager** and agree to perform the assigned tasks following some established protocol of communication between the task interpreter and the agent resource instances.

It is further assumed that information about the agent resource instances and their capabilities are available to the **Resource Manager**. Further it is assumed that each agent resource instance maintains a ‘To Do’ list (we shall call them *agendas*) where task instances are placed once the **Resource Manager** has bound an agent resource instance to a task instance. Figure 4.10 illustrates this concept.

Here a *Task Coordination Model* is provided to the *Task Interpreter*, which is a central component of such a framework. The *Task Interpreter* instantiates tasks according to the coordination model specification, identifies the agent and other re-

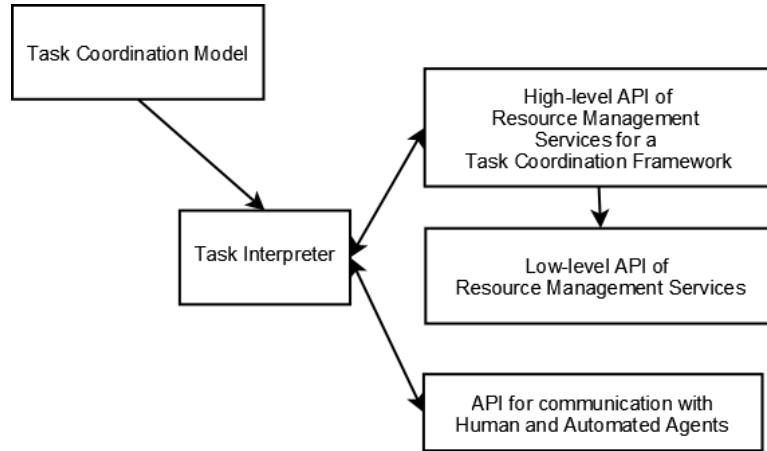


Figure 4.10: Resource Manager API for a Task Coordination Framework

source requirements for a task instance and uses that requirements information to invoke different resource management services. These resource management services are provided by components of our **Resource Manager** that implements the high-level API to support such task coordination.

Figure 4.11 shows the life cycle of a task instance in this framework. Once a task is *instantiated*, a suitable agent is identified, the *Task Interpreter* places the instantiated task on the ‘To do’ list of the agent resource that has been bound to the task instance. The **Resource Manager** provides this identification as a service, making the assignment of the task instance to the resource instance that will be working as the agent resource for this task. The task instance transitions into a *posted* state once it has been placed on the ‘To do’ list of the selected agent resource. It is then the responsibility of the agent resource to notify the *Task Interpreter* once it starts working on the task instance. At that time, the task instance transitions into the *started* state. The agent resource then either successfully completes the execution of the task instance, in which case it transitions into the *completed* state, or, in case the agent resource fails to complete the task instance and terminates it, the

task instance then transitions into the *terminated* state. Figure 4.11 shows the state diagram illustrating this life cycle of a task instance.

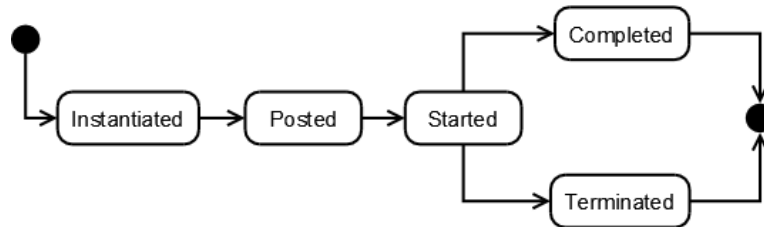


Figure 4.11: Life cycle of a Task instance

In this scenario, a set of potential agent resource instances is identified as a result of instantiating a task instance. If there is no identifiable resource instance that meets the specified requirements for a task instance, then the *Task Interpreter* can go no further with the instantiation process.

To support the execution of such a task instance life cycle, there needs to be a corresponding resource usage life cycle. The high level task coordination framework described above thus suggests that the resource usage life cycle be slightly different depending on whether the resource instance is bound to a task instance as an agent resource or as a non-agent resource.

In case a resource instance is bound to a task instance as an agent resource, one possible life cycle for the use of that resource instance is shown in figure 4.12. We shall use this usage life-cycle as an example to define a higher level API for our prototype resource management service.

As shown in figure 4.12, an agent resource instance, as part of its usage cycle, first becomes a member of a set of resource instances that has been identified as the set of candidates for becoming the agent for a given task instance. This transition happens as a result of an *identify* operation performed by the *Resource Manager*. Once the client has identified a set of potential agent resource candidates, it sends a

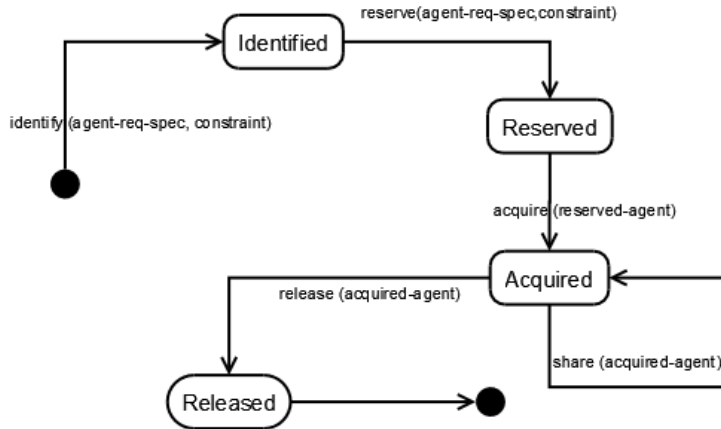


Figure 4.12: Life cycle for the usage of an agent resource instance

request to *reserve* one of these candidate resource instances. A successful *reservation* results in a binding between an agent resource instance and the task instance. In this scenario, a *reservation* is followed by an *acquisition* of the reserved agent. There is a subtle difference between a *reservation* and an *acquisition* in this task coordination framework. A *reservation* binding between a task instance and an agent resource indicates that the agent resource is willing to work on the task at some point. An *acquisition* binding refers to the scenario where an agent resource is bound to a task instance with the expectation that the agent is going to start working on it in the immediate future. The underlying assumption for making this difference is that an agent resource may have different capacity for reservation allocation and acquisition allocation. An agent may be willing to get reserved for  $m$  tasks, while it is capable of performing  $n$  tasks simultaneously. Here  $m$  refers to reservation capacity and  $n$  refers to allocation capacity and often  $m > n$ . When a resource instance is not getting bound to a task instance as an agent resource, it does not need to go through the reservation state. Figure 4.13 illustrates the state diagram that depicts the life cycle of a non-agent resource instance.

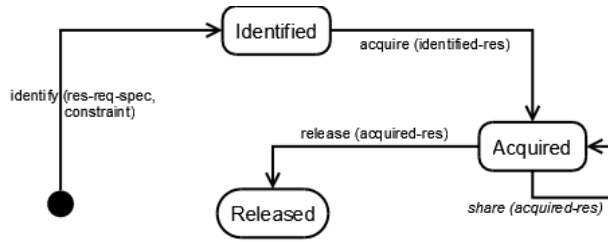


Figure 4.13: Life cycle for the usage of a non-agent resource instance

Figure 4.14 illustrates the relationship between the life cycle of a task instance and the usage of an agent resource instance by combining the state diagrams shown in figure 4.12 and figure 4.11. To distinguish the states of the two entities, the states of a task instance is shown in shaded rectangles whereas states of an agent resource instance have been shown in clear rectangles.

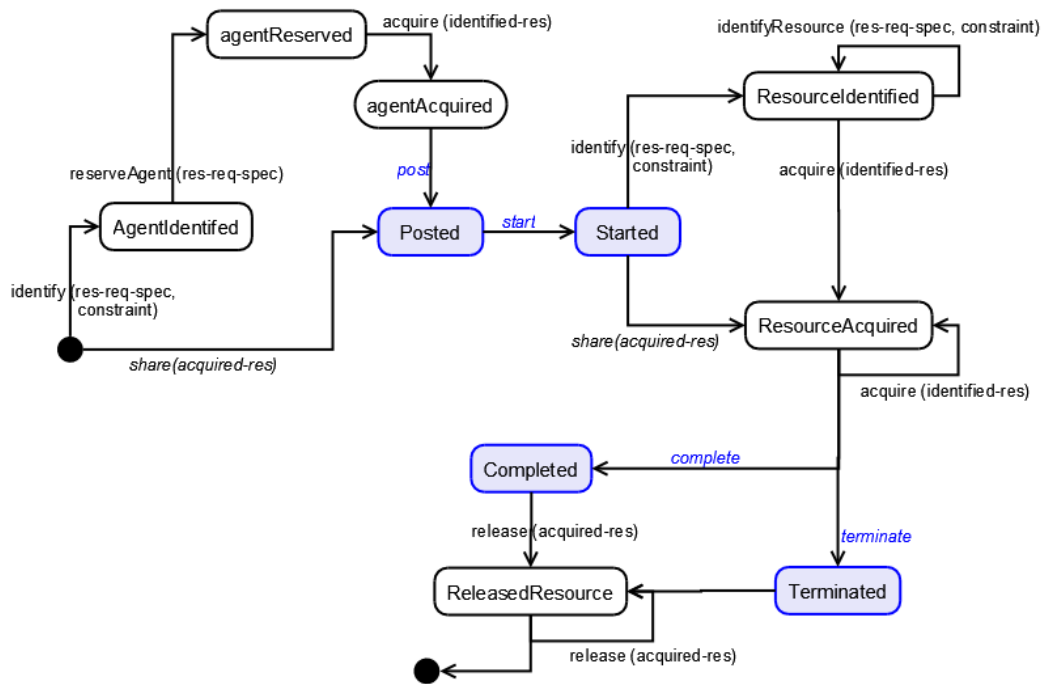


Figure 4.14: Combined state diagram of a task instance and an agent resource instance

The state transition events shown in figure 4.11 also defines the **Resource Manager** abstract API that would be needed to support such a task coordination model. The following discussion describes these operations. The API is primarily based on the agent and non-agent related operations between a **Task Instance** object and the **Resource Manager**. The protocol between the resource client (**Task Instance**) and the resource server (**Resource Manager**) is illustrated using a message sequence chart in figure 4.15. For a non-agent resource instance, the **Task Instance** sends the requests: *identify*, *acquire*, and *release*. For an agent-resource, the task instance sends the requests: *identify*, *reserve*, *acquire*, and *release*. There is also a request named *share*, which is mostly used for notification purposes from resource clients to the resource management service. Each message from a **Task Instance** to the **Resource Manager** results in a corresponding message back to the caller. All communication between the two entities are asynchronous.

Figure 4.15 captures the interactions between a resource client and the resource management service. The task coordination framework presented here expects the resource management service to process multiple resource requests together. Accordingly, we have specified the high level **Resource Manager** API such that it will process as a group all of the requests that have been sent once it has received a message from the client that all the requests in the group have indeed been sent.

- public abstract void *identify*(*TaskInstance*, *RequiredResourceSpec*, *ResourceCollectionSpec*)

This method describes a request to the **Resource Manager** to check for the existence of a resource instance, as specified in the resource request (using **RequiredResourceSpec** class). A request constraint can also be specified using the parameter **ResourceCollectionSpec**, which places additional requirements on the specification of the required resource instance. The first parameter of this method identifies the instance of the task that is the client for this request.

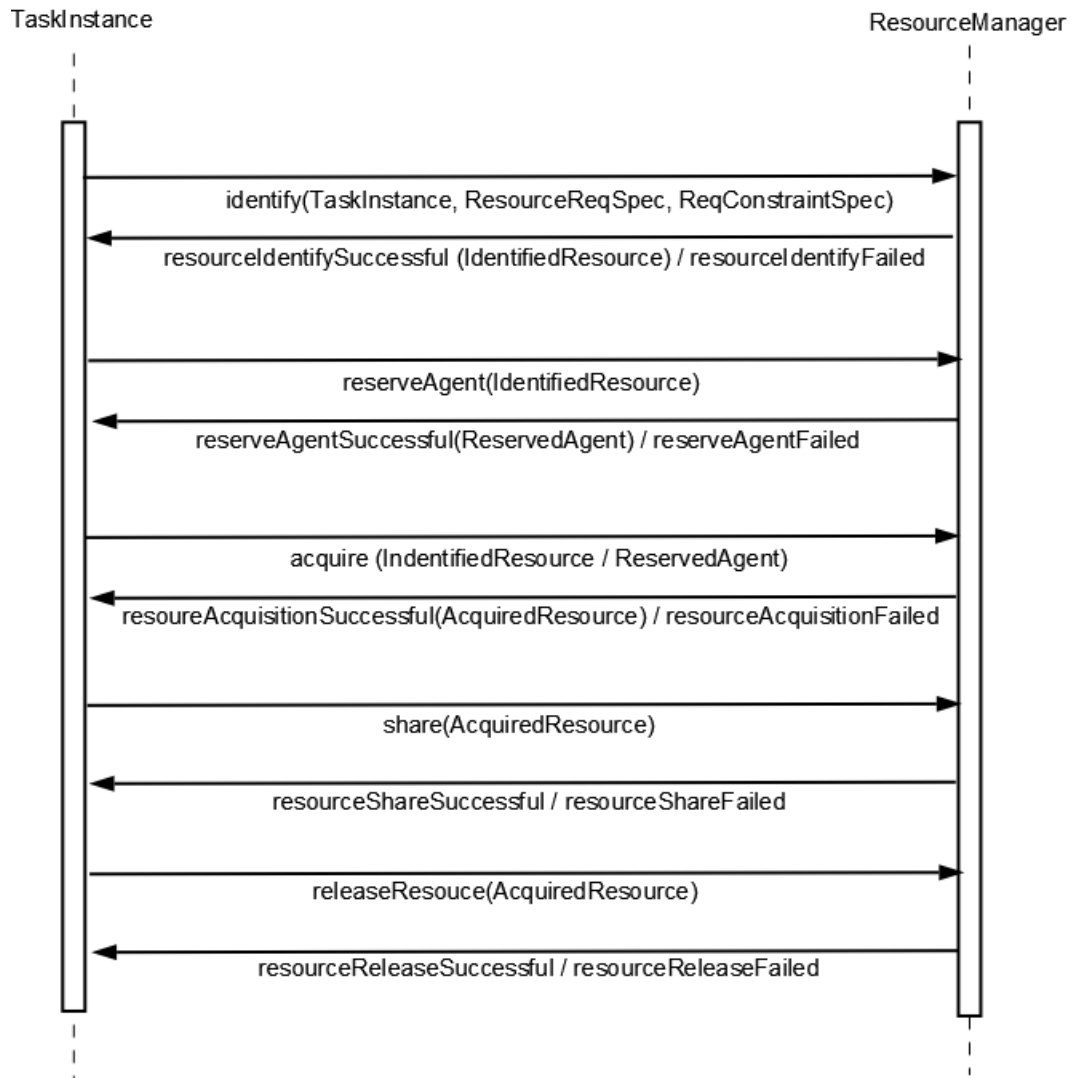


Figure 4.15: Message Sequence Chart defining the protocol between Task Instance and the Resource Manager

- public abstract void *reserveAgent(TaskInstance, IdentifiedResource)*

This method describes the reservation operation used to reserve an agent resource from an already identified set of resource instances. The resource manager notifies the **Task Instance** with a reservation successful or reservation failed message depending on the type of the call. If the method call is a blocking call, the **Task Instance** blocks until **Resource Manager** returns with an agent resource instance reserved to carry out this task.

- public abstract void *acquire(TaskInstance, IdentifiedResource)*

This method represents the operation of acquiring a resource instance that has already been identified. In the case of an agent resource instance, this acquisition happens after the reservation process, as shown in figure 4.15. The parameter passed to this method is a **Task Instance** object and an identified or reserved resource object that has been returned by the **Resource Manager** as a result of earlier method calls.

- public abstract void *share(TaskInstance, AcquiredResource)*

Once a resource instance has been acquired, it may be shared among multiple task instances. This method specifies that such a sharing operation of an **Acquired Resource** can be taken advantage of by a **Task Instance**.

- public abstract void *release (TaskInstance, AcquiredResource)*

This method is invoked at the end of a **Task Instance**'s life cycle and effects the release of a resource instance that had been allocated to the **Task Instance**. It is particularly important that the release method be used conscientiously at the end of execution of a task that has had any reusable resource instances allocated to it.

- public void *close(ThreadGroup)*

This method represents the closing of a set of requests that are to be processed



atomically (i.e. as a group). A parameter of type *ThreadGroup* is provided as part of this method. This parameter represents the thread group all these requests are a part of.

### 4.3.2 Request Structure to Support a Task Coordination Framework

To support the resource management services required by the task coordination framework introduced in section 4.3, we modeled a set of classes to represent different types of resource requests originating from a resource client. Figure 4.16 shows the set of classes we have defined for this purpose. The top level class is named *ResourceRequest*. It has five direct subclasses: *IdentifyRequest*, *AcquireRequest*, *ReserveAgentRequest*, *ShareRequest*, and *ReleaseRequest*. There are two subclasses of *AcquireRequest* class: *AcquireIdentifiedRequest* and *AcquireReservedRequest*. The classes have been structured to support the *command* [35] design pattern.

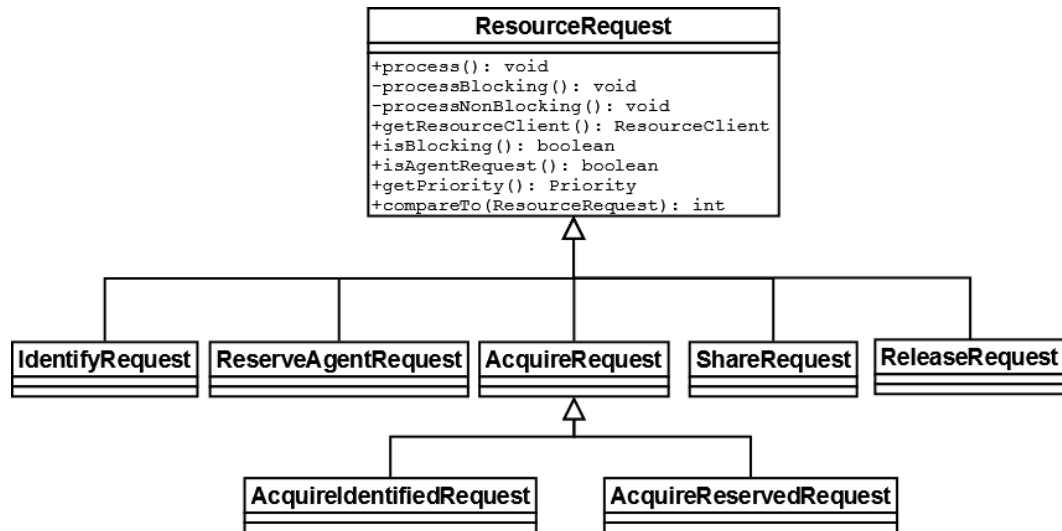


Figure 4.16: Resource Request Class and its subclasses

Each of these request classes represents a specific type of request enumerated in the task coordination model described in section 4.3. The base class’s *process()* method is the primary operation that each of the subclasses implement. In the above context,

each request for a resource management service could be blocking or non-blocking. To support processing of both these categories of request processing, each request class also implements two private methods: *processBlocking()* and *processNonBlocking()*. There is priority information associated with request classes. The classes also implement an interface that allows the priorities of two `ResourceRequest` objects to be compared to each other.

### 4.3.3 Support for Blocking Reservation and Acquisition

The preceding section introduced five different types of request classes capturing the different resource management services corresponding to the usage life cycle of an agent or non-agent resource instance. These requests are generated from the different resource clients in the course of execution of a task structure specified with the task coordination framework we have specified. We noted that some of these requests can be sent to the `Resource Manager` as blocking requests, and some as non-blocking requests. For *identify* and *share*, the processing is same for both blocking and non-blocking requests. It is the *reserve agent*, *acquire*, and *release* operations that require different processing depending upon whether the requests are blocking or non-blocking. To support blocking requests, we have designed a `Singleton` [35] class named `PendingRequests` that keeps track of all pending requests for the `Resource Manager` and processes the requests at appropriate times. Figure 4.17 shows the design of this class. There are two groups of requests maintained by `PendingRequests` that are waiting to be processed. The first one is a group of agent reservation requests and the second one is a group of acquisition requests. Since the task coordination framework described here requires only one agent resource instance but may require multiple non-agent resource instances for every task instance, the list of acquisition requests has been modeled as a group of acquisition request groups. This design also

supports atomic acquisition of resource instances. The following list elaborates the operations we have defined for this class:

<b>PendingRequests</b>
+pendingReserveAgentRequests: PriorityQueue<ReserveAgentRequest>
+pendingAcquireRequestGroups: PriorityQueue<AcquireRequestGroup>
+
+addAcquireRequestGroup(): void
+removeAcquireRequestGroup(): void
+addReserveAgentRequest(): void
+removeReserveAgentRequest(): void
+processNewRequests(List <ResourceRequest>): void
+processReserveAgentRequests(): void
+processAcquireRequestGroups(): void
+processReleaseRequests(): void
+getPendingAcquireRequests(): int
+getPendingReserveAgentRequests(): int

Figure 4.17: Resource Request Class and its subclasses

- `public void addAcquireRequestGroup(AcquireRequestGroup)`

The parameter `AcquireRequestGroup` represents a group of `AcquireRequests` as discussed in section 4.3.2. This method adds an `AcquireRequestGroup` to the group of acquisition requests that are waiting to be processed.

- `public void removeAcquireRequestGroup(AcquireRequestGroup)`

This method removes an `AcquireRequestGroup` from the group of waiting acquisition requests.

- `public void addReserveAgentRequest(ReserveAgentRequest)`

This method adds a new `ReserveAgentRequest` to the list of agent reservation requests that are waiting to be processed.

- `public void removeReserveAgentRequest(ReserveAgentRequest)`

This method removes an existing `ReserveAgentRequest` from the list of agent reservation requests that are waiting to be processed.

- `public void processNewRequests(List<ResourceRequest>)`

The parameter of this method is a list of `ResourceRequest` objects. This method processes a set of new requests sent by a `ResourceClient`.

- `public void processReserveAgentRequests()`

This method processes the set of agent reservation requests that are currently in the pending queue waiting to be processed. The order in which the pending requests are processed is implementation dependent.

- `public void processAcquireRequestGroups()`

This method processes the set of acquisition request groups that are currently in the pending queue waiting to be processed. An `AcquireRequestGroup` is made up of a set of resource acquisition requests originating from a single resource client. There are two important things to note about the implementation of this method. First, like the agent reservation requests, the order in which the acquisition request groups are processed is implementation dependent. Second, a specific implementation of this method will determine whether atomic acquisition of resource instances in a group are supported or not.

- `public void processReleaseRequests()`

This method specifies the operation of processing a set of requests for releasing resources. Depending on specific implementation, the release operation of a resource instance may result in the processing of a set of pending requests.

- `public int getPendingAcquireRequests()`

This method returns the number of acquisition request groups waiting to be processed by the `Resource Manager` at any point in time.

- `public int getPendingReserveAgentRequests()`

This method returns the number of agent reservation requests that are waiting to be processed by the Resource Manager at any point in time.

## CHAPTER 5

### EVALUATION SETUP

As part of this dissertation, we have developed a prototype resource management framework following the architecture we described in chapter 4. To evaluate this prototype, we looked for a execution platform to test the flexibility and effectiveness of our approach. The ideal candidate for such a platform would be a generic one that can model and possibly support execution and/or simulation of real world system that require resource management services. We intended to also identify an evaluation platform that has a clean and explicitly separate concern for requesting and using resources. Having such a clean interface for resource requests and use would enable us to model resource entities separately and provide a clear focus to examine the interaction amongst issues related to resource management services and other requirements that are fundamental to system execution. To this end, we decided to work with a framework that supports such clean separation of the resource management concern, namely the Little-JIL process definition language [84] and its execution environment [17]. Processes, workflow and many multi-agent systems are primarily task coordination systems that define the coordination of resource usage, which includes both agent and non-agent resources, to enable execution of a set of interdependent tasks. The Little-JIL process definition and execution framework provides a vehicle for defining the coordination of agent and other types of resources, the flow of artifacts and resource instances for achieving a complex goal. As such the Little-JIL framework creates a particularly appropriate platform for developing and evaluating new resource management services. The following sections provide a introduction to

the Little-JIL process definition language, its execution infrastructure, Juliette, and the simulation framework named JSim we have developed on top of Juliette.

## 5.1 Little-JIL Process Programming Language

Little-JIL is a process definition language [84] that, along with its interpreter Juliette [16], supports specification and execution of processes involving different agent and non-agent resources. Here agent resource instances are the ones that are capable of carrying out tasks by themselves. Non-agent resources, on the other hand, are the resource instances that are required and used by an agent resource instance to carry out an assigned task.

The Little-JIL process program includes three orthogonal components: a) a coordination specification, b) a resource requirement specification including constraints, and c) specification of artifacts and their flow. The most immediately noticeable aspect of a Little-JIL process program is the visual depiction of the coordination specification of the process. This component of the Little-JIL process program looks initially somewhat like a task decomposition graph, in which processes are decomposed hierarchically into steps. The steps are connected to each other with edges that represent both control flow and artifact flow. Each step contains a specification of the type of agent resource needed in order to perform the task associated with that step. Thus, for example, in the context of an emergency department process, the agents would be entities such as the doctors, nurses, registration software etc. The collection of steps assigned to an agent resource defines the interface that the agent must satisfy to participate in the process. It is important to note that the coordination specification includes a description of the external view and observable behavior of such agent resources. But a specification of how the agent resources themselves perform their tasks (their internal behaviors) is NOT a part of the coordination specification. The behaviors of agents can be defined in a separate specification component. It is impor-

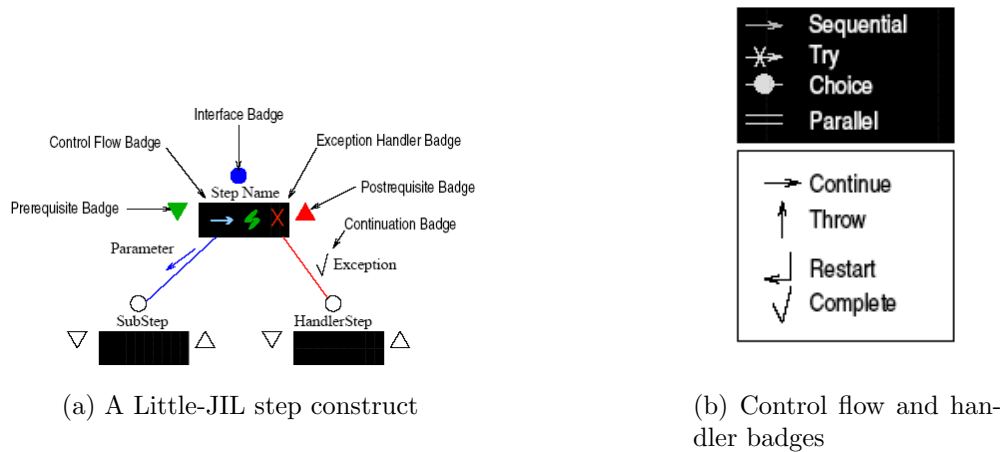


Figure 5.1: Little-JIL iconography

tant to note that Little-JIL enforces a sharp separation of concerns, separating the internal specification of what resource is capable of performing what work and how agent resources carry out their work, from the specification of how they coordinate with each other in the context of carrying out the overall process.

The central construct of a Little-JIL process is a step. Steps are organized into a hierarchical tree-like structure. The leaves of the tree represent the smallest specified units of work, each of which is assigned to an agent resource that has characteristics consistent with those defined as part of the definition of the step. The tree structure defines how the work of these agent resources will be coordinated. In particular, the agent assigned responsibility for executing a parent node is responsible for coordinating the activities of the agents assigned to execute all of the parent’s children. Figure 5.1a shows the graphical representation of a Little-JIL step with its different badges and possible connections to other steps. The *interface badge* is a circle on the top of the step name that connects a step to its parent. The interface badge represents the specification of any and all artifacts that are either required for, or generated by, the step’s execution. Of greater importance for the work described in this dissertation, the interface badge also represents the specification of any and all resources needed



in order to support the execution of the step. Chief among these resources is the single resource designated as the step's execution agent. Below the circle is the step name. A step may also include pre-requisite and/or post-requisite badges, which are representations of steps that need to be executed before and/or after (respectively) this step for the proper performance of the step's execution. A simple form of pre and post-requisites can be simple predicates that need to be evaluated by the process execution engine. A pre-requisite is shown with an upside down triangle on the left of a step bar. Similarly a post-requisite is shown with a regular triangle on the right of a step bar. Inside the central black box of the step structure, there are three more badges. On the left is the control flow badge, which specifies the order in which the child sub-steps of this step are to be executed. A child (substep) of a step is connected to the parent by an edge emanating from the parent and terminating at the child. Artifact flows between the parent and child are indicated by annotations on this edge.

On the right of the step bar is an *X* sign, which represents the exception handler capabilities of the step. Attached to this badge by red-colored exception edges are any and all handlers defined to deal with exceptions that may occur in any of the descendants of this step. Each handler (with the exception of simple handlers) is itself a step, and is annotated to indicate the type of exception that it handles. Here too, artifact flow between the parent and the exception handler step is represented by annotations on the edge connecting them. This edge also bears an annotation indicating the type of exception handled. In the middle of the step bar is a *lightning sign*, which represents the message handling capabilities of the step. Attached to this badge by message handling edges (also known as reaction handling edges) are any and all handlers defined to deal with messages that may emanate from any step in the process definition. A message can be generated from outside the process as well. The message handling capability is quite similar to the exception handling capability, but, while exception handlers respond only to exceptions thrown from within their

substep structure (a scoped capability), message handlers can respond to message thrown from anywhere (an unscoped capability). If there are no child steps, message handlers, or exception handlers, the corresponding badges are not depicted in the step bar.

One of the important features of the language is its ability to define control flow. There are four different non-leaf step kinds, namely *sequential*, *parallel*, *try* and *choice*. Children of a *sequential* step are executed one after another from left to right. Children of a *parallel* step can be executed in any order, including in parallel, depending on when the agents actually pick up, and begin execution of, the work assigned in those steps. A *try* step attempts to execute its children one by one starting from the leftmost one and considers itself completed as soon as one of the children successfully completes. Finally a *choice* step allows only one of its children to execute, with the choice of which child being made by the agent assigned to execute the step.

The pre-requisites and post-requisites associated with each step act essentially as guards, defining conditions that need to hold true for a step to begin execution or to complete successfully. Exceptions and handlers are control flow constructs that augment the step kinds. The exceptions and exceptions handlers work in a manner that is similar in principle to the way in which they work in well known contemporary application programming languages. Exceptions indicate an exceptional condition or error in the process execution flow, and handlers are used to recover from, or fix, the consequences of those situations. When an exception is thrown by a step, it is passed up the tree hierarchy until a matching handler is found. There are control flow semantics involved with handler steps to indicate how the program flow will continue once a raised exception has been handled by the defined handler. Figure 5.1 shows four different types of continuation semantics for handlers. With these semantics, a process definer can specify whether a step will continue execution, successfully

complete, restart execution at the beginning, or rethrow the exception for a higher level parent step to handle.

As noted above, a complete Little-JIL process definition also contains definitions of *artifacts* and *resources* to complement this coordination definition. Artifacts are entities such as data items, files, or access mechanisms that are passed between parent and child steps. The resource management capability is expected to provide language to specify resource requests that are required for the completion of a step. Section 5.3 discusses the request language supported by ROMEO. A process programmer uses this resource request language to define the required resources for every step. Juliette is the execution framework that is used to execute processes written in Little-JIL. Section 5.4 discusses the architecture of Juliette. Juliette executes Little-JIL programs by interpreting steps according to their specified sequences. While interpreting a step, Juliette makes requests to the **Resource Manager** for first reservation and then allocation of the designated agent resource followed by allocation requests of other resources that are required for that step to get executed. The Juliette interpreter then notifies the selected agent by putting the tasks to be done in the agent's *agenda*, which is an abstraction for something akin to a 'to do' list for agents. This is done by the using a distributed Agenda Management System [55]. Agents, in turn, decide which work to pick up from their list of tasks waiting in their agenda and to start working on. It is expected that agents know how to do the tasks and when to notify back the interpreter that it has been completed. The resource (and thus agent) definition, as mentioned above, is separate from, and orthogonal to, the Little-JIL coordination definition. How an agent carries out a particular task is independent of the coordination dictated by the process. Of course, however, the outcome of a process is dependent on the assignment decisions of tasks to agents, which is provided by the resource management service.

The Little-JIL framework provides us with a very useful and apposite setup to study the effectiveness of our approach for modeling resource objects, specifying the resource requests, defining the constraints, and performing the matchmaking and selection operation to assign resources to tasks. Being able to study these aspects of resource management carefully should allow us to evaluate our approach of modeling and managing resource instances in a complex and dynamic environment.

## 5.2 Modeling an ED Process using Little-JIL

To illustrate how the Little-JIL language can be used to define a process, we present a simplified process of how care is provided to patients that arrive in a hospital ED. In subsequent discussions, we shall refer to this process as the ‘VerySimpleED’ process. In figure 5.2, `TreatPatientsAsTheyArrive`, is the root step of the process. Hanging from the root step is a sub-process structure defined as a reaction handling scope. The root step is modeled to receive a message of type `PatientArrivalMessage`. Each such message, when received, results in the instantiation of the process sub-tree `TreatOnePatient`. This sub-tree defines the process of providing patient care to each of the patients arriving at the ED. When a new patient arrives, a triage nurse comes and performs triage on the patient. This activity is captured by the (`TriagePatient` step). The required resource characteristic for the agent that can be assigned the `TriagePatient` task is defined as part of the step, but, is not visible in the figure. To view the agent and other resource requirements as well as parameters that are used or produced by a step, one has to select the step and explicitly look at its interface. This has been done to reduce the visual clutter while modeling or reading a process definition. Later we will discuss the exact interface and the request language that allows a process programmer to specify the required resource characteristics. Let us suppose a triage-nurse is required to carry out this task. As part of performing triage, a triage-nurse assigns a acuity level for the patient. At this point, according to the

process definition of figure 5.2, the patient goes for registration. The `RegisterPatient` step requires an agent resource of type `Clerk`. Here the patient's insurance and other information is collected and an *id-band* is generated and placed in the patient's arm. The patient then goes inside the ED for treatment. However, the required resource for placing the patient inside an ED is a `bed` resource. If all the beds inside the ED is occupied, the patient waits in the waiting room until a bed becomes available.

Once a bed becomes available, the waiting patient is placed in it and a nurse first comes to do an assessment (`RNAssessment` step), followed by a visit by the doctor (`MDInitialAssesmmment` step). After assessing the patient, according to the above defined process, the doctor orders some tests for the patient. Consequently, the patient will go through the step `Tests`, which represents the activities of tests being performed on the patient. In a more complete process, this step is elaborated and a patient only goes through the test related steps that have been ordered for the patient. In addition to the tests, there are bedside procedures performed on the step. These include activities such as 'suturing', 'casting' or 'intubation'. There are some procedures that could be done by a nurse (RN) and there are other procedures that need to be done by a doctor (MD). The `RNProcedure` and `MDProcedure` refer to these tasks. Once the patients have been treated, tested and performed procedures on, the attending doctor performs a final assessment (`MDFinalAssessmentAndDecision` step) and makes a decision about either discharging the patient or admitting the patient. Consequently an ED nurse needs to perform some paperwork for admitting or discharging the patient (`RNPaperwork` step). Throughout this process, there is a parameter named *patientInfo* (not explicitly shown in the figure) that passes through each step. This parameter carries information related to the current state of the patient for whom treatment is being provided. As agents carry out different steps, they may use information carried in the *patientInfo* parameter coming into a step as well as a set of values inside the parameter that are then available to latter steps

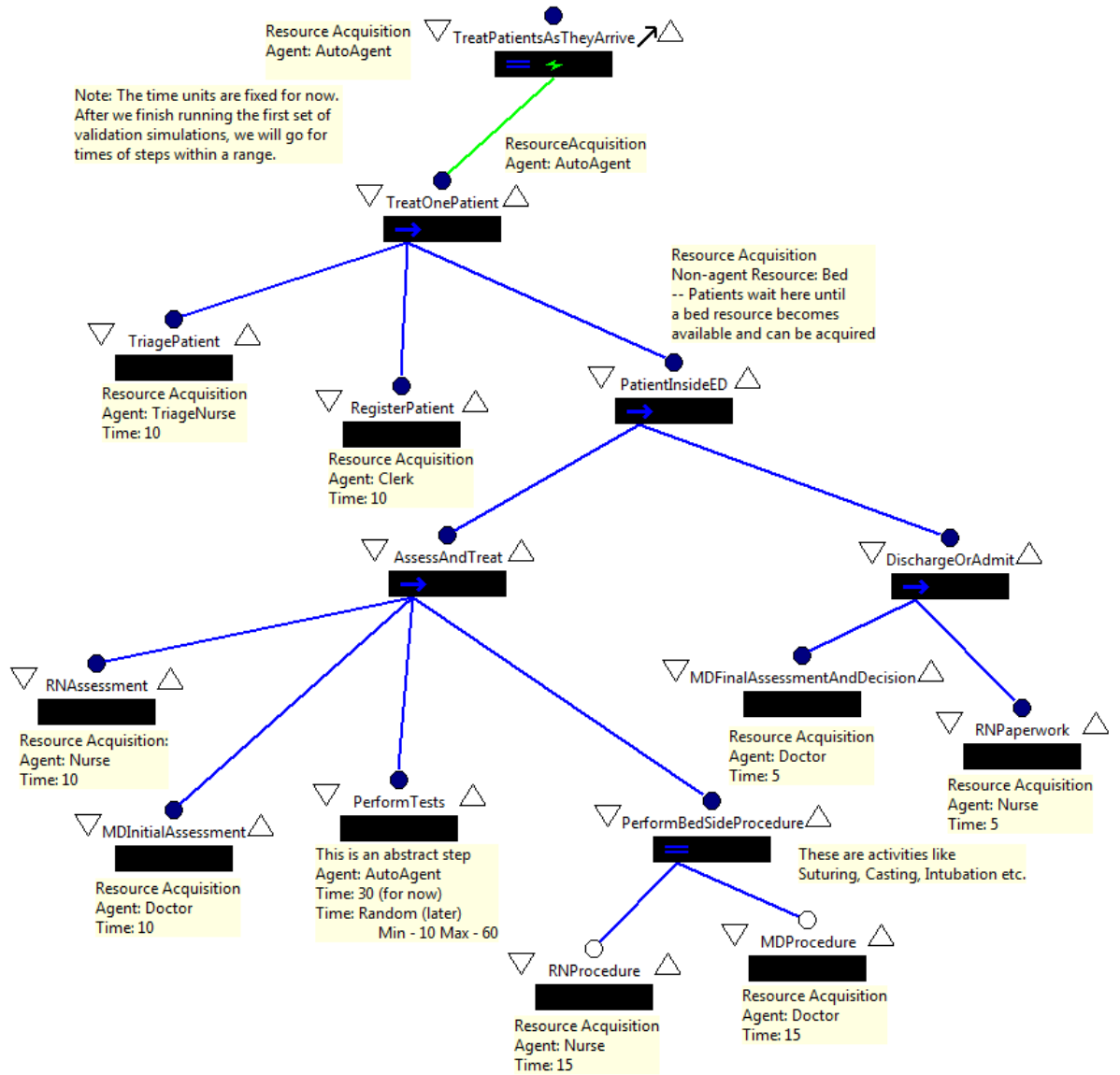


Figure 5.2: A Little-JIL definition of a very simple ED process

(and their agents) subsequently. In chapter 6, we present more elaborate processes with examples of how state information within the *patientInfo* parameter can be used to determine if a step needs to be carried out or not. The above process is largely sequential and abstracts out some of the complexities of the patient care process in a hospital ED. Nevertheless, it introduces the basic activities involved in providing patient care in a hospital ED as well as illustrates how processes are modeled using Little-JIL. In section 5.4, we shall introduce an execution infrastructure that can take such a process model and execute it.

### 5.3 Resource Request Specification in Little-JIL

In this section, we present the syntax and semantics of specifying required resource instances as part of defining a step in a Little-JIL process. We note that a Little-JIL step defines either a scope (non-leaf step) for a set of tasks or an actual unit of work (leaf-step) that needs to be performed by an agent resource. In either case, each step needs to declare specification of the required agent resource instance. The resource requirements related declarations are attached to a Little-JIL step in what is known as the interface of a step. The declaration of the agent requirement for a step gets attached to a parameter associated with each step. This special parameter is given the name *agent*. Thus *agent* is a keyword in Little-JIL that can not be used to name a parameter that specifies a non-agent resource requirement or an artifact associated with the step.

#### 5.3.1 Resource Acquisition and Resource Use

During a process execution, each step instance can get access to a resource instance either by explicitly *acquiring* the resource instance or inheriting it from and thus *sharing* it with its parent step instance. Resource sharing between parent and child step instances is modeled in the static process by binding the parameters representing

the resource instance at the parent and the child over the edge connecting them. In Little-JIL this binding is represented the same way an artifact flow from parent to child [84] is modeled. For *agent* resource instances, Little-JIL allows implicit specification of a step instance inheriting its agent from parent. In this case, if there is no agent resource specified for a step, it is assumed to be coming from its parent. The syntax for specifying explicit resource acquisition (as opposed to sharing from parent) of in a Little-JIL step is to use a filled circle ● in a step’s interface.

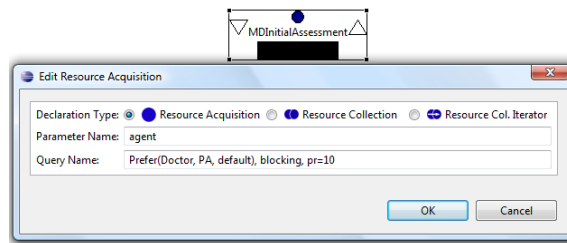


Figure 5.3: Resource acquisition syntax

Figure 5.3 shows a Little-JIL step interface with resource acquisition specification. Here, the step `MDInitialAssessment`’s resource acquisition specification includes a preferred list of queries and some additional information regarding the protocol to be used while acquiring the resource and the relative priority of the step. Note that the parameter name for this resource is *agent*, which indicates that this resource is required as an agent to carry out the task instantiated out of the `MDInitialAssessment` step. Each item in the preference list refers to a resource query that is to be specified in the resource model with which the process will be executed. The specification of ‘blocking’ as part of the request specifies the requirement that if there is no resource instance available to satisfy the request, the resource management service should wait until such resource becomes available and return with the assignment of the requested resource instance.



The syntax for specifying a resource use is an empty circle  $\bigcirc$  with a parameter name representing the resource instance for that step. As noted earlier, a resource use specifies the sharing of the resource instance with its parent. Figure 5.4 shows how a resource use request is specified in a step’s interface. Here the non-leaf step `AssessAndTreat` is specifying the use of a ‘Bed’ resource instance that has been acquired in one of its predecessor steps.

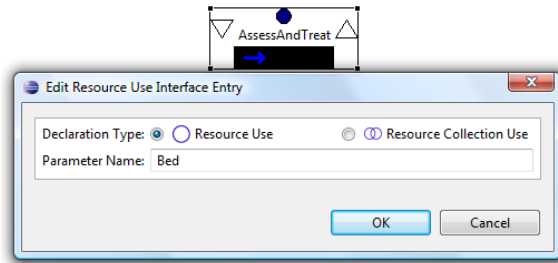




Figure 5.4: Resource use syntax


If a Little-JIL step instance declares a agent acquisition requirement, the agent is first *identified* and then *reserved* through the resource management service before the step instance is *posted* to the assigned agent’s *agenda*. This is done to ensure the existence of available agent resources that match the steps agent requirements. For non-agent resources, a step instance first sends identification requests to the resource manager and then sends acquisition requests once its agent resource that has already been reserved decides to start the step. This resource usage life cycle is very much the same as what we illustrated using figure 4.11 and 4.12 in chapter 4.



### 5.3.2 Request Constraints

Little-JIL allows specification of additional constraints that augments the declarations of required resource specifications in Little-JIL steps. The syntax of these request constraints are such that they can be declared independently in any scoping (i.e. non-leaf) step. Resource requirements in any step can be specified to be ad-

ditionally constrained by a resource request constraint specified in its parent step. There are two groups of request constraints that can be specified in Little-JIL language: a) Resource-Collection constraint () , and b) Resource-Iterator constraint ()


### 5.3.2.1 Resource Collection Constraint

The *Resource Collection* constraint is specified by using an icon composed of two overlapping circles (). Like all other resource related declarations, there are two parts in specifying a *Resource Collection* constraint: a parameter and a specification of the query that defines the constraint. In our request language for specifying such constraints, we allow one or more query names separated by commas. For example, a *Resource Collection* constraint with a parameter name *caregiver* may declare “**doctor, nurse**” as its specification. ROMEO will take this specification and instantiate the constraint into a collection of resource instances that includes both the **doctor** resource instances and the **nurse** resource instances.

The *Resource Collection* constraint can optionally have a maximum cardinality specification associated with it. For example, the declaration of “**doctor,nurse,5**” specifies that the constraining collections cardinality, when instantiated, may not exceed 5. Similar to the *resource acquisition* and *resource use* declarations we discussed in section 5.3.1, there are two separate icons for *Resource Collection* constraint declaration () and *Resource Collection Use* declaration (). These declarations and icons appear as annotations attached to the interface of Little-JIL steps. A *Resource Collection* constraint can be copied from a parent step to any of its child steps if the child step has a parameter declared as a *Resource Collection Use* and that parameter is bound to the parent step’s constraint declaration.

When a resource acquisition at a step is constrained by a *Resource Collection* constraint specified in its parent step, the edge connecting the two steps carries an explicit annotation capturing the constraining relationship.

### 5.3.2.2 Resource Iterator Constraint

The iconography used for declaring a *Resource Iterator* constraint looks like two overlapping circles with an arrow through them (). The *Resource Iterator* constraint has the exact same syntax as the *Resource Collection* constraint. The request language supported by ROMEO accepts declaration of a single query name or a set of query names separated by commas as the specification of a *Resource Iterator* constraint. Like *Resource Collection* constraint, Little-JIL language allows specification of copying a parameter declared as a *Resource Iterator* constraint at a parent step to a parameter declared as a *Resource Collection Use* in any of its child step. Like before, When a resource acquisition at a step is constrained by a *Resource Iterator* constraint specified in its parent step, the edge connecting the two steps carries an explicit annotation capturing that constraining relationship.

### 5.3.3 Resource Exceptions

If a resource identification or acquisition for a step instance fails, it may result in termination of the step instance and consequently an exception may get thrown. In Little-JIL execution environment, when a step instance is instantiated for execution, all its required resources are first identified, the agent resource is then reserved and when the reserved agent decides to start the task, acquisition requests for the agent and other identified resources are sent to the **Resource Manager**. While processing the identification requests, if the **Resource Manager** fails to find any matching resource instances for any of the requests, it replies to requesting entity with a **ResourceIdentificationFailed** message, which results in a **ResourceUnknown** exception being thrown by the step.

Figure 5.3 showed an example of specifying agent resource requirement for a step, where the resource acquisition has been specified to be blocking. Requests for resources can also be ‘non-blocking’. When the **Resource Manager** fails to acquire a resource blocking resource instance, it places it in the queue of pending requests to satisfy it once resources become available. If, however, the request is ‘non-blocking’, the resource manager returns a **ResourceAcquisitionFailed** message, which results in the termination of the step instance by throwing **ResourceUnavailable** exception. In the execution framework of Little-JIL (Juliette), both **ResourceUnknown** and **ResourceUnavailable** are subtype of **ResourceException** class.

## 5.4 Juliette: the Little-JIL Process Execution Environment

We have noted earlier in this chapter that Little-JIL, our chosen process modeling language, has rigorous semantics that allow enactment of a coordinated set of activities by participating agent resources. Such process execution is achieved by facilitating the communication among agent resources that are made responsible for and who, in turn, carry out the activities defined in the process model. The execution engine ensures that the temporal constraints specified in the process definition is maintained. It also provides mechanism for acquisition of resource instances and the flow of artifacts needed to enable and complete an activity. Figure 5.5 provides a high-level depiction of the Little-JIL execution architecture. The **Step Sequencer** is a central feature of this system, receiving requests for execution of the process (for example, a notification of a patient arrival), and then supervising the forward progress of process execution as steps complete.

The **Step Sequencer** performs its work by accessing the Little-JIL process to determine which step(s) are to be executed next (based upon information about step(s) that have completed), and then assembles the items needed to get the step executed. Most specifically, the **Step Sequencer** consults the **Resource Manager** to convey requests

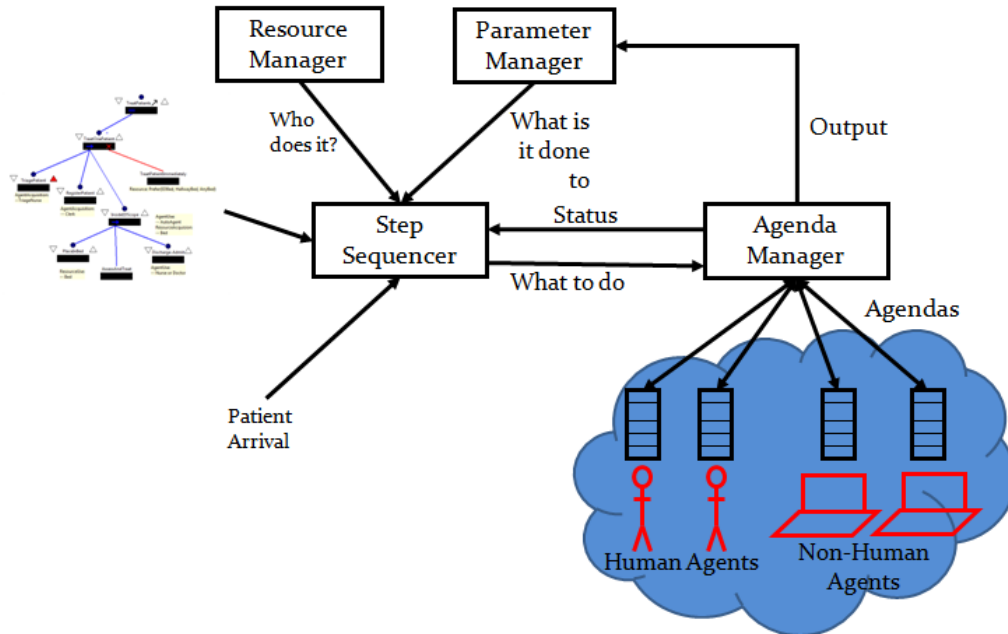


Figure 5.5: Architecture of Juliette

for resources (an agent resource and other supporting resources) that are instances of the types defined as being needed by the step being executed. The Resource Manager is responsible for searching its internal repository of resource instances and selecting those that seem particularly well-suited for meeting the needs of the requesting step. Determination of which resource is best suited often requires understanding the circumstances under which the step is being performed. Information about circumstances is generally obtained through an inquiry about the state of the process execution. Once the Resource Manager has identified the needed resource instances, the step is placed as an item on the agenda of the selected agent resource. This agenda item also includes the input and output arguments for the step. These arguments are accessed through a Parameter Manager. During execution each agent must monitor its agenda, select a step to be performed, perform the step, and signal step completion once result values have been bound to the appropriate output arguments.

Note that the monitoring of an agenda of a non-human agent (e.g. an MRI or an electronic health record system) is likely to be done by automatic polling. Live agents (e.g. doctors, nurses, and registration clerks) must monitor their agendas themselves. In all cases, an agent may have the capacity to perform more than one step at a time, and so may have multiple agenda items open simultaneously. An agent signals completion of a step by placing an annotation in the step instances agenda item, and passing the agenda item through the Agenda Management system back to the Step Sequencer, which proceeds with execution of subsequent steps.

## 5.5 JSim: The Simulation Environment

We have primarily evaluated our resource management service, ROMEO, in an environment that can simulate process models defined in Little-JIL. For this purpose, we have developed a simulation engine on top of the Juliette execution framework described in section 5.4. This simulation capability, named JSim, has been built by making relatively modest additions and modifications to Juliette. Figure 5.6 shows the simulation system architecture. This architecture allows any combination of human and non-human agents to be simulated. In Figure 5.6 workstation icons indicate the human and non-human agents to be simulated. Note that the main additions to the execution system are a simulation TimeLine, facilities for simulating the behaviors of all of those agents that are being simulated (Agent Behaviors), and a facility for collecting the results of a simulation run. In addition, the Step Sequencer has been modified (e.g. so that it accepts instructions about when to proceed to the next step from the TimeLine), and the user now provides information about the distribution of external messages (e.g., patient arrival messages). The following description provides a brief description of how JSim works using the example of simulating a hospital ED process.

A simulation begins with the user providing an arrival distribution specification, and specifications of agent behaviors, through the Agent Behaviors module. To begin, JSim initializes the TimeLine to zero to start off a simulation run, initializes the root step of the Little-JIL process to be the step currently being executed, and places a start event for that step in the TimeLine. The simulation then proceeds as an iterative loop in which the most proximate event in the TimeLine is picked up and simulated. An event can represent posting, starting or completion of a step instance, arrival of an external message etc. The perpetuation of the simulation results from the fact that each step is responsible for placing in the TimeLine one or more events that represent such key activities as step completion, spawning of substeps, etc. Each such event has a designated simulated time at which it is to occur. Thus, for example, a step completion event is generated at the start of the simulation of the step, and the time of this event reflects how long it is expected to take for the steps agent to complete the performance of the step. The TimeLine module keeps all events in sorted order, so that the Step Sequencer can easily determine which event is to be simulated next.

The Step Sequencer then proceeds very much as it does when executing the process. In particular, it picks up the events to be simulated in order and for each event, consults the Resource Manager to obtain the needed resources (including the agent) and the Parameter Manager to obtain the needed input arguments. Once all needed resources and arguments have been obtained, the Step Sequencer packages them into an agenda item and delivers the agenda item to the Agenda Manager for placement on the agenda of the agent assigned to perform the step. Performance of the step, in turn, results in more events being placed in the TimeLine. To determine the times at which different events, such as starting or completion of a task, are to be performed by an agent, JSim uses the Agent Behaviors module, which has been initialized with information about how to model agent behaviors. To specify simulated agent behaviors in a flexible way, we have developed an XML based rule language [83] called the JSim

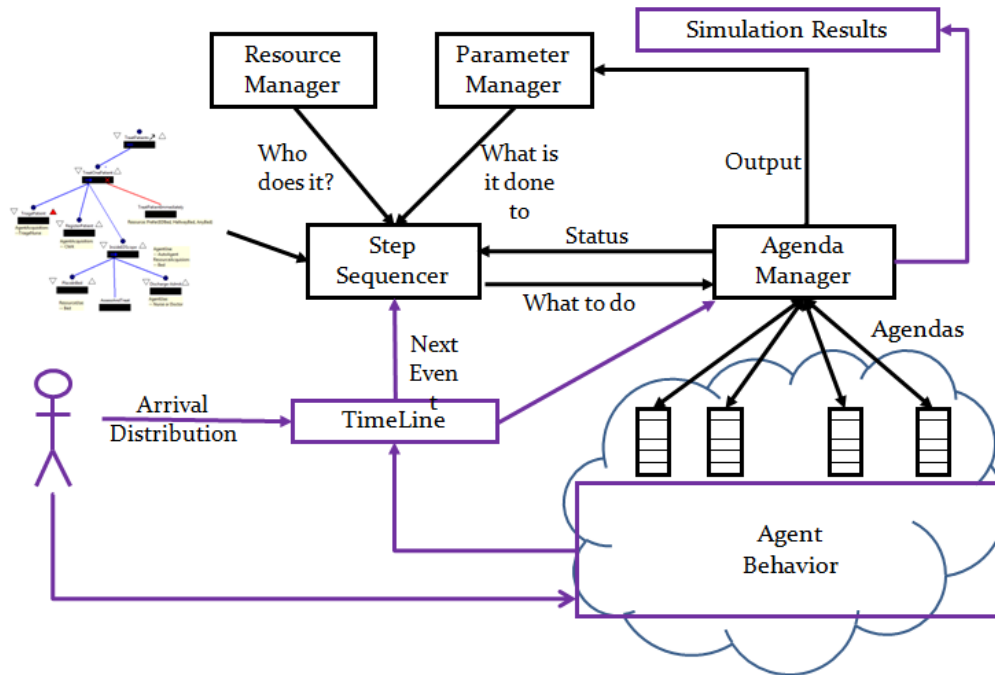


Figure 5.6: Architecture of JSim

Agent Behavior Specification (JABS) language. Examples of how agent behaviors are specified using JABS is briefly discussed in section 5.5.1. This Agent Behaviors specification replaces the actual interaction with live agents in a JSim simulation. JSim allows the specification of agent behavior to be done primarily in two ways:

- **Stepwise:** There is a specification for how a step execution is to be simulated, and the specification does not vary with different instantiations in the process or for different agents that may perform the step.
- **Agentwise:** There is a specification for how to simulate the behavior of each different agent that may be assigned to carry out each of the steps to which it might be assigned. Thus, for example, this type of specification allows the possibility that different agent instances may require different amounts of time to perform the same step.



JABS also allows nested specification of agent behavior in order to allow combinations of the above two approaches. In both cases, if the step uses input parameters or produces output parameters, the Agent Behavior Specification must define how the agent uses and converts its input arguments into outputs. It is important to note here that JABS also allows for specification of simple statistical distributions when defining how long an agent takes to complete a task. Thus we can simulate statistical variations in the amount of time an agent may take while performing different instances of the same task. In addition to modeling execution time, JSim also allows for estimation and modeling of the lag time between step assignment and initiation of step execution. In case of simulating the patient care process in an ED, these estimates have been developed based on interviews with ED professionals, and analysis of statistical data.

### 5.5.1 JSim Agent Behavior Specification (JABS)

Figure 5.7 shows an example of part of an Agent Behavior specification. This specification is provided as part of the configuration of a specific simulation. It is assumed that the Little-JIL process for which this agent behavior is specified has steps named `PlacePatientInBed`, `CompleteRegistration`, and `MDAssessment`. Some of these steps receives a parameter name *patientInfo* that carries characteristics and state information. An agent who is made responsible for such a state can modify fields of that parameter.

In Little-JIL, each step is formally defined using a finite state automaton. During the execution or simulation of a Little-JIL process, a step goes through the states *posted*, *started*, *completed*, and/or *terminated*. Although not required, for very fine-grained control, JABS supports specification of the behavior of an agent upon entry into each of these states. Figure 5.7 shows examples of how such behaviors can be specified upon entry into the *started* execution state for some of the steps in the

```

<step name="PlacePatientInBed">
  <started>
    <complete>
      <fixed value="10" />
    </complete>
  </started>
</step>
<step name="CompleteRegistration">
  <started>
    <group>
      <set-field parameter="patientInfo">
        <field name="isRegistrationDone">
          <boolean value="true" />
        </field>
      </set-field>
    </group>
  </started>
</step>
<agent name="ha001-doctor">
  <step name="MDAssessment">
    <started>
      <complete>
        <linear-range min="10" max="20"/>
      </complete>
    </started>
  </step>
</agent>

```

Figure 5.7: Example of Agent Behavior Specification

process shown in figure 5.2. For example, for the step `PlacePatientInBed`, the first rule in the example specifies that once started, this step will get completed after 10 time units regardless of which agent resource instance is assigned this task. The second rule in the example specifies the time it takes for any agent assigned to the step `CompleteRegistration` to be computed using a uniform distribution between 10 and 20 simulation time units. The agent behavior specified as part of completing the `CompleteRegistration` step also sets the value of a boolean field inside the *patientInfo* parameter to become true. The third rule, which is a nested rule, specifies that a specific doctor agent, with id ‘HA001’ takes somewhere between 10 to 20 simulation time units when assigned the task of performing `MDAssessment`.

### 5.5.2 Simulation Outputs

As an output of the simulation, JSim produces a trace file, which holds the following information:

- Which agent resource instance was assigned to which task at what time
- When did the agent resource instance start working on that task
- When did the agent complete the task

In the trace output, an instance of a task needs to be differentiated from other task instances that are generated out of the same Little-JIL step. In the case of ED simulations, we have done that differentiation based on the *patientInfo* parameter that carries an id for the patient it represents. Additionally ROMEIO prints out an allocation file that lists which resource instance and with what capacity was assigned against each request generated during a simulation run. ROMEIO also prints out another trace file that describes the utilization level of each resource instance that was active during a particular simulation.

## CHAPTER 6

### CASE STUDIES AND EXPERIENCES

Our primary evaluation vehicle has been JSim, a discrete event simulation infrastructure that was developed on top of the Little-JIL execution engine, Juliette. We have discussed the architecture of both Juliette and JSim in chapter 5 where the discussion focused on describing how a resource management service is central to both process execution and simulation. This chapter now discusses the case studies we have performed using ROMEO, our prototype Resource Manager implementation, in conjunction with JSim and Juliette. In the following discussion, we use the term ROMEO-JSim to refer to our simulation infrastructure. All the process models and many of the experiments presented here are results of many discussions with a domain expert, an emergency department physician and ex-director of a large ED in the United States. The experiments are described in the following subsections.

#### **6.1 Validating Simulation Results**

We have built considerable flexibility into our simulation infrastructure through its factored architecture. This architecture affords expedited access to each of the key architectural components, and many different points at which the parameters used to configure these components can be tuned and adjusted quickly. This has been useful in facilitating the tuning and setup of specific simulations or sets of simulations. These experience have reinforced our view that resource management facilities such as ROMEO can be critical components of a discrete event simulation infrastructure. In fact, we have observed that it is not uncommon for the output of a

simulation to be strongly affected by how ROMEO assigns resource instances to tasks in the simulation. Thus correct behavior of ROMEO should be expected to be closely connected to the validity of the results produced by JSim. The first set of steps we took to gain some confidence in the simulation results produced by ROMEO-JSim included modeling a simplified hospital ED process and running simulations with a scenario where patients arrive at a fixed interval (constant rate). We performed a number of sanity checks on the results of these simulations. For example, we manually inspected output traces of the initial ED simulations to establish that each simulated patient was processed by the steps that it was supposed to go through, the agent resource instances were assigned to the ED tasks as expected, and each agent started and completed steps according to the behavior specified in the simulation input configuration. The following subsections describe some other sanity checks we performed to gain confidence on our simulation results.

### **6.1.1 Impact of Varying a Bottleneck Resource**

After the initial inspection of the outputs of some ED simulations, we ran some more simulations using a simplified ED process. Our objective was to look at the trends in simulation results with some specific changes in the resource mix. For the purpose of these simulation runs, we used the process shown in figure 6.1. We refer to this process as ‘SimpleED’ in the following discussion.

In the ‘SimpleED’ process, when a patient arrives at the ED, s/he first gets seen by a triage-nurse (**TriagePatient** step) and consequently gets a triage acuity level assigned. The patient then goes to the registration clerk for registration (**RegisterPatient** step). The registration clerk collects information from the patient including insurance information and puts it in the patient’s record. The registration clerk also generates and places an id-band on the patient. The patient then goes inside the treatment area of the ED (often referred to as main-ED) if a bed is available. If all

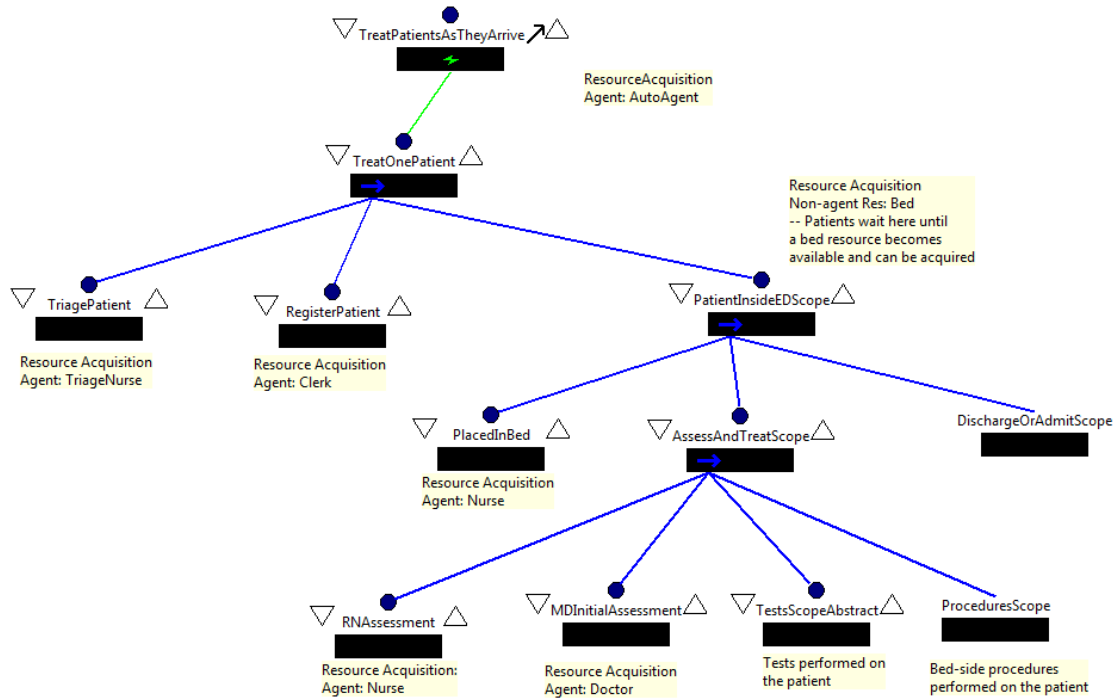


Figure 6.1: A Simple ED process in Little-JIL

beds inside the main-ED are occupied, the patient waits in the waiting room until a bed becomes available. This is modeled by a *blocking* acquisition request for a bed resource instance in `PatientInsideEDScope` step. Once a bed is successfully acquired, the patient is placed in a bed (`PlacedInBed` step) inside the main-ED. The bed placement activity also includes the activity of having the patient change into hospital clothing. Usually a **nurse** resource instance is made responsible (i.e. made the agent) for the bed placement step. Inside the main-ED, the patient is first seen by a **nurse** in the `RNAssessment` step, followed by an assessment by the attending doctor (`MDInitialAssessment` step). The doctor assessment may result in some tests. These test related activities have been represented as a single abstract step named `TestsScopeAbstract`. There are also some bedside procedures that may be performed on the patient as shown by the step reference named `ProceduresScope`. Figure 6.2 shows the elaboration of `DischargeOrAdmitScope`. Once all the tests and procedures

are done, the attending `doctor` makes a final assessment of the patient and decides whether to admit the patient or to discharge her/him. This is represented by the `MDFinalAssessmentAndDecision` step, which is depicted in a diagram (figure 6.2) that is shown separately from the one shown in figure 6.1. At the end of this ‘SimpleED’ process, `RNPaperWork` step is performed. Usually this step is performed by a `nurse` resource instance.

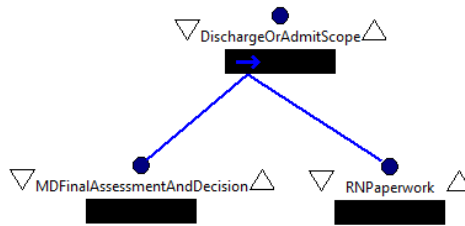
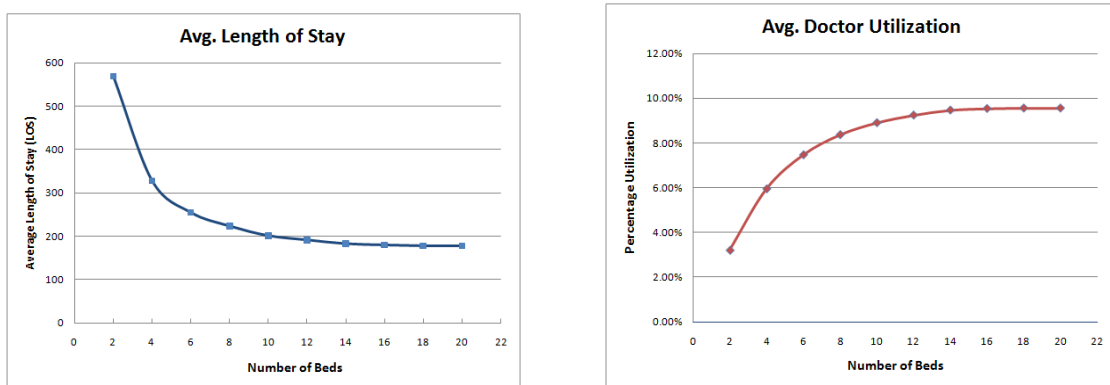


Figure 6.2: Discharge part of ‘SimpleED’ process

Both this description, and actual observed experiences, suggest that in the ‘SimpleED’ process, `beds` are potentially bottleneck resources. Thus we decided to study the actual criticality of bed resources by contriving a set of simulations where the numbers of beds was varied. In doing so we hoped to both gain better understanding of the importance of bed resources, and also validate the accuracy of our simulations. Thus, we ran simulations of patient flow through this process using different numbers of available beds, and, we collected data about the length-of-stay (LOS) for each patient for each such simulation. These simulations were repeated with a resource mix that varied only in the number of available bed resource instances. With fewer beds, we expected to see many of the patients spending more time waiting and thus a high average LOS. With more beds added to the resource mix, the simulation output was expected to show reduced average LOS. We plotted the average LOS against the increased number of beds and the graph did demonstrate the expected behavior.

Figure 6.3a, shows the output of this experiment. As more and more beds were added into the resource mix, the LOS metric improved (i.e. reduced). However, the improvement diminished with the increase of this one resource only and there was no impact of adding that resource after a certain point. This graph is basically demonstrating a simple case of the ‘law of diminishing returns’. Of course these results apply only to a scenario in which patients arrive at fixed time intervals. Different patient arrival distributions might lead to different results. But this set of simulations did increase our confidence in the soundness of the simulation system, thus also suggesting that similar simulation runs be tried with different patient arrival patterns and different resource mixes.



(a) Avg. LOS with increased beds

(b) Avg. doctor util. with increased beds

Figure 6.3: Validating Simulation Results

Thus, in particular, it seemed reasonable that if more beds are added to the resource mix, the utilization levels of other resource instances might increase. This is because having more beds seems to result in more patients simultaneously getting treatment inside the main-ED and thus requiring services from other resource instances such as doctors, nurses, etc. Consequently, that suggests that these other resources would be utilized more heavily with increases in the number of beds in the simulation. Further, we conjectured that this improvement would be less and less as we continue to increase only the bed resource instances keeping the quantities of all the



other resources fixed. To further validate the behavior of ROMEO-JSIM, we collected utilization levels of all resource instances (not just the bed resource instances) used in the previously described set of simulations. This set of simulations used a resource mix that contained 2 triage nurses, 2 registration clerks, 4 doctors and 4 nurses. We took the utilization levels of each of the doctors as determined by the ROMEO-JSim simulation runs, and computed average doctor utilization for each. Figure 6.3b shows the graph of these results. As we expected, the average doctor utilization improved as more and more patients were allowed inside the ED simultaneously as a result of adding more beds. However, the improvement in the utilization was diminishing and gradually flattened out as the number of beds continued to increase.

### 6.1.2 Little's Law

In queuing theory, there is an intuitive, yet remarkably simple, equation that describes the steady state behavior of a resource utilization based system known as Little's law. It states that the long term average number of customers in a stable system  $L$  is equal to the long term average arrival rate  $\lambda$  multiplied by the long term average time the customer spends in the system (i.e. length-of-stay),  $W$ .

$$L = \lambda \times W$$

A system is considered to be stable if it is non-preemptive and if the rate at which the customers arrive at the beginning service station is the same as the rate at which they go on to the next service station and so on such that the customers leave the overall system at the same rate as well. In other words, in this definition of a stable system, there should not be any queue occurring in any part of the system that continues to grow larger as the simulation proceeds. It is possible to create such a situation with our simulation infrastructure, ensuring that we have enough resource instances for each of the service stations (triage, registration, assessment, procedures,

etc.) of our ‘SimpleED’ process to assure that arriving patients do not have to wait anywhere as a result of resource contention. By doing this we wanted to determine whether our ROMEO-JSim systems behavior was consistent with the predictions of Littles Law.

To do this, we configured the arrival of one hundred (100) patients to the ED following a *Poisson* distribution. We set the mean of the distribution to be thirty (30), which translates to roughly two (2) patients arriving per hour. We set up the simulation with fixed service times (step times) for each of the services and loaded the simulation set up with a large number of resource instances:

- Beds: 100
- Nurse: 100
- Doctor: 100
- TriageNurse: 100

To keep track of the number of patients in the system at any point in time, we incremented a variable each time the first step of the process (`TriagePatient`) was started, and decremented the variable each time the last step of ‘SimpleED’ process (`RNPaperwork`) was completed. An output recording the value of this variable was produced in addition to the simulation trace. At the end of the simulation, we computed the mean number of patients in the ED from this output. The result of the simulation shows the following:

- Average LOS ( $W$ ):  $137.42 = 2.29$  hours
- Mean number of patients in the ED ( $L$ ) : 4.58
- Patient arrival rate ( $\lambda$ ): 2 per hour

So, we can see that in this particular stable system, as simulated by our infrastructure, the following holds:

$$\lambda * W = 2 * 2.29 = 4.58 = W$$

We note this is a very special case of a stable system. Ideally, we need to run a system for a very long time such that the arrival rate for each service station stabilizes to a fixed rate and compute the parameter values under such a simulation. Nevertheless this exercise provided us with one more sanity check on the validity of our simulation infrastructure.

### 6.1.3 Comparing with a Commercial Simulation Product

To gain more confidence in the inner workings of the ROMEO-JSim infrastructure and simulation results produced by it, we decided to compare our results with a well established commercial discrete event simulation product: Arena [27, 40]. Arena is an object-based, hierarchical modeling tool that has been used in a wide range of simulation applications. Of particular relevance to our studies is the fact that many ED simulation studies have used Arena as their modeling tool [30, 68, 7, 29].

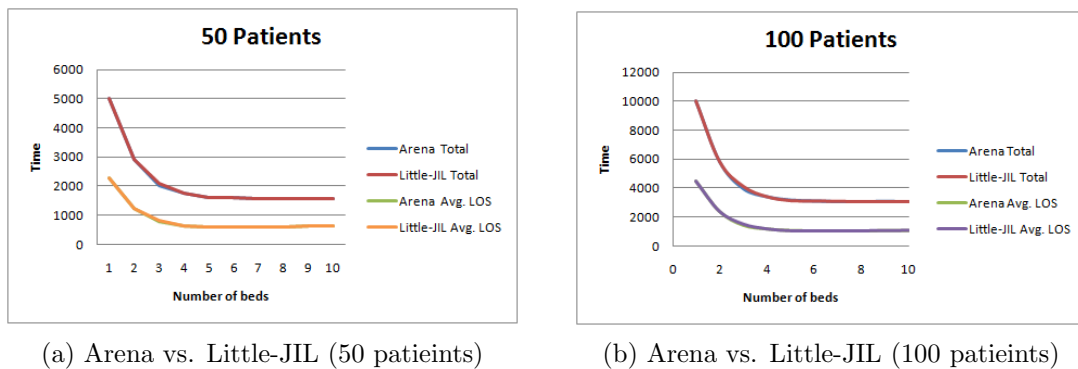


Figure 6.4: Comparing Little-JIL based simulations with Arena

For purposes of comparison, we modeled a very simple ED process using both ROMEO-JSim and Arena and ran simulations with patients arriving at a fixed rate.

The process we used for this study is very similar to the one showed in figure 5.2. The task times were kept fixed and only one type of resource was varied, namely the bed resource. Figure 6.4 shows the comparison graphs. We plotted both how long the simulation ran in total time as well as the average LOS for each patient. We simulated fifty (50), one hundred (100) and two hundred (200) patients. Here we have shown the simulation results produced by scenarios with fifty (50) and one hundred(100) patients. We went through a couple of iterations of simulations and inspections to ensure that details of both the Arena and the ROMEO-Jsim models were describing the exact same process. As one can see, the simulation tools produced exactly the same results for the different numbers of patients.

## 6.2 Capturing ED Domain Policies

This dissertation aims to evaluate the usefulness of our approach of resource and request modeling as well as our proposed overall resource management architecture. One evaluation exercise that seemed particularly challenging was to see if our approaches were effective in expediting the evaluation of complex domain policies. We expected that such studies might sharpen an evaluative focus on identifying the effectiveness of the flexibility derived from our proposed separation of concerns, and from the various tuning parameters incorporated. Our hypothesis has been that the correct separation of concerns and apposite resource and request modeling mechanism would allow one to model a wide variety of simulations quickly, thus speeding the cycle of asking interesting what-if questions, getting back answers, and then formulating follow-up what-if questions suggested by the answers. Consequently, we have tried to model a number of different policies from the ED domain. One of the first policies we modeled is the following:

In a hospital ED, the doctor who performs `InitialAssessment` on the patient must be the same doctor who performs the `FinalAssessment` and makes the decision regarding discharging or admitting the patient.

We have used the notion of *restricted-group-constraint* that we introduced through definition 14 in section 3.4 to support simulations that implement this policy. We introduced the syntax for implementing this notion as a constraining request in section 5.3.2. In ROMEO-Jsim, a Little-JIL *resource-collection constraint* is used to implement this type of constraining request. Using this, we have specified in the ‘SimpleED’ process that the resource requirements specified for steps MDInitialAssessment, MDProcedure, and MDFinalAssessmentAndDecision not only require a doctor, but that the doctor required for each of these steps has to be the same as for the other steps. We specified this by declaring a *resource-collection constraint* in the PatientInsideED-Scope non-leaf step.

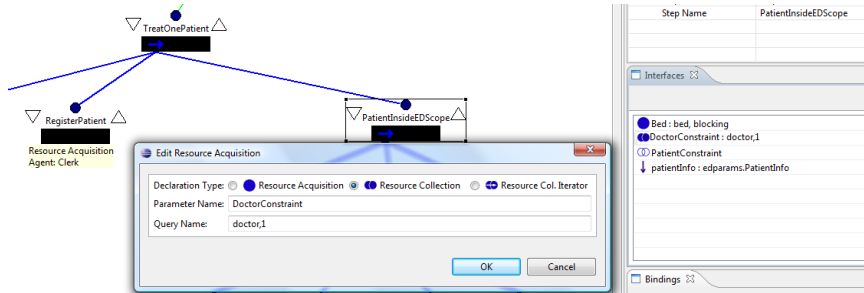


Figure 6.5: Declaration of the same-doctor constraint

The constraint, as shown in figure 6.5, is specified using a parameter named `DoctorConstraint`. This parameter is copied down the tree from parent step to child step through binding annotations on the edges. At the leaf step level, where agent acquisition is specified, the edge connecting the leaf step (`MDInitialAssessment`) to its parent `AssessAndTreatScope` needs to declare the constraining annotation between the parameter (`agent`) specifying the request at the leaf step to the parameter specifying the constraint (`DoctorConstraint`) in the parent step. ROMEO, when passed the request for maintaining such constraints and resource instances for steps, ensures that all requests for resource instances that are returned actually do satisfy the constraint. When the constraining collection (e.g. the set of resource instances

instantiated out of the `DoctorConstraint`) is given a maximum cardinality of 1, it amounts to specifying that both the steps whose resource requirements are constrained by this *resource-collection* constraint, must get the same resource instance assigned to them.

### 6.2.1 Impact of Same-Doctor Constraint

While running experiments with ROME0-Jsim for comparison with Arena, we ran a number of simulations using our model and varying the mix of resources. We were trying to determine the optimum resource mix for a given set of patients and a specific arrival rate of  $n$  patients per hour where  $n = 1, 2, \dots, 20$ .

Table 6.1: Optimum resource mix for different patient arrivals

<b>Arrival Rate</b>	Number	Number	Number	Number	Number
pts/hr	MDs	RNs	TNs	Clerks	Beds
1	1	1	1	1	2
2	1	2	1	1	5
3	2	2	1	1	5
4	2	2	1	1	10
5	3	3	1	1	9
6	3	3	1	1	13
7	4	4	2	2	13
8	4	4	2	2	16
9	5	5	2	2	16
10	5	5	2	2	20
11	5	5	2	2	21
12	5	5	2	2	23
13	5	5	2	2	22
14	6	6	2	2	20
15	6	6	2	2	20
16	6	6	2	2	21
17	6	6	2	2	22
18	7	7	3	3	43
19	7	7	3	3	43
20	7	7	3	3	37

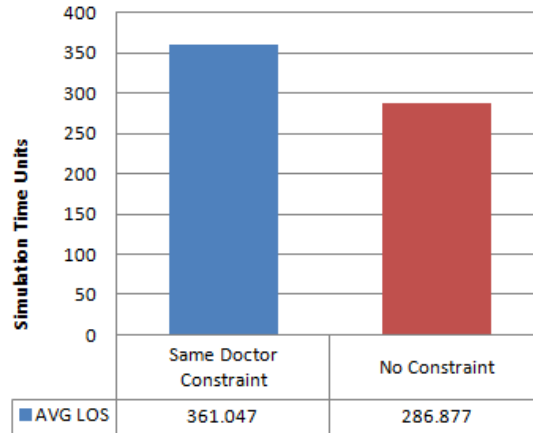


Figure 6.6: The impact of adding doctor constraint

Table 6.1 shows the result of that experiment where one hundred (100) patients were simulated being treated using the ‘VerySimpleED’ (figure 5.2) process. From these results we found that for the ‘VerySimpleED’ process, with seven (7) patients arriving per hour, the optimum resource mix is a combination thirteen (13) beds, four (4) doctors, four (4) nurses, two (2) triage-nurses, and two (2) clerks. We then repeated this experiment by using this same resource configuration but with a slightly elaborated ‘SimpleED’ process that has just a few additional steps that simply shift the simulation by a fixed time interval.

We ran simulations of patients going through the ‘SimpleED’ process with and without the doctor constraint. For this particular experiment we simulated three hundred (300) patients arriving at a constant rate and flowing through the process. Figure 6.6 shows the average LOS for the two simulation configurations. As intuitively expected, this experiment shows that adding the constraint increases the average LOS. In this particular setup, having the constraint maintained all the time adds on average 25.85% time to each patient’s stay in the hospital ED. We were interested in discovering the impact of such policy change through simulation, but were particularly interested to note how easy it was to perform such studies by setting up the different

simulations. In this particular example, the additional effort to specify the constraints throughout the process took a Little-JIL process definer about thirty-five (35) minutes of extra modeling time. To set up the simulation where there were no constraints, it was only a matter of removing the constraint annotations on the edges between the steps where resource-acquisitions were declared (e.g. `MDAssessment` and `FinalMDAssessmentAndDecision` etc.). To set up the new simulation without the doctor constraints thus needed only a few minutes. Placing the constraint specification as part of the request model thus made it remarkably easy to set up these different simulations, and supported our sense of the power obtained through the separation of concerns and flexibility built into our approach to resource management.

### 6.2.2 Dynamic Substitution

Another focuses of this dissertation has been to determine the value of providing strong support for the modeling of complex situations where resource instances' behaviors may change dynamically. More specifically, we were interested in capturing and supporting scenarios where a resource instance can be used to satisfy a request only under some specific execution state, but not under other execution states. Such state-dependent behavior of resource instances might, for example enable them to dynamically substitute for each other, depending upon contexts established by the execution state of the process. To study the effectiveness of our resource management service in specifying and simulating scenarios with dynamic substitutability, we again considered the 'SimpleED' process shown in figure 6.1. We set up a simulation experiment with the following resource substitutability scenario:

- `PlacePatientInBed` and `RNPaperwork` for discharging a patient do not always have to be done by a regular registered nurse (RN). The triage nurses who are specified as the performers of triage operations can substitute for an RN for performing the placement of a patient in a bed or for performing discharge pa-



perwork when the ED is overcrowded, all RNs are busy, and a resource instance of type `TriageNurse` is available.

- `RNPaperwork` can also be performed by a registration clerk when the ED is overcrowded, the clerk is idle, and there is no nurse available for performing the discharge paperwork.

We measure the crowdedness of the ED based on the number of patients who have gone through `TriagePatient` and `RegisterPatient` and are waiting for a bed to become available inside the main-ED. This scenario, in the real world, describes the patients who are sitting in the waiting room of an ED after being triaged and registered. To simulate this scenario, we changed the agent resource request for `PlacePatientInBed` and `RNPaperwork` from `Nurse` to `default`. In ROMEIO, a request for a default agent resource means whichever resource instance is capable of providing the service. Here the service is the activity, i.e. placing a patient in bed (`PlacePatientInBed`) or doing the paper work for discharging a patient (`RNPaperwork`). As part of the repository that holds descriptions of the resource instances of a domain, ROMEIO maintains a table named `GuardFunction` that defines the services a resource instance is capable of offering under various system execution states.

ServiceName	ResourceGroup	Condition
PlacePatientInBed	Nurse	true
PlacePatientInBed	TriageNurse	StateServer.PendingRequests("bed") > N
RNPaperwork	Nurse	true
RNPaperwork	TriageNurse	StateServer.PendingRequests("bed") > N
RNPaperwork	Clerk	StateServer.PendingRequests("bed") > N

Table 6.2: Guard function defining services offered by resource instances

Table 6.2 shows how ROMEIO defines the guard function on the *capabilities* or *offered-services* of a group of resource instances such as nurses or registration clerks. In rows 2, 4, and 5 of table 6.2, the *N* gets replaced with a specific number for a

particular simulation run. In most of our simulation experiments presented here, we have worked with  $N = 3$ . In performing this simulation experiment, we kept the resource mix from our last set of experiments (i.e. 13 beds, 4 doctors, 4 nurses, 2 triage nurses and 2 clerks). However, we added a little more variability in our ED process model. Instead of fixed step execution times, we specified the time taken to perform each step execution as being provided by a triangular distribution. Moreover, we generated patient arrivals using a *Poisson* distribution with a mean inter-arrival time of nine (9) minutes, which translates to roughly seven (7) patients per hour. With these additions to the simulation specification, we ran each simulation five (5) times with three hundred (300) patients and looked at the average of the simulation results. To study the impact of such dynamic substitution, we computed the number of patients waiting in the waiting room (i.e. waiting for a bed to become available) every time a new patient arrived. We plotted this data in figure 6.7.

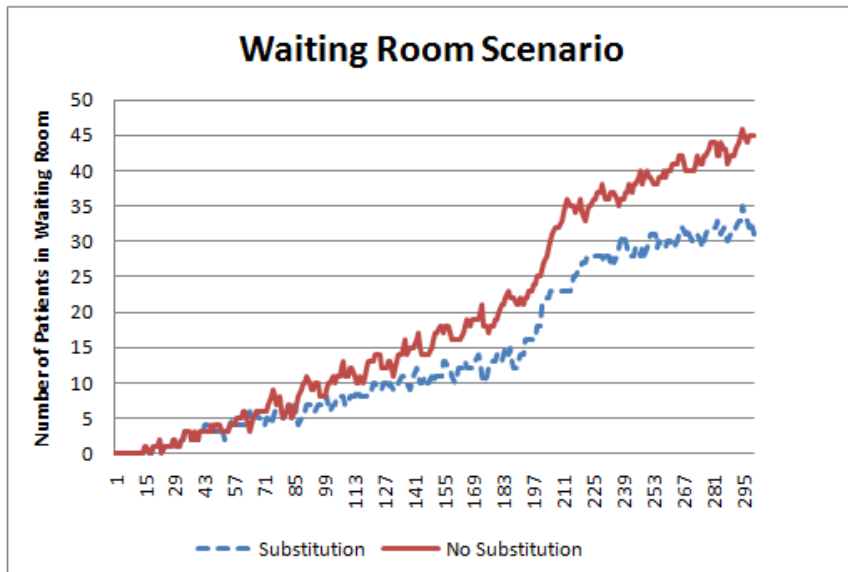


Figure 6.7: The impact of dynamic substitution in the waiting room.

As one would expect, allowing dynamic substitution impacts the LOS of the patients positively as evidenced by the level of crowdedness in the ED. It is our con-

ture that such dynamic substitution allows us to model complex resource usage in a real world dynamic environment more accurately.

Like the last experiment, here again, we were interested in studying the ease of setting up such simulations using our ROMEO-JSim infrastructure. We found once we decided on the substitution we wanted to experiment with, all the changes in the setup to devise the new simulation experiment took us less than thirty (30) minutes. However, we did not stumble upon an interesting substitution at our first attempt. A few hours of trying different combinations led to the substitution rules we presented here. But it seems important to note that the power and flexibility of our approach to resource specification and management enabled us to try many different combinations in the space of a few hours, thereby materially facilitating the progress of this experimentation.

The above dynamic substitution scenarios point to a domain policy where a nurse dedicated to performing Triage is given the responsibility of some other tasks when the ED gets overcrowded. The initial results reported in figure 6.7 led to the question of whether it might not actually be more realistic to model a somewhat more complex domain policy stating that a triage nurse is allowed to substitute for a regular nurse in some activities only when there is at least one triage nurse left available to attend to a newly arrived patient. To model this more complex domain policy, all we had to do was to add the following condition in the **GuardFunction** table. This table, in the ROMEO resource repository, keeps track of the execution state dependent dynamic *capabilities* of resource instances. The time for changing this setup was less than five (5) minutes.

<b>ServiceName</b>	<b>ResourceGroup</b>	<b>Condition</b>
PlacePatientInBed	TriageNurse	StateServer.PendingRequests("bed") > N && StateServer.Available("TriageNurse") > 1

Table 6.3: Elaboration of substitution condition for triage nurses

We repeated the experiment with this additional constraint governing the dynamic substitution of resource instances. Figure 6.8 shows the impact on average LOS of patients as a result of allowing dynamic substitutability of resource instances following different domain policies.

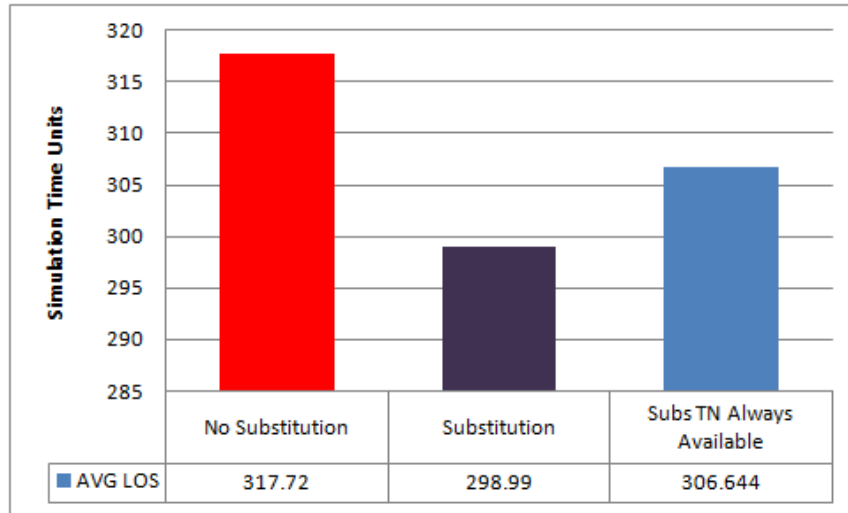


Figure 6.8: Impact on LOS with dynamic substitutions.

### 6.2.3 Dynamically Changing Process based on Resource Availability

In articulating his ideas regarding the potential future direction for research related to resource scheduling [76], Prof. Stephen Smith suggested that one of the most promising and least explored areas of resource scheduling research is the combination of adaptive planning and dynamic provisioning of resources. His paper also indicated that existing tools are not very effective in studying such dynamic combination. In this section we explore the ability of our resource management and simulation framework to support some dynamic changes in the process flow based on the runtime availability of resource instances and its impact on LOS for patients. In this experiment, we have used a more elaborate ED process than the ones presented earlier ('VerySimpleED' and 'SimpleED'). We shall refer to this process as 'EDCare'.

This process, as shown in figure 6.9, models exceptional situations and handling of exceptional flow. It also captures more parallelism, specifies more detail using more steps, and uses pre and post requisites in Little-JIL language all in attempting to get closer to the real world patient care process in an actual hospital ED.

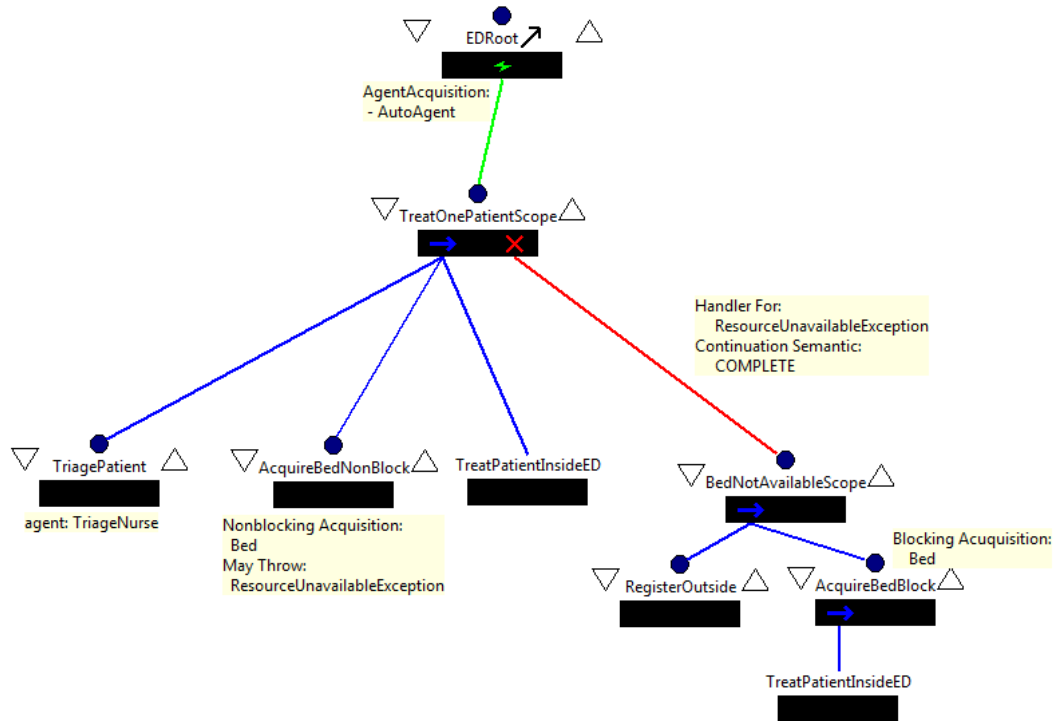


Figure 6.9: The root diagram of ‘EDCare’ process

Consultation with our ED domain expert (an ED physician and ex-director of a large ED in the United States) resulted in investigating the simulation of an ED process in which different incoming ED patients follow different paths depending on the degree of availability of `bed` resource instances. In particular, the domain expert was interested in looking at the impact of a policy where after going through triage, a patient is placed immediately into the main-ED without waiting for registration to be done. In some large hospitals, there are facilities to perform a two step registration inside the treatment area of the ED, known as *quick registration*. In a quick regis-

tration scenario, a clerk collects minimal information about the patient to generate an id-band and then the rest of the registration, which includes collecting of such information as insurance etc., is completed in parallel with the treatment process. In other words, the treatment of the patient is started immediately after triage if beds are available and the registration is completed at some point during the patient's stay at the ED. This scenario is modeled in figure 6.9. The nominal flow of this process takes a patient through the TriagePatient step, and then immediately tries to acquire a bed through the AcquireBedNonblock step. If the bed acquisition is successful, the flow of the process continues to the process sub-tree rooted at TreatPatientInsideED. In case the step AcquireBedNonblock fails to acquire a bed immediately it throws a ResourceUnavailable exception which propagates up to TreatOnePatientScope step, where a separate process (BedNotAvailableScope) is defined to handle the ResourceUnavailable exception. The handler process, starts by performing the step RegisterOutside and then tries to acquire a bed with a blocking request call.

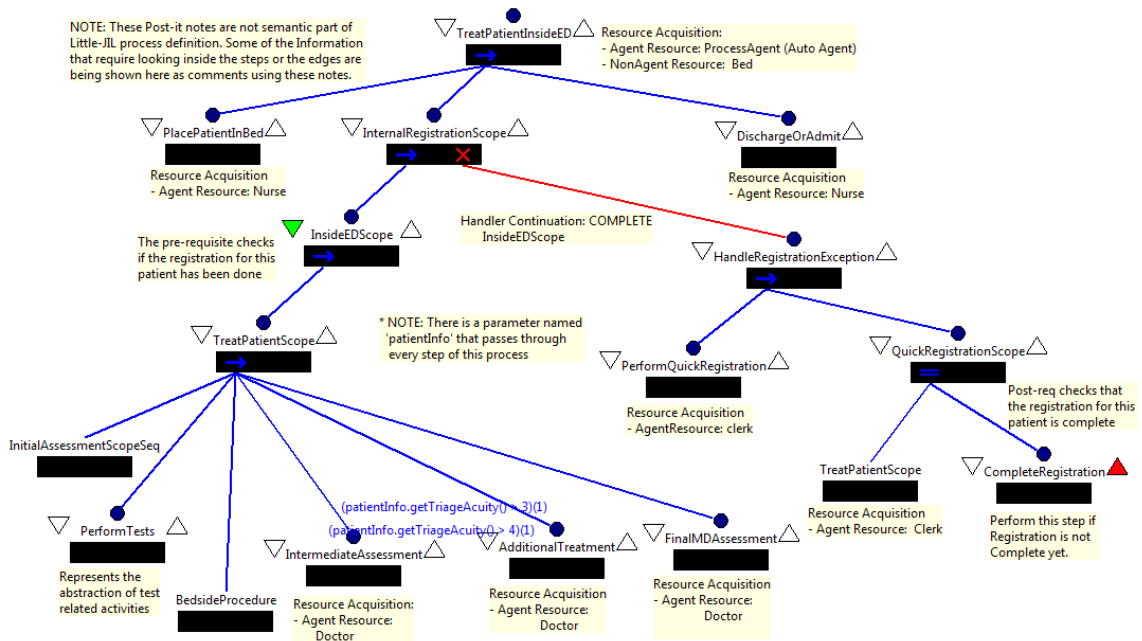


Figure 6.10: The patient care process inside the treatment area in EDCare

Figure 6.10 shows the process inside the treatment area of the ED once a `bed` has been acquired and the patient has been placed inside the main-ED. The process is largely self-explanatory with comments placed using yellow ‘post-it’ notes regarding resource acquisition and resource usage. Like other ED processes presented in this study, there is a parameter named *patientInfo* that is instantiated with the information about each patient and flowed through the process from step to step as an artifact. There is a Boolean field in the *patientInfo* object named *isRegistrationComplete* that describes whether registration has been completed for a patient or not.

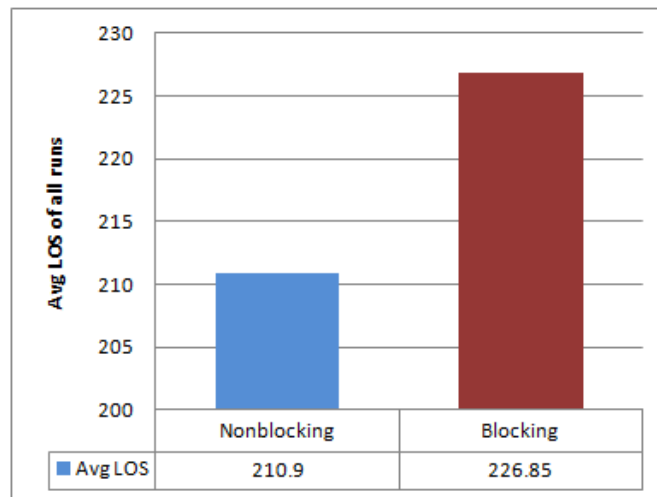


Figure 6.11: The impact of blocking and nonblocking bed acquisition

If a patient goes through the `RegisterOutside` step, we specify the agent behavior such that after successful completion of the test, the *patientInfo* parameter flowing out of this step will have the value *true* for *patientInfo.isRegistrationComplete* field. We specify this behavior using JABS [83] syntax in the JSim input configuration file. While executing `InsideEDScope`, JSim will check to see if the pre-requisite is satisfied. In this process, the pre-requisite checks that registration for this patient has been completed. If the pre-requisite fails, it will throw a `RegistrationNotDone` exception, which will propagate to its parent step, `InternalRegistrationScope`, where

the exception will be handled. The handler subtree will then specify that the patient go through `QuickRegistration`, with registration being completed in parallel with other treatment related steps. We ran the simulation a number of the times with the following configurations:

- Patients: 100
- Beds: 20
- Doctor: 4
- Nurse: 4
- Clerk: 2
- TriageNurse: 2

For this set of simulation runs, the patient arrivals were generated using a *Poisson* distribution with mean inter-arrival time of six (6), which translates into roughly ten (10) patients per hour. The execution times of the steps were specified using a triangular distribution. We ran each configuration of the process five (5) times and looked at the average of all the average LOS measures from the simulation runs. Figure 6.11 summarizes the output of these simulations. As intuitively expected, the situation where a patient is immediately placed inside the ED when a bed is available results in improved patient flow. However, like earlier experiments, our focus was also on observing how easy or difficult it is to set up our simulation and resource management infrastructure for such an experiment. So, we timed ourselves in doing the simulation setup. The process augmentation required about 2 hours and 24 minutes. Once we had the process model elaborated, switching from a non-blocking request scenario to the blocking request scenario required less than twenty (20) minutes to complete.



#### 6.2.4 Impact of Request Priority

In this experiment, we looked at the impact of specifying relative priorities for resource requests, based on the step from which a request is generated. We used the ‘VerySimpleED’ process shown in figure 5.2 as the basis for this set of simulations. We were interested in simulating what impact, if any, is observed if the simulation specifies the order in which an agent does tasks based upon their specified priority. We were also interested in illustrating how our model would specify such an ordering based on the relative priorities of the process steps. To illustrate the scenario with an example, consider the case where there are two tasks for two different patients, patient-1 and patient-2, that can be assigned to a doctor. Let us suppose the task of `MDFinalAssessmentAndDecision` for patient-1 and the task of `MDInitialAssessment` for patient-2 are both ready for the doctor to perform. If a doctor who can perform both the tasks becomes available, and if both tasks are assigned to such a doctor, what impact does it have if the doctor always does one task before the other. For example, if we put a relative higher priority on `MDFinalAssessmentAndDecision` over `MDInitialAssessment`, this means the doctor always gives higher priority to discharge-related tasks over attending to a new patient. The intuitive idea behind experimenting with such a domain policy would be to investigate if such a policy can improve patient flow in a crowded ED.

Task Groups	Step Name	Agent Resource
Group 1	<code>MDInitialAssessment</code>	Doctor
	<code>RNInitialAssessment</code>	Nurse
Group 2	<code>MDProcedure</code>	Doctor
	<code>RNProcedure</code>	Nurse
Group 3	<code>MDFinalAssessmentAndDecision</code>	Doctor
	<code>RNPaperwork</code>	Nurse

Table 6.4: Task groups for relative priority experiment

<b>Step Name</b>	<b>Min</b>	<b>Mode</b>	<b>Max</b>
TriagePatient	3	5	10
RegisterPatient	5	7	10
RNAssessment	5	7	10
MDInitialAssessment	5	10	15
PerformTests	30	30	30
RNProcedure	5	15	30
MDProcedure	5	15	30
MDFinalAssessmentAndDecision	5	7	10
RNPaperwork	5	10	15

Table 6.5: Triangular distribution of the step execution times in VerySimpleED

To assign relative priorities, we identified three task (step) groups. Table 6.4 shows these groups and the agent resource requirement specified in each Little-JIL step. The requests associated with each of the above steps also declares a numeric value for its priority specification. We set up three different scenarios with different priority combinations for the task groups. For this set of simulations, we used a triangular distribution for the step execution times. Table 6.5 shows the step execution times we used.

We ran each simulation configuration five (5) times and computed the average LOS each time. Table 6.6 shows the average of all runs for each different priority combination. Although small, the simulation results show a modest improvement in the average LOS when we gave discharge related tasks higher priorities than other tasks. The results also show that for this particular process and patient arrival scenario, the best priority combination was to have higher priority for discharge related tasks (`MDFinalAssessmentAndDecision`, followed by tasks that are performed toward the beginning of the patient care process (`MDInitialAssessment` and `RNInitialAssessment`), followed by tasks that are performed on patients at the middle of their stay at the ED (`MDProcedure` and `RNProcedure`).

This experiment, like the other presented in this chapter, was not meant to suggest that we have succeeded in obtaining definite answers to questions about which

prioritization policies produce reduced LOS results in the ED. Rather, our focus was to demonstrate that such interesting ‘what-if’ questions can lead to interesting observations. As usual, we also tried to get a sense of how easy or difficult it was for us to set up this experiment using ROMEO-JSim. We used a novice user of ROMEO-JSim to setup this experiment and clocked how long it took for the user to set up the required configuration. We found that changing the relative priorities to set up each configuration of the experiments took a small number of minutes (less than 10 minutes in each case).

RNAssessment	RNProcedure	RNPaperwork	Length of Stay
MDInitialAssessment	MDProcedure	MDFinalAssessment AndDecision	
No Priority	No Priority	No Priority	699.26
Low Priority	Low Priority	High Priority	684.00
Medium Priority	Low Priority	High Priority	663.77

Table 6.6: Impact on LOS based on different priority combination

### 6.3 Resource Sharing in a Multi-department ED

The final simulation related case study we present was performed on an elaborate patient care process that modeled two separate departments within an ED, namely main-ED and fast-track-ED. In this experiment, we modeled patients with different acuity levels and used that acuity information to decide at run time which department a patient was going to be placed in. ROMEO modeled resource instances that included information such as which department (main-ED or fast-track-ED) the resource instance was a part of. This process also elaborately modeled different tests and bedside procedures that are usually ordered on patients visiting an ED. Figure 6.12 shows the root diagram of this process. For the purposes of this discussion, we shall refer to this process by the name ‘EDCare2’.

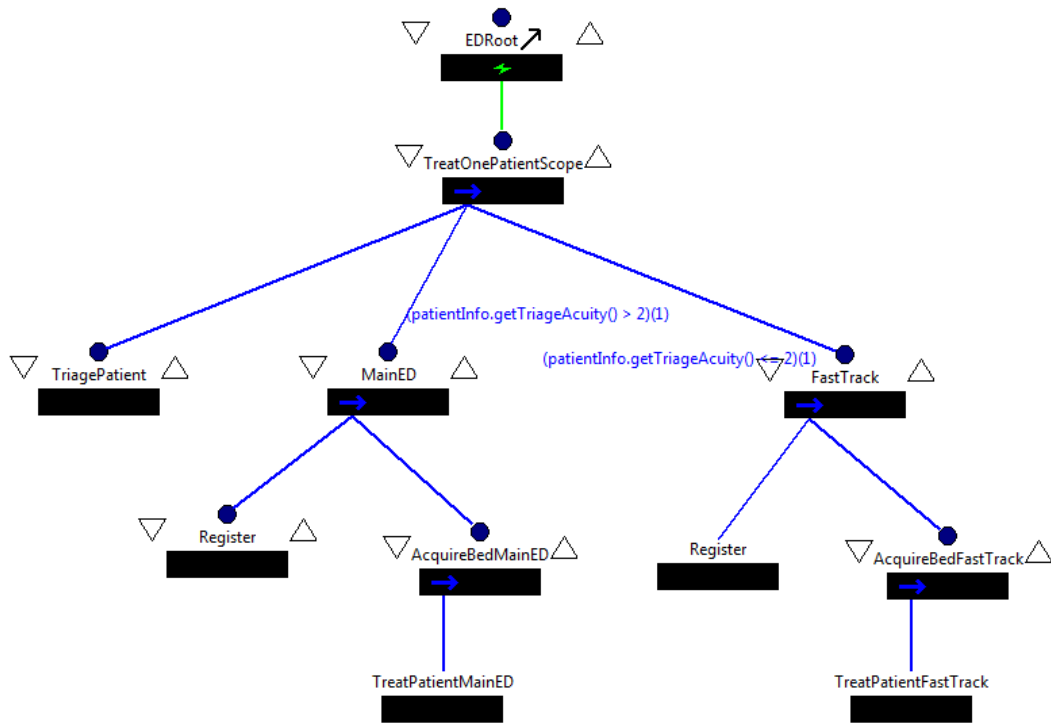


Figure 6.12: Root diagram of EDCare2 process

The ‘EDCare2’ process was developed by elaborating the ‘EDCare’ process introduced in section 6.2.3. The immediately noticeable difference in the root diagram is the presence of two separate sub-trees rooted at the **MainED** step and the **FastTrack** step. The edges connecting these two steps to their parent have predicates associated with them that evaluate the *patientInfo* parameter’s *triageAcuity* value. If the patient’s acuity is less than or equal to two (2), the patient is placed in fast-track-ED, otherwise the patient is placed in the main-ED.

Figure 6.13 shows what steps a patient goes through once s/he is placed inside the main-ED. This is similar to the main-ED part of the ‘EDCare’ process. However, the ‘EDCare2’ processes, **PerformTests** and **BedsideProcedureMainED** refer to elaborated process structures shown in another diagram. The primary difference between the main-ED and fast-track-ED subprocesses are that the former has more tests, bedside procedures, and assessment steps than the latter. Figure 6.14 shows some elabora-

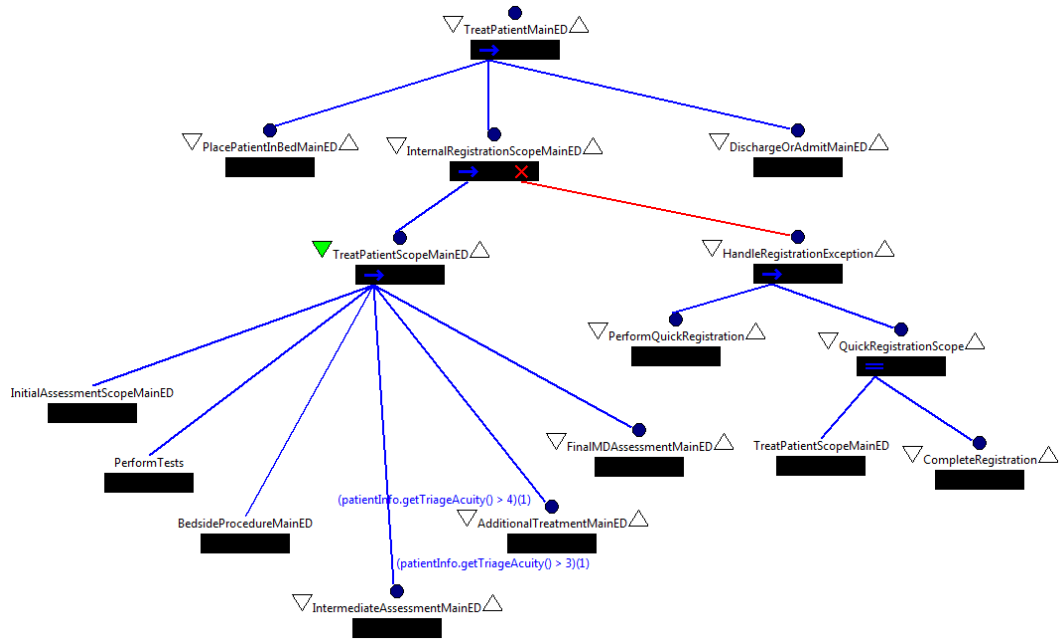


Figure 6.13: Patient care inside main-ED in EDCare2 process

tion of the **PerformTests** subprocess. Each test in this process model is guarded by predicates depending on the Boolean valued fields within the *patientInfo* parameter object. Note that **PerformTests** is a parallel non-leaf step, which indicates that the tests ordered on a patient can take place in any order including simultaneously. However, there is a **PatientConstraint** resource collection constraint specified with a maximum cardinality of one (1). Each of the testing steps (i.e. **ObtainSample**, **PerformXray**, **PerformECG**, **PerformCTScan**), although not **PerformLab**, needs to acquire a **Patient** resource instance constrained by this constraining collection. ROMEO, when presented with the acquisition request for a **Patient** resource with the constraining collection of maximum cardinality one (1), ends up only allowing one of the steps to be successful in acquiring the patient. Thus, this constraining mechanism allows us to model the constraint that all the steps that require the patients physical presence may occur in any order but no combination of them may take place concurrently.

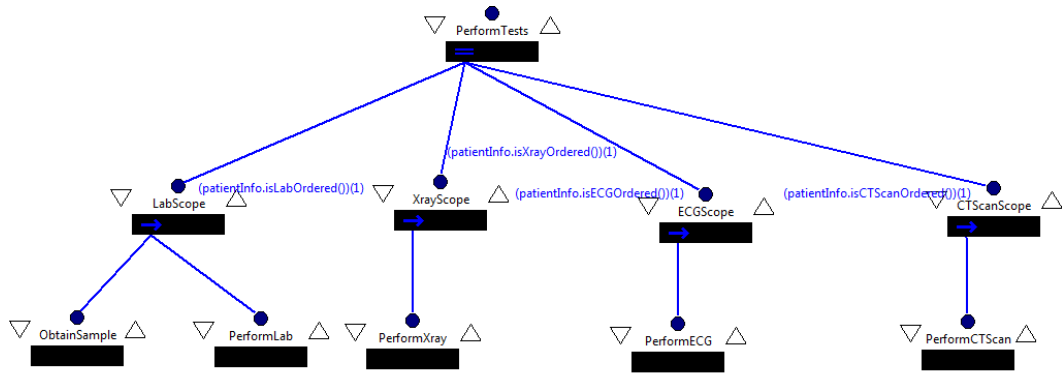


Figure 6.14: Elaboration of step PerformTests

Figure 6.15 shows the bedside procedures that have been modeled in the ‘EDCare2’ process for patients going into the main-ED. We show the rest of the processes in appendix ???. With this detailed ‘EDCare2’ process, we were interested in running a set of simulations that respected the following domain policy:

If a patient is waiting in the main-ED to be seen by a **doctor** and if all doctors in the main-ED are busy but a **doctor** in the fast-track-ED area is available, that **doctor** can be assigned to see the patient in the main-ED. However, main-ED **doctor** is not allowed to see a fast-track-ED patient even if the **doctor** is available while all fast-track-ED doctors are occupied.

We specified this constraint by using a preference specification in the resource request. Specifically, in all the steps where a **doctor** is required in the main-ED, we used the query, *prefer(doctormained, doctorfasttrack)*. Figure 6.16 shows an example of the resource requirement specification for step MDAssessmentMainED. ROMEQ, when presented with this preferential list of queries, tries to fulfill the request with the leftmost query. If no resource instance can be found to be available to satisfy the preferred query, ROMEQ, goes through the preferential order attempting to satisfy the request. For the steps in fast-track-ED process that require a **doctor**, we did not specify any such preferential request.

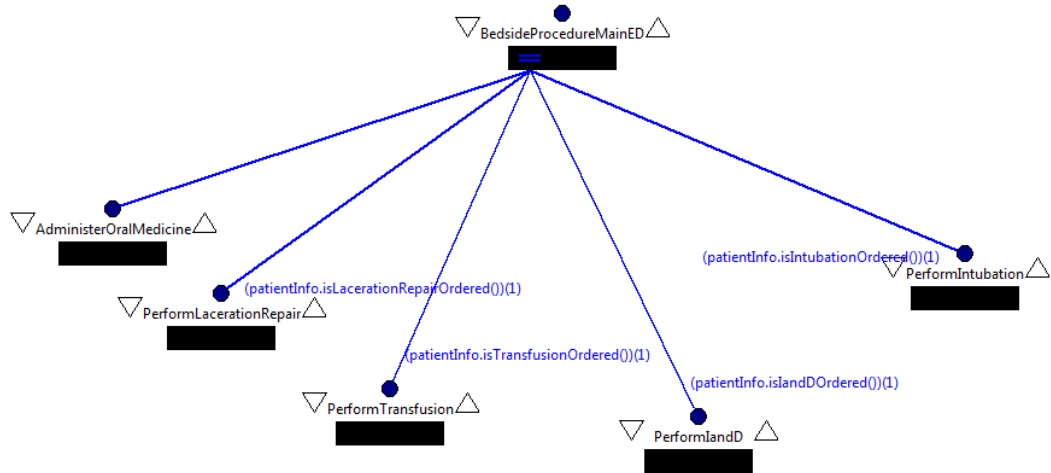


Figure 6.15: Elaboration of step BedsideProcedureMainED

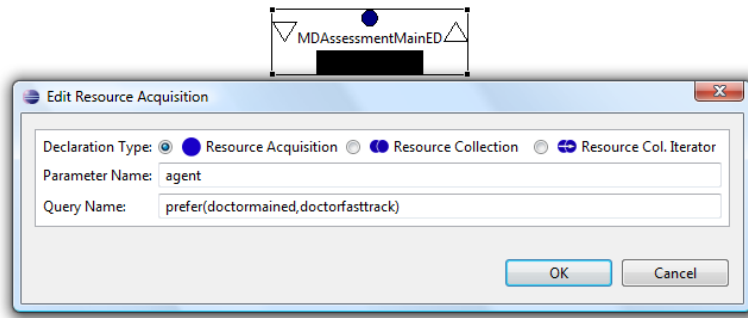


Figure 6.16: Resource requirement specification for the MDAssessmentMainED step

With the above-described process, we set up some simulation runs. Table 6.7 lists the different resource mixes used for these simulation runs. The patient characteristic inputs to these steps were all distributed according to a triangular distribution based on expert opinion provided by our ED domain expert. The patient arrival scenario was generated using a *Poisson* distribution with an inter-arrival mean time of 9 minutes. Patients' acuities were determined randomly upon arrival. In the generated mix, 10% of patients were given an acuity of 1, 20% were given an acuity of 2, 30% were given an acuity of 3, 20% were given an acuity of 4, 10% were given an acuity of 5, and 10%

Resource Group	Available
Patients	100
MainED Beds	25
FastTrack Beds	5
MainED Doctors	4
FastTrack Doctors	1
MainED Nurses	4
FastTrack Nurses	1
Clerks	2
TriageNurses	2

Table 6.7: Resource mix for running simulations with ‘EDCare2’ process

were given an acuity of 6. We simulated the process first with the preferential resource requirements and then ran them again by removing the preferential specification from the requests. Like other experiments, we studied the impact of this change in the process to patient LOS. The average of the output of all simulation runs is shown in figure 6.17

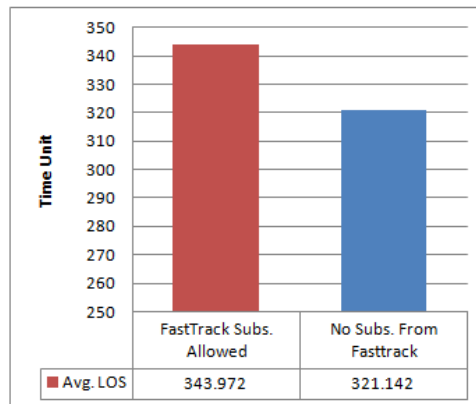


Figure 6.17: The impact of allowing fast-track doctor in main-ED

Initially, this result may seem to be somewhat counter intuitive. One might expect the results to show that if a doctor is brought in from the fast-track-ED when the doctor is sitting idle, it should improve the patient flow and reduce LOS. However, figure 6.17 shows the opposite. One root cause of this behavior might be the



same-doctor constraint that we have specified in the ‘EDCare2’ process. Note that when a patient is placed in the main-ED s/he goes through more steps involving a doctor than if s/he is placed in the the fast-track-ED. Because of the same-doctor constraint, however, the fast-track-ED doctor becomes responsible for the patient for all subsequent treatments. Since we had only one fast-track doctor in these simulations, this means that one fast-track-ED doctor might potentially wind up being assigned relatively more work than others. Examination of the resource utilization outputs produced by ROMEO showed that this is indeed what had happened. Doctors in the main-ED had an average 13.54% utilization level, but the fast-track-ED doctor had an average utilization level of 31.39%.

Once again, with this case study, we did not aim to find a definitive answer regarding the impact of some specific domain policies on an ED. We were more interested in studying how easy or difficult it is for someone to set up such an experiment and to run the simulation. We gave a novice user (someone who has been acquainted with Little-JIL, JSim, and ROMEO for less than two months) of ROMEO-JSim the task of setting up and running these simulations. Elaboration of the process, setting up of the new resource model, and specifying the resource requirements for the process steps were all done by this user in a little less than four hours and thirty minutes. This indeed was a very encouraging sign regarding the usability and flexibility of our simulation infrastructure.

## **6.4 Experiences with Processes in Other Domains**

The primary evaluation vehicle for the studies presented in this dissertation has been the ROMEO-JSim simulation infrastructure, especially as it has been applied in a particular domain, namely the patient care processes in a hospital ED. We developed ROMEO, however, to be incorporated as a key component of a generic architecture that is capable of supporting other types of applications in different domains,

running on different platforms. To evaluate ROMEO and the resource management approaches it implements in such other applications, domains, and platforms, we integrated ROMEO into Juliette, an execution engine for the Little-JIL language. We then tested ROMEO's effectiveness in modeling resource instances, requests, and constraints, and in supporting execution-time resource allocation, in a completely different domain supporting execution of a large process. To this end, we chose a process that has been developed to drive an *Online Dispute Resolution* (ODR) activity.

The specific dispute resolution, or mediation, process we have experimented with was developed in collaboration with the National Mediation Board (NMB), the U.S. government agency charged with resolution of all labor-management disputes in the U.S. transportation industries (principally airlines and railroads). The need for mediating disputes has been growing steadily without commensurate increases in human resources at NMB. Thus NMB was very interested in incorporating process-based ODR into their activities. [44, 21, 75] describe work that has been done in developing, using and analyzing these mediation processes. In this work, the ODR processes were developed with the aim of gaining a better understanding of NMB mediation process requirements, training new mediators, and supporting NMB's process with automation. It is the required support of the resource management service in facilitating the automation of this mediation process is what we have focused on. The mediation process we experimented with is a very large and complex one with more than 100 steps, and with numerous instances of parallelism and numerous instances of exception handling. Of particular interest to us was the fact that there are some types of resource requirements in this ODR process that were not present the ED processes we have discussed so far, notably the use of a *Resource Iterator* constraint.

The primary agent resource instances specified in NMB's dispute resolution process are a **Mediator** and a number of **Participants** (disputants). The process assumes that there are two sides in every dispute, with each side being represented by

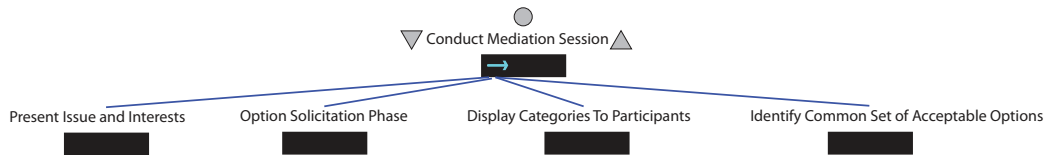


Figure 6.18: High level view of the mediation process

one or more disputing **Participants**. The resource requirements declared for some steps specified such constraints as that the agent resource instance for the step needed to be a **Participant** from a side that is the same as, or opposite to, the side of a **Participant** that performed some other step. This was modeled by specifying certain parts of the process as many times as instances of the **Participant** resource type was available at run time. Figure 6.19 shows an example of how this was specified using the Little-JIL process definition language, which was a key part of the platform for our experimentation with the ODR process.

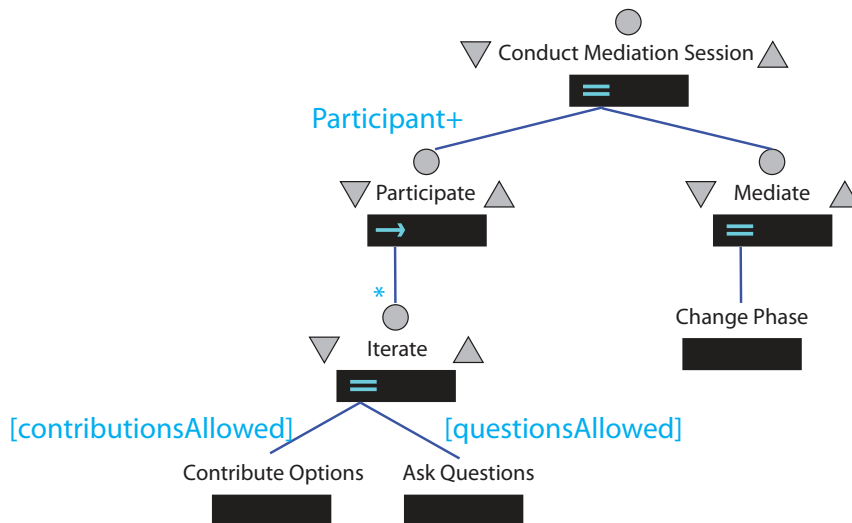


Figure 6.19: Elaboration of a part of the mediation process

Figure 6.18 shows the high level definition of the The Little-JIL process definition, which can be viewed as a carefully structured multi-party conversation. The nominal

process defines a set of sequential steps where the issues underlying a dispute are first elucidated by the **mediator**, followed by submission of ideas by the **disputants**, then summarization of these ideas by the **mediator**, and finally iteration by the **mediator** through all of the suggestions for a resolution that had been submitted by the **disputants**. This process defines this iteration to continue until agreement on a resolution has been reached, or until it is agreed that agreement cannot be found.

For reasons described in [75], the high-level process shown in figure 6.18 was elaborated and refactored to a process that describes a more role-oriented view of how this sort of mediation is performed by NMB. Figure 6.19 shows an important portion of that process model. In this process, the **Participate** step requires a **Participant** type of agent resource instance and the **Mediate** step requires a **Mediator** type of resource instance. For the **Participate** step, the annotation *Participant+* on the incoming on edge to that step specifies that a request for a **Participant** type of agent resource instance will be sent to the **Resource Manager** iteratively for as long as the **Resource Manager** continues to succeed in assigning an instance of the **Participant** resource in response to the previous one of these requests. The execution engine (Juliette), in turn, will instantiate the sub-tree of the process rooted at step **Participate** for each agent resource instance assigned by the **Resource Manager**. This is internally accomplished by Juliette sending a request to ROMEO to declare a *Resource Iterator* constraint using the query name *Participant*. Each subsequent agent resource acquisition request for step **Participate** is sent to ROMEO with this *Resource Iterator* constraint.

Once initial systems work required to integrate ROMEO with the the Little-JIL execution environment had been accomplished, the configuration of ROMEO to support execution of the mediation process was simple and quick. It took us less than half a day's work to define the resource instances, and their attributes and capabilities required for the mediation process. We note that this process also required

the specification of a number of non-human agent resource instances (e.g. agents to manage a data repository, anonymize participant contributions, etc.) in addition to the `Participants` and `Mediator`. Our success in being able to support the execution of a large process such as the above described mediation process using our prototype Resource Manager, ROMEO, gives us additional confidence about the effectiveness of both our resource management architecture and the ROMEO prototype implementation as well.

## CHAPTER 7

### CONCLUSIONS AND FUTURE WORK

Modeling and managing entities that we commonly refer to as resources seems to be a ubiquitous problem in systems in many different domains. This dissertation aimed to identify some basic, as well as special, problems encountered in managing resources in environments that are quite complex and dynamic. We also intended to find out the separate concerns software engineers need to consider while developing resource management services that are able to address these problems in such domains. We also wanted to study ways to support the implementation and enforcement of complex domain policies regarding resources in such highly dynamic environments. We proposed to develop a generic resource management service architecture, build a prototype, and evaluate our approach by primarily driving simulations of patient care service in a large and busy hospital ED. The evaluation was also aimed at looking at the applicability of our approach to other domains that are very different from a hospital ED.

#### **7.1 Summary of the Research**

In this dissertation, we have first articulated the generic resource management problem we decided to study. We have proposed a set of definitions to precisely define such notions as resource instances, request structures, domain-specific resource management constraints, and objectives of resource allocation activities. We have enunciated our proposed ideas about representing the dynamic nature of some resource instances in a complex environment. We have identified a set of separate con-

cerns that need to be addressed by software engineers who need to develop systems that provide resource management services in such complex and dynamic environments. We have also proposed a generic resource management service architecture and developed a prototype following this architecture. We have presented in detail the major components of this prototype and specified their interactions in a generic **Resource Manager**. Using our proposed design, we have developed ROMEO, a resource management service and customized it to serve a task coordination framework based on Little-JIL process definition language. Our work then concentrated on evaluating the effectiveness of ROMEO in supporting simulations and executions of complex processes. For this evaluation purpose, we developed a simulation infrastructure named JSim on top of Juliette, Little-JIL's execution environment. We ran a variety of simulations of patient care processes in EDs using our ROMEO-JSim infrastructure. We also used ROMEO to support the actual execution (rather than just the simulation) of a large mediation process.

A central premise, hypothesized and explored in this thesis, was a novel way of thinking about resource instances in dynamic domains, namely defining them with a set of *guarded capabilities*, some of which may be dependent on the execution state of the system. This led us to think about how to represent execution states of a running system and what types of system state information might be important for representing the *guard functions* on the *capabilities* of a resource instance that define the resource instance's ability to satisfy a request at a given execution state of the system. We have also identified a small set of common types of attributes of resource instances that seem able to support specification of a large variety of resource instances in complex domains. We believe that our research supports our hypothesis that specifying resource instances as having sets of *guarded capabilities* provides a useful abstraction for modeling many of the complex dynamic behaviors of resources instances in such domains as hospital EDs.

One important component of our work was to look closely at what is needed in order for a resource request language to specify complex resource requirements, many of which are often dependent on the run time state of an executing system. One of the contributions of this research has been to come up with a relatively simple request model, which has turned out to be remarkably powerful in describing complex runtime requirements on the allocation of resource instances. Our research made it clear that domain policies in dynamic and complex environments can usually be translated into constraints on resource allocation. Our request specification language has allowed us to model a wide variety of interesting domain policies in the ED domain. We believe that our approach and tools are capable of modeling many more domain policies than the ones we have demonstrated in our case studies.

We have found that the separate concerns we identified have helped us to develop a flexible resource management architecture. These concerns not only helped us to identify the components of a **Resource Manager**, but also facilitated the discovery of an effective interaction model among the components. Our experience has also shown that the architecture proposed in this dissertation can nicely support a task coordination framework such as the Little-JIL technology that was used in our experimentation. Our experience with ROMEO, the prototype **Resource Manager** we built, has been quite gratifying. We have successfully developed a simulation infrastructure that is supported by ROMEO. This simulation environment, ROMEO-JSim, has been successfully used to simulate a number of different ED scenarios. Due to the factored architecture of the environment, there are many well defined mechanisms for supporting the specification of detailed configuration information. Our experimentation has indicated that these mechanisms were sufficient to support rapid tuning of resource management behavior need to quickly set up interesting ‘what if’ question regarding ED simulations. This in turn supported expedited iterations among simulation



runs, thereby facilitating the study of relative effectiveness of different resource mixes, policy decisions, and process configurations in meeting needs in the ED domain.

It is important to note that in designing and implement our ROMEO-JSim infrastructure our emphasis was on expressive power and flexibility. As a result ROMEO-JSim is not optimized for performance, and executes considerably more slowly than commercial tools such as Arena. Consequently, simulation runs can take a lot longer with ROMEO-JSim than with Arena. This seems to represent yet another instance of a situation in which there is probably a software engineering trade off that can be made for flexibility vs. performance. We suggest that having the capabilities to support quickly setting up different simulation scenarios facilitates the investigation of interesting questions. In situations where rapid setup of very diverse and demanding simulations is desired, foregoing speed is probably a tolerable trade-off. Moreover, we believe that the speed of our prototype can almost certainly be improved considerably with additional engineering effort.

Besides simulating hospital EDs, we have successfully integrated ROMEO to test its ability to support live execution of a mediation process. The resource requirements in the executable mediation process led us to exercise some more features of our resource and request modeling capabilities.

## **7.2 Future Directions for the Research**

This dissertation has produced a useful baseline from which exciting future research in a few different directions can be pursued. Some continuations of this work may require long term research, while some might be much more immediate. The following discussion includes a few promising and important future directions.

### 7.2.1 More validation of ROMEO-JSim

Discrete event simulation (DES) is an effective and highly popular method for performing ‘what if’ type of analyses of complex systems. Almost all DES environments focus on the resource usage patterns of the systems under simulation. However, the model of the real world and its use of resources, especially under complex constraints, often has to be simplified in order to support these studies. In the simulation studies we have discussed in our related work section in chapter 2, contained very few attempts to validate model and simulation results by matching them with input and output data from the real world. The ones that have attempted to perform such validation often failed to do so. Connelly and Bair’s [22] work is a good example of that failure.

We conjecture that the types of detailed modeling capabilities, such as dynamic resource substitutability and resource constraint specification, that we have incorporated into ROMEO-JSim infrastructure should provide a better basis for recreating complex real world scenarios more closely, thereby offering a better chance that simulation results might be more successful in matching observed real world behaviors. Evaluating this conjecture seems to require elaborating such (e.g ED) processes in more detail and running simulations based on actual data collected from real world processes. Such evaluation will also require elaboration of the specifications of resource instance characteristics, specifications of incoming patient mixes, and ED operational scenarios such as shift changes. We believe that ROMEO incorporates facilities for supporting the detailed specifications of all of these types of information. Thus, one interesting direction for future work is to design simulation experiments that exercise these capabilities, and determine the degree to which they can be successful in supporting simulations that match actual observed behavior. Invariably we expect that detailed specifications of some types will be of more value than others in supporting replication of real-world behaviors. Identification of the relatively more

effective types of specification could then be important in suggesting which types of specification warrant further research.

### **7.2.2 Infrastructure Improvement**

Although ROMEO-JSim is capable of modeling and simulating complex processes quickly, the specification of input configurations and organization of simulation outputs are still largely done manually. There is considerable need, and ample opportunity, for improving the simulation infrastructure so that it becomes more user-friendly and convenient to use. Taking the lead from commercial products, we have added some convenience features to ROMEO-JSim. There is still much to be done, however. An important part of this improvement is to create a useful user interface for setting up and running new simulations. Our initial exploration in this direction has suggested that this task can be a lot more complex than it initially appears to be. In fact, there is likely to be some very interesting user-interface research in this area.

Another closely related area of future work is to investigate how best to organize and present simulation results to users. Presenting the simulation results in the right way should lead to the formulation of more interesting questions. Thus it can be worthwhile to investigate how best to capture and represent the outputs of ROMEO-JSim simulations.

### **7.2.3 Experimenting with Intelligent Scheduling**

Stephen Smith, one of the leading researchers on scheduling in recent times, argues that although significant milestones have been achieved in scheduling research, there is still a lot to be done in this area [76]. In his invited paper regarding the future of scheduling research, he pointed out many advances of scheduling research in different domains. However, Prof. Smith suggested that most solution techniques studied so far defined scheduling as a static, well-defined optimization task like some sort of puzzle solving activity. But scheduling is typically an ongoing iterative process,

and thus there seems to be considerable room for improvement in the heuristics and other scheduling techniques that have been developed to derive schedules that can be expected to come close to matching results obtained from exhaustive searches. Indeed our observation of the ED domain is consistent with the observation that scheduling is best done dynamically, as the demands and constraints of the domain can be expected to change dramatically and quickly.

As one of the major challenges of this research area, Prof. Smith identified the need for generating schedules under complex constraints, objectives, and preferences that arise directly from the actual domain. We observed this to be the case in the ED domain, and we believe that our resource management approach and ROMEO have been designed to meet these challenges, and should be able to deliver encouraging results. Thus the JSim-ROME0 infrastructure should be a promising vehicle for exploring integration of different intelligent scheduling techniques. This, we believe, would be a fruitful direction of future research.

## BIBLIOGRAPHY

- [1] *BPEL4WS Specification, Version 1.1*. IBM, 2005.
- [2] *BPMN 1.0: OMG Final Adopted Specification*. Object Management Group, 2006.
- [3] Aalst, Wil M. P. van der, and Hofstede, A. H. M. ter. Yawl: Yet another workflow language. Tech. rep., Eindhoven University of Technology, 2002.
- [4] Allen, Rob. Workflow: An introduction technical report. Tech. rep., Open Image Systems Inc., 2001.
- [5] Aron, Mohit, Druschel, Peter, and Zwaenepoel, Willy. Cluster reserves: A mechanism for resource management in cluster-based network servers. In *In Proceedings of the ACM SIGMETRICS Conference (2000)*, pp. 90–101.
- [6] Baader, Franz. *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press, 2003.
- [7] Baesler, Felipe F., Jahnsen, Hector E., and DaCosta, Mahal. Emergency departments i: the use of simulation and design of experiments for estimating maximum capacity in an emergency room. In *WSC '03: Proceedings of the 35th conference on Winter simulation (2003)*, Winter Simulation Conference, pp. 1903–1906.
- [8] Banga, Gaurav, Druschel, Peter, and Mogul, Jeffrey C. Resource containers: A new facility for resource management in server systems. In *Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation (OSDI), New Orleans, Louisiana (February 22-25 1999)*.
- [9] Bechhofer, S., Broekstra, J., Decker, S., M. Erdmann, D. Fensel, Goble, C., Harmelen, F. van, Horrocks, I., Klein, M., McGuinness, D. L., Motta, E., Patel-Schneider, P. F., Staab, S., and Studer, R. An informal description of standard oil and instance oil (white paper). <http://www.ontoknowledge.org/oil/download/oil-whitepaper.pdf>, 2000.
- [10] Berners-Lee, Tim, Wendler, Jan, and Lassila, O. The semantic web: A new form of web content that is meaningful to computers will unleash a revolution of new possibilities. *Scientific American* 284 (2001), 34–43.
- [11] Bobrow, Daniel G., DeMichiel, Linda G., Gabriel, Richard P., Keene, Sonya E., Kiczales, Gregor, and Moon, David A. Common lisp object system specification. *SIGPLAN Notices* 23, SI (1988), 1–142.

- [12] Brasseur, M. Le, and Perdreaul, G. Process weaver: from case to workflow applications. In *IEEE colloquium on CSCW and Software Process* (1995).
- [13] Brckers, Alfred, Lott, Christopher M., Rombach, H. Dieter, and Verlage, Martin. Mvp-l language report version 2, 1995.
- [14] Breuel, Thomas M. Implementing dynamic language features in java using dynamic code generation. In *In Proceedings 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems, TOOLS 39* (2001), pp. 143–152.
- [15] Canals, G., Boudjlida, N., Derniame, J. C., Godart, C., and Lonchamp, J. Alf: A framework for building process-centred software engineering environments. In *Software Process Modelling and Technology* (1994), J. Kramer B. Nuseibeh, A. Finkelstein, Ed., John Wiley and Sons, pp. 153–185.
- [16] Cass, Aaron G., Lerner, Barbara Staudt, Sutton, Stanley M., McCall, Eric K., Wise, Alexander E., and Osterweil, Leon J. Little-jil/juliette: a process definition language and interpreter. In *International Conference on Software Engineering (ICSE)* (2000), pp. 754–757.
- [17] Cass, Aaron G., and Osterweil, Leon J. Process support to help novices design software faster and better. In *Automated Software Engineering* (Long Beach, CA, 2005), pp. 295–299.
- [18] Chambers, C., Ungar, D., and Lee, E. An efficient implementation of self a dynamically-typed object-oriented language based on prototypes. *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA) 24*, 10 (1989), 49–70.
- [19] Chandra, A., and Shenoy, Prashant J. Effectiveness of dynamic resource allocation for handling internet. In *3rd USENIX Symposium on Operating Systems Design and Implementation (ODSi)* (New Orleans, LA, 2003), pp. 3–37.
- [20] Chun, Andy Hon Wai, Chan, Steve Ho Chuen, Lam, Garbbie Pui Shan, Tsang, Francis Ming Fai, Wong, Jean, and Yeung, Dennis Wai Ming. Nurse rostering at the hospital authority of hong kong, 2000.
- [21] Clarke, Lori, Gaitenby, Alan, Gyllstrom, Daniel, Katsh, Ethan, Marzilli, Matthew, Osterweil, Leon J., Sondheimer, Norman K., Wing, Leah, Wise, Alexander, and Rainey, Daniel. A process-driven tool to support online dispute resolution. In *dg.o '06: Proceedings of the 2006 international conference on Digital government research* (New York, NY, USA, 2006), ACM, pp. 356–357.
- [22] Connelly, Lloyd G., and Bair, Aaron E. Discrete event simulation of emergency department activity: A platform for system-level operations research. *Academic Emergency Medicine* 11, 11 (2004), 1177–1185.

- [23] Dami, S., Estublier, J., and Amiour, M. Apel: A graphical yet executable formalism for process modeling. *Automated Software Engineering: An International Journal* 5, 1 (1998), 61–96.
- [24] Decker, Stefan, Sintek, Michael, Billig, Andreas, Henze, Nicola, Harth, Andreas, and Leicher, Andreas. Triple - an rdf rule language with context and use cases, 27-28 April 2005.
- [25] Decker, Stefan, Tangmunarunkit, Hongsuda, and Kesselman, Carl. Ontology-based resource matching in the grid - the grid meets the semantic web. Tech. rep., Information Sciences Institute, University of Southern California, 2004.
- [26] Draeger, Margaret A. An emergency department simulation model used to evaluate alternative nurse staffing and patient population scenarios. In *Proceedings of the 24th Conference on Winter Simulation* (Arlington, VA, USA, 1992), J. J. Swain Wilson, D. Goldsman, R. C. Crain, and J. R., Eds., pp. 1057–1064.
- [27] Drevna, Michael J., and Kasales, Cynthia J. Introduction to arena. In *WSC '94: Proceedings of the 26th conference on Winter simulation* (San Diego, CA, USA, 1994), Society for Computer Simulation International, pp. 431–436.
- [28] Du, Weimin, and Shan, Ming-Chien. Enterprise workflow resource management. In *RIDE '99: Proceedings of the Ninth International Workshop on Research Issues on Data Engineering: Information Technology for Virtual Enterprises*. IEEE Computer Society, Washington, DC, USA, 1999, p. 108.
- [29] Duguay, Christine, and Chetouane, Fatah. Modeling and improving emergency department systems using discrete event simulation. *Simulation* 83, 4 (2007), 311–320.
- [30] Evans, Gerald W., Gor, Tesham B., and Unger, Edward. A simulation model for evaluating personnel schedules in a hospital emergency department. In *WSC '96: Proceedings of the 28th conference on Winter simulation* (Washington, DC, USA, 1996), IEEE Computer Society, pp. 1205–1209.
- [31] Fadel, Fadi George, Fox, Mark S., and Gruninger, Michael. A generic enterprise resource ontology. In *The third IEEE workshop on enabling technologies: Infrastructure for collaborative enterprises (WET ICE '94)* (Morgantown, West Virginia, 1994).
- [32] Foster, Ian, and Kesselman, Carl. Globus: A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing* 11, 2 (1997), 115–128.
- [33] Fox, Mark S. The tove project towards a common-sense model of the enterprise. 25–34.
- [34] Fox, Mark S., and Gruninger, Michael. Ontologies for enterprise integration. In *Conference on Cooperative Information Systems* (1994), pp. 82–89.

- [35] Gamma, Erich, Helm, Richard, Johnson, Ralph, and Vlissides, John M. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1996.
- [36] García, Marelys L., Centeno, Martha A., Rivera, Camille, and DeCario, Nina. Reducing time in an emergency room via a fast-track. In *WSC '95: Proceedings of the 27th conference on Winter simulation* (Washington, DC, USA, 1995), IEEE Computer Society, pp. 1048–1053.
- [37] Gere Jr., W. S. Heuristics in job shop scheduling. *Management Science* 13, 3 (1966), 167–190.
- [38] Goldberg, Adele, and Robson, David. *Smalltalk-80: The Language and its Implementation*. Longman Higher Education, 1983.
- [39] Gruber, T. R. Toward principles for the design of ontologies used for knowledge sharing. *International Journal of Human-Computer Studies* 43 (1995), 907–928.
- [40] Hammann, John E., and Markovitch, Nancy A. Introduction to arena. In *WSC '95: Proceedings of the conference on Witer Simulation* (San Diego, CA, USA, 1995), Society for Computer Simulation International.
- [41] Harel, David, and Naamad, Amnon. The statestate semantics of statecharts. *ACM Trans. Softw. Eng. Methodol.* 5, 4 (1996), 293–333.
- [42] Hendler, J., and McGuinness, D. L. The darpa agent markup language. *IEEE Intelligent Systems* 15, 6 (2000), 67–73.
- [43] Hobbs, Jerry R., Lassila, Ora, and Narayanan, Srin. Toward an ontology of resources. <http://www.daml.org/2001/09/resources/>, 2001.
- [44] Katsh, Ethan, Osterweil, Leon J., Sondheimer, Norman K., and Rainey, Daniel. Early lessons from the application of process technology to online grievance mediation. In *dg.o 2005: Proceedings of the 2005 national conference on Digital government research* (2005), Digital Government Society of North America, pp. 99–100.
- [45] Kee, Yang-Suk, Yocum, Ken, and Chien, Andrew A. Improving grid resource allocation via integrated selection and binding. In *International Conference on High Performance Computing and Communication* (Munich, Germany, 2006).
- [46] Khare, R. K., Powell, E. S., Rheinhardt, G., and Lucenti, M. Adding more beds to the emergency department or reducing admitted patient boarding times: Which has a more significant influence on emergency department congestion?, 2009.
- [47] Klyne, Graham, and Carroll, Jeremy J. Resource description framework(rdf): Concepts and abstract syntax. <http://www.w3.org/TR/rdf-concepts/>, 2004.



- [48] Kulkarni, Purushottam, Ganesan, Deepak, and Shenoy, Prashant. Senseye: a multi-tier camera sensor network. In *ACM Multimedia* (2005), pp. 229–238.
- [49] Kumar, A., and Kapur, R. Discrete simulation application - scheduling staff for the emergency room. In *IEEE Winter Simulation Conference* (Washington, D.C., 1989), E. A. MacNair Heidelberger, K. J. Musselman, and P., Eds., pp. 1112–1120.
- [50] Litsios, Socrates. A resource allocation problem. *Operations Research* 13, 6 (1965), 960–988.
- [51] Liu, Chuang, and Foster, Ian. A constraint language approach to grid resource allocation. In *Twelfth IEEE International Symposium on High Performance Distributed Computing (HPDC-12)* (2003).
- [52] Mailler, Roger, Vincent, Regis, Lesser, Victor, Middlekoop, Tim, and Shen, Jiaying. Soft real-time, cooperative negotiation for distributed resource allocation. In *AAAI Fall Symposium on Negotiation Methods for Autonomous Cooperative Systems* (Falmouth, MA, 2001).
- [53] Matthias Kloppmann(IBM), Dieter Koenig(IBM), Frank Leymann(IBM) Gerhard Pfau(IBM) Alan Rickayzen(SAP) Claus von Riegen(SAP) Patrick Schmidt(SAP) Ivana Trickovic(SAP). Ws-bpel extension for people. Tech. rep., IBM, August 25 2005.
- [54] McCaig, Linda F., and Nawar, Eric W. National hospital ambulatory medical care survey: 2004 emergency department summary. Tech. rep., Centers for Disease Control and Prevention, National Center for Health Statistics, June 23, 2006 2006.
- [55] McCall, Eric K., Clarke, Lori A., and Osterweil, Leon J. An adaptable generation approach to agenda management. In *ICSE '98: Proceedings of the 20th international conference on Software engineering* (Washington, DC, USA, 1998), IEEE Computer Society, pp. 282–291.
- [56] McGuinness, Deborah L., and Harmelen, Frank van. Owl web ontology language. <http://www.w3.org/TR/owl-features/>, 2004.
- [57] McGuire, F. Using simulation to reduce length of stay in emergency departments. In *IEEE Winter Simulation Conference* (Orlando, FL, 1994), J.D. Tew Seila, S. Manivannan, D.A. Sadowski, and A.F., Eds., pp. 861–867.
- [58] Medeiros, D. J., Swenson, Eric, and DeFlicht, Christopher. Improving patient flow in a hospital emergency department. In *WSC '08: Proceedings of the 40th Conference on Winter Simulation* (2008), Winter Simulation Conference, pp. 1526–1531.
- [59] Mellor, P. A review of job shop scheduling. *Operations Research* 17, 2 (1966), 161–171.

- [60] Monch, Lars, Stehli, Marcel, and Schulz, Roland. An agent-based architecture for solving dynamic resource allocation problems in manufacturing. In *14th European Simulation Symposium* (Dresden, Germany, 2002), A. Verbraeck Krug and W., Eds., SCS Europe BVBA Technical University of Ilmenau.
- [61] Panwalkar, S. S., and Iskander, Wafil. A survey of scheduling rules. *Operations Research* 25, 1 (1977), 45–61.
- [62] Pinson, L. J., and Wiener, R. S. *Objective-C*. Addison-Wesley, 1991.
- [63] Plesums, Charles. Introduction to workflow. Tech. rep., Computer Sciences Corporation, Financial Services Group, 2001.
- [64] Raman, Rajesh, Livny, Miron, and Solomon, Marvin H. Matchmaking: Distributed resource management for high throughput computing, 1998.
- [65] Raman, Rajesh, Livny, Miron, and Solomon, Marving. Policy driven heterogeneous resource co-allocation with gangmatching. In *International Symposium on High Performance Distributed Computing (HPDC '03)* (2003), pp. 80–89.
- [66] Rossetti, Manuel D., Trzcinski, Gregory F., and Syverud, Scott A. Emergency department simulation and determination of optimal attending physician staffing schedules. In *1999 Winter Simulation Conference* (1999), pp. 1532–1540.
- [67] Russell, Nick, Aalst, Wil van der, Hofstede, Arther, and Edmond, David. Workflow resource patterns. In *17th International Conference on Advanced Information Systems Engineering (CAISE '05)* (Portugal, 2005), Pastor J. eCunha and Falcao, Eds., vol. 3520 of *Lecture Notes in Computer Science*, Springer Verlag, pp. 216–232.
- [68] Samaha, Simon, Armel, Wendy S., and Starks, Darrell W. The use of simulation to reduce the length of stay in an emergency department. In *WSC '03: Proceedings of the 35th conference on Winter simulation* (New Orleans, Louisiana, 2003), Winter Simulation Conference, pp. 1907–1911.
- [69] Saunders, Charles E., Makens Paul K., and Leblanc, Larry J. Modeling emergency department operations using advanced computer simulation systems, February 1989.
- [70] Schreiber, A. Th., Wielinga, B., and Breuker, J. *KADS. A Principled Approach to Knowledge-Based System Development*, vol. 11. Academic Press, London, 1993.
- [71] Schreiber, A. Th., Wielinga, B., Hoog, R. de, Akkermans, H., and Velde, W. van de. Coomonkads: A comprehensive methodology for kbs development. In *IEEE Expert* (1994), pp. 28–37.
- [72] Schull, M. J., Kiss, A., and Szalai, J. P. The effect of low-complexity patients on emergency department waiting times, 2009.

- [73] Shalit, Andrew. *The Dylan Reference Manual*. Apple Computer, Inc., 1998.
- [74] Shenoy, Prashant J., and Vin, Harrick M. Cello: a disk scheduling framework for next generation operating systems, 1998. <http://doi.acm.org/10.1145/277851.277871>.
- [75] Simidchieva, Borislava, Osterweil, Leon J., and Wise, Alexander. Structural considerations in defining executable process models, 2009.
- [76] Smith, Stephen. Is scheduling a solved problem? In *The Next Ten Years of Scheduling Research* (San Francisco, California, USA, 2003), Peter Cowling Kendall and Graham, Eds., pp. 116–120.
- [77] Storrow, A. B., Zhou, C., and Gaddis, G. Decreasing lab turnaround time improves emergency department throughput and decreases emergency medical services diversion: A simulation model, 2008.
- [78] Takakuwa, Soemon, and Shiozaki, Hiroko. Functional analysis for operating emergency department of a general hospital. In *WSC '04: Proceedings of the 36th conference on Winter simulation* (2004), Winter Simulation Conference, pp. 2003–2011.
- [79] Ungar, David, and Smith, Randall B. Self: The power of simplicity. *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '87)* 22, 12 (1987), 227–241.
- [80] Uргаonkar, Bhuvan, Shenoy, Prashant J., and Roscoe, Timothy. Resource overbooking and application profiling in shared hosting platforms. In *Operating System Design and Implementation (OSDI)* (2002), p. 44.
- [81] Vasko, Martin, and Dustdar, Schahram. A view based analysis of workflow modeling languages. In *PDP '06: Proceedings of the 14th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP'06)* (Washington, DC, USA, 2006), IEEE Computer Society, pp. 293–300.
- [82] Wiinamaki, Allan, and Dronzek, Rainer. Emergency departments i: using simulation in the architectural concept phase of an emergency department design. In *WSC '03: Proceedings of the 35th conference on Winter simulation* (2003), Winter Simulation Conference, pp. 1912–1916.
- [83] Wise, Alexandar. Jsim agent behavior specification. <http://laser.cs.umass.edu/documentation/jsim/language.html>, 2008.
- [84] Wise, Alexander. Little-jil 1.5 language report. Technical Report 51, University of Massachusetts Amherst, October 2006.
- [85] Zhang, Weixiong. Modeling and solving a resource allocation problem with soft constraint techniques. Tech. Rep. WUSEAS-2002-13, Washington University in St. Louis, 2002.