DOKUZ EYLÜL UNIVERSITY
GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES

FORMAL METHODS AND PROGRAMMING TOOLS FOR MODELING ANT COLONIES

by

Emine EKİN

February, 2006
İZMİR

FORMAL METHODS AND PROGRAMMING TOOLS FOR MODELING ANT COLONIES

A Thesis Submitted to the
Graduate School of Natural and Applied Sciences of Dokuz Eylül University
In Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy in Computer Engineering

by
Emine EKİN

February, 2006
İZMİR

Ph.D. THESIS EXAMINATION RESULT FORM

We have read the thesis entitled **“FORMAL METHODS AND PROGRAMMING TOOLS FOR MODELING ANT COLONIES ”** completed by **Emine EKİN** under supervision of **Prof. Dr. Tatyana YAKHNO** and we certify that in our opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Doctor of Philosophy.

Prof. Dr. Tatyana YAKHNO

Supervisor

Prof. Dr. İrem ÖZKARAHAN	Prof.Dr. Cüneyt GÜZELİŞ
Committee Member	Committee Member
Prof. Dr. Fazlı CAN	Prof. Dr. Alp KUT
Jury Member	Jury Member

Prof.Dr. Cahit HELVACI

Director

Graduate School of Natural and Applied Sciences

ACKNOWLEDGEMENTS

I appreciate Prof. Dr. Tatyana YAKHNO, for her supervision, support, contribution, care, and patience despite all my obstinacy. She led me to understand who I want to be.

I would like to thank to committee members Prof. Dr. İrem ÖZKARAHAN, and Prof. Dr. Cüneyt GÜZELİŞ, for their valuable comments, suggestions during the whole work.

I also want to thank to Prof. Dr. Fazlı CAN, who is one of the final committee members, for his comments on the document, encouraging me for the further studies. My thanks also go to Prof. Dr. Alp KUT, who is final committee member as well, for his remarks on the document.

There is a special friend, who listened me, sometimes cooked for me, encouraged, convinced, and forced me in achieving the completion, so, I would like to express my heartfelt thanks to Berna. Thanks to Çağdaş as well, for his contribution particularly on early development phases, and for continuing moral support. I want to also thank to all other friends, for their patience on waiting the completion of this study; to the students for their interest on ants.

I extend my thanks to all members of Department of Computer Engineering for their understandings during the whole work.

Finally, I would like to express my special gratitude to my brother Yalçın who always supports me for everything I need, and also my parents, Beşri and Eşe, just because being my parents, taking care of me from hundreds of kilometers away.

Emine EKİN

FORMAL METHODS AND PROGRAMMING TOOLS FOR MODELING ANT COLONIES

ABSTRACT

Nature inspired algorithms have growing up interest in the area of optimization, and the class of ant colony optimization algorithms is one of the recently developed instances of such algorithms. Ant Colony Optimization, ACO, algorithms rely on the basic behavior of ants that is known as foraging behavior, which helps the ant colonies to find the shortest path among a number of possible choices. This is achieved by laying down a chemical substance, called pheromone, on the ground while moving; and preference of the paths with high pheromone level by the successor ants. This type of behavior is also called social behavior in more abstract level, and covers many biological phenomena.

There are two directions in dealing with the social behavior observed in ant colonies; one is transferring the idea to solve optimization problems, leading to the ant colony algorithms, and the other one is formal modeling of the behavior followed with a proper verification schema for better understanding of the relationship between the local interactions of individuals in colonies, and the global dynamical behavior of the colony. Through formal modeling, and a proper verification approach not only social behavior, but various aspects of ant behavior can be investigated.

In this thesis, we have followed both directions. There are some applications developed employing ACO algorithms for solving a real world problem, and a problem from operations research area. In addition, the application developed for Traveling Salesman Problem serves for better understanding of the algorithm. However, much of the efforts have been spent for formal modeling, verification, and developing an automated modeling tool.

Among a variety of formal modeling languages, Weighted Synchronized Calculus of Communicating Systems, WSCCS, has been chosen which is a probabilistic state based transition process algebra. However, modeling itself brings no insight unless it is combined with a verification schema. Verification aims to confirm the correctness of the abstract model against its specification, and also to bring front the properties of colony being studied via asking some questions to the model. Model checking is a technique of verification, and concerns to verify the model for a given property.

In order to verify the correctness of the model, model checking approach has been employed. Since model checking can be performed via temporal logics, probabilistic Computation Tree Logic is another issue dealt with which is then extended to be able to cover the notion of action.

Combining the model checking and formal modeling by WSCCS can be accomplished through transforming the model into a discrete state space with corresponding transitions. Therefore, Labeled Kripke Transition Systems (LKTS) is another formalism introduced, and extended to wrap the probability and action in its state transitions.

The main achievements addressed in the thesis are: the ACO applications developed to solve optimization problems, an investigation of WSCCS for modeling ant colonies, extending the CTL and LKTS such that both systems allow representing probabilistic action occurrences which is the most important property of WSCCS, designing a model checking schema that permits to query the model for probabilistic action occurrences, and implementing a tool in order to automate the whole process.

Keywords: Ant Colony Optimization (ACO), social behavior, Weighted Synchronized Calculus of Communicating Systems (WSCCS), model checking, probabilistic computation tree logic (PCTL), labeled Kripke transition systems (LKTS).

KARINCA KOLONİLERİNİN MODELLENMESİ İÇİN BİÇİMSEL YÖNTEMLER VE PROGRAMLAMA ARAÇLARI

ÖZ

Eniyileştime problemlerinin çözümünde doğadan esinlenerek geliştirilen algoritmalar giderek artan bir yere sahiptir. Karınca kolonisi eniyileştirme algoritmaları da doğa kaynaklı algoritmaların yakın dönemde geliştirilen örneklerinden biridir. Karınca kolonisi algoritmaları, karıncaların yiyecek arama sürecindeki olası bir çok yol arasından en kısa olanı bulabilme yeteneği baz alınarak geliştirilmiştir. Bu yetenek hareket eden karıncaların yeryüzüne kimyasal bir madde bırakmaları, ve takipçilerinin de bu maddeyi algılayarak daha yoğun madde olan yolları tercih etmeleri olarak açıklanabilir. Birçok biyolojik sistemde de görülen bu davranış sosyal davranış olarak isimlendirilmektedir.

Karınca kolonilerinde görülen sosyal davranış iki ayrı inceleme alanı ile karşımıza çıkar. Birincisi, bu davranışın eniyiliştirme problemlerinin çözümünde kullanılabilecek algoritmalar tarafından taklit edilmesi, diğeri ise koloni içindeki bireylerin davranışları ile bunun koloni üzerindeki etkilerinin araştırılmasına ve daha iyi anlaşılmasına yönelik olarak geliştirilen biçimsel modelleme yöntemleri. Biçimsel modellemenin avantajı uygun bir doğrulama yöntemi ile birleştirildiğinde yalnızca sosyal davranışın değil, karınca kolonilerinin birçok özelliklerinin araştırılabilmesidir.

Bu tezde, her iki alanda da çalışmalar yapılmıştır. Karınca kolonisi en iyileştirme algoritmaları kullanılarak gerçek hayattan, ve yöneylem araştırması alanından alınan problemler çözülmüş benzer eniyiliştirme algoritmaları ile karşılaştırmaları verilmiştir. Ayrıca gezgin satıcı problemi için geliştirilen uygulama ile algoritmanın temel özelliklerinin, parametrelerin birbirlerilerine olan etkilerinin anlaşılması sağlanmıştır. Ancak, asıl içerik biçimsel modelleme, doğrulama, ve ikisini birleştiren bir modelleme aracının geliştirilmesi üzerine yoğunlaşmıştır.

Birçok biçimsel modelleme dili arasından olasılıksal olarak durumlar arası geçişe olanak sağlayan bir işlem cebri olan Bağlantılı Çalışan Sistemler için Ağırlıklı

Eşlemeli Hesap "Weighted Synchronized Calculus of Communicating Systems" (WSCCS) seçilmiştir. Ancak, modelleme, doğrulama ile birleştirilmediği sürece incelemesi yapılan sistem hakkında herhangi bir yargıda bulunmamıza olanak sağlamaz. Doğrulama işlemi, oluşturulan soyut modelin sistem tanımı ile karşılaştırılarak teyit edilmesidir. Ayrıca, model üzerinde yapılabilen bazı sorgulamalar sayesinde sistemin özellikleri de incelenebilir. Model denetimi bir doğrulama yöntemi olup verilen bir özelliği sistemin sağlayıp sağlamadığının araştırılmasıdır.

Doğrulama yöntemi olarak seçilen model denetimi yaklaşımı ancak zamana bağlı mantık "temporal logics" ile yapılabildiği için, olasılıksal hesap ağacı mantığı "probabilistic Computational Tree Logic" (pCTL), bu tezde incelenen, ve karınca kolonilerinin toplu davranışlarını da kapsayabilmesi için genişletilen diğer bir biçimsel yöntemdir.

Son olarak WSCCS modelleme ile model doğrulama işleminin birleştirilmesi için geliştirilen modellerin ilgili geçişlerle birlikte ayrık durum uzayına dönüştürülmesi zorunludur. "Labeled Kripke Transition Systems" ayrık durum uzayı biçimselliği olarak tanıtılmış, ancak durum geçişlerinde olasılık ve eylemlerin temsil edilebilmesi için tanımı genişletilmiştir.

Bu tezde elde edilen başlıca kazanımlar; karınca kolonisi algoritmalarının eniyileştirme problemlerinin çözümünde kullanılması için geliştirilen uygulamalar; karınca kolonilerinin modellemesinde kullanılmak üzere WSCCS dilinin detaylı incelenmesi; CTL zamana bağlı mantığı ve LKTS ayrık durum uzayı biçimselliklerinin WSCCS dilinin temel özelliği olan olasılığa bağlı faaliyetleri desteklemek üzere genişletilmesi; modeller üzerinde yine olasılığa ve eyleme bağlı sorgulamalar yapabilmek üzere bir model denetleme şemasının tasarlanması; ve tüm bu işlemleri otomatik olarak yapabilmek için bir yazılım aracının geliştirilmesi olarak sayılabilir.

Anahtar Kelimeler: Karınca Kolonisi Optimizasyonu, sosyal davranış, Bağlantılı Çalışan Sistemler için Ağırlıklı Eşlemeli Hesap, model denetleme, olasılıksal hesaplama ağacı mantığı, etiketli Kripke geçiş sistemleri.

CONTENTS **Page**

Page

Page

Page

CHAPTER ONE

INTRODUCTION

1.1 Introduction

How do the actions and interactions of individuals lead to dynamics of the population which they make up? The researchers working in several areas are interested in answering this question very often to gain benefit. For biologists, the answer is important because they attempt to understand the rules of living organisms. However, from a computer scientist's point of view, understanding such a relationship has introduced a new family of algorithms known as nature inspired algorithms. Also, the field of artificial life concerns to deal with such systems, and build up computer simulations of living organisms.

The idea studied in the thesis has been borrowed from the behavior of biological systems so called social insects, particularly ant colonies. The behavior of an ant colony is often studied as if it were a single entity or "super organism". This super organism may perform a variety of complex tasks as nest building, food foraging, or reproduction. The reason behind calling ants as social insects is that, while any ant cannot perform any of these complex tasks, when they are gathered as colony they can carry out any task. The assumptions took us figuring out that, either each ant exhibit a remarkable intelligence or that the colony has a collective intelligence above and beyond that the ants composing the colony.

The idea of collective intelligence, which is also called social behavior, has been directed the computer scientists in two ways: (i) Copying the behavior in a proper form for solving some certain types of problems, leading to the Ant Colony Optimization Algorithms, (ii) Modeling the behavior for a better understanding of the relationship between individual interactions and the global behavior of colony.

The organization of this chapter is as follows: The behavior of real ant colonies which constitutes the basics of the thesis is introduced first. Explaining the necessity of modeling, particularly modeling ant colonies, and a brief definition of formal modeling, the languages involved in modeling and realizing colonies have been presented. A number of computer tools for modeling languages have also been mentioned.

1.2 Real Ants' Behaviors

The ants in nature are known to be blind; therefore they are not able to perform any task individually. However, they exhibit some certain type of behavior which allows them to survive as a colony. This type of behavior is known as *social behavior*, and is observed on social insects, i.e., insects living in colonies. Social behavior is nothing but the behavior where the survival of the colony is more important than survival of any individual. That is, ants are social insects, and their behavior is directed towards to the survival of the colony rather than that of any individual belonging to the colony.

The fact that ants are social insects, and in turn show social behavior can be best observed while they are looking for food, i.e., the foraging behavior, particularly how they can form the shortest path from their nest to the food source. Observations figured out that the basis of this ability is a kind of chemical substance called pheromone (Dorigo & Gambardella, 1997b). Basically, an ant moving around or looking for food, or transporting food to the nest, deposits some pheromone trail on the path it follows. If an ant perceives pheromone trail while moving around, it is tempted to follow that previously followed path. While moving along that path, it leaves a pheromone trail increasing the existing one. The emerging behavior is a form of *autocatalytic* or *positive feedback behavior* (Dorigo, Maniezzo & Colorni, 1991; Ekin & Yakhno, 2001; Yakhno & Ekin, 2002). The probability of choosing a path is affected by the pheromone trail laid down on the path. Thus, the more ants follow a path, the more attractive that path becomes for the rest of ants. Furthermore, since the probability of an ant choosing a path increases with the number of ants that have already chosen that path, at the end, almost all ants will choose to follow the shortest path, even if each ant's decision always remains probabilistic (Bonabeau, Dorigo, & Theraulaz, 1999).

1.2.1 Bridge Experiment to Observe Pheromone Following

To observe pheromone trail laying and following behavior of some ant species has been investigated in controlled experiments known as double bridge experiments by Bonabeau, Dorigo, & Theraulaz, 1999; Deneubourg, Aron, Goss, & Pasteels, 1990; and Goss, Aron, Deneubourg, & Pasteels, 1989. A similar experiment has been taken from Dorigo & Gambardella, 1997b. The main idea in the experiment is that, if an obstacle appears on regular path of an ant colony, Figure 1.1B, in some amount of time they construct another path around the shorter side of the obstacle, Figure 1.1D.

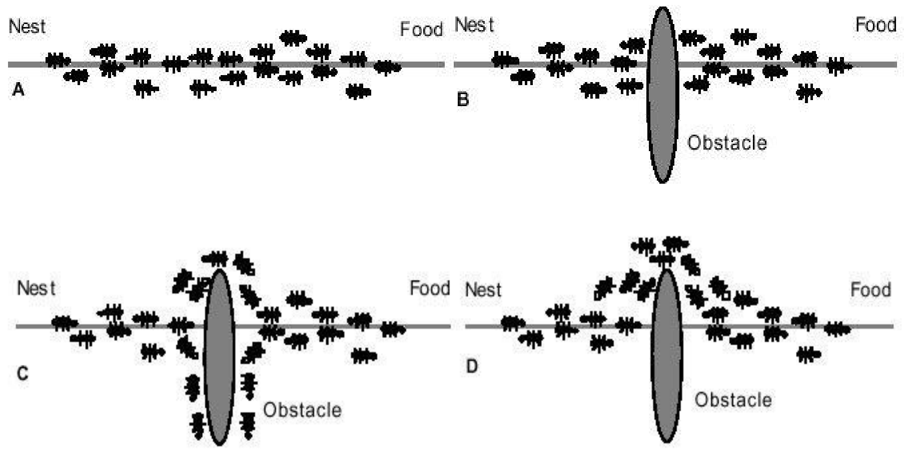


Figure 1.1 Obstacle experiment from Dorigo & Gambardella, 1997b.

In Figure 1.1A, ants are moving on a straight line on both directions and following pheromone trail, probably carrying food from food source to nest, or going to the food source. In Figure1.1B, an obstacle has been put that breaks the straight line, and interrupts the pheromone. Then, the ants those are just in front of the obstacle cannot continue following pheromone trail, thus they decide either turning right or left randomly Figure1.1C. No matter whether an ant turns left or right, it deposits pheromone on the path it follows. Clearly, the ants those selecting the shorter side arrives the food/nest depending on their directions, and when they attempt to turn back there is a high probability to select the path with pheromone. As a result, the ants those selecting the shorter path will rapidly reconstitute the pheromone trail as compared to the ants selecting the longer path. Hence, the shorter path will be preferred by more ants, and will deposit more pheromone in turn. Due to this positive feedback behavior, in some amount of time, all ants prefer the shorter path, Fig1.1D.

1.3 Formal Modeling of Systems

1.3.1 Why Modeling is Required?

Advantage of modeling lies in easiness and safeness of understanding and manipulating a model before realizing the system in question. In other words, models are created as an aid for predicting and understanding any phenomena.

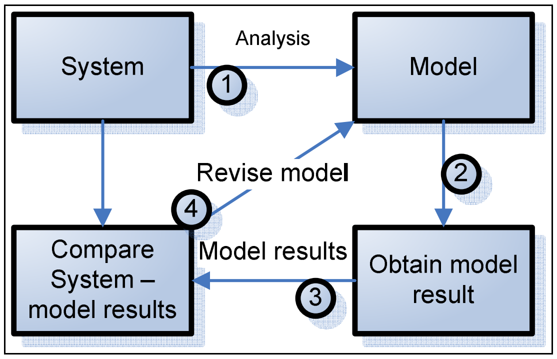


Figure 1.1 Systems modeling.

Any system, which has components working in parallel, communicating with each other and with the environment is considered as a *distributed computing system*. Biological systems, banking systems, air flight control systems, etc. are all distributed computing systems. It is known that concurrent systems carry a much higher risk for unexpected and unintended behavior. In order to assure that such a designed system is functioning properly we need to be able to verify and analyze it against specifications of its intended behavior (Figure 1.1). Since a formal model systematically describes the structure and behavior of systems in an abstract manner, it helps system analysis. By manipulating the model, it is hoped that new knowledge about the modeled system can be obtained without the danger, cost, or inconvenience of manipulating the real system itself. However, modeling is not only required in system analysis and design field, but in study all aspects of systems, especially biological systems.

1.3.2 The Need for Modeling Ant Colonies

We have concentrated on social insect colonies, particularly Ant System; and attempted to formalize ant system behavior.

The behavior of real ants introduced above has been extracted from a series of experiments with real ants on laboratories. However, such experiments are sometimes unable to be performed, and difficult to realize in any other environment. Formal modeling, then, plays an important role by creating abstract models of colonies to both observe the behavior of colonies and answering some philosophical questions regarding the behavior including the keywords “why”, “when”, and “how”. Combining the abstract model with analysis completes the treatment.

Another reason for modeling ant systems is the common behaviors of ant systems with some other insects. These behaviors are, *reproduction*, *adapting to the environment*, *performing the actions independent from other agents*, and *social behavior*. Social behavior, is nothing but an emergent property, and states that, the system itself performs more than the sum of its components.

1.3.3 How Modeling is Performed?

Most modeling uses mathematics. The important features of many physical phenomena can be described numerically and the relations between these features described by equations or inequalities. Particularly in the natural sciences and engineering, properties such as mass, position, momentum, acceleration, and forces are describable by mathematical equations. To successfully utilize the modeling approach, however, requires knowledge of both the modeled phenomena and the properties of the modeling technique. Thus, mathematics has been developed as a science in part because of its usefulness in modeling the phenomena of other sciences. For example, the differential calculus was developed in direct response to the need for a means of modeling continuously changing properties, such as position, velocity, and acceleration in physics (Peterson, 1981).

Deciding the modeling language to be used strongly depends on the system that is going to be modeled.

Ant System is considered as a multi agent system and is a distributed computing system. The term “behavior of a system” corresponds to the “process” in modeling context, and the “behavior of the components” corresponds to “action”. And if the action is a communication with another component, it is going to be called “interaction”.

A model of a Multi Agent System, particularly Ant System should be able to represent the following properties:

- The actions of agents, and the consequences of the environment against these actions,
- The functioning of an agent, with regard to both its observable behavior and its internal changes,
- The interaction of agents with each other, and, in particular different types of communication,
- The evolution of the system itself. This property is extremely important in modeling Ant Systems since they exhibit social behavior as a colony.

The language for modeling distributed computing systems should allow formal specification of individual components making up the system, and provide means for proving properties of component interactions. There are many traditional and recently developed modeling techniques satisfying these requirements. Also, the language should allow formalizing the aforementioned common properties of social insects, e.g. reproduction, adapting to the environment, being capable of performing actions independently.

1.4 Classification of Modeling Languages

In order to build up a formal model of any multi agent system various formalisms are required. All formalisms together must satisfy the requirements of the modeling in terms of aforementioned functional properties, behavioral properties, and structural properties. Ferber, 1999 divides the languages involved in modeling multi agent systems into five classes.

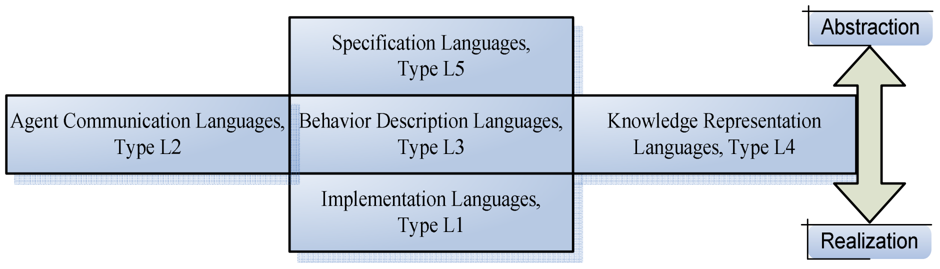

Specification Languages, Type L5
Agent Communication Languages, Type L2
Behavior Description Languages, Type L3
Knowledge Representation Languages, Type L4
Implementation Languages, Type L1
Abstraction
Realization

Figure 1.2 The languages involved in realization of any Multi Agent System.

Figure 1.2 shows various languages those are used to design and develop a multi agent system modeling tools. On the most abstract level, L5 class languages appear where L1 class languages are usually the traditional programming languages as C++, Java, or Smalltalk.

Type L5: Specification Languages: Languages belonging to this class were developed from mathematics, thus have a meta perspective over the modeling task. L5 class languages are used to define what is intended by multi agent system by the notions of action and interaction.

Type L4: Knowledge Representation Languages: This class essentially contains AI- based logic languages which have syntax and semantics to make inference. The language that is an element of L4 class is used to describe the resources of an agent, the information an agent store; which are later used to make reasoning to decide the actions to be performed. Rule- based and blackboard-based languages are examples of this class, where Prolog (Clocksin & Mellish, 2003) and CLIPS, which is developed in 1985 for NASA are well-known instances of rule-based languages. Languages for the structured representation of the knowledge such as semantic nets are also considered as L4-type languages.

Type L3: Behavior Description Languages: Any language in L3 describes what is happening in multi agent system in an abstract manner. The examples of this class are mainly production rule-based, those are arising from automata or from distributed systems. Petri Nets (Petri, 1962) is a well-known example of L3 class language.

Since one must be sure that any multi agent system being modeled must be complete and consistent in its component's actions, and those actions are defined by an instance of L3 class, this class is probably the most important layer in designing a multi agent system.

Type L2: Agent Communication Languages: The languages in this class provide a basis to define the interactions between the agents by means of information exchange. Involving agent communication languages, the system allows its heterogeneous agents to coordinate their actions, and cooperate for a common goal. Knowledge Query and Manipulation Language, KQML is a well-known L2 class language (Finin, Weber, Wiederhold, Genesereth, Genesereth, McKay, McGuire, Pelavin, Shapiro & Beck, 1993).

Type L1: Implementation Languages: They are used to programming a multi agent system, and expected to be able to cover all the computing structures of both the agents and the environment, all the behavioral properties of the multi agent system including inter-agent and inter-environment actions, the activities of transmitting and receiving messages. The languages most frequently used as implementation language are Lisp, C/C++, Java, Prolog, or Smalltalk.

A large number of formal modeling techniques exist in designing multi-agent systems; however, two classes can be distinguished: algebraic models, which tend to describe agents in mathematical notion, and which are L5 type languages; and operative models, which use structures which are a priori more related to computing. These are L3 type languages. The former are fundamental, since they determine all further developments in multi agent system by defining the basics.

The previously listed properties of Ant System are refined below so as to decide the language class that is used to model the property:

- L3 and L5 type languages are involved in modeling the actions of agents, and the consequences of the environment against these actions,
- Modeling the functioning of an agent, with regard to both its observable behavior and its internal changes makes use of L3 and perhaps L5 type languages,
- The interaction of agents with each other, and, in particular different types of communication involves L2 and L3 type languages,
- The evolution of the system itself. This property is extremely important in modeling Ant Systems since they exhibit social behavior as a colony, and L5 type languages are used to model the evolution of the whole system.

All these languages are obviously related to each other, although need not to be similar.

1.5 Existing Modeling Languages & Available Tools

The languages which are capable of modeling distributed systems with mentioned properties are Cellular Automata, Petri Nets, and Process Algebra.

1.5.1 Cellular Automata

A *cellular automaton* is a discrete model studied in computability theory, mathematics, and theoretical biology. It consists of an infinite, regular grid of *cells*, each in one of a finite number of *states*. The grid can be in any finite number of dimensions. Time is also discrete, and the state of a cell at time *t* is a function of the state of a finite number of cells called the *neighborhood* at time *t-1*. These neighbors are a selection of cells relative to some specified, and do not change (Though the cell itself may be in its neighborhood, it is not usually considered a neighbor). Every cell has the same rule for updating, based on the values in this neighborhood. Each time the rules are applied to the whole grid, a new generation is produced. Applying these rules as many times as desired, the evolution of a population is observed. (Wolfram, 1984; Toffoli & Margolus, 1987).

Cellular automata (CA) were originally conceived by Ulam & von Neumann in the 1940s to provide a formal framework for investigating the behavior of complex, extended systems (von Neumann, 1966). Over the years CAs have been applied to the study of general phenomenological aspects of the world, including communication, computation, construction, growth, reproduction, competition, and evolution (see, e.g., Burks, 1970; Smith, 1969; Toffoli & Margolus, 1987; Perrier, Sipper & Zahnd, 1996). One of the most well-known CA rules, the "game of life" was conceived by Conway in the late 1960s (Gardner, 1970; Gardner, 1971) and was shown by him to be computation universal (Berlekamp et al., 1982).

The systematic study of CAs was pioneered by Wolfram and studied extensively by him (Wolfram, 1983; Wolfram, 1984a; Wolfram, 1984b). He investigated CAs and their relationships to dynamical systems, identifying the qualitative subclasses of CA behavior, with analogs in the field of dynamical systems.

In summary, Cellular Automata provide a bottom up modeling in discrete state space. However, cellular automata are best suited to understanding spatial dynamics, and often confound with the interactions between individuals. Thus, cellular automata do not meet the requirements of modeling the following properties of Ant System: (i) actions of an agent independent from other agents (ii) reproduction (iii) different types of communication between different types of agents.

1.5.2 Cellular Automata Tools

It has usage in Public Key Cryptography (Wolfram, 1985). Almost all tools for CA are Java™ based, and therefore available to public. The tools differ in the neighborhood structures they support, the population size, and the transition function.

Conway's Game of Life: One of the most popular CA applications. The Game of Life is not a typical computer game. It is a 'cellular automaton', and was invented by Cambridge mathematician John Conway. It consists of a collection of cells which, based on a few mathematical rules, can live, die or multiply. Depending on the initial conditions, the cells form various patterns throughout the course of the game.

CelLab for Windows: A rare Windows® application, allowing the users to enter their own rules, and select one of predefined neighborhood structures. The experiment can be saved to execute later again.

EvoCellLab, *Modern Cellular Automata* are other free Java™ softwares for evolving and designing cellular automata.

1.5.3 Petri Nets

Petri Nets may seem to be a powerful alternative in modeling distributed systems, since they are specifically designed to model concurrent systems with interacting components. Petri Nets have been developed from the early work of Carl Adam Petri in his doctoral dissertation (Petri, 1962), in which the basis of theory of communication between asynchronous components was formulated. He was primarily focused on describing causal relationships between events. After the work of Petri C.A., the Computation Structures Group at Massachusetts Institute of Technology (M.I.T.), USA, made enormous effort on Petri Nets. The use and study of Petri nets has spread widely in the late 70's. There are several extension proposed to make Petri Nets more powerful in designing systems.

Roughly speaking, a Petri Net has four components: a set of places, *P;* a set of transitions which is a disjoint set from places, *T*; an input function, *I,* that maps a transition to a bag of places, known as *input places of the transition;* output function, *O,* which maps a transition to a bag of places, known as *output places of the transition.* It can be seen as a directed multigraph where places and transitions depict the nodes, and the input / output functions draw the incoming and outgoing directed edges of the nodes. Note that, in a multigraph, there can be more than one directed edge from a transition to a place or vice versa.

Executing, or running up a Petri Net means, enabling the transitions to get fired, which are controlled by tokens. Tokens are assigned to and can be thought to reside in the places of a Petri net. The number and position of tokens may change during the execution of a Petri net in a way that a transition fires by removing tokens from its input places and creating new tokens which are distributed to its output places. The key point is that in order to fire a transition, or, enable a transition, each of its input places has at least as many tokens in it as arcs from the place to the transition. Multiple tokens are needed for multiple input arcs. After firing the transition all of its enabling tokens are removed from its input places and then deposited into each of its output places one token for each arc from the transition to the place. Multiple tokens are produced for multiple output arcs. Transition firings can continue as long as there exists at least one enabled transition. When there are no enabled transitions, the execution *halts.*

Although Petri Nets are well-studied and widely used formalisms in modeling concurrently communicating systems, there are some difficulties using Petri Nets in modeling Ant Systems: (i) synchronous behavior, (ii) reproduction of the agents, (iii) adapting the environment (iv) social behavior.

1.5.4 Petri Nets Tools

Petri Nets are one of the most studied subjects in the area of computer science, both its formal development and application development sides. Thus there is huge number of Petri Nets tools available.

The tools can be classified by availability (free versus commercial), by the environments they are working on (SunOS, MacOS, MS Windows, Java etc.), by the Petri Net type they are supporting (colored Petri Nets, Timed Petri Nets, high-level Petri Nets, stochastic Petri Nets etc.), by the components they make use of (graphical editor, model checking, animation, saving as bitmap, code generation, etc.).

Petri Net Kernel: There is no restricted Petri Net type; rather, the Petri net type can be specified by the programmer.

Predator: Stochastic Petri Nets can be modeled by Predator, but it has advantage of loading analysis modules dynamically, which means enabling users to write their own analysis modules.

YAWL: YAWL (Yet Another Workflow Language) is a workflow language / workflow management system based on Petri nets and a rigorous analysis of existing workflow management systems and workflow languages. Unlike traditional systems and (high-level) Petri nets it provides direct support for most of the workflow patterns. YAWL supports the control-flow perspective and the data perspective and is based on a web-based infrastructure.

Maria: Works on any platform on which C++ is supported. High-level Petri Nets, Place/Transition Nets, Modular high-level nets and any Labeled state transition systems can be designed by Maria. It is a reachability analyzer and a Linear Time Logic (LTL) based model checker, capable of handling tens of millions of reachable states and enabled transitions. Maria also includes a distributed algorithm for checking safety properties. The algorithm can be executed on multi-processor computers as well as on workstations in TCP/IP networks.

The Model-Checking Kit: It is a collection of programs which allow to model a finite-state system using a variety of modeling languages, and verify it using a variety of checkers, including deadlock-checkers, reachability analyzer, and model-checkers for the temporal logics CTL and LTL. The checkers are taken from different existing tools like SMV. They use state-of-the-art techniques to avoid the state explosion problem, such as symbolic model checking and partial orders. The most interesting feature of the Kit is that independently of the description language chosen by the user all checkers can be applied to the same model. It is not necessary to model the system in different modeling languages.

Petri Net Toolbox: Works with MATLAB® 6.0 or later. It can be used for simulation, analysis, and design of discrete event systems, based on Petri Net models. Five types of Petri Net models are accepted: untimed, transition-timed, place-timed, stochastic, and generalized stochastic nets. The timed nets can be deterministic or stochastic. Places may have finite or infinite capacity. Priorities or probabilities can be assigned to conflicting transitions. A graphical user interface (GUI) allows you to draw, store, and retrieve Petri Net models, as well as to start the procedures for simulation, analysis, and design. All the procedures available in the Petri Net Toolbox are implemented as M-files and are designed in a modular fashion to start from the GUI.

Simulaworks: Runs on Windows® 2000, XP operating system. It is a general purpose simulator, which supports an open set of simulation languages and AI techniques. SimulaWorks can be used to construct various system types like control loops, signal processing, logic and hardware circuits, manufacturing systems, production lines, traffic systems, complicated algebraic calculation, optimization and search problems, state diagrams and even graphs, presentation using graphics, run-able flow charts, etc. Also sometimes hybrid system of those systems can be simulated, like control and signal processing, etc. Many libraries such as Control, Mathematics, GPSS, Petri Nets, Active Flow Chart, Smart Draw, Hardware/Logic are implemented within Simulaworks.

1.5.5 Process Algebra

There is another class of techniques for modeling distributed computing systems, especially biological systems, Process Algebra. It means algebraic/axiomatic approach while talking about behavior. (L5 type language) Historically, process algebras have developed as formal descriptions of complex computer systems, especially those involving communicating, concurrently executing components. The crucial idea in definition of Process Algebra is the algebraic structures of the concurrent processes. It uses a state based approach with labeled transitions, where states and transitions correspond to agents and actions respectively. The finite state space difficulty arisen in Petri Nets is overcome with aggregation of the states, which is a result of algebraic structures of processes in Process Algebra.

There are many examples of Process Algebra. Milner's Calculus of Communicating Systems, CCS, is one of the well known process algebras (Milner, 1989). Hoare's Calculus of Sequential Processes, CSP, is another cornerstone in Process Algebra (Hoare, 1978). Both CCS and CSP have several extensions. Another example process algebra which is more recently developed is Hillston's Performance Evaluation Process Algebra, PEPA (Hillston, 1996). Since we are interested in concurrently communicating systems rather than sequentially operating systems, CCS is more appropriate for modeling ant system behavior.

CCS uses the notions of agents, and actions, and proven to be very successful in modeling distributed computing systems. It is completely nondeterministic in the sense that given a choice between two actions, there is no mechanism, probabilistic, or otherwise, by which to choose between them. It also makes no attempt to synchronize the actions of agents acting in parallel. This makes CCS powerful when determining whether or not a series of actions will always or never occur. But it cannot determine their likelihood or time till occurrence. Synchronous CCS, SCCS, was developed to model systems where agents are synchronized (Milner, 1983). Finally, Weighted SCCS, denoted by WSCCS, adds further notions of priority and probability of actions to SCCS (Tofts, 1994). Neither CCS nor its mentioned variants accounts for time explicitly. Instead, synchronization scheme, which is introduced in SCCS, assumes actions occur at each tick of a global discrete time clock. Stochasticity has been attached into these discrete time algebras, which is used to predict the system behavior. For example the model underlying WSCCS is a discrete time Markov chain, and PEPA is based on continuous Markovian processes.

1.6 Structure of the Thesis

The main goal of the study is to investigate behaviors of real ant colonies, using the idea in solving optimization problems, developing a formalism for modeling and analyzing the behavior and a implementing computer tool to automate analysis. The content of the thesis is constructed according to this outline.

In Chapter 2, the class of Ant Colony Optimization Algorithms, and its state of the art application areas are described, where

In Chapter 3, the applications of traveling salesman problem, 2D map problem, and single machine total tardiness problem have been presented.

Chapter 4 is devoted to informal explanation of the selected formal modeling language, WSCCS, whose formal definition is provided in Appendix A.

Chapter 5 is a literature survey chapter and gives the formal verification definition, several approaches of formal verification. Detailed information on model checking, which is the verification technique employed in analysis of models, and several formalisms involved in model checking approach such as computation tree logic, discrete state space systems are other issues addressed in Chapter 5.

Chapter 6 is the longest chapter since it combines all previous formal notions to make up a complete system for modeling, and verifying ant colonies, and relating those formalisms with software development in order to automate whole task. It also explains how each formalism is enlarged so that it covers basic behavior of ants.

In Chapter 7, some detailed examples of ant colony models have been demonstrated to show various aspects of WSCCS, model checking, and computation tree logic together with the results obtained from WSCCS++.

The thesis completes with the conclusions drawn from particularly Chapter 3 and Chapter 6.

CHAPTER TWO

ANT COLONY OPTIMIZATION ALGORITHMS

2.1 Historical Background

One of the main directions of this thesis is to investigate the stigmergetic model of communication in general, particularly, foraging behavior of ants. Ant Colony Optimization (ACO) algorithms were first introduced by (Dorigo, 1992), in his doctoral dissertation, and are a class of probabilistic techniques for solving combinatorial optimization problems which can be reduced to finding cost efficient paths on graphs.

As the name suggests, ACO algorithms have been inspired by the behavior of real ants. The idea behind is that, although the real ants are blind, they can construct the shortest paths from their nest to the food sources. The ant colonies accomplish this task by using a collective decision making strategy (Colorni, Dorigo & Maniezzo, 1992), which has been successfully applied to several NP-complete (Garey & Johnson, 1979) optimization problems ranging from Traveling Salesman Problem to Telecommunication Routing Problems (Bullnheimer, Hartl & Strauss, 1999; Colorni, Dorigo & Maniezzo, 1992; Colorni, Dorigo, Maniezzo & Trubian, 1994; Costa & Hertz, 1997; Dorigo & Gambardella, 1997a; Gambardella, Taillard & Dorigo, 1999; Schoonderwoerd, Holland, Bruten & Rothkrantz, 1997).

The first ACO algorithm, called Ant System (AS), has been applied on Traveling Salesman Problem, TSP (Dorigo, Maniezzo & Colorni, 1991; Dorigo, Maniezzo & Colorni, 1996; Lin, 1965), which is a well known NP-complete problem. Starting with the Ant System algorithm, several improvements have been proposed and applied on TSP again (Gambardella & Dorigo, 1996; Stützle & Hoos, 1998). Although the promising results have been obtained by AS and its variants, the algorithm was not superior over some other well-known metaheuristics, such as Tabu

Search (Glover & Laguna, 1996; Osman & Kelly, (Eds.), 1996). However, it has shown that ACO algorithms are as good as some other general purpose heuristics, for example evolutionary computation (Fogel, 1993). Further studies on obtaining the ACO algorithms with better computational performances have resulted in successful applications of ACO algorithms in a variety of fields.

In the following sections, the artificial ants constituting the basics ACO algorithms are introduced first. After describing the Traveling Salesman Problem, on which ACO algorithms are defined, the subsequent sections contain definitions of some well-known and successful ACO algorithms, the techniques used to improve the performance of ACO algorithms, and the application areas of ACO algorithms in the literature.

2.2 Artificial Ants

In ACO, the main idea is the indirect communication based on pheromone trails of a colony of ants.

Artificial Ant System is a distributed system in which an artificial ant is defined as a simple computational agent that iteratively constructs solutions for the problem to solve. Partial problem solutions are seen as states and each ant moves from one state to another, corresponding to more complete partial solution (Dorigo, Maniezzo & Colorni, 1996; Ferber, 1999). The pheromone trail they leave is a kind of distributed numeric information that is modified by the ants to reflect their solution quality.

There are three properties transferred from real ants to artificial ants (Dorigo & Gambardella, 1997b):

1. The preference of paths with higher pheromone level.
2. The higher growth on the amount of trail on shorter paths.
3. The trail based communication among ants.

The artificial ants (agents) also have some extra abilities to make them useful for solving combinatorial optimization problems.

1. They live in a discrete time environment; and at each time unit they make a movement corresponding to the problem definition.
2. They are not completely blind; they are given the ability of knowing the problem definition, and the objective function.
3. The artificial ants have some memory to keep some details related with the problem. They are also able to make some simple computations.

Artificial ants choose the action in the search space using a probabilistic action choice rule, which is determined by the objective function of the given problem.

For each artificial agent, the probability of moving one state to another depends on combination of two values (Dorigo, Maniezzo & Colorni, 1996):

1. The attractiveness of the move, as computed by some heuristics indicating *a priori* desirability of that move.
2. The trail level of the move, indicating how proficient it has been in the past to make that particular move; it represents therefore *a posteriori* of the desirability of that move.

```
Randomly select initial positions for ants
For each ant:
   Perform
      Move to next state
      Update ant memory
      Update trail on visited edge(local update-optional)
   Until goal state is reached
End For
Global trail updating
```

Figure 2.1 The general framework for ACO algorithms.

Figure 2.1 depicts the most general ACO algorithm. There are two main steps: solution construction, and trail updating. While examining any ACO algorithm, it is necessary to clarify both parts.

2.3 Traveling Salesman Problem

Traveling Salesman Problem (TSP) is a well-known NP-complete problem and is almost a standard test problem for the metaheuristics, which are heuristic methods for solving a very general class of computational problems by combining user given black-box procedures — usually heuristics themselves — in an efficient way, in the area of combinatorial optimization. Informally TSP can be defined as following: given a set of cities and the costs of traveling from one city to another, finding the cheapest round trip route that visits each city exactly once, and returning the initial city (Papadimitriou & Steiglitz, 1982).

From the formal side, it is a complete graph with $G = (V, E)$ such that V is a finite set of nodes, representing the cities; E is the finite set of edges, fully connecting the nodes in V; a cost function, d_{ij}, giving the distance between the cities i and j, for each $(i, j) \in E$. The problem is to find the minimal cost Hamiltonian circuit of the graph, where Hamiltonian circuit is a closed tour visiting each node exactly once with a cost value calculated by summing up all the distances composing the tour. Note that the distances between two cities need not to be symmetric. If $d_{ij} = d_{ji}$, then the TSP instance is called “symmetric TSP”, otherwise it is called “asymmetric TSP”.

2.4 Different Classes of ACO Algorithms

We have classified ACO algorithms into two according to the solution construction rules.

- Ant System (AS) based algorithms: The instances of this class are Ant Density, Ant Quantity, Ant Cycle algorithms, which are direct variants of the one known as AS (Dorigo, 1992; Dorigo, Maniezzo & Colorni, 1991); Elitist Ant (Dorigo, 1992; Dorigo, Maniezzo, & Colorni, 1996), MAX-

MIN AS (Stützle & Hoos, 1997; Stützle & Hoos, 1998), Rank Based Version of AS (Bullnheimer, Hartl & Strauss, 1999). All members of this class employ the same random transition rule, which explores the edges rather than exploiting the pheromone information. However, the trail updating schemas are different.

- Q-Learning based algorithms: Ant-Q (Gambardella & Dorigo, 1995) and Ant Colony System (ACS) (Dorigo & Gambardella, 1997) are the two examples of Q-learning based algorithms. State transition rule of this class is known as pseudo-random proportional since it depends on a learning parameter, which enables to balance the exploitation of previously accumulated information and the exploration of new edges. The difference between Ant-Q and ACS appears in trail updating rules.

2.4.1 Ant System-AS

Ant System was the first ACO algorithm (Dorigo, 1992; Dorigo, Maniezzo & Colorni, 1991). Its importance relies on being a prototype for the further ACO algorithms with successful applications. When AS was first introduced its application was the TSP. There are three different AS algorithms:

- *ant-density*,
- *ant-quantity*,
- *ant-cycle.*

The algorithms differ in pheromone update mechanisms.

2.4.1.1 Tour Construction

In AS, ants are simple agents, initially positioned on random cities. The priori desirability of the moves, which is a heuristic information depending on the problem, are computed by the Euclidian distances between two cities, where posteriori desirability is the pheromone level on the edge being passed. Consider the following formula which gives the probability of selecting city `j`, which is not visited before, being at city `i` by the ant k at time step t:

$$p_{ij}^{k}(t) = \frac{\left[\tau_{ij}(t)\right]^{\alpha} * \left[\eta_{ij}\right]^{\beta}}{\sum_{l \in N_i^k} \left[\tau_{il}(t)\right]^{\alpha} * \left[\eta_{il}\right]^{\beta}} \quad \text{if } j \in N_i^k \qquad \text{(Eq.2.1)}$$

$\tau_{ij}(t)$: the pheromone level between cities `i` and `j` at time t.

η_{ij}: the heuristic information on moving from `i` to `j`, which is inversely proportional with the distance between `i` and `j`. $\eta_{ij} = 1/d_{ij}$

N_i^k: being on `i`, the set of unvisited cities for the ant k.

α, β: two parameters determining the relative influence of pheromone trail and heuristic information respectively.

Parameters α, β affect the probability of selecting city j as following: Let $\alpha = 0$, then the probability becomes proportional to η_{ij}, meaning that the closest cities are more likely to be selected. In this case, the AS algorithm corresponds to a stochastic greedy algorithm (Cormen, Leiserson & Rivest, 1990). When $\beta = 0$, only the pheromone accumulation is at work, and this leads to a rapid emergence of a *stagnation* behavior (Dorigo, Maniezzo & Colorni, 1996), which is the case all ants following the same path and construct the same solution, and resulting in suboptimal solutions.

The tour construction phase ends when all ants iteratively constructs theirs tours.

2.4.1.2 Trail Update Mechanism

There are two different types of trail updating independent from the ACO algorithm in use. First one is local trail updating. Local trail updating is applied in tour construction phase, after selecting city, and is optional. Second one is global trail updating. Global trail updating is common in all ACO algorithms, although different formulas appear in different algorithms.

Global Trail Updating. By the Eq.2.2, the pheromone levels on edges, those are used in the solutions by some ants, are updated.

$$\tau_{ij}(t+1) = (1-\rho) * \tau_{ij}(t) + \rho * \sum_{k=1}^{m} \Delta\tau_{ij}^{k}(t) \qquad \text{(Eq.2.2)}$$

$\tau_{ij}(t+1)$: the pheromone value on edge (`i,j`) to be used in next iteration, i.e., (t+1);

$\tau_{ij}(t)$: the pheromone value on edge (`i,j`) after recently finished iteration;

ρ: evaporation coefficient. As in biological counterpart, the pheromone level decreases in time, and the parameter $0 \le \rho < 1$ controls this evaporation. In other words, it helps to forget "bad" experience. Assume an edge (`r,s`) is not used in any solutions, in this case, the pheromone deposited on (`r,s`) decreases exponentially;

$\sum_{k=1}^{m} \Delta\tau_{ij}^{k}(t)$: the total change in amount of pheromone level on edge (`i,j`) that is caused by being used in solution of ant k. Calculation of this amount differs by the algorithm used, that is Ant-Cycle, Ant-Density, Ant-Quantity.

- Ant Density: a unit of pheromone quantity is left on the edge (`i,j`) every time an ant uses the edge (`i,j`).

$$\Delta\tau_{ij}^{k}(t) = \begin{cases} 1, \text{ if ant k uses the edge (i,j)} \\ 0 \text{ , otherwise} \end{cases}, \qquad \text{(Eq.2.3)}$$

- Ant Quantity: a quantity that is inversely proportional to distance between `i` and `j` is deposited on the edge (`i,j`), i.e., the shorter the edge is, the more pheromone trail is deposits.

$$\Delta\tau_{ij}^{k}(t) = \begin{cases} \frac{1}{d_{ij}} \text{ , if ant k uses the edge (i,j)} \\ 0\text{, otherwise} \end{cases}, \qquad \text{(Eq.2.4)}$$

- Ant Cycle: a quantity that is proportional to the solution quality is deposited on the edge (i, j), i.e., the shorter the tour is, the more pheromone its edges receive.

$$\Delta\tau_{ij}^{k}(t) = \begin{cases} \frac{1}{L^k}, & \text{if ant k uses edge (i,j)} \\ 0, & \text{otherwise} \end{cases} \qquad \text{(Eq.2.5)}$$

L^k : The length of the tour constructed by ant k.

Local Trail Updating. The update rules defined for Ant-Quantity and Ant-Density algorithms may also be used as local trail updating rules. The only difference is that, if local update is at work, the rules are applied after each state transition.

2.4.2 Other AS-Like Algorithms

After introduction AS and its three variants, several other algorithms based on AS were proposed. The algorithm called *Elitist Ants* was introduced in (Dorigo, 1992; Dorigo, Maniezzo, & Colorni, 1996). This algorithm allows only the ant, which constructs the shortest tour, to update the pheromone levels on its edges.

MAX-MIN Ant System is also a direct improvement over AS (Stützle & Hoos, 1997; Stützle & Hoos, 1998). In MAX-MIN approach, only the ant which constructs the shortest tour updates the pheromone trails belonging to its edges, as in elitist strategy. The difference is on the values of pheromone levels. The pheromone level on each edge is initialized with a value, $\tau_{\max}$, and the changes are restricted to the interval $[\tau_{\min}, \tau_{\max}]$. Setting these explicit limits on trail levels confines the range of possible values of the probability of selecting an edge, and in turn helps to avoid stagnation behavior, which is one if the reasons of poor results of AS algorithms.

Bullnheimer, Hartl & Strauss (1999) proposed another modification of AS, called *Rank-Based version of AS, AS_{Rank}*. In AS_{Rank} algorithm ants are sorted according to their solution quality, and a prefixed number of ants are allowed to update the trail strengths in proportion with their order.

2.4.3 Ant-Q Algorithm

Ant-Q algorithm is a Q-learning based ACO algorithm proposed in (Gambardella & Dorigo, 1995). Q-learning is a reinforcement learning algorithm that works by estimating the values of state-action pairs (Watkins & Dayan, 1992). State transition rule differs from the AS based approaches as shown in Section 2.4.3.1. Trail update mechanism is similar to those of aforementioned algorithms, only the ant constructing the cheapest tour is allowed global trail updating, while local updating is applied by all ants. Although the update formula itself is similar to those of priori algorithms, the pheromone quantity to be added is calculated according to the Q-learning algorithm.

2.4.3.1 Solution Construction

Being on a city `i`, to select the next city to move to, there are two options decided by learning parameter, and a random number: first case favors transitions towards the nodes connected by short edges and with large amount of pheromone, that is exploitation of the good solutions. Second transition rule is same as that of AS based approaches, and helps to explore the probable good solutions, and therefore avoiding early stagnation behavior.

$$v_{ij} = \begin{cases} \arg\max\limits_{l \in N_i^k} \left[(\tau_{ij})^{\alpha} * (\eta_{il})^{\beta} \right] & \mathrm{q} \le \mathrm{q}_0 \\ V & \mathrm{q} > \mathrm{q}_0 \end{cases}, \qquad \text{(Eq.2.6)}$$

where:

arg max stands for the *argument of the maximum*, that is, the value of the given argument for which the value of the given expression attains its maximum value. This is well-defined only if the maximum is reached at a single value;

q is a random number uniformly distributed in [0,1];

q0 is a learning parameter, $0 \le q_0 \le 1$, such that, the higher the q0 is, the smaller the probability of making a random choice;

V is a random variable which is calculated as:

$$p_{ij}^{k} = \begin{cases} \dfrac{\left(\tau_{ij}\right)^{\alpha} * \left(\eta_{ij}\right)^{\beta}}{\sum\limits_{l \in N_i^k} \left(\tau_{il}\right)^{\alpha} * \left(\eta_{il}\right)^{\beta}} & \text{if } j \in N_i^k \\ 0 & \text{otherwise} \end{cases} . \qquad \text{(Eq.2.7)}$$

2.4.3.2 Trail Update Mechanism

$$\tau_{ij}(t+1) = (1-\rho) * \tau_{ij}(t) + \rho * \left(\Delta\tau_{ij}^{best}(t) + \gamma * \max_{z \in N_j} \tau_{jz}(t) \right) \qquad \text{(Eq.2.8)}$$

The formula is composed of a reinforcement term, and of the evaporated evaluation of the next state. In general, the reinforcement term $\Delta\tau$ can be either local, or global, i.e., calculated when all ants finish their tour, and calculated by the Eq.2.9:

$$\Delta\tau_{ij} = \begin{cases} 1/L_{best} & \text{if } (\text{i,j}) \in \text{ shortest tour} \\ 0 & \text{otherwise} \end{cases} \qquad \text{(Eq.2.9)}$$

where L_{best} is the shortest tour length.

2.4.4 Ant Colony System- ACS

ACS is based on Ant-Q algorithms, uses the same state transition and trail updating rules except the amount of pheromone trail to be added in trail updating rule (Eq.2.10):

$$\tau_{ij}(t+1) = (1-\rho) * \tau_{ij}(t) + \rho * \Delta\tau_{ij}^{best}(t) \qquad \text{(Eq.2.10)}$$

Since the computational effort for ACS update rule is less than that of Ant-Q, ACS algorithm is more preferable in applications of ACO algorithms.

2.5 Local Search to Improve the Performance of ACO Algorithms

One of the most usual techniques to improve the performance of ACO algorithms is the use of local search techniques (Bullnheimer, Hartl & Strauss, 1999; Maniezzo & Colorni, 1999). This approach entails employing a local optimization technique to refine the solutions obtained after one or several iterations. In spite of using local search procedures usually improve the efficacy of the ACO algorithm; it increases the number of evaluations at each iteration and therefore the runtime of the learning method, thus loosing efficiency.

Local search algorithms start with a complete solution and attempts to generate a better solution in an appropriately defined *neighborhood* of the initial solution. In its most basic version, known as *iterative improvement*, the algorithm searches the neighborhood for an improving solution. If such a solution is found, it replaces the current solution and local search continues. These steps are repeated until no improving neighborhood solution is found anymore in the neighborhood of the current solution and the algorithm ends in a *local optimum*. When the problem size and in turn the neighborhood of the solution gets larger, this iterative improvement algorithm becomes poor in both solution quality, and computation time.

The choice of appropriate neighborhood structure is crucial for the performance of local search algorithm and has to be decided according to the problem. A neighborhood structure defines the set of solutions those can be reached from the current solution in one step of the local search algorithm. The followings are different neighborhood structures that might be used for different problems producing different effectiveness.

- An example neighborhood for the TSP is k-opt, in which neighbor solutions differ by at most k edges. Figure 2.2 demonstrates an example 2-opt neighborhood, where 2-opt local search algorithm systematically test for the whether the current tour can be improved by replacing two edges.

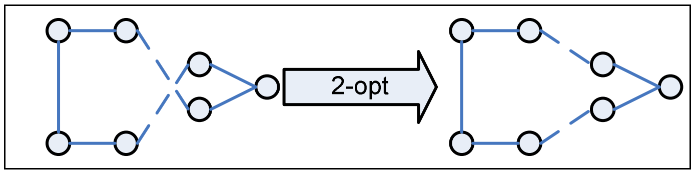


Figure 2.2 Schematic illustration of the 2-opt algorithm.

- Another local search technique involves interchanging the adjacent pairs in any sequence, and yielding the neighborhood of size n-1, where the number of elements in the sequence is n. This schema is known as *adjacent pairwise interchange*, API. The neighborhood generated by this way is relatively small and easy to generate, but it also limits the number of new choices to look at, so there is a tradeoff. If the current sequence were 1, 2, 3, ,…, n, then the API neighborhood of the solution would be exactly as following.

2, 1, 3, 4,…, n-2, n-1, n	(interchange 1 and 2)
1, 3, 2, 4,…, n-2, n-1, n	(interchange 2 and 3)
……….	
1, 2, 3, 4,…, n-1, n-2, n	(interchange n-1 and n-2)
1, 2, 3, 4,…, n-1, n, n-2	(interchange n-2 and n)

- Interchanging not only the adjacent pairs, but all possible pairs causes a different neighborhood structure to be obtained. This method is known as *general pairwise interchange*, GPI. The neighborhood size is $n(n-1)/2$ for an initial sequence of size n. GPI generates a larger neighborhood than API, however, the number of new choices to look at is increased, meaning that the tradeoff exists in GPI as well.

- There are two other local search techniques closely related with each other. In the first one, the jobs in the sequence are shifted circularly one position left. This is repeated as the number of jobs. If an improvement on the solution quality is met, the algorithm halts and returns the new sequence. Otherwise, a different neighbor structure, in which all possible jobs *k* inserted into first position and each *(k-1)* jobs are moved one back position, has been employed. If any improvement occurs, the procedure is terminated.

Application of local search varies by problem and independent from the ACO algorithm in use. The local search might be carried out by the best ant only, or by all ants after each iteration. The global trail updating rule works on the solution obtained with local search.

2.6 Application Areas of ACO Algorithms

The applications of ACO algorithms can be divided into two distinct classes: the static problems, and the dynamic problems. The former one defines the set of problems whose definition is fixed and does not change over time. ACO algorithms were first applied on the problems belonging to this class, such as TSP, quadratic assignment problem, sequential ordering problem, vehicle routing problems, shop scheduling problems, graph coloring problems, etc. Since any static combinatorial optimization problem must be converted to TSP (Sipser, 1996), providing an appropriate heuristic function, the algorithms developed for TSP can be adopted for solving other static problems.

The latter kinds of problems are the problems, whose characteristics are changed while being solved. Any network routing problems fall into this class, i.e., circuit-switched network routing, packet-switched network routing. And the algorithms developed for solving dynamic problems are the most recent improvements on ACO algorithms.

2.6.1 Quadratic Assignment Problem-QAP

QAP concerns assigning a set of `n` facilities to `n` locations so that the cost of assignment, which is a function of the way performing the assignment, is minimized. Some real world applications of QAP are planning the buildings on a land, arrangement of departments in hospitals, minimizing the total wire length in electronic circuits, and ordering the associated data in magnetic tapes. QAP is the first problem attempted to be solved with AS-like approaches, since it is directly mapped onto TSP. (Maniezzo, Colorni & Dorigo, 1994; Gambardella, Taillard & Dorigo, 1999).

Maniezzo, Colorni & Dorigo (1994) has applied the AS algorithm on QAP, AS-QAP, and obtained the results of approximately the same quality with other metaheuritics, such as simulated annealing. More recently, Maniezzo & Colorni (1999) and Gambardella, Taillard & Dorigo (1999) developed two variants of AS-QAP, added a local optimizing feature. The resulting algorithms compared to other metaheuristics were performed better for almost all test cases.

More advanced ACO algorithms were also applied to QAP. In (Stützle & Hoos, 1998), MAX-MIN ant system for QAP was described. The HAS-QAP algorithm defined in (Gambardella, Taillard & Dorigo, 1999) was tested on some real world problems, and known to be the best performing algorithms for the problems of this class.

2.6.2 Sequential Ordering Problem-SOP

Sequential Ordering Problem is defined as, finding minimal weight Hamiltonian path satisfying the precedence constraints among nodes on a weighted graph. Real world problems associated with SOP are single vehicle routing with predefined pick-up and delivery nodes, production planning and transportation in flexible manufacturing systems. SOP corresponds to Asymmetric TSP, and therefore was attacked by the researchers since the beginning of the development of first ACO algorithms.

Gambardella & Dorigo (1997) has proposed a method, Hybrid Ant System for solving SOP, which was an extension of ACS algorithm. The HAS-SOP algorithm has been tested on a huge number of problem sets, and compared with a number of heuristics, and in all cases, HAS-SOP algorithm has been evaluated as the best-performing algorithm in terms of both solution quality and computation time.

2.6.3 Single Machine Total Tardiness Problem-SMTTP

SMTTP is a well known NP-complete problem from Operations Research, and has been attacked by several exact solution algorithms (Della Croce, Tadei, Baracco, & Grosso, 1996; Fisher, 1976; Lawler,1977; Szwarc & Mukhopadhyay, 1996), as well as heuristic algorithms (Fry, Vicens, Macleod & Fernandez,1989; Potts & Van Wassenhove, 1991; Wilkerson & Irwin, 1971).

It is a permutation problem as all other static combinatorial optimization problems, and defined as following: sorting N jobs each of which has processing time p_i, and due date d_i, with subject to minimize the tardiness T, i.e., in case of late delivery, the time between delivering the job and its due date. Other important properties of the problem are: all jobs to be sorted are available at starting time, the machine is available for processing all jobs, once a job is started to be processed, it is completed without interruption, i.e., no preemption, and each job has equal importance.

Eq.2.11 defines the objective function for a set of N jobs, in terms of processing times and due dates. Note that, if a job is finished before its due date, it is not taken into account.

$$T = \sum_{i=1}^{N} \max(0, \sum_{j=1}^{i} p_j - d_i) \qquad \text{(Eq.2.11)}$$

As in QAP, and SOP, it is necessary to provide a heuristic function that is used to guide the ants together with the pheromone information. There are some well known heuristic functions for SMTTP in the literature (Morton & Pentico, 1993). EDD is the most primitive heuristic function, that sorts the jobs according to ascending due dates. η_{ij} stands for the heuristic information where the last scheduled job is i, and j is one of the jobs to be sorted.

$$\text{EDD: } \eta_{ij} = 1/d_j \qquad \text{(Eq.2.12)}$$

Since EDD rule is so poor in performance, it is modified in a way that, once a job is scheduled, all the remaining jobs are resorted in ascending fashion iteratively according to the modified due dates (MDD), see the Eq.2.13.

$$\text{MDD: } \eta_{ij} = \frac{1}{\max(\varphi + p_j, d_j)} \qquad \text{(Eq.2.13)}$$

where φ is the total processing time of all scheduled jobs.

Another alternative incorporating heuristic function is the Shortest Processing Time, (SPT), which sorts the jobs according to the processing times in ascending fashion. SPT generates feasible results in cases where all the jobs are late even in optimal solution.

$$\text{SPT: } \eta_{ij} = 1/p_j \qquad \text{(Eq.2.14)}$$

2.6.3.1 Trail Update

The global trail update rule is same as any ACO algorithm, except the trail intensity to be added.

$$\tau_{ij}(t+1) = (1-\rho) * \tau_{ij}(t) + \rho * \Delta\tau_{ij}(t) \qquad \text{(Eq.2.15)}$$

$$\Delta\tau_{ij}(t) = 1/T^* \qquad \text{(Eq.2.16)}$$

T^* is the minimum total tardiness in current iteration.

Besides global update rule, a local trail update also employed to improve the performance. Selecting a job to be appended to the current partial solution, the rules in Eq.2.17 and Eq.2.18 update corresponding pheromone trail.

$$\tau_{ij}(t+1) = (1-\rho) * \tau_{ij}(t) + \rho * \tau_0 \qquad \text{(Eq.2.17)}$$

$$\tau_0 = \frac{1}{n * T_{EDD}} \quad \text{(Eq.2.18)}$$

T_{EDD} is the total tardiness of the solution generated according to the EDD rule.

2.6.4 Single Machine Total Weighted Tardiness Problem-SMTWTP

The Single Machine Total Weighted Tardiness Problem is the generalized form of SMTTP (Morton & Pentico, 1993). In addition to the processing times and due dates, some weight values are assigned for each job, and the total tardiness is calculated by summing up the time exceeding the due date multiplied by weight value.

Eq.3.19 defines the objective function for SMTWTP. It is same as that of SMTTP except the tardiness for each job is multiplied by its corresponding weight value.

$$T = \sum_{i=1}^{N} \left(w_i * \max(0, \sum_{j=1}^{i} p_j - d_i) \right) \quad \text{(Eq.2.19)}$$

Since SMTWTP is NP-complete, several heuristics have been proposed for its solution, such as Earliest Due Date or Apparent Urgency (Abdul-Razaq, Potts & Van Wassenhove, 1990; Potts & Van Wassenhove, 1991), and metaheuristics like Simulated Annealing (Potts & Van Wassenhove, 1991), Tabu Search, and Genetic Algorithms (Crauwels, Potts & Van Wassenhove, 1998).

Although the heuristics functions given for SMTTP can be used for SMTWTP, since they do not involve weight term, some other heuristics with weight information are preferred while solving SMTWTP with ACO algorithms.

Among many others, four problem specific heuristic functions are examined in this research since they are known to be computationally efficient relative to the rest. The first is Weighted Shortest Processing Time which sorts the jobs according to the w_i / p_i.

WSPT: $\eta_{ij} = \frac{w_j}{p_j}$ (Eq.2.20)

The second rule, called AR, was derived by Alidaee & Ramakrishnan, 1996; k is a look ahead parameter that relates the number of critical jobs, and nearly critical jobs, and $\overline{p}$ is the average processing time of all jobs, and t is the current time.

AR: $\eta_{ij} = \frac{w_j}{p_j} * \left[1 + \frac{\max\left(d_j - t - p_j, 0\right)}{k * \overline{p}} \right]$ (Eq.2.21)

Third rule is a variant of exponential function, termed MRV, developed by (Morton, Rachamadugu & Vepsalainen, 1984). In Eq.2.22, t is current time, $\overline{p}$ is the average processing time of the job not scheduled yet, k is a look ahead parameter as in AR rule.

MRV: $\eta_{ij} = w_j * e^{-\frac{\max\left(d_j - t - p_j, 0\right)}{k * \overline{p}}}$ (Eq.2.22)

A basic modification on MRV which is a combination of WSPT and AR yields another rule, which is well known Apparent Urgency rule (Potts & Van Wassenhove, 1991).

AU: $\eta_{ij} = \frac{w_j}{p_j} * e^{-\frac{\max\left(d_j - t - p_j, 0\right)}{k * \overline{p}}}$ (Eq.2.23)

2.6.4.1 Trail Updating

To update the trails after each iteration, Eq.2.15, and Eq.2.16 are used. As for the local updating, Eq.2.17 and Eq.2.24 is used, where T_{WSPT} is the total weighted tardiness of the sequence generated by the WSPT rule.

$$\tau_0 = \frac{1}{n * T_{WSPT}} \qquad \text{(Eq.2.24)}$$

2.6.5 Routing Problems

A very successful application of ACO on dynamic problems is AntNet (Di Caro & Dorigo, 1998a; Di Caro & Dorigo, 1998b). AntNet was applied to routing in packet-switched networks like Internet.

Routing problems are basically defined as, given a graph representing a communications network, finding the minimum cost path between each pair of nodes. Note that, although the definition seems to be analogous to TSP definition, the problem becomes extremely difficult when the costs on the edges are time-varying stochastic variables, which is the problem tackled by the AntNet.

2.6.5.1 Solution Construction

In AntNet, each ant searches for a minimum cost path between a pair of given nodes of the network. To achieve this goal, ants are initially positioned onto each network node (s), and move to randomly chosen destination nodes (d), i.e., starting from the node s, each ant moves to d by hopping from one node to another. As in all ACO approaches, deciding the next node to hop is based on a probabilistic action choice rule which is a function of local pheromone and heuristic information.

In ACO approaches designed for solving static problems the pheromone trails are associated to edges, however in AntNet, trails are associated to arc-destination pairs. The idea behind is the following: An edge might be useful while traveling to a node x, but the same edge is very likely improper while traveling to another node y. Formally speaking, each (i,j) directed edge has n-1 trail values attached to it where n is the number of nodes in the graph. Each edge also has an associated heuristic information independent from the target node. The heuristic function employed is another difference of AntNet from static ACO applications.

$$\eta_{ij} = 1 - \frac{q_{ij}}{\sum_{l \in N_i} q_{il}} \qquad \text{(Eq.2.25)}$$

where q_{ij} is the length of the queue (in bits) waiting in channel from i to j.

The channels with shorter queue will have a higher heuristic value.

In static problems the goal is to minimize the cost, which is usually the tour length and closely related to the heuristic function. However, in telecommunications routing problems the objective is to minimize the arrival time of the packets, and the heuristic function given in Eq.2.25 has no time component, and therefore is not used in cost calculation. Instead, since ants are using same channels as packages and experiencing same time delays, the time elapsed while moving from s to d, T_{sd}, is considered as a measure of solution quality.

2.6.5.2 Trail Updating

After an ant, say k, completed its path, to deposit pheromone it moves back to its source node along the same path, using a high priority queue in order to allow a fast propagation of the collected information.

$$\tau_{ijd}(t+1) = (1-\rho) * \tau_{ijd}(t) + \rho * \Delta\tau^{k}(t) \qquad \text{(Eq.2.26)}$$

After the pheromone trail on the visited edges has been updated, the pheromone value of all the outgoing connections of the same node i, relative to the d, evaporates in such a way that the pheromone values are normalized and can continue to be usable as probabilities.

$$\tau_{ijd}(t) = \tau_{ijd}(t) \Big/ \left(1 + \Delta\tau^{k}(t)\right), \forall j \in N_i \qquad \text{(Eq.2.27)}$$

where N_i is the neighbors of the node `i`.

AntNet is the most well known application of ACO algorithms for dynamic problems. It was compared to the many state-of-the-art algorithms on a large set of problems under a variety of traffic conditions. More details can be found in (Di Caro & Dorigo , 1998b).

CHAPTER THREE

CASE STUDIES ON ACO ALGORITHMS

3.1 TSP Application

The application has been developed for Microsoft Windows® platform on a Pentium 4, 1.4GHz PC using Borland C++ Builder 6.0 Development Environment. Since the TSP is the first problem on which ACO was applied, we have developed an application on TSP to appraise parameter settings for further works.

All the tests reported in this section are based, where not otherwise stated, on the *Eil51* problem, which is a 51-cities problem described in TSPLIB (Reinelt, 1991). All the tests have been carried out for 100 cycles with 10 ants and averaged over five trials.

ACS has been selected as it is known to be the most beneficial algorithm for static problems. In ACS, there are four parameters affecting the solution quality. They are:

- α : Pheromone coefficient, relative influence of pheromone information.
  $0 \leq \alpha \leq 1$, increment rate is 0.1;
- β: Heuristic coefficient, defines the relative influence of the heuristic function on probability.
  $0 \leq \beta \leq 1$, increment rate is 0.1;
- ρ : Evaporation coefficient, level of pheromone that disappears in time, and correspondingly, level of pheromone that will be added.
  $0 \leq \rho < 1$, increment rate is 0.1;
- $q0$: Learning rate, the smaller learning rates yield in making decisions probabilistically.
  $0 \leq q0 \leq 1$, increment rate is 0.1.

Three distinct groups of tests were performed. In first group, it has been intended to obtain the best appropriate parameter settings. Second group was used to decide the update mechanism to be applied; either global best update or iteration best update, and the in last group of tests, the results of ACO algorithm with the parameters found in first tests have been compared and contrasted with some other well-known heuristics.

3.1.1 Parameter Settings

Almost all applications in the literature the parameters mentioned above are given trivially. What we have intended to accomplish here is not only to obtain the set of best fitting values, but also to observe how they infer each other.

In all figures a represents α, b represents β, and r represents ρ.

2.6.5.3 Learning Rate versus Evaporation Coefficient

Setting heuristic and pheromone coefficients to some constant values, the relationship between learning rate and evaporation coefficient has been observed.

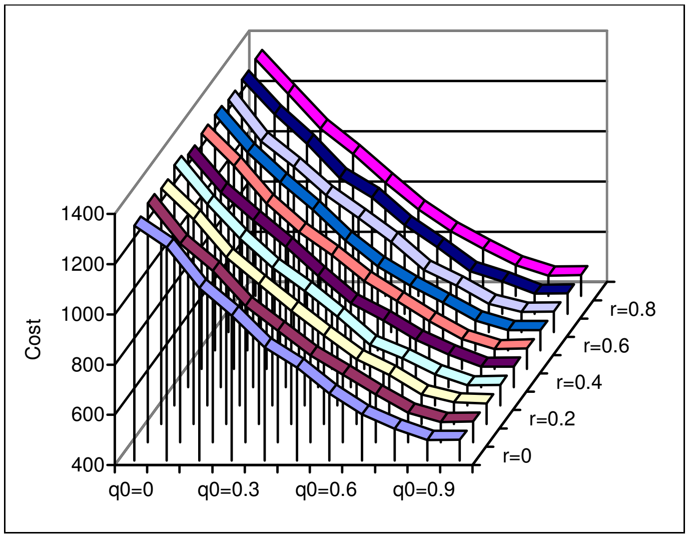


Figure 3.1 The performance of learning rate vs evaporation coefficient for $\alpha = 0, \beta = 0$.

In Figure 3.1, all ρ values exhibit the approximately the same behavior against the values of q0. However, learning rate variations result in quite important changes on solution qualities. So, if neither heuristic nor pheromone information is in use, learning rate determines the solution quality. Figure 3.2 shows another view for the same settings which is clearer in determining the inference between learning rate and evaporation.

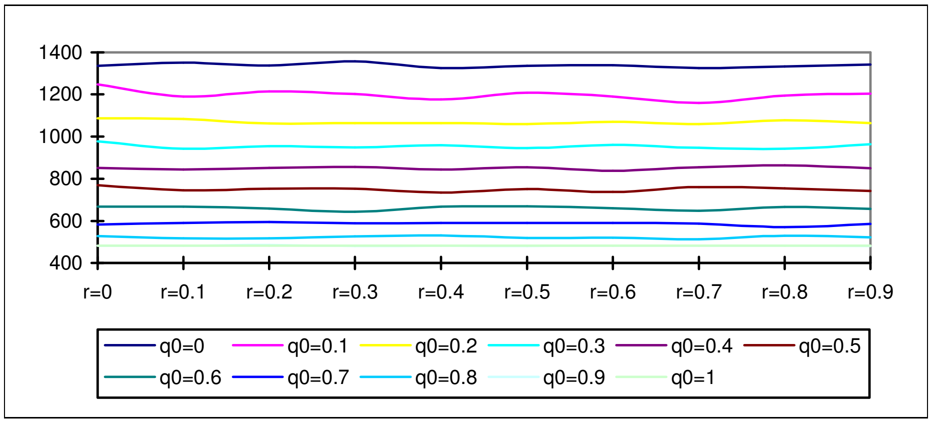


Figure 3.2 The behavior of learning rate against evaporation for $\alpha = 0, \beta = 0$.

From Figure 3.2, it can be concluded that, setting the learning rate to the upper bound causes the solutions to stuck in local optimum, which is a solution optimal within a neighboring set of solutions. The reason behind is, for $q0 = 1$, no probabilistic decision is allowed, only the state which has the shortest edge, and largest pheromone strength is chosen. In addition, all solutions generated for $q0 = 1$ are same since neither heuristic nor pheromone information is involved. However for small values of q0, it was expected to obtain more probabilistic results, which is not the case. The reason is, settings of the other two parameters, $\alpha = 0, \beta = 0$, disallows using any kind of pheromone and / or heuristic information, i.e., distance.

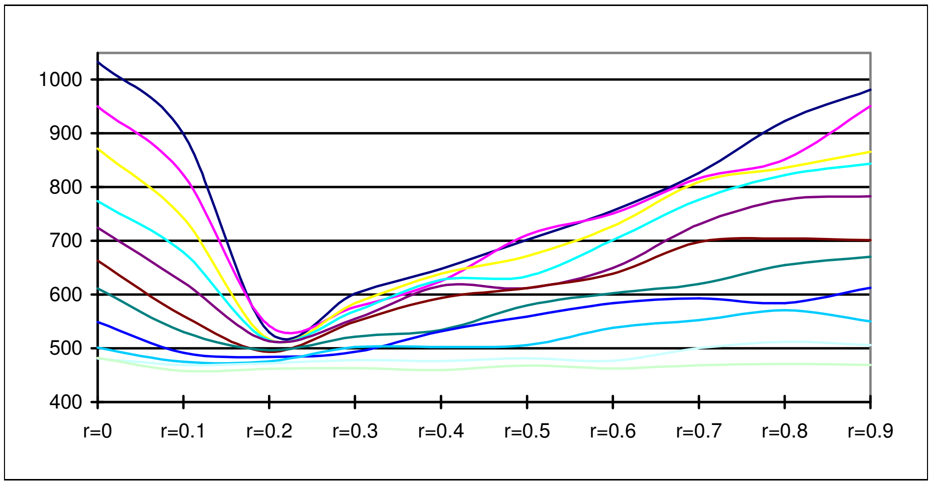


Figure 3.3 The behavior of learning rate against evaporation for $\alpha = 1, \beta = 1$ (The legend is same as that of Figure 3.2).

Figure 3.3 shows the behavior of learning rate against evaporation coefficient when using all available information of both heuristic and pheromone strength with equal weight. There are some important points to be emphasized: the maximum value of cost has decreased; for $q0 = 1$, there is a slight variation on solutions; and, there is a certain breakpoint for almost all values of q0, at $\rho \approx 0.2$. While the solution qualities are getting better until this point, they tend to get worse for the larger evaporation rates.

2.6.5.4 Heuristic Information versus Learning Rate (β vs q0)

In this group, the effect of pheromone strength and evaporation rate has been set to fixed values. This class of tests are vital in the sense that, pheromone strength and evaporation rate are closely related to each other, and setting them to some fixed values, it is possible to acquire how the other two parameters are functioning.

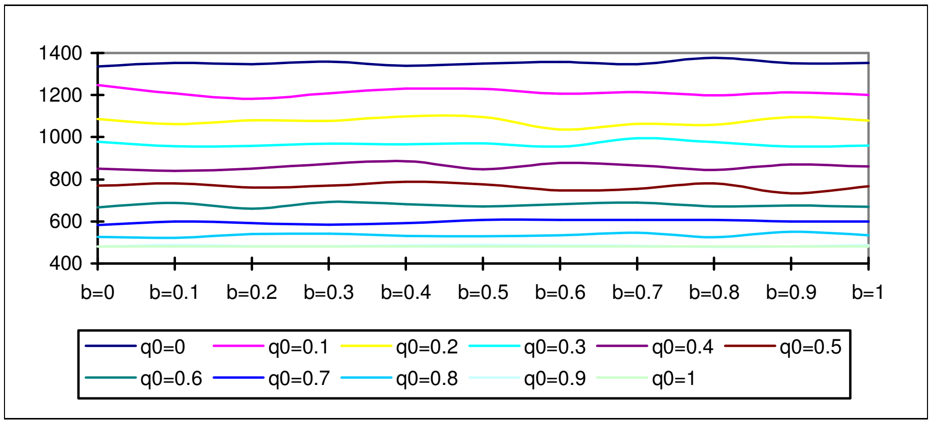


Figure 3.4 The behavior of learning rate against heuristic information for $\rho = 0, \alpha = 0$.

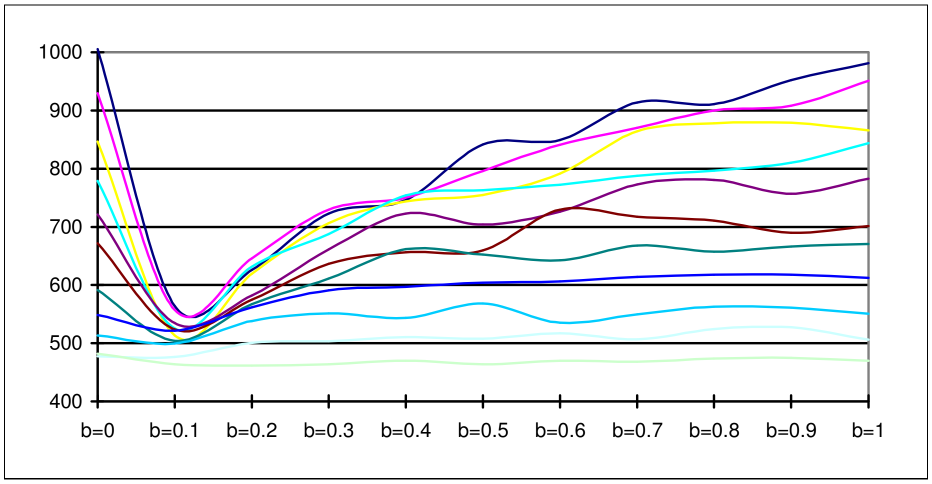


Figure 3.5 The behavior of learning rate against heuristic information for $\rho = 0.9, \alpha = 1$.

For the initial values of evaporation and pheromone coefficients, meaning that no pheromone addition and using no heuristic; all learning rate values exhibit the same behavior, and heuristic information usage does not affect the solution quality (Figure 3.4). On the other hand, when evaporation ratio is set to 0.9, meaning that loosing almost all available pheromone information, and adding some amount reflecting only current solution quality (using a little amount of previously deposited pheromone), and using the heuristic information without empowering, we obtain a certain breakpoint on pheromone evolution at $\beta \approx 0.2$.

2.6.5.5 Heuristic Information versus Evaporation Rate(β vs ρ)

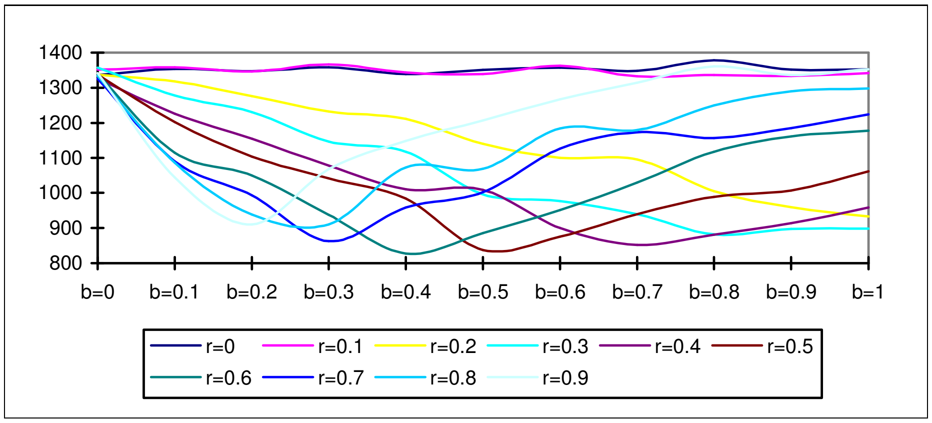


Figure 3.6 The function of ρ against β for $\alpha = 0, q0 = 0$.

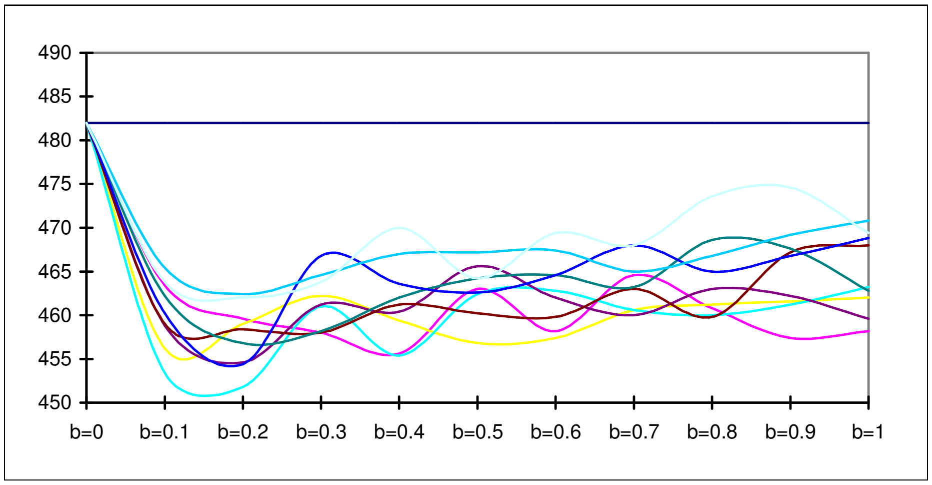


Figure 3.7 The function of ρ against β for $\alpha = 1, q0 = 1$.

In Figure 3.6, when no pheromone in use, and all solutions are generated by probabilistic action choice rule; the behavior of $\rho = 0$ and $\rho = 0.1$ are same, and has no change in solution qualities, while other ρ values are operating similarly, but having different minimal breakpoints on different heuristic usages. When pheromone strength is used without empowering, and all solutions are directed towards exploitation, all ρ values except 0, behave in a similar fashion, and there is a slight breakpoint at $\beta \approx 0.2$.

2.6.5.6 Pheromone Strength versus Evaporation(α vs ρ)

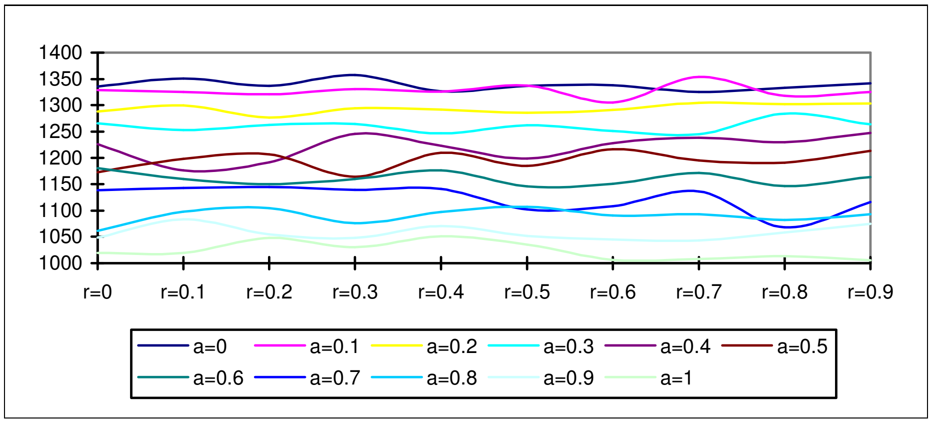


Figure 3.8 The function of α against ρ for $\beta = 0, q0 = 0$.

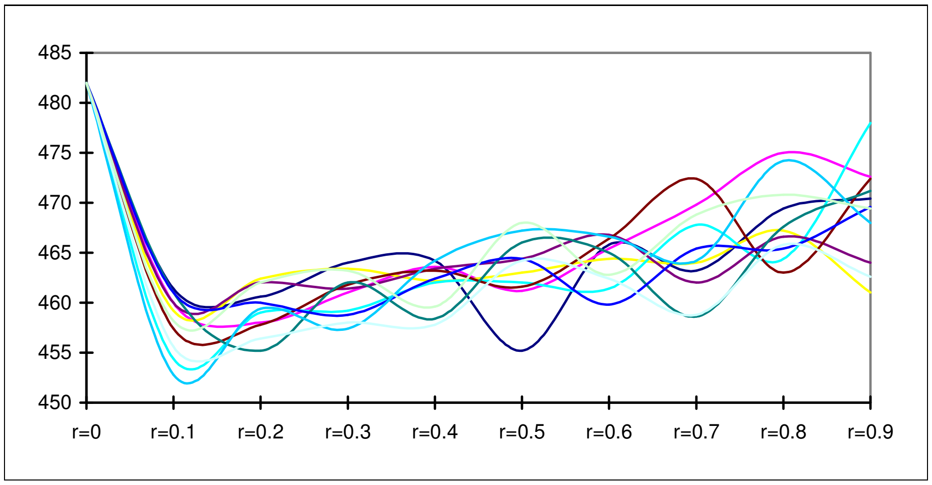


Figure 3.9 The function of α against ρ for $\beta = 1, q0 = 1$.

In Figure 3.8, all pheromone values are operating similarly, although there are slight differences, there is no contribution at all (there is not heuristic information involved). However, in Figure 3.9, when using all available heuristic information, it can be observed that the costs start to get increase when evaporation increases.

3.1.2 Conclusions

The conclusions extracted from tests can be figured out as following:

- Learning rate parameter dominates all others, meaning that, for the values close to the upper limit of its range, all the solutions generated starts to be the same independent from other parameter settings, which is an undesired situation. Note that, this is not a stagnation behavior, because stagnation is such a behavior that, in early iterations each ant acts autonomously, and in later iterations they impose to generate approximate and probably the same solutions because of the pheromone effect. However, for the large values of learning parameter, this is not the case. All ants, starting from the first iteration propagated the same solutions.
- As long as evaporation is higher than pheromone increment, pheromone effect does not make any contributions on the solution quality. Therefore, the pheromone coefficient should be greater than the evaporation ratio.
- It is better to use the heuristic information as much as possible, however, since pheromone strength is carrying much more information about the solution qualities, heuristic usage should not dominate over pheromone. If such a situation occurs, the behavior of ants gets closer to stochastic search as defined in ACO algorithms.

3.2 2D Projected Map Application

The problem is similar to the well-known TSP, and an application from pipeline and network design (Ekin & Yakhno, 2001; Yakhno & Ekin, 2002). There is a landscape containing some obstacles like mountain ranges, hills, rivers etc., which increases the cost of laying a highway on. Then this map is projected onto a 2 dimensional plane. The projected map contains certain polygons with their costs, which corresponds to the obstacles on the landscape. There is an initial position, which can be considered as the starting city in TSP, and a final position, which is the starting city again in TSP. The problem is to find the minimum cost path on this projected map, from initial position to the final position, which is similar to finding the shortest closed tour in TSP. Figure 3.10 demonstrates an example map with 9 polygons. The cost each polygon is marked as an integer inside of the polygon. The initial point is marked with “+”, and the destination point is marked with “*”.

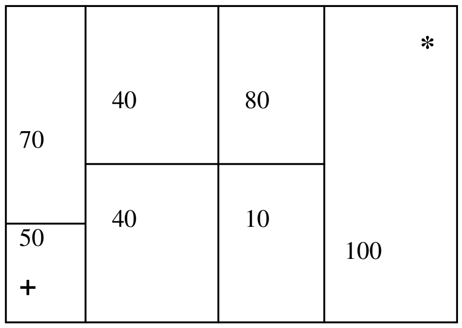


Figure 3.10 An example map with 7 polygons.

While applying ACO algorithms on these kinds of problems, the point that should be considered first is the specification of the map. To specify the map in an appropriate way, we used 2 dimensional coordinate spaces, translated the map to origin and defined as the following:

```
outerMostPolygon(Left_Bottom_Coord, Right_Top_Coord)
polygon(Left_Bottom_Coord, Right_Top_Coord, Cost_Of_Polygon)
initialPosition(coord)
finalPosition(coord)
```

The `OuterMostPolygon` defines the limits of the map, and since it is not a real polygon having a cost, it does not have `Cost_Of_Polygon` property. However, to prevent the ants to go out of the map, a virtual cost is assigned to the `OuterMostPolygon`. This cost is considered only when moving along the `OuterMostPolygon'` s borders.

A cost assigned to a polygon defines the cost of moving from one point to another in that polygon. There are some special cases while calculating the cost while moving from N_A, to N_B. Suppose that there are two neighboring polygons P_A, P_B.

1: N_A is in P_A, N_B is on border of P_A and P_B. Then

```
Cost_Of_Moving N_A _to_ N_B = cost_Of_ P_A
```

2: N_A is on border of P_A and P_B, N_B is in P_B. Then

```
Cost_Of_Moving N_A_to_ N_B = cost _Of_ P_B
```

3: N_A and N_B are both on the border of two polygons.

```
Cost_Of_Moving N_A_to_N_B = averageOf( cost_Of_P_A , cost_Of_P_B )
```

4: N_A and N_B are both in the P_A, they are not on a straight line, but on a diagonal.

```
Cost_Of_Moving N_A_to_N_B = sqrt( 2 ) * cost_Of_P_A
```

Calculating the costs of all possible traveling directions, the map becomes a 3-dimensional cost matrix. Being on `x,y`, the first dimension keeps the axis of current position (`x`), second keeps ordinate of current position (`y`), and the last dimension keeps the costs of moving one of eight possible neighbor nodes.

3.2.1 Modifications on ACO Algorithm

Applying ACS algorithm on defined problem, a number of ants having memory to remember the visited nodes, are placed on the initial position on the map, then attempt to construct the solution by moving from one node to another. The main algorithm of modified ACS is:

```
For a numberOfCycles
   Randomly select initial positions for ants
   while all ants are neither in trap nor in final state
      For each ant:
        Move to next state
        Update ant memory
        local update-optional
      End
   End
   Global trail updating
End
```

Figure 3.11 Modified ACO algorithm.

2.6.5.7 Moving Strategy

The moving strategy is as following:

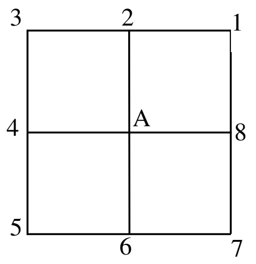


Figure 3.12 Being on node A(i, j), possible nodes to move to.

An ant being on node `A(i,j)` can move eight nodes which are indexed as in Figure 3.12. The index value, i, and j, correspond to `z`, `x`, and `y` respectively in the below cost matrix, which is the definition of the problem.

$$\cos ts_{xyz} \rightarrow C_{xyz} \qquad \text{(Eq.3.1)}$$

The coordinates of indexed nodes are calculated as the following:

$$\begin{array}{ll} 0 \rightarrow (i+1, j) & 4 \rightarrow (i-1, j) \\ 1 \rightarrow (i+1, j+1) & 5 \rightarrow (i-1, j-1) \\ 2 \rightarrow (i, j+1) & 6 \rightarrow (i, j-1) \\ 3 \rightarrow (i-1, j+1) & 7 \rightarrow (i+1, j-1) \end{array} \qquad \text{(Eq.3.2)}$$

Selecting any of these nodes depends on a probability that combines the cost of that possible move, and the pheromone trail level on that edge, if any. Ants probabilistically choose the node, which has lower cost and higher pheromone level.

The probability of selecting node k being on (`i, j`), in the cycle of h,

$$P_{ijk}^{h} = \begin{cases} \dfrac{\left[W_{ijk}^{h}\right]^{\alpha} * \left[1/C_{ijk}\right]^{\beta}}{\sum\limits_{all\_m} \left[W_{ijm}^{h}\right]^{\alpha} * \left[1/C_{ijm}\right]^{\beta}} & \text{if k is not visited} \\ 0 & \text{otherwise} \end{cases} \qquad \text{(Eq.3.3)}$$

$W[x][y][z]$ is the matrix that keeps pheromone trails and has the same structure that of Eq.3.1. α and β are two parameters that determines the relative influence of cost and pheromone levels. Setting α to 0, causes the pheromone trails not to be considered by ants, they choose the node, which has cheapest cost. However, setting β to 0 results in moving towards the higher pheromone trail regardless of cost.

According to Eq.3.3, each ant moves step by step in each time unit. At this point, computer simulations showed that, using this simple probabilistic rule, many of the ants go trap states, in which there is no unvisited node to move to, before reaching the final position.

In the problem definition, since there is no predefined paths, an ant does not know where to move to reach the final state, and this behavior corresponds to search state of real ant's behavior, which is not modeled in AS. Therefore, the ants walk around randomly if there is no deposited pheromone trail, and they may go into trap node.

To decrease the probability of going into a neighbor that causes the ant to stop searching, we propose to check how many available (not visited) neighbors the node that the ant plans to move has. But, this primitive heuristic does not guarantee not to move into dead configurations. The ants somehow should be directed towards the final position, and they should be encouraged selecting the neighbor which makes them closer to final position in terms of distance. In order to obtain this type of heuristic information, the Euclidian distance between the possible next node and the final position is calculated and added to the probability function as a posteriori knowledge.

The resulting probability function of selecting node k being on (i,j), in the cycle of h is:

$$P_{ijk}^{h} = \begin{cases} \dfrac{\left[W_{ijk}^{h}\right]^{\alpha} * \left[1/C_{ijk}\right]^{\beta} * \left[aN_{k}\right]^{\delta} * \left[1/dist_{kf}\right]^{\lambda}}{\sum\limits_{all\_m} \left[W_{ijm}^{h}\right]^{\alpha} * \left[1/C_{ijm}\right]^{\beta} * \left[aN_{m}\right]^{\delta} * \left[1/dist_{km}\right]^{\lambda}} & \text{if k is not visited} \\ 0 & \text{otherwise} \end{cases} \qquad \text{(Eq.3.4)}$$

aN_k and aN_m are the number of unvisited neighbors of node k and m respectively. $dist_{kf}$ is the Euclidian distance between the node k and the final position. δ and λ the two parameters defining the effect of the available neighbor and distance heuristics.

According to Eq.3.4, all ants are expected to reach the goal state after a transition phase. However, there is still a chance, which takes them into dead states. Therefore, instead of checking if all ants are reached the final state, also their position is controlled whether it is a trap state or not, to detect if current cycle is ended or not. When an ant is realized to be in trap state, it is killed in that specific cycle, which means, it stops searching for further movements. When all ants are in either trap state or final state, the solution is found, and the trail update mechanism is triggered.

2.6.5.8 Trail Update Mechanism

When the end of a cycle condition is met, the pheromone trails on each connected two nodes are updated. While updating pheromone trails, the evaporation characteristic of real pheromone trails is considered as a first step.

$$W^{l}(t+1) = W(t+1) * (1-\rho) \qquad \text{(Eq.3.5)}$$

ρ is the evaporation coefficient.

The ants those are either in final state or in a trap state are at the end of cycle. The solutions constructed by the ants are said to be either complete solution or partial solution, with respect to the state the solution ends. The partial solutions, list of nodes that causes ants to fall into trap state, are punished, whereas the complete solutions, the paths reaching the final state, are rewarded.

$$W_{ijk}(t+1) = W_{ijk}^{l}(t+1) + \sum_{\text{for all ants\_m}} \Delta W_{ijk}^{m}(t) \qquad \text{(Eq.3.6)}$$

W_{ijk} is the pheromone trail level on edge between nodes (i, j) and the neighbor indexed by k. ΔW_{ijk}^{m} is the pheromone trail deposited by ant k, on each edge it visited, calculated as following:

$$\Delta W_{ijk}^{m} = \begin{cases} 1/Cost^{m}(t) & \text{if ant m visits edge (i,j)-k, complete solution} \\ -1/PCost^{m}(t) & \text{if ant m visits edge (i,j)-k, partial solution} \\ 0 & \text{if ant m does not use edge (i,j)-k} \end{cases} \qquad \text{(Eq.3.7)}$$

The amounts of pheromone trail on all visited edges, including the ones that are part of partial solutions are updated according to Eq.3.7. By Eq.3.7, the cheaper the tour is, the more pheromone trail its edges receive, and more likely to be chosen in following iterations. Unlikely, the possibility of selecting edges included in partial solutions decreases. And for the last case, if an edge is not chosen by ants, its associated pheromone trail decreases exponentially.

3.2.2 Experimental Results

We have followed ACS algorithm with proposed modifications in which all ants update pheromone trails after each move, i.e. local trail updating, where just the ant that generates the best solution in each iteration is allowed for global trail updating.

The test includes small number of ants, and small number of cycles. All presented results are average of five trials on each set.

Table 3.1 The percentage of ants that can construct solutions for different δ and λ values. This test is performed with 1 ant and for 1 cycle. Costs are average costs.

	λ=0		λ=1		λ=2		λ=5	
	Solution	Cost	Solution	Cost	Solution	Cost	Solution	Cost
δ=0	%0	0	%60	5852	%80	1952	%100	1573
δ=1	%40	3263	%20	1955	%40	1509	%100	1508
δ=2	%80	1989	%80	936	%80	1035	%100	858
δ=5	%20	751	%60	1187	%100	1588	%100	753

Table 3.1 shows that, if no heuristic is used to direct an ant towards the final position, it is not able to reach that position. Since no ACO specific information is used for this test, the improvements on the path costs show that, those heuristics are useful not only for guiding the ants but may also for optimization process.

Table 3.2 Number of ants=5, number of cycles=5, α=1, β=1, ρ=0. 9. The costs are average costs.

	λ=0		λ=1		λ=2		λ=5	
	Solution	Cost	Solution	Cost	Solution	Cost	Solution	Cost
δ=0	12%	2562	28%	2100	57.6%	1651	89.6%	1040
δ=1	16%	1708	44%	1634	63.2%	1243	96.8%	922
δ=2	28%	1740	50.5%	1453	76%	1259	96%	845

In Table 3.2, the effects of δ and λ can be evaluated more clear. Increasing λ causes more ants to be alive and also cheaper cost. Nevertheless, this cheaper cost effect is more likely occurs just because of the specified problem, that is, problem specific.

As the last experiment, remaining all parameters same as Table 3.2, 100 ants employed for 10 cycles the system. In this test the minimum cost of the path from initial position to final is found as 518.

3.3 SMTTP Application

We have followed ACS algorithm in which all ants update pheromone trails after each move, i.e. local trail updating, where just the ant that generates the best solution in each iteration is allowed for global trail updating.

In generic ACO algorithm, ants are initialized with assigning some randomly chosen nodes. The randomization makes no distinction between the generated solutions qualities. However, it is not the case in SMTTP problem. In order to generate feasible solutions more quickly, ants should find the "best" job to be scheduled first. Selecting the first job to be scheduled can be managed in several ways, such as the job with the shortest processing time, or the job with the longest processing time, or the job with the earliest due date, etc. In the tests we have performed, the job with the earliest due date is assigned as the first job in the schedule for all ants.

Another point in SMTTP application is local search. ACO algorithms have been proven to be very successful if combined with a local search procedure. The procedure has applied on iteration best solutions generated by the ants before updating the trail intensities. Circular shifting which was defined in Section 3.5 has been used as for the local search method.

MDD heuristic was used in action choice rule. The tests were performed with the following parameters:

$$\begin{aligned}&\alpha = 0, \beta = 3, \rho_G = 0.9, \rho_L = 0.5, q0 = 0.5\\&nofCycles = 5000\\&nofAnts \in \{20, 40, 60, 80, 100\}\end{aligned}$$

Almost for all problems, the solutions generated by cooperative work of 100 ants were the best among all others.

3.4 Problem Sets

Two different problem sets have been experimented in this study.

3.4.1 Problem Set 1

Problems Set 1 was taken from (Baker,1974). Set 1 contains 16 problems each of which has 8 jobs.

3.4.2 Problem Set 2

This set contains 800 problems created by the method suggested in (Potts & Van Wassenhove, 1982). The method generates problems with various degrees of difficulty by means of two factors, tardiness factor (TF), and range of due dates (RDD).

$$TF = 1 - \frac{\sum_{j=1}^{n} d_j}{n \sum_{j=1}^{n} p_j} = 1 - \frac{d_{avg}}{C_{\max}} \qquad \text{(Eq.3.8)}$$

TF indicates the tightness of the due dates, and the difference between average due dates and the completion time of possible longest job. For small values of TF, the jobs get looser, which is the case that EDD rule creates optimal schedule. On the other hand, if all jobs are subject to be completed at starting time, i.e., $d_{avg} = 0$, then TF=1.

$$RDD = \frac{d_{\max} - d_{\min}}{C_{\max}} \qquad \text{(Eq.3.9)}$$

If all jobs are to be completed at the same date, then RDD=0. One can conclude that, small values of RDD indicates that the jobs should be finished in a short time, where larger RDD means there is more time to process jobs.

For each problem, processing times for *n* jobs are generated from a uniform distribution [1,1000]. Computing the total processing time P, the due dates are calculated from a uniform distribution defined as:

$$[P\left(1-TF-RDD/2\right),P\left(1-TF+RDD/2\right)] \qquad \text{(Eq.3.10)}$$

The problems have been generated with the following characteristics:

$$TF \in \{0.2,0.4,0.6,0.8\}$$
$$RDD \in \{0.2,0.4,0.6,0.8\}$$
$$N \in \{20,40,60,80,100\}$$

For each problem size, there are 16 different difficulty levels, and for each problem having different size and difficulty level, 10 problems were generated, yielding 800 problems in total.

3.4.3 Results for Problem Set 1

Table 3.3 Results for the set of problems taken from literature.

TF,RDD	***API***	***Best***	***PSK***	***Best***	***SA***	***Best***	***ACO***
(0.2, 0.2)	755	1	765	0	755	1	**755**
(0.2, 0.4)	**668**	1	**668**	1	**668**	1	**668**
(0.2, 0.6)	378	1	378	1	378	1	378
(0.2, 0.8)	708	0	711	0	675	1	675
(0.4, 0.2)	364	1	364	1	**364**	1	364
(0.4, 0.4)	478	1	478	1	478	1	**478**
(0.4, 0.6)	820	1	820	1	820	1	**820**
(0.4, 0.8)	1364	0	1364	0	1360	1	**1364**
(0.6, 0.2)	63	1	63	1	**63**	1	63
(0.6, 0.4)	1300	1	1300	1	**1300**	1	1300
(0.6, 0.6)	662	1	662	1	662	1	**662**
(0.6, 0.8)	531	1	**531**	1	531	1	531
(0.8, 0.2)	271	1	271	1	**271**	1	271
(0.8, 0.4)	936	1	**936**	1	936	1	936
(0.8, 0.6)	708	1	**708**	1	708	1	711
(0.8, 0.8)	1285	0	**1285**	0	1280	1	1285
Total	11291	13	11304	12	11249	16	11261

Table 3.3 has no significant contribution on performances of the metaheuristic algorithms since each problem set contains very small number of jobs, i.e. 8 jobs. Therefore we have performed similar experiments on problem sets containing more jobs.

3.4.4 Results for Problem Set 2

Table 3.4 summarizes the outcomes of the experiments. It also shows that how many times each algorithm performed best, compared to the others for the problems of size 100. The numerical results in Table 3.4 for API, PSK, and SA heuristics were provided by Bilge Bilgen from Dept. of Industrial Engineering, DEU (Bilgen & Özkarahan, 2002).

There are three important elements of SA algorithm and has been used with the following settings:

- Initial sequence: EDD rule has used to create the initial schedule.
- Neighborhood searching structure: General Pairwise Interchange, GPI, in which all possible pairs are exchanged, except the problems for N=80. Adjacent Pairwise Interchange, API, is the neighborhood structure in N=80 problem set.
- Cooling strategy: Total number of iterations is 100, 2 is the initial temperature. At iteration 50, the temperature is lowered by factor 0.25.

Table 3.4 Results for the for problems of N=100. API refers to Adjacent Pairwise Interchange technique as the main algorithm, and PSK means Panwalkar-Smith-Koulamas heuristic. (Panwalkar, Smith, & Koulamas, 1993).

TF,RDD	*API*	*Best*	*PSK*	*Best*	*SA*	*Best*	*ACO*
(0.2, 0.2)	22783	3	22783	3	21835	9	**21824**
(0.2, 0.4)	**258**	10	**258**	10	**258**	10	**258**
(0.2, 0.6)	0	10	0	10	0	10	0
(0.2, 0.8)	0	10	0	10	0	10	0
(0.4, 0.2)	189877	1	189877	1	**182500**	10	185517
(0.4, 0.4)	126295	3	126262	3	124600	9	**123720**
(0.4, 0.6)	67305	4	67256	4	66450	6	**66206**
(0.4, 0.8)	5790	8	5790	8	5808	8	**5760**
(0.6, 0.2)	509251	0	510224	0	**483650**	10	502780
(0.6, 0.4)	435224	1	435224	1	**430590**	9	432963
(0.6, 0.6)	364665	4	364613	4	363580	7	**363365**
(0.6, 0.8)	296823	8	**296813**	8	302175	3	297878
(0.8, 0.2)	991931	0	992052	0	**965190**	10	981721
(0.8, 0.4)	953567	9	**953544**	10	958410	2	955985
(0.8, 0.6)	939036	9	**939023**	10	943315	3	942852
(0.8, 0.8)	969283	9	**969283**	3	974550	3	973845
Avg.	367005.5		367062.6		363931.9		365917.1

In Table 3.4, considering the average total tardiness values in the last row, the results reveals that, both SA and ACO algorithms perform better than the other heuristic algorithms and ACO and SA should be preferred to the other two methods.

If we compare SA versus ACO, it can be said that those two metaheuristics with the identified parameter settings, and for these data sets, are competing which each other.

Table 3.5 Results for the problem sets of the sizes 20, 40, 60, 80. No local search has been employed for the problems of size 80 in ACO algorithms, while for SA, API has been used as local search technique.

TF,RDD	***N=20***		***N=40***		***N=60***		***N=80***	
	SA	**ACO**	**SA**	**ACO**	**SA**	**ACO**	**SA**	**ACO**
(0.2, 0.2)	1508	**1491**	4527	**4480**	9285	**9206**	16220	**155299**
(0.2, 0.4)	**331**	**331**	326	**325**	451	**449**	471	**466**
(0.2, 0.6)	**55**	**55**	**0**	**0**	**0**	**0**	**0**	**0**
(0.2, 0.8)	**182**	**182**	**0**	**0**	**0**	**0**	**0**	**0**
(0.4, 0.2)	9160	**9156**	**32410**	32414	**63800**	64308	126000	**113010**
(0.4, 0.4)	6201	**6192**	19450	**18832**	44180	**44012**	85670	**75676**
(0.4, 0.6)	4090	**4070**	10705	**10634**	18103	**17988**	39300	**35657**
(0.4, 0.8)	2275	**2258**	2902	**2891**	8912	**8766**	8901	**7875**
(0.6, 0.2)	**21120**	21124	**83640**	83986	**179800**	183851	429900	**299429**
(0.6, 0.4)	**19000**	**19000**	66610	**66553**	145400	**144936**	381900	**264987**
(0.6, 0.6)	**20721**	20730	63120	**63002**	131040	**130778**	283700	**204506**
(0.6, 0.8)	**18490**	18608	**58230**	58247	115190	**113906**	295100	**208442**
(0.8, 0.2)	**45720**	42776	**153650**	154663	**359200**	362076	923500	**626824**
(0.8, 0.4)	**43790**	45056	**145000**	145111	335400	**334742**	915900	**573505**
(0.8, 0.6)	**39220**	39319	**175600**	176350	**360400**	360763	904200	**582975**
(0.8, 0.8)	**42980**	43069	**163200**	163902	341900	**341340**	872900	**590533**
Avg.	17177.7	17088.6	61210.6	61336.9	132066.3	132320.1	330228.9	233699

Tables 3.5 shows and compares the results of SA and ACO for the problem sets of size 20, 40, 60, 80. The results are interesting in the point that for the problems of size 80, ACO always outperforms SA although no local search was employed in ACO. The reason is, instead of GPI, API technique was employed for SA with the previous parameter settings. Therefore, one can conclude that, the neighborhood structure is quite important, and if GPI were used for ACO algorithms, the results would have been outperform for other sets as well. See Table 3.6 demonstrating the results of ACO algorithm for the problem set 80 with and without local search technique of circular shifting.

Table 3.6, ACO results for N=80, with and without neighborhood search.

TF,RDD	*Without Local Search*	*With Local Search*
(0.2, 0.2)	155299	155256
(0.2, 0.4)	466	466
(0.2, 0.6)	0	0
(0.2, 0.8)	0	0
(0.4, 0.2)	113010	111482
(0.4, 0.4)	75676	75113
(0.4, 0.6)	35657	35552
(0.4, 0.8)	7875	7837
(0.6, 0.2)	299429	297298
(0.6, 0.4)	264987	263097
(0.6, 0.6)	204506	203291
(0.6, 0.8)	208442	207728
(0.8, 0.2)	626824	621962
(0.8, 0.4)	573505	571696
(0.8, 0.6)	582975	581652
(0.8, 0.8)	590533	588501
Avg.	233699	232558.2

CHAPTER FOUR

WEIGHTED SYNCHRONISED CALCULUS OF COMMUNICATING SYSTEMS

4.1 WSCCS as a Multi Agent System Modeling Language

WSCCS is known to be very efficient in modeling biological systems, particularly social insect colonies, rather than its predecessors, i.e., CCS, SCCS. But the range is not limited with biological systems. Any distributed computing system can be modeled by WSCCS. For example, problems from Operations Research, such as a problem involving first come first served based resource allocation system, or a problem of probabilistic task allocation falls into domain of problems that can be modeled using WSCCS. Modeling computer systems is also in concern of WSCCS.

WSCCS was first introduced by Tofts, 1994. It appears as an extension of Milner's Synchronized Calculus of Communicating Systems, SCCS (Milner, 1989). The first improvement over CCS is the synchronization of actions of the agents, which helps a clear understanding the expression of communication between agents. However, the synchronization extension does not remove the possibility of asynchronous action occurrence, since the calculus is still CCS. Adding weight, priority information to the synchronized CCS actually causes a major change over CCS. The new formalism is not completely nondeterministic, since the weights, and probabilities are used to compute the choice of an action to be performed. However, it is not completely deterministic either. For the two actions having same probability to be chosen, again a nondeterministic action choice rule works.

When modeling in WSCCS, each individual ant is thought as an autonomous computer program. Each program changes its state both according to weight rules, and through communication with other programs. An ant in WSCCS is described by listing all agents which comprise its possible states of behavior, and the weight that actions occur to change the agent. Composing the agents in parallel and allowing them to communicate with each other, the entire model of ant colonies is obtained.

In the rest of the chapter, the reason behind employing WSCCS as a language for modeling Ant Colonies rather than other languages is explained first. Then the language constructs, semantics, and equational theory are presented.

4.2 Why to Model Ant Colonies by WSCCS?

The difficulties in modeling Ant Systems using Cellular Automata, and Petri Nets has been mentioned before. The main characteristics of WSCCS allow us to defeat those problems.

In Cellular Automata approach, it is not permitted to model the individuals independently, i.e. all agents subject to perform the same set of rules. Therefore only the colony level behavior can be observed. In WSCCS each agent is defined uniquely in its own actions, and if an agent is communicating with either the environment or with another agent a new agent, which is an aggregation of the communicating ones is defined. By this way all the difficulties arising in Cellular Automata approach can be solved.

The main reason for not preferring Petri Nets is its asynchronous behavior, since we are attempting to model synchronous systems. WSCCS is synchronized. Another reason is reproduction behavior. Reproduction is a primitive behavior of any biological system, means that some new agents might be added to the colony with its own set of actions during execution. However, in Petri Nets, once a system is modeled and execution is started, no new rules and / or agents can be added. WSCCS accomplishes the behavior as following: Any different types of possible agent are defined before the system begins execution, and during its run, a number of agents can created according to their definitions. The important point is, only the definitions of agents exist, the number of actually running agents is unknown.

4.3 Definition of the Calculus

WSCCS is a language for modeling the behavior of concurrent communicating systems hence it offers mechanisms to formalize these behaviors. Besides, it is also expected to offer means for expressing alternative actions, the probabilistic action choice rules, and actions occurring sequentially. Furthermore one should be able to verify the system which is supported by equational reasoning.

The language of WSCCS contains three basic classes of objects:

- Agents,
- Actions,
- Weights.

And there are several operations defined on these objects:

- Action performing (. - dot),
- Weighing an action (:),
- Choice between actions (+),
- Parallel composition of agents ($\times$),
- Restriction on the actions to be performed (<),
- Communication between agents,
- Prioritization of action in an agent expression (`prio`),
- Renaming an action ($f_E\left(Act_E\right)$),
- Fixing an agent (fix).

The subsequent sections introduce the syntactical rules of WSCCS along with several examples, although the full syntax is attached to the Appendix A.

4.3.1 Action Performing (.)

This rule indicates that an agent (it might be defined implicitly or explicitly) performs the action *act*, and becomes another agent. In other words, from one state, with a transition *act*, it reaches another state.

- $act.E$: current agent is implicit. It performs the action *act*, and becomes the agent E.
- $P \overset{def}{\equiv} act.E$: current agent is explicit, P.

4.3.2 Weighing an Action (:)

WSCCS is a probabilistic calculus, thus occurrence of an action is biased by means of weight. The reason behind calling the bias term as weight instead of probability is that, weights can be interpreted as probabilistic specifications via the concept of relative frequencies. Usage of weights also allows giving priorities to some actions. In addition, weights are operationally more convenient to use, because one does not have to ensure that they normalize to any particular value.

The weight term which is denoted by $w_i = nw^k$, where w_i is known to be the weight of an action, contains two components in form of multiplication which is usually omitted:

- $n \in Z^+$ is the relative frequency of occurrence of an action with which the weight is associated,
- $k \in N$ w^k gives the action's priority.

Weights are attached to the actions by the operator : (colon) as following:

- $6w : a$ the weight of action a is $6w$, where 6 is the relative frequency of action a, and w is the priority. Note that $k = 1$ for this example.

- $2:b$ the weight of action b is 2, where the relative frequency is 2, and there is no priority assigned to the action b, since there is no component of w, note that this case holds for $k=0$. That means, nw^0 is denoted by n.

A relative frequency expression, e, is formed with the following syntax, where k ranging over a set of variable names, and c ranging over a fixed set, N or R.

$$e \rightarrow k \,|\, c \,|\, e+e \,|\, e*e$$

Moreover the following assumptions are used for relative frequency expressions, where e and f are relative frequency expressions:

$$e+f=f+e \qquad \text{(Eq.4.1)}$$

$$(e+f)+g=e+(f+g) \qquad \text{(Eq.4.2)}$$

$$e*f=f*e \qquad \text{(Eq.4.3)}$$

$$(e*f)*g=e*(f*g) \qquad \text{(Eq.4.4)}$$

$$e*(f+g)=e*f+e*g \qquad \text{(Eq.4.5)}$$

That means, both addition and multiplication are commutative, and associative; multiplication is distributive over addition.

4.3.3 Choice between Actions (+)

At any given time, an agent may have several actions, and only one of them can be performed. + operator indicates there are several choices in actions of an agent. These actions can be given with weights, where the weight expression might contain either a relative frequency component, or a priority component, or both of them.

When an agent is able to perform more than one action at any time, the actions whose priorities are equal to the highest priority may occur. The probability of actions with equal weights is then determined by their relative frequency. An agent performs an action with a probability according to the actions weight and become a different agent.

$$EX \overset{def}{\equiv} 6w : a.A + 4w : b.B + 2 : \#.C \tag{Ex.4.1}$$

The agent *EX* has three choices in its actions, and only one can be performed. *6w,4w,2* are weights, where 6, 4, and 2 gives the relative frequencies; and w^1, w^1, w^0 are the priorities of actions *a, b,* and *#* respectively. *A,B,C* are agents. The + operator indicates a choice between actions.

It is worth to mention here that, relative frequencies are used to compute the probability of occurrence of action. Consider another agent $EX2$ to explain how the relative frequencies are involved in probability:

$$EX2 \overset{def}{\equiv} n_a : a.A + n_b : b.B + n_c : c.C + n_d : d.D \tag{Ex.4.2}$$

n_a, n_b, n_c, n_d are constants,

a, b, c, d are actions,

A, B, C, D are resulting agents of corresponding actions.

$$EX2 \overset{a[n_a/n]}{\rightarrow} A$$
$$EX2 \overset{b[n_b/n]}{\rightarrow} B$$
$$EX2 \overset{c[n_c/n]}{\rightarrow} C$$
$$EX2 \overset{d[n_d/n]}{\rightarrow} D$$

All presumptive transitions for the agent EX2.

The first transition means that, the agent $EX2$ may perform the action a with the probability $[n_a / n]$ where $n = n_a + n_b + n_c + n_d$, sum of all relative frequencies, and becomes agent A. The rest of the transitions are construed at the same way.

Normalizing all the transition weights, i.e., setting the sum to 1, probabilities for each transition are obtained.

4.3.4 Parallel Composition of Agents (×)

This is a core operation of any distributed computing system. It states that, the agents are acting concurrently. They may either communicate, or work independently. The agents are synchronized in this type of activity, which means, at each global time step, each agent must perform an action.

In the following primitive example, there are two ants, *ant1, ant2*, working in parallel. It is not clear whether they are communicating or not.

$$ant1 \times ant2$$

If the concurrently acting agents have more than one weighted actions to be performed, the following set of rules are applied on weights. Assume that $k > k'$.

$$nw^k + mw^{k'} = mw^{k'} + nw^k = nw^k \qquad \text{(Eq.4.6)}$$

$$nw^k + mw^k = mw^k + nw^k = (n+m)\,w^k \qquad \text{(Eq.4.7)}$$

$$nw^k * mw^{k'} = mw^{k'} * nw^k = (nm)w^{k+k'} \qquad \text{(Eq.4.8)}$$

4.3.5 Restriction on the Actions to be Performed (<)

Applying the restriction operator to an agent expression, only the actions appearing at right hand side of the operator are permitted.

$$EX3 \overset{def}{\equiv} 6w : a.A + 4w : b.B + 2 : \#.C + 3 : \#.D < \{\#\} \qquad \text{(Ex.4.3)}$$

In Ex.4.3, the agent $EX3$ has three choices in its actions, $\{a, b, \#, \#\}$, and it is restricted to perform only the unit action, $\#$. There are two options:

$2:\#.C$ or $3:\#.D$

The probability of actions with equal weights is determined by the relative frequencies. In this example, although the actions to be performed are same, #, the resulting agent is different.

Suppose that E1 performs an action a and becomes F1 (Ex.4.4), and concurrently E2 performs the complementary action !a, and becomes F2 (Ex.4.5). They are working in parallel, and limited to perform jointly only the action a (Ex.4.6), which is not possible. Because, E1 performs a, and at the same time E2 performs !a by the definition of parallel composition. And the resulting action becomes again #, which is not allowed by the restriction operator. Then it is said that this $E1 \times E2$ agent deadlocks under the restriction of environment level action being a, that means it is equivalent to agent 0.

$$E_1 \overset{def}{\equiv} a.F_1 \tag{Ex.4.4}$$

$$E_2 \overset{def}{\equiv} !a.F_2 \tag{Ex.4.5}$$

$$E_1 \times E_2 < \{a\} \equiv a.F_1 \times !a.F_2 < \{a\} \tag{Ex.4.6}$$

4.3.6 Communication

There are two basic requirements to achieve communication between two agents: (i) parallel composition of agents, (ii) observing unit action by the environment.

The agents willing to communicate must work in parallel. Thus, parallel composition forms the basis of communication.

Another issue in describing communication by WSCCS is the notion of complementary actions. The *complementary actions* are nothing but the two actions such that being performed by some agents concurrently, the resulting action that may be observed from the environment is the unit action, or identity action, denoted by #.

Communication is such a behavior that it cannot be observed by the environment, meaning that either both agents performed unit actions, or they communicated through performing complementary actions.

Restriction is an indirect requisite in communication, which is essential when the agents, who are planned to communicate, have more than one action to be performed. Restriction is used to force the agents to communicate by disallowing the occurrence of the actions which are not complementing each other. We can prevent the agents to communicate using restriction operator as well.

With the agents defined in Ex.4.4, and Ex.4.5, assume that, they are working in parallel, with no restriction at all (Ex.4.7). It is expected these two agents to communicate.

$$E_1 \overset{def}{\equiv} a.F_1 \qquad \text{(Ex.4.4)}$$

$$E_2 \overset{def}{\equiv} !a.F_2 \qquad \text{(Ex.4.5)}$$

$$E_1 \times E_2 \equiv a.F_1 \times !a.F_2 \qquad \text{(Ex.4.7)}$$

By the expression (Ex.4.7) $E1 \times E2$ performs #, since $a \times !a = \#$. There is no need to force the communication. However, in the following example, where $E_2^{'}$ has more actions to be performed, if the communication is planned, restriction operator should be employed.

$$E_2^{'} \overset{def}{\equiv} !a.F_2 + 2w : c.F_3 \qquad \text{(Ex.4.8)}$$

$$E_1 \times E_2^{'} \equiv a.F_1 \times \left(!a.F_2 + 2w : c.F_3 \right) \qquad \text{(Ex.4.9)}$$

Ex.4.9 defines the parallel acting E_1 and $E_2^{'}$ agents. There are two possible directions in action performing such that:

$$
\begin{aligned}
E_1 \times E_2^{'} &\equiv a.F_1 \times (!a.F_2 + 2w : c.F_3) \\
&\equiv \underbrace{a.F_1 \times !a.F_2}_{1} + \underbrace{a.F_1 \times 2w : c.F_3}_{2}
\end{aligned}
$$

If Part 1 is chosen, communication occurs, where Part 2 involves only parallel action performing behavior. Assume that, we want E_1 and $E_2^{'}$ agents to communicate always. Then, the $E_2^{'}$ agent must be prohibited to perform action c, in other words, the environment level action must be forced to unit action, $\#$, as following:

$$
\begin{aligned}
E_1 \times E_2^{'} < \{\#\} &\equiv a.F_1 \times (!a.F_2 + 2w : c.F_3) < \{\#\} \\
&\equiv a.F_1 \times !a.F_2 + a.F_1 \times 2w : c.F_3 < \{\#\} \\
&\equiv \underbrace{\#.(F_1 \times F_2)}_{1} + \underbrace{2w : (a \times c).(F_1 \times F_3)}_{2} < \{\#\}
\end{aligned}
$$

There is no way for Part 2 to happen because of the restriction.

4.3.7 Prioritization of Action(s) in an Agent Expression

This operator, `prio(E)`, is used to extract the prioritized parts of the agent expression E. Prioritization disables considering relative frequencies, and in turn probabilistic choice. Consider the example (Ex.4.10):

$$
EX \overset{def}{\equiv} 6w : a.A + 4w : b.B + 2 : \#.C \qquad \text{(Ex.4.10)}
$$

If we apply the prioritization operator on the agent EX, the following result is obtained, such that the actions those do not have priority are ignored.

$$
\begin{aligned}
prio\,(EX) &\equiv prio\,(6w : a.A + 4w : b.B + 2 : \#.C) \\
&\equiv 6w : a.A + 4w : b.B
\end{aligned}
$$

After prioritization an agent, the actions those have the priority equal to the highest priority may occur.

4.3.8 Renaming Actions in an Agent Expression ($f_E\left(Act_E\right)$)

The actions in the agent expression E that is pointed by the function, $f_E\left(Act_E\right)$, are renamed. The renaming function does not affect the unit action, #, and applying the function on a complementary action produces the same result as applying the function on the regular action and taking its complement. That is:

$$f_E : Act \to Act$$
$$f\left(\#\right) = \#$$
$$\forall a \in Act, f\left(!a\right) = !f\left(a\right)$$

Where Act denotes the set of actions of the agent.

Assume that we have two agents, EX , and $EX3$ which are previously defined. The following is an example of renaming the actions in agent EX .

$$EX \overset{def}{\equiv} 6w : a.A + 4w : b.B + 2 : \#.C$$
$$EX3 \overset{def}{\equiv} 6w : a.A + 4w : b.B + 2 : \#.C + 3 : \#.D < \{\#\}$$

$$f_{EX}\left(a\right) = new\_a$$
$$f_{EX}\left(b\right) = new\_b$$

Renaming function does not affect the other agents' actions even if they have common action(s). So, the agents become:

$$EX \overset{def}{\equiv} 6w : new\_a.A + 4w : new\_b.B + 2 : \#.C$$
$$EX3 \overset{def}{\equiv} 6w : a.A + 4w : b.B + 2 : \#.C + 3 : \#.D < \{\#\}$$

4.3.9 Fixing an Agent (fix)

This operation is used to avoid recursive agent definitions by choosing any component from a family of agent definitions depicted inside the operator.

$$fix_j\left(\{X_i = E_i : i \in I\}\right)$$

See below example:

$$A \overset{def}{\equiv} p:\#.P+(1-p):\#.A \tag{Ex.4.11}$$

$$P \overset{def}{\equiv} 1:\#.P \tag{Ex.4.12}$$

This definition does not satisfy the syntactical rules of WSCCS in two ways. First, the weights are not integers. This problem can be overcome by multiplying both weights by a constant c, which must exist as long as p is a rational number. The second problem is, both agents are defined recursively, meaning that the agent labels A, and P appear on both sides of the definitions. To tackle with this difficulty, the fixing operator is used.

A family of recursively defined expressions, F, is defined, then the agents A and P are set to the elements of this family.

$$A_1 \overset{def}{\equiv} c * p : \#.P + c * (1 - p) : \#.A_1$$
$$A_2 \overset{def}{\equiv} 1 : \#.A_2$$
$$F = \{A_1, A_2\}$$
$$A = fix_1(F)$$
$$P = fix_2(F)$$

Figure 4.1 Non-recursive definitions of agents A, and P.

Although the fixing operation is essential for describing agents those are convenient to semantics, this non-recursive forms are not widely used while dealing with the applications of WSCCS (Sumpter, 2000).

4.4 WSCCS Semantics

The semantics of WSCCS, as well as any formal language, define how to decide the validness of any given series of actions. For each syntactically correct agent expression, proof trees are generated according to the semantic rules formally given in Appendix A.2.

In the following example, there exist two types of agents, A and B performing different actions with different weight and/or priorities. And the property to be checked is whether or not these two agents performing some actions while working concurrently. In other words, to explore the validness of $A \times B$ agent performing the given action.

Ex.4.13 and Ex.4.14 define the two agents A and B.

$$A \overset{def}{\equiv} p : a.C + q : !b.D \tag{Ex.4.13}$$

$$B \overset{def}{\equiv} x : b.T + y : !a.R \tag{Ex.4.14}$$

Figure 5.2 demonstrates assuring the syntactical correctness of given agent definition A. Proving that the agent definitions are correct, the question is to show that $A \times B \overset{\#}{\rightarrow} C \times R$ is a valid transition.

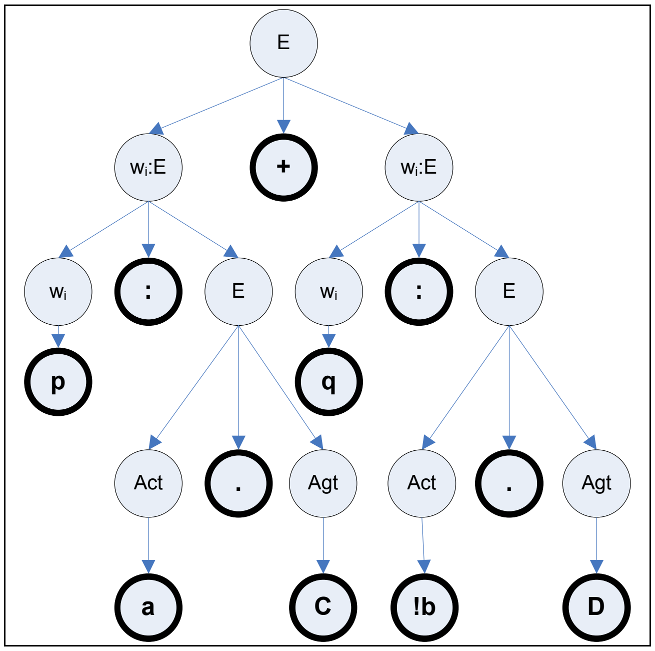


Figure 4.2 The derivation tree of agent A. The definition of B is checked in the same way.

Below, all possible transitions of $A \times B$ are listed, and only one of them would be occurred. (Multiplication of actions, that is concurrently occurring actions, is denoted usually by juxtaposition, which is omitted in examples. Also the multiplication symbol which exists between weights is omitted.)

$$\begin{aligned} A \times B &\equiv (p : a.C + q : !\,b.D) \times (x : b.T + y : !\,a.R) \\ &\equiv (p : a.C \times x : b.T) + (p : a.C \times y : !\,a.R) + \\ &\quad\ (q : !\,b.D \times x : b.T) + (q : !\,b.D \times y : !\,a.R) \end{aligned}$$

Applying the multiplication rule for weights:

$$\begin{aligned} A \times B \equiv\ & px : (a.C \times b.T) + py : (a.C \times !\,a.R) + \\ & qx : (!\,b.D \times b.T) + qy : (!\,b.D \times !\,a.R) \end{aligned}$$

Then applying transitional semantics, the proof tree in Figure 4.3 is obtained for the weighted choice of $py:(a.C\times !a.R)$. The probability of occurrence of this choice is:

$py/(px+py+qx+qy)$.

$$\dfrac{\dfrac{}{a.C \xrightarrow{a} C}\text{ by rule A1} \wedge \dfrac{}{!\,a.R \xrightarrow{!\,a} R}\text{ by rule A1}}{a.C \times !\,a.R \xrightarrow{\#} C \times R}\text{ by rule A2}$$

Figure 4.3 Proof tree for $A \times B \xrightarrow{\#} C \times R$

4.5 Congruence and Equational Theory

There are two ways two decide whether two agents are equal, or similar. First technique, called congruence involves comparing the two agents in algebraic context, using direct bisimulation, or relative bisimulation; whereas the second one employs equational theory which is based on formal techniques to make the comparison at syntactical level. Consider the example agents in Figure 4.4. It is possible to show the congruence of X and Y agents by demonstrating that both making the same transitions, then one can say that relative bisimulation exists between the two. Alternatively, using equational theory, evidence is provided for their equivalence (Figure 4.5).

$$X \overset{def}{\equiv} (w:!a.A+1:\#.R)\times(w:a.A+1:\#.P)<\{\#\}$$
$$Y \overset{def}{\equiv} \left(w^2:\#.(A\times A)+1:\#.(R\times P)\right)<\{\#\}$$
$$X \overset{r}{\sim} Y$$

Figure 4.4 X and Y agents, and relative bisimulation between them.

4.5.1 Proof by Equational Theory

$$X \overset{def}{\equiv} (w:!a.A+1:\#.R)\times(w:a.A+1:\#.P) < \{\#\}$$

$$= (w:!a.A\times w:a.A)+(w:!a.A\times 1:\#.P)+$$

$$(1:\#.R\times w:a.A)+(1:\#.R\times 1:\#.P) < \{\#\} \text{ (by Exp4)}$$

$$= w^2:(!a.A\times a.A)+w:(!a.A\times \#.P)+$$

$$w:(\#.R\times a.A)+1:(\#.R\times \#.P) < \{\#\} \quad (\text{by multiplication of weights})$$

$$= w^2:(!a\times a).(A\times A)+w:(!a\times \#).(A\times P)+$$

$$w:(\#\times a).(R\times A)+1:(\#\times \#).(R\times P) < \{\#\} \text{ (by Exp1)}$$

$$= w^2:\#.(A\times A)+w:!a.(A\times P)+$$

$$w:a.(R\times A)+1:\#.(R\times P) < \{\#\} \text{ (by multiplication of actions)}$$

$$= \underbrace{w^2:\#.(A\times A)+1:\#.(R\times P) < \{\#\}}_{Y} \text{ (by Res1)}$$

Figure 4.5 Proof for $X \overset{r}{\sim} Y$ by equational theory. The rules Exp4, Exp1, Res1 are defined in Appendix A.

CHAPTER FIVE

FORMAL VERIFICATION

5.1 Introduction

It is clear that while systems are becoming more complex, the criticality of ensuring their correct operations is also increasing. To be able to build up any system secure and reliable, one must be sure that the system to be realized is operating correctly in all senses. In context of hardware and software systems, the act of proving or disproving the correctness of an abstract model that will be realized against its formal specifications is called *verification*. Here, specification refers to some property of the system expressed in temporal logic.

There are roughly two approaches to formal verification. The first approach is model checking, which consists in a systematic and always automatic exploration of the entire mathematical model. Usually this consists into exploring all of states and transitions in the model, by using smart and domain-specific abstraction techniques to consider whole bunches of states in a single operation and reduce computing time.

The second approach is theorem proving. It consists in using a formal version of mathematical reasoning about the system. This is usually only partially automated and is driven by the user's understanding of the system to validate.

The rest of the chapter is organized as follows: The two techniques of verification, theorem proving and model checking are introduced first. Then discrete state space modeling languages and property modeling languages – temporal logics- are expounded as the required formalisms in model checking.

5.2 Verification by Theorem Proving

Theorem proving is a technique where both the system and its desired properties expressed as mathematical logic formulas. The logic is given by a formal system, which defines a set of axioms and a set of inference rules. Then, theorem proving is the process of finding the proof of property from the axioms of the system by applying inference rules. Steps in the proof appeal to the axioms and rules, and possibly derived definitions and intermediate lemmas. Although the proof could be constructed by hand, since systems becoming large and complex, it is not reasonable. Hence, there has been a great effort on developing theorem provers with various degree of automation, and also proof checkers have been proposed to confirm the correctness of the proof. Another way is the interactive theorem prover. By definition, it requires human assistance which causes the process to be slow, and error-prone.

The initial attempts on obtaining automated theorem provers appeared in 60s. Gilmore (1960), Wang (1960a), and Prawitz, Prawitz, & Voghera (1960) presented theorem provers for the full first-order predicate calculus. Although this mechanization constituted an important proof of concept, the practical utility of the theorem provers was limited. Robinson (1965) introduced the principle of *resolution*, and it was a central point for the development of automated theorem provers till the end of 70s. Despite this great effort, all the early theorem provers suffered from the same limitation that had plagued the previous generation of mechanical proof procedures: the combinatorial explosion of the proof search space.

The 70s also witnessed the appearance of logic programming, originally attributed to (Colmerauer & Roussel, 1992; Kowalski , 1988). Colmerauer and colleagues implemented a specialized resolution based theorem prover called *PROLOG* (stands for the French “PROgrammation en LOGique”) which implemented Kowalski’s procedural interpretation of Horn clause logic[1]. The result was not only an automated theorem prover, but a programming language, which is still widely used.

Otter (Wos, Overbeek & Boyle, 1992), SETHEO (Letz, Schumann & Bayerl, 1992), and PTTP (Stickel, 1986) are the state-of-the art automated theorem provers developed during 70s and early 80s, which are known to be able to solve many benchmark problems with extremely high inference rates, and efficient use of storage.

Resolution based methods generate proofs that are not readily understood by human users. This disadvantage, as well as the difficulties in combining the resolution with other nonlogical inference techniques such as induction, led to studying other approaches, such as allowing human interaction, an inserting some heuristic algorithms.

Robert Boyer and Strother Moore have introduced a series of theorem provers that use heuristics to develop proofs by induction and rewriting. The NQTHM series of provers (Boyer & Moore 1979, 1988) and its successor ACL2 (Kaufmann & Moore, 1994, 1996) are highly automated, but still requiring user guidance to accomplish difficult proofs. Though, the Boyer-Moore prover has been used to prove program and hardware correctness (Bevier, Hunt, Moore & Young,1989; Hunt, 1987), as well as mathematical theorems including Gödel's incompleteness theorem, which has appeared in PhD. dissertation of Shankar (Shankar, 1994). Shankar has also proposed another theorem prover, PVS (Shankar, 1993; Shankar, Owre, & Rushby, 1993). Shankar (1994) claimed that there is no sharp distinction between theorem provers and proof checkers but instead, it is defined by the intended use of the system, or the degree of automation.

[1] A clause is said to be Horn clause if it has at most one positive literal. For example, $\neg P(x) \wedge \neg Q(x) \vee R(x)$.

There are some certain systems known to be “proof checker”. Automath (van Benthem Jutting, 1979), and LCF that refers to Logic for Computable Functions (Gordon, Milner, & Wadsworth, 1979) are the two proof checker in the early years of the field. LCF has been used to verify program properties and to check the correctness of unification algorithm. Several well-known algorithms / applications have evolved from LCF, including HOL (Gordon & Melham, 1993), Nuprl (Constable, 1986), and Isabelle (Paulson, 1988). HOL (Higher order logic) is a widely used system with extensive libraries that is employed primarily for verification of hardware and real time systems. Nuprl is based constructive type theory, and has been primarily used as research and teaching tool in the areas of constructive mathematics, hardware verification, software engineering. Isabelle is a generic, interactive theorem prover based on the typed lambda calculus, supports proof in any logic whose inference rules can be expressed as Horn clauses. Isabelle also represents a synthesis between two largely distinctions in area of automated reasoning: resolution based theorem proving and interactive theorem proving. Coq is another proof checker tool proposed in (Dowek, Felty, Herbelin, & Huet, 1993) which is useful for formalizing and verifying hard problems in mathematics and program verification.

5.3 Verification by Model Checking

Model checking is a widely used verification technique that is model-based, automated, using a property verification approach, mainly useful for verifying concurrent and reactive systems, typically in a post development stage. The overall behavior of a distributed system is modeled as a transition system, whose states represent the global states of the distributed system, and whose transition relation gives the possible evolutions of the system. It can be checked whether such a transition system is a model of a temporal logic formula.

In contrast to the theorem proving model checking is completely automatic and fast. Model checking is used to check partial specifications; therefore it can provide useful information about a system’s behavior without checking the whole system. Above all, model checking outputs counterexample if the given property doesn’t hold by the model, which usually represents the subtle errors in design, and thus can be used to help in debugging.

5.3.1 History of Model Checking

Two general approaches in model checking are used in practice today. The first, *temporal model checking*, is a technique proposed by (Clarke & Emerson, 1981), and by (Queille & Sifakis, 1982). In this approach, specifications expressed as temporal logic formulas, and systems are modeled as finite state transition systems. Then with an efficient search procedure, the problem is to check whether the given finite state transition meets the given model of specification.

In the second approach, *automata based model checking*, both the specification and the system are given as automata and a comparison is performed to evaluate whether or not the behavior of the system conforms to that of specification. Different notions of conformance have been explores, including language inclusion (Har'el & Kurshan, 1990; Kurshan, 1994), refinement orderings (Cleaveland, Parrow & Stefen, 1993; Roscoe, 1994), and observational equivalence (Cleaveland, Parrow & Stefen, 1993; Fernandez, Garavel, Kerbrat, Mateescu, Mounier & Sighireanu, 1996). Vardi & Wolper (1986) demonstrated how the temporal logic based model checking could be combined with automata based model checking.

The technique suffers from state explosion; so, until 90s model checking was only used for finite state models. Hence, some infinite state spaces can have finite representations, which can be used for different verification methods. For example, Burkart & Steffen, 1994 presented an algorithm for the model checking of alternation-free mu-calculus (Bradfield & Stirling, 2001), which is a modal logic, on pushdown processes. The published methods study restricted process algebras such that the infinite transition graphs are pushdown-automata graphs.

Model checking is an appealing direction in verification. The Model Checking Group at Carnegie Mellon University (CMU) has developed a number of model checkers for several systems, such as CV, which is a model checker for VHDL;

CSML and MCB, a language for supporting compositional description of the finite state machines, and model checking of CTL; SMV a symbolic model checker for CTL which employed Binary Decision Diagrams (BDD) first; and a tool which combines model checking and theorem proving features, SyMP. Edmund M. Clarke, who is first suggested temporal logic based model checking, and currently a member of Model Checking Group at CMU have had enormous effort on model checking and proposed model checking algorithms for a wide range of problems including finite state concurrent systems (Clarke & Grumberg, 1987), sequential circuits (Clarke, Burch, Long, McMillan,& Dill, 1994), models with large states (Burch, Clarke, McMillan, Dill, & Hwang, 1990), protocol verification (Clarke & Kurshan, 1996).

There are some other well-known and successful model checking tools. The SPIN (Gerth, Peled, Vardi, & Wolper,1995; Holzmann, 1991) uses partial order reduction to overcome state space explosion, and a linear time temporal logic for property specification. SPIN is mostly used in modeling communication protocols. The Concurrency Workbench-CW- (Cleaveland, Parrow & Stefen, 1993) verifies CCS processes for properties expressed in μ-calculus. Another tool The Concurrency Workbench of the New Century (CWB-NC), (Cleaveland & Sims, 1996) provides users with a number of different languages to specify and to verify the finite state concurrent systems. CWB-NC uses CTL as property specification language. And Java Pathfinder, which is a model checker for Java bytecode programs.

Model checking is now powerful enough that is becoming widely used in industry. As an example, in 1995 civil engineers at North Caroline State University used the CWB-NC to analyze the timing properties of distributed active structure control system. The system under study was designed to make buildings more resistant to earthquakes. The system was first modeled by CCS language, which consisted of approximately $2.12*10^{19}$ states and therefore could not be analyzed by hand, or even by semiautomatic theorem provers. They have used the features of CWB-NC constructing a much smaller system automatically so that the timing properties could be analyzed.

Other successful industrial sized case studies in model checking are too long to list. The proof for model checking is the future of verification is that industry is building their own model checkers, or by simply using the existing ones.

5.3.2 Model Checking Approaches

Model checking is a highly automated verification technique. Informally, a model checker is a procedure that decides whether a given model M satisfies a given logic formula ϕ, abbreviated $M \models \phi$. M is an abstract model of the system in question, usually a discrete state space model, in our work, and ϕ, typically drawn from a temporal or modal logic specifies a desirable property. The model checker then attempts to figure out whether the model enjoys the given property.

There are two main directions in which model checking problem can be specified: (i) Global Model Checking, (ii) Local Model Checking.

Global Model Checking: Assumes that the global state space of the transition system is already constructed, and considering every state of the transition system attempts to decide whether each state meets the specification. This leads to state explosion problem. Indeed for most systems of practical interest, to construct the complete global state space is not reasonable, since it is too large.

Local Model Checking: Considers whether a certain state in state space meets the given property. In contrast to global model checking, local model checking must determine the modelhood of a single state.

Obviously, the solution of global model checking problem comprises of local model checking of all states, thus the two problems are closely related although have different application areas. Verifying properties of hardware systems is a classical application of model checking, which is usually considered as local model checking problem because of the state space growth. Therefore, another means of local model checking is fighting the state explosion problem.

Global model checking, on the other hand, is more useful for problems as data-flow analysis. Such applications use structures those are rather small in comparison to those arising in verification activities, and thus the state explosion problem holds less importance.

Model checking can be implemented by several different approaches, some well-known examples are temporal model checking, automata theoretic model checking, and tableu based model checking. Temporal model checking and automata based model checking was briefly defined in Section 5.3.1.

Tableu based methods are composite methods and, they attempt to solve local model checking problem by subgoaling. Main idea behind is, if a proof tree can be constructed that witnesses that the given state has the property. Otherwise, if no proof tree is found, then it is disproof of the given state does not hold the property. This methods deal with only a small fraction of state space, and therefore a good way to fight with state explosion.

Figure 5.1 demonstrates the approaches along the axes of temporal logic involved, and the type of problem that is suitable, i.e. global or local.

	Temporal Logic		Problem Type	
	Branching	Linear	Global	Local
Temporal model checking	X		X	
Automata theoretic model checking		X	X	X
Tableu based model checking	X	X		X

Figure 5.1 Classification of Model Checking Approaches (Müller-Olm, Schmidt & Steffen, 1999).

The model checking technique to be employed strongly depends on the problem to be checked. Also, the properties to be questioned are important to form the problem as either local or global.

5.4 Discrete State Space Modeling

Formal reasoning about systems having discrete state spaces involves two steps: (i) Building a formal model of this state changing system which describes, in particular, functioning of the system, or some abstraction thereof, and (ii) Using a kind of logic to specify and verify properties of the system.

Usually, a discrete state space system is characterized by the following properties:

- At each particular time moment, the system is in a particular *state*;
- This state can be characterized by the values of some variables called the *state variables*.
- There is a notion of an action, and actions may result in a state change of the system, that is a change of the value of some variables. These actions are usually called as *transitions*.

When building a formal model of systems with discrete state spaces, one can use different levels of abstractions of the real system for the above properties depending on the level of details interested in. In the following subsections we are going to introduce three ways of modeling discrete state spaces all of which are connected to the constructing graphs representing the state space of the transition system being investigated: (i) Labeled transition systems, where the edges -transitions- of the graph are labeled with single actions; (ii) Kripke structures, where the nodes – states- are labeled with a set of atomic propositions, (iii) and a combination of the two techniques Labeled Kripke Transition Systems, where both nodes and edges are decorated with labels.

5.4.1 Labeled Transition Systems

We first introduce the ingredients in the model of labeled transition system (LTS) informally, and then provide its formal definition.

In the model of labeled transition systems, agents are represented by nodes of graphs and performing an action is understood as moving along an edge, labeled with action name that goes out of that state. Therefore, a labeled transition system consists of a set of states (or agents), a set of labels (or actions), and a transition relation $\rightarrow$ describing changes in agent states: if an agent p can perform an action a and become an agent p', we write $p \xrightarrow{a} p'$. Sometimes an agent is singled out as the initial agent in the labeled transition system under consideration. Formally speaking:

A labeled transition system is a triplet $M = (S, Act, \rightarrow)$, where:

- S is a finite non-empty set, called the *states* of M, ranged over by s, $p \in S$ is the initial state;
- Act is a set of *actions*, ranged over by a;
- $\rightarrow \subseteq S \times Act \times S$ is a transition relation, for every $a \in Act$. As usual, we shall use the more suggestive notation $s \xrightarrow{a} s'$ instead of $(s, a, s') \in \rightarrow$, and write $s \overset{a}{\not\rightarrow}$ (read "s refuses a") if and only if $s \xrightarrow{a} s'$ for no state s'.

The definition of labeled transition systems allow situations such as, starting from the initial state if defined, some states may never be reached. This motivates us to introduce the notion of *reachable* states. We say that a state p' in the transition system is reachable from p if and only if there exists a path from p to p'. The set of such states is called the set of *reachable* states. Before dealing with the formal definition of reachable states, it is necessary to introduce the extended transition relation.

The transition relation can be extended to the elements of Act^* (the set of all finite strings over Act including the empty string λ) The definition is as follows:

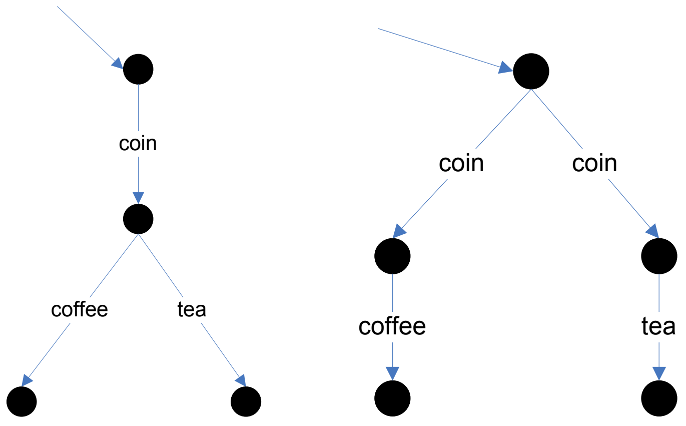
- $s \xrightarrow{\lambda} s, \forall s \in S$,
- $s \xrightarrow{aw} s'$ if and only if there is $t \in S$ such that $s \xrightarrow{a} t$ and $t \xrightarrow{w} s'$, for every $s, s' \in S, a \in Act$, and $w \in Act^*$. Note that this inductive definition states that, extended transition relation is a reflexive and transitive closure of its original version.

In other words, if $w = a_1 a_2 ... a_n$ for $a_1, a_2, ..., a_n \in Act$ then we write $s \xrightarrow{w} s'$ whenever there exist states $s_1, ..., s_{n-1} \in S$ such that:

$$s \xrightarrow{a_1} s_1 \xrightarrow{a_2} s_2 \xrightarrow{a_3} s_3 \xrightarrow{a_4} ... \xrightarrow{a_{n-1}} s_{n-1} \xrightarrow{a_n} s'.$$

Then, reachable states of a transition system can be defined as follows:

Let $T = (S, Act, \rightarrow)$ be a labeled transition system, and let $p \in S$ be its initial state. It is said that $p' \in S$ is reachable in transition system T if and only if $p \xrightarrow{w} p'$. The set of reachable states contains all states reachable in T.



(a) (b)

Figure 5.2 Labeled transition systems for two different coffee / tea vending machines.

Figure 5.2 presents labeled transition system for two different vending machines offering tea and coffee. Both machines serve tea or coffee after inserting a coin, but from the customer's point of view, the machine in (a) should be avoided since it decides internally serving whether tea or coffee. In contrast the machine in (b) leaves the decision to the customer.

5.4.2 Kripke Structures

In Kripke Structures, states are described by the values of atomic propositions only (Kripke, 1963). The definition of Kripke structures is as follows:

Let V be a set of propositional variables. A *Kripke Structure* over V is a tuple $M = (S, I, \rightarrow, L)$, where:

- S is a finite non-empty set, called the *states* of M,
- $I \subseteq S$ is a finite non-empty set of states, called the set of *initial values* of M,
- $\rightarrow \subseteq S \times S$ is a set of pairs of states, called the *transition relation* of M,
- $L : S \rightarrow 2^V$ is a function, called *labeling function* of M.

Any Kripke structure can be considered as a directed graph whose nodes are the states and labeled by sets of propositional variables which are true in current states (basic local properties), labeling function assigns the propositional variables of each state. The edges of the graph are the elements of transition relation: there is an edge from s to s' if and only if $(s, s') \in \rightarrow$, or in more common notation $s \rightarrow s'$, where both $s \in S$ and $s' \in S$. Such a graph is called the *state transition graph* of the Kripke Structure.

Figure 5.3 demonstrates an example Kripke structure whose propositions take the form of $variable = number$; the structure represents the states that arise while the program's components , x and y , trade two resources back and forth.

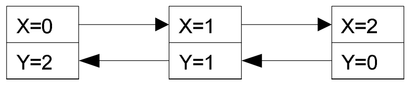


Figure 5.3 An example Kripke Structure. Note that the transitions are unspecified.

5.4.3 Labeled Kripke Transition Systems

Labeled Kripke Transition Systems (LKTS) takes its name since it is a combination of Kripke structures and Labeled Transition Systems. Therefore, both the nodes and edges are annotated with agents' and actions' labels respectively.

Let V be a set of propositional variables. A LKTS over V is a tuple $M = (S, Act, \rightarrow, L)$, where:

- S is a finite non-empty set, called the *states* of M,
- Act is a set of *actions*, ranged over by a;
- $\rightarrow \subseteq S \times Act \times S$ is the transition relation,
- $L : S \rightarrow 2^V$ is a function, called *labeling function* of M.

By this definition, LKTS generalizes both Labeled Transition Systems and Kripke Structures: A Kripke Structure is a LKTS with an empty set of actions, Act, and a Labeled Transition System is a LKTS with a trivial interpretation.

Figure 5.4 represents the LKTS graph for the below code:

```
z:=0, i:=0 //assignments
while not_equal(i,y) // propositional variable
    z := z+x
    i := i+1
end.
```

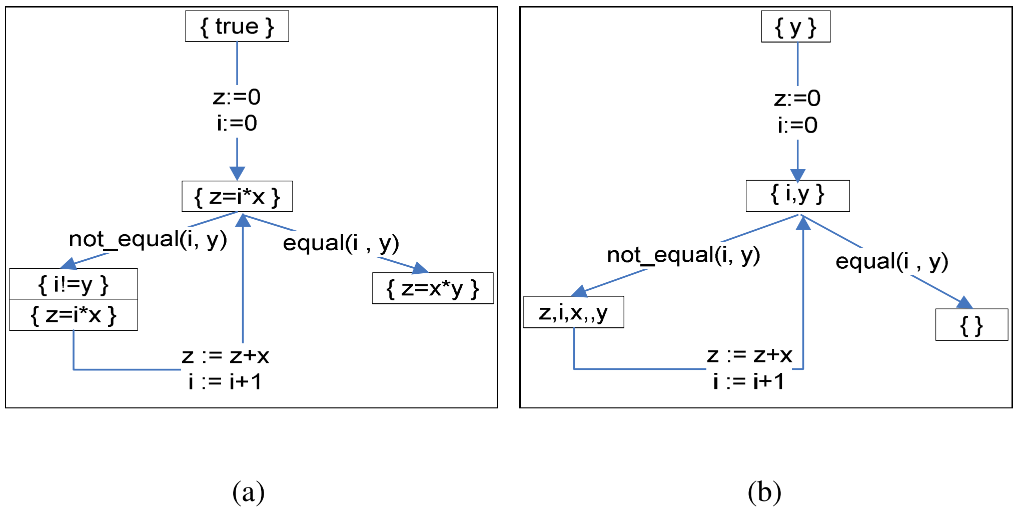


Figure 5.4 Two examples of labeled Kripke transition system graphs.

Both Figure 5.4(a) and Figure 5.4(b) label the edges with the program phrases. The LKTS in Figure 5.4 (a) uses properties that are logical propositions of the form $variable := expression;$. The system in Figure 5.4 (b) uses the variables of the program as the state variables.

5.5 Temporal Logics

In logic, the term *temporal logic,* which is sometimes also refer to *tense logic*, is used to describe any system of rules and symbolism for representing, and reasoning about, propositions qualified in terms of time. In a system of temporal logic, various temporal operators, or so called “modalities” are provided to describe and reason about how the truth values of assertions vary over time. Typical temporal operators include *sometimes* P which is true now if it becomes true in future, *always* Q which is true now if Q is always true in future.

Although it was first studied by Aristotle, temporal logic introduced by Arthur Prior (Prior 1957, 1967, 1969), and is thoroughly described in (Rescher & Urquhart, 1971). Temporal logic has been proposed as applying both to specification and verification of program behavior, and to the specification of system behavior. The first application for using temporal logic to describe program behavior came from Pnueli (Pnueli, 1977, 1981, 1985a). Lamport, 1983; 1986 published research on specifying concurrent systems with temporal logic. Clarke is a well known scientist in the area and has many publications on applying temporal logic on finite state concurrent documents (Bradfield & Stirling, 2001; Clarke, Emerson, & Sistla, 1986; Clarke & Grumberg , 1987).

Temporal logic can be classified along several axes: propositional versus first order, global versus compositional, points versus interval, past versus future, branching versus linear (Emerson, 1990). In the following, the linear time and branching time temporal logic will be compared briefly.

5.5.1 Linear Time Temporal Logic versus Branching Time Logics

Concerning a concurrent system by means of temporal logic, there are two ways regarding the time. One is the classical view of time, linear time. At each time step, there is one possible future moment. The other possibility is that time has branching, tree-like nature. At each moment, time may split into alternate branches representing different futures.

The modalities of temporal logic reflect the properties of time assumed in semantics. Therefore, the modalities in linear time temporal logic used to describe the events along a single time line. In contrast, the branching time temporal logic modalities reflect the branching nature of time by allowing quantification over possible futures.

In general, linear time temporal logic is used to check whether a given system satisfies a given property such that starting from a given state, if all paths satisfy the property. For example, in a labeled transition system, two states that generate the same language satisfy the *same* linear time properties. On the other hand, branching time logics describe properties depending on the branching structure of the model at hand. Two states generating the same language by using different branching structures can often be distinguished by a branching time formulae. Both approaches have been applied to program reasoning, and it is a matter of property to be queried as to decide whether branching or linear time temporal logic is to be employed

(Emerson & Halpern, 1986; Lamport, 1980; Pnueli, 1985b). Due to the greater selectivity, branching time logics are often better in analyzing concurrent systems. Linear time logics are preferred when only path properties are of interest.

Several branching time temporal logics introduced with several properties, advantageous, and disadvantageous, and for several applications. Among many variants of branching time logics, we'll exemplify Hennessy-Milner logic, and μ-calculus since they are cornerstones in the area, and computation tree logic will be introduced in detail that is best appropriate to model WSCCS state space.

Hennessy-Milner Logic (HML): It is a simple modal logic introduced by Hennessy and Milner in (Hennessy & Milner, 1985; Milner, 1989). HML is interpreted over labeled transition systems and able to express properties for a limited depth, therefore it has limited usage in area of verification, particularly in model checking. However, it forms the core of other branching time logics, which are known to be more powerful and efficient in model checking.

Modal μ-calculus: It is a very small, yet expressive branching time logic that extends HML by fixpoint operators (Kozen, 1983). μ-calculus is also interpreted on labeled transition systems. The fixpoint operators provided with μ-calculus are "at least" and "greatest" operators. It is a logic, although much studied and also widely used in model checking applications (Andersen, Stirling, & Winskel, 1994; Bradfield & Stirling, 2001; Cleaveland, Klein, & Steffen, 1992; Stirling & Walker, 1989), has a reputation for being hard and difficult to understand compared to, say, computation tree logic. μ-calculus also has relationships with some other formalisms, in particular automata, and games. Much of this work may seem to be of mathematical interest; however, it is also of practical interest since automata have been used for model checking and, recently games have also been found to have uses in tools. (Kaviola,1995; Stirling, 1995)

5.6 Computation Tree Logic, CTL

CTL was introduced in Clarke & Emerson (1981) and also known to be the first temporal logic for which an efficient model checking procedure was proposed (Clarke, Emerson, & Sistla, 1986). CTL is interpreted with respect to the states of the transition systems.

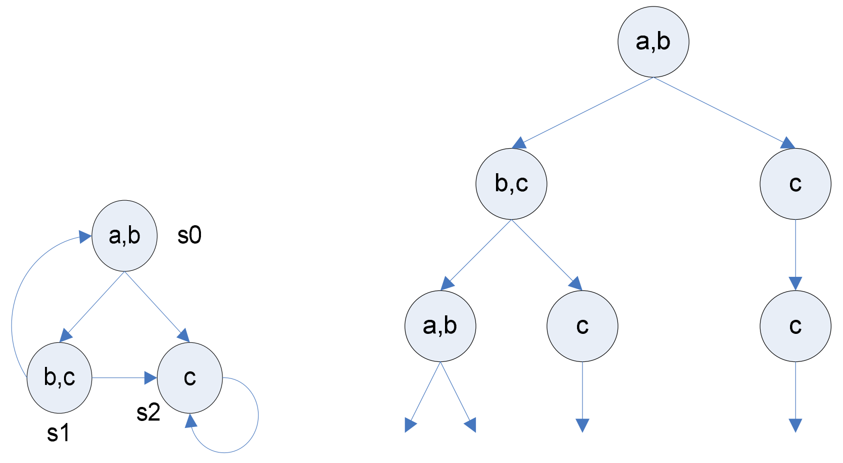
The branching time temporal logic CTL can conveniently express all interesting properties of the system under study and allows one to verify these properties in polynomial time proportional with the size of the number of the nodes in state transition system. The notion of computation tree is introduced first, then in the following the syntax and semantics of CTL is presented as follows along with several examples.

5.6.1 Computation Tree

CTL formulas describe properties of computation trees that show all possible execution paths in the transition system. A state transition graph of Kripke structure is converted to a computation tree by designating a particular state in a Kripke structure as the initial state, and unwinding the structure into an infinite tree with the designated state at the root.

More formally, let $M = (S, I, \rightarrow, L)$ be a Kripke structure model with $s \in S$. A *path* is an infinite sequence of states $(s_0, s_1, s_2, ...)$ such that $\forall i, s_i \rightarrow s_{i+1}$.

The initial state is important since the propositional variables which are true in that state define the initial conditions of given model. Therefore, the initial state is unique in any system, and cannot be changed.



(a) (b)

Figure 5.5 (a) Transition graph of Kripke structure , (b) Infinite computation tree of (a). The labels on the nodes represent propositions those are true on current node, and $s0$ is the initial state.

5.6.2 Syntax of CTL

The objects in CTL are a set of propositional variables, say V, which are the propositional variables of the state transition system under study. The operators are borrowed from propositional logic, and $\wedge$, or $\vee$, negation $\neg$, and implication $\Rightarrow$. There are two modalities regarding the paths, or so called path quantifiers, which are very similar to the quantifiers in Boolean formulas, but quantifying paths instead of formulas: (i) *for all paths* A ,and (ii) *some path* E. There is a list of operators, *temporal operators,* referring to the path under investigation. Note that path quantifiers and temporal operators are compound in CTL, which means there is never an isolated path quantifier or an isolated temporal operator. Let $\phi, \Psi \in V$. Then:

- $X\phi$: refers to the "evaluate the value of ϕ" at next state in the path, **Next,**
- $F\phi$: ϕ holds for some state on the path, **Finally,**
- $G\phi$: ϕ holds for all states on the path, **Globally,**
- $U(\phi, \Psi)$: ϕ holds on the path at least until Ψ holds. **Until**.

CTL formulas can be categorized as state formulas, and path formulas. ϕ is a CTL state formula, then CTL formulas are defined inductively as following:

- Any propositional variable $\phi \in V$ is a CTL formula.
- The constants $True$ and $False$ are CTL formulas.
- If ϕ and Ψ are state formulas then
  - $\neg\phi$,
  - $\phi \vee \Psi$, $\phi \wedge \Psi$, $\phi \Rightarrow \Psi$,

  are also state formulas, and
  - $EX(\phi)$, $AX(\phi)$,
  - $EF(\phi)$, $AF(\phi)$,
  - $EG(\phi)$, $AG(\phi)$,
  - $EU(\phi,\Psi)$, $AU(\phi,\Psi)$.

  are path formulas of CTL.

In any CTL formula, the unary operators have the highest priority, $\wedge$, and $\vee$ are bound next, the binary path modalities have the lowest priority.

5.6.3 Semantics

Let $M = (S, I, \rightarrow, L)$ be a Kripke structure model for CTL, with $s \in S$, and ϕ, Ψ are CTL formulas. The standard notation has been used to represent truth in a structure: $M, s_0 \models \phi$ means that the formula ϕ holds at state s_0 in model M. Understanding the model M, it is simply written as: $s_0 \models \phi$. Then the satisfaction relation $s_0 \models \phi$ is inductively defined by:

1. $s_0 \models True$ and $s_0 \not\models False$, $\forall s \in S$,
2. $s_0 \models p$ iff $p \in L(s_0)$,
3. $s_0 \models \neg\phi$ iff $s_0 \not\models \phi$ in other words $\neg(s_0 \models \phi)$,
4. $s_0 \models \phi \wedge \Psi$ iff $s_0 \models \phi \wedge s_0 \models \Psi$,
5. $s_0 \models \phi \vee \Psi$ iff $s_0 \models \phi \vee s_0 \models \Psi$,

6. $s_0 \models \phi \Rightarrow \Psi$ iff $s_0 \not\models \phi \lor s_0 \models \Psi$,
7. $s_0 \models AX(\phi)$ iff for all states s_1 such that $s \rightarrow s_1$, $s_1 \models \phi$
8. $s_0 \models EX(\phi)$ iff for some s_1 such that $s \rightarrow s_1$, $s_1 \models \phi$
9. $s_0 \models AG(\phi)$ iff **for all paths** $s, s_1, s_2 ...$ we have $s_i \models \phi$ for all i,
10. $s_0 \models EG(\phi)$ iff **there exists a path** $s, s_1, s_2 ...$ such that $M, s_i \models \phi$ for some i,
11. $s_0 \models AF(\phi)$ iff **for all paths** $s, s_1, s_2 ...$, and for some i, $s_i \models \phi$,
12. $s_0 \models EF(\phi)$ iff **there exists a path** $s, s_1, s_2 ...$ such that for some i, $s_i \models \phi$,
13. $s_0 \models AU(\phi, \Psi)$ iff **for all paths** $s, s_1, s_2 ...$

    $\exists i \left(i \geq 0 \land s_i \models \Psi \land \forall j \left(0 \leq j < i \Rightarrow s_j \models \phi\right)\right)$,
14. $s_0 \models EU(\phi, \Psi)$ iff **there is a path** $s, s_1, s_2 ...$

    $\exists i \left(i \geq 0 \land s_i \models \Psi \land \forall j \left(0 \leq j < i \Rightarrow s_i \models \phi\right)\right)$

The following abbreviations which are used to translate F modalities into U modalities, and G modalities into duals of F modalities, are also useful in CTL formulas:

- $AF(\phi) \equiv AU(True, \phi)$, means that ϕ holds in future along from every path originating from s_0,
- $EF(\phi) \equiv EU(True, \phi)$, means that there exists a path emanating from s_0 to a state at which ϕ holds.
- $EG(\phi) \equiv \neg AF(\neg \phi)$, means that there exist some path from s_0 such that ϕ holds at every state on the path.
- $AG(\phi) \equiv \neg EF(\neg \phi)$, means that ϕ holds every state on every path originating from s_0.

Moreover, each of the eight temporal operators can be expressed in terms of three operators EX , EG , and EU :

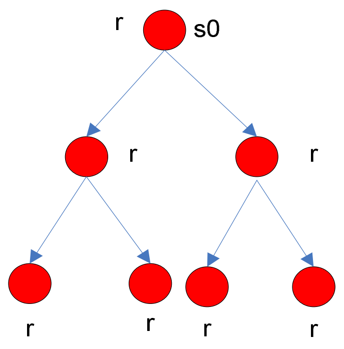
- $AX(\phi) \equiv \neg EX(\neg\phi)$
- $AG(\phi) \equiv \neg EF(\neg\phi)$
- $AF(\phi) \equiv \neg EG(\neg\phi)$
- $EF(\phi) \equiv EU(True, \phi)$
- $AU(\phi, \Psi) \equiv \neg EU(\neg\Psi, (\neg\phi \wedge \neg\Psi)) \wedge \neg EG(\neg\Psi)$

The semantics of CTL completes with the following recursive definitions of Globally, Finally, and Until temporal operators.

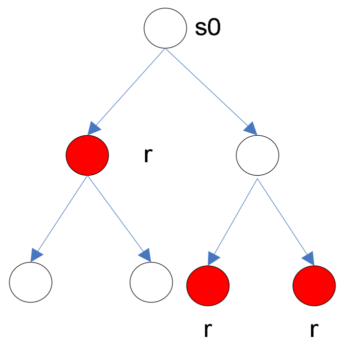
- $AG(\phi) \equiv \phi \wedge AX(AG(\phi))$, $EG(\phi) \equiv \phi \wedge EX(EG(\phi))$
- $AF(\phi) \equiv \phi \vee AX(AF(\phi))$, $EF(\phi) \equiv \phi \vee EX(EF(\phi))$
- $AU(\phi, \Psi) \equiv \Psi \vee (\phi \wedge AX(AU(\phi, \Psi)))$,
- $EU(\phi, \Psi) \equiv \Psi \vee (\phi \wedge EX(EU(\phi, \Psi)))$.

On Figure 5.6 four modalities of CTL have been demonstrated.

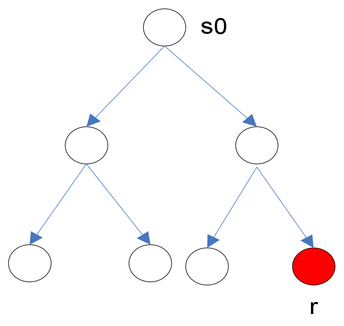
- (a) : $s \models AG(r) = true$, because, $AG(r)$ means the proposition r is true in all paths (A), and in all states of path (G).
- (b): $s0 \models AF(r)$ is true, $AF(r)$ means, the proposition r is true in all paths (A), and in some states of the path (F)
- (c): $s0 \models EF(r)$ searches some paths (E), and some state on paths (F) on which r is true. The formula evaluates true.
- (d): $s0 \models EG(r)$, to be true, there must be some path (E), on which r is true on all states (G) on the path.



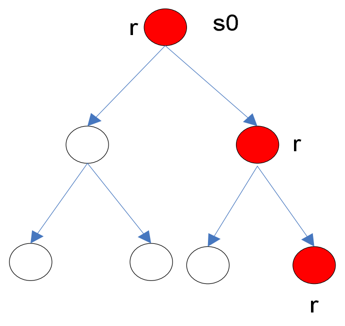
(a) $s0 \models AG(r)$



(b) $s0 \models AF(r)$



(c) $s0 \models EF(r)$



(d) $s0 \models EG(r)$

Figure 5.6 Four most widely used operators of CTL. Red nodes indicate that the proposition r is true on the node, and s_0 is the initial state.

Example

The computation tree (Figure 5.7) and some CTL formulas are presented. The CTL formulas are evaluated by the semantics rules.

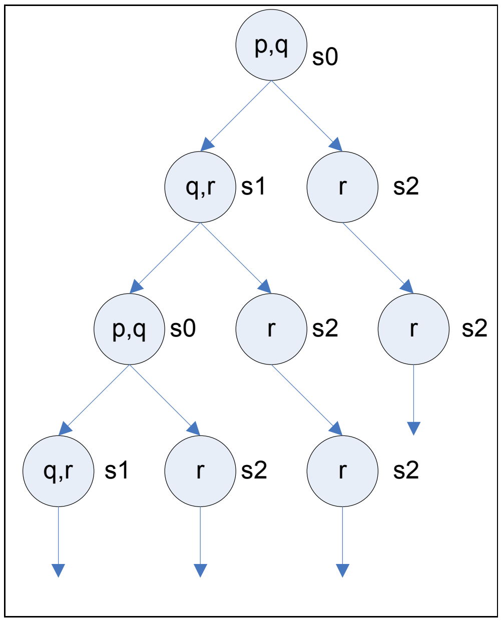


Figure 5.7 An infinite computation tree for a given model M, $s0$ is the root of the tree.

- $M, s0 \models p \wedge q$, *True*, since both p and q are propositional variables of state $s0$,
- $M, s0 \models \neg r$, *True,*
- $M, s0 \models True$, *True,*
- $M, s0 \models EX(q \wedge r)$, *True,*
- $M, s0 \models \neg AX(q \wedge r)$, *True,*
- $M, s0 \models \neg EF(p \wedge r)$ *True,*
- $M, s0 \models EG(r)$, *False,*
- $M, s2 \models EG(r)$,*True,*
- $M, s2 \models AG(r)$ *True,*
- $M, s0 \models AF(r)$,*True,*
- $M, s0 \models EU((p \wedge q), r)$, *True,*
- $M, s0 \models AU(p, r)$, *True.*

CHAPTER SIX

DESIGN OF WSCCS++

6.1 Introduction

One clear benefit to the biological modeling research community of using process algebras are the software tools available for such systems (see e.g. PEPA and Concurrency Workbench of New Century for CCS and SCCS). Such tools enable models to be specified using an appropriate algebra, performing model checking and then simulated. Additional functions such as graphical output of simulation results, and theorem proving may also be supported.

The WSCCS++, the modeling tool we have developed, makes use of some early phases of a compiler, as lexical analysis and syntax analysis, however omits some core units as preprocessing, or type-checking. Also some further phases regarding executable code generation are replaced with model checking, and a text based simulation of the results.

This chapter firstly introduces available tools different process algebras. Then following sections explains every module in architecture of WSCCS++ in level order and in details.

6.2 Available Tools for Process Algebra

Process algebras have been mentioned and exemplified previously in this text. Among many others, Calculus of Communicating Systems (CCS), Calculus of Sequential Processes (CSP), Performance Evaluation Process Algebra (PEPA), and extensions of CCS such as Synchronized Calculus of Communicating Systems (SCCS), and Weighted Synchronized Calculus of Communicating Systems (WSCCS), which is the language for which we intended to develop a tool, were introduced as being cornerstones of Process Algebraic languages.

- **The PEPA Workbench:** The PEPA Workbench is available in two editions, one for ML and the other for Java. The ML version of the PEPA Workbench transforms PEPA descriptions into a form suitable for solution by another solution tool (these include Maple, Matlab and Mathematica). The Java version of the PEPA Workbench can solve models without the need for a separate solution tool. PEPAroni is a discrete-event simulator for PEPA.
- **The Edinburgh Concurrency Workbench (CWB):** Both the tool running on Unix and source code is public and presented in (Cleaveland, Parrow, & Stefen, 1993). It offers a text-based user interface. The languages supported by CWB are CCS, and SCCS. The main goal that CWB achieved is to incorporate verification methods of equivalence checking, preorder checking, and model checking based on μ-calculus. Processes are interpreted as labeled transition graphs. CWB has been applied to verifying communication protocols of Alternating Bit Protocol, the CSMA/CD protocol, and mutual exclusion algorithms.
- **Concurrency Workbench of the New Century (CWB-NC):** Cleaveland & Sims, (1996) presented CWB-NC which is actually a successor of CWB. CWB-NC is also freely available. The latest version has been released in 2000 with the following *additional* features:
  - Runs on Win32, RedHat Linux, and Solaris 2.x environments,
  - Tcl/Tk based graphical user interface, and interactive simulation of systems,
  - Prioritized CCS, Timed CCS, CSP, and LOTOS (Bolognesi, & Brinksma, 1989) are also supported,
  - Reachability analysis,
  - Bisimulation and observational equivalence checking,
  - CTL based model checking.

  The connection phase of the UNI (Version 3.0) protocol used in ATM networks was formalized in CCS and verified with CWB-NC. The largest finite-state machine handled in the course of the analysis contained about 60,000 reachable states. The timing behavior of an active-structure control system was analyzed. The functional behavior of different variations of a railway signaling system was also analyzed. The language used to define the system borrowed constructs from several different process algebras, while the system's requirements were specified using μ-calculus and GCTL* formulas. The functional behavior of the SCSI-2 Bus Protocol was analyzed with CWB-NC as well.

- **Process Algebra Compiler (PAC):** PAC is the latest descendant of CWB-NC. (Sims, 1999). PAC runs with CWB-NC in its core, and is a tool that eases the task of changing the design language accepted by CWB-NC. That is, PAC is nothing but a front-end generator for CWB-NC.
- **The Timing and Probability Workbench (TPWB):** Fredlund (1994) presented the workbench, which was designed to deal with timed probabilistic CCS, called TPCCS. The tool was essentially providing three features: (i) simulation of TPCCS models, (ii) strong bisimilation equivalence checking to determine whether or not two TPCCS specifications are equal, (ii) model checking based on TPCTL (Timed Probabilistic CTL).

6.3 Architecture of WSCCS++

The WSCCS has been successfully applied to the several biological systems especially social insect colonies (Sumpter, 2000; Tofts, 1991). However, this approach has never been widely adopted, probably because it is not easily accessible to those unfamiliar with the techniques of computer science. This is unfortunate, since WSCCS provides natural ways to formalize much of the modeling work which is currently conducted on social insects. Therefore, developing a modeling tool for WSCCS increases the utilization of calculus even for the biological experts.

WSCCS++ is proposed to help the researchers in modeling distributed systems, particularly biological systems; to checking the syntactical rules as well as the semantic structure of the model. The tool also offers means to apply prioritization and renaming operators. This part of the tool covers information related only with the definition of the model. It gives no insight about its behavior. However, after checking the model specification against the grammar of WSCCS, the user is able to analyze how the model works.

The difficulty arose while applying and integrating the formal background introduced in Chapter 5 is that, WSCCS is a probabilistic transition based language, and none of those systems deal with probability. Thus, to each module except the lexical and syntactical analysis, some solution for probabilistic analysis has been added.

Figure 6.1 is the full architectural diagram of WSCCS++, and the rest of this chapter explains each layer of WSCCS++ in detail.

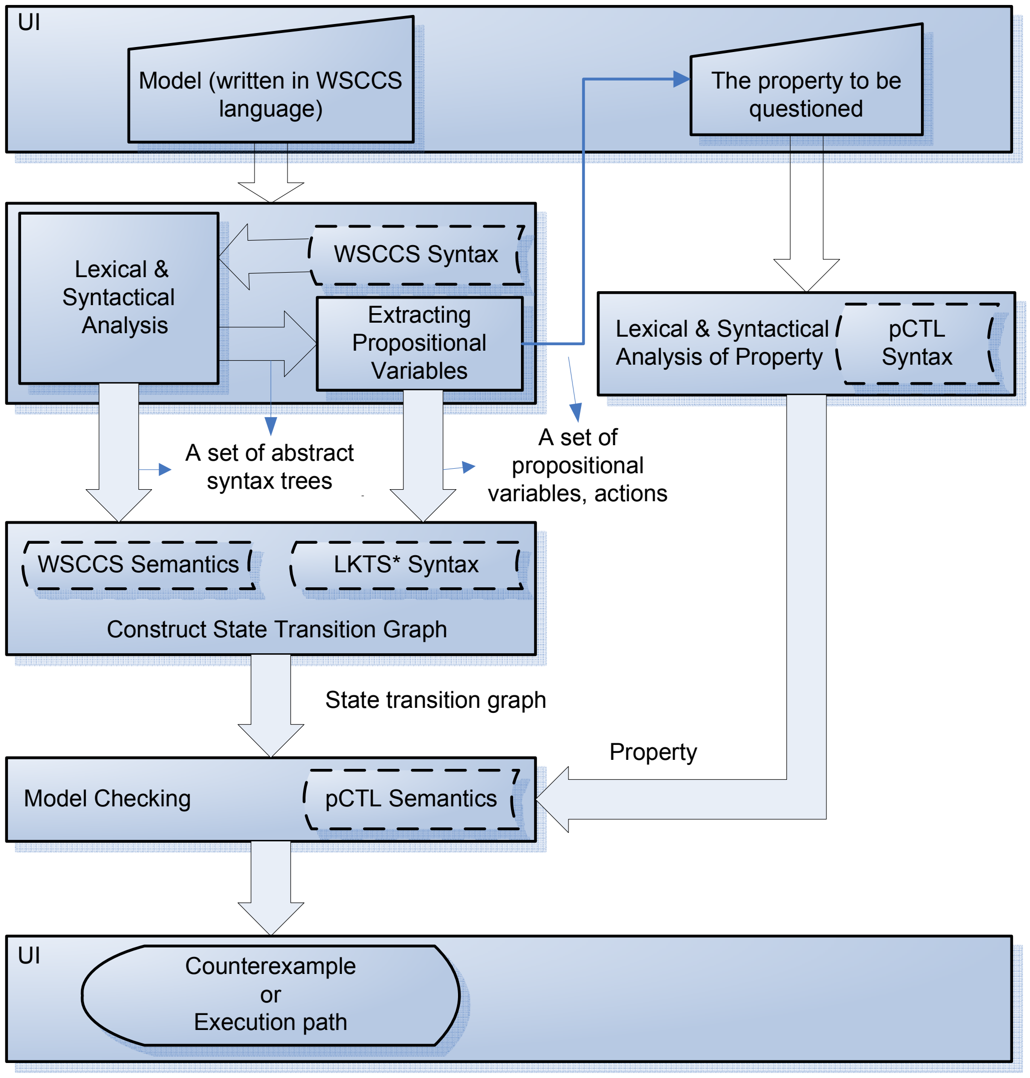

UI
Model (written in WSCCS language)
The property to be questioned
Lexical & Syntactical Analysis
WSCCS Syntax
Extracting Propositional Variables
Lexical & Syntactical Analysis of Property
pCTL Syntax
A set of abstract syntax trees
A set of propositional variables, actions
WSCCS Semantics
LKTS* Syntax
Construct State Transition Graph
State transition graph
Property
Model Checking
pCTL Semantics
UI
Counterexample or Execution path

Figure 6.1 Detailed architectural diagram for WSCCS++.

6.4 Lexical Analyzer—Lexer— for WSCCS++

The first step in any compiler like program is to separate the input text into logical pieces, i.e. strings, called *tokens*, and checking these strings against their definition supplied by the designer. Thus, the input of the Lexer is a stream of characters, and the output is a stream of tokens.

What is called a token depends on the language at hand, but in general each token is a substring of source program that is to be treated a single unit. The form of the token to be recognized by lexical analyzer is specified by regular expression. Using these regular expressions, lexical analyzer examines the source text, and generates the list of tokens as output.

6.4.1 Tokens in WSCCS++

There are two major groups of tokens. The first group is the specific strings, language keywords. The name of the token for these strings is "*keyword*". These keywords were added to the language by the designer, in order to distinguish the agent declaration of the source text.

Any model written in WSCCS Model Simulator looks like:

```
Define;
{list of agents}  as agent;
{list of actions} as action;
{list of variables} as variable;
End;
Expression;
{agent_expressions};
…
End;
```

The `Definition` block introduces several strings to be used in WSCCS model definition. The core part in source text is the `Expression` block, since this part holds agents' definitions. Note that the language is planned to be a case sensitive language, meaning that "`Define`" and "`define`" are different tokens. The following is the list of keywords appearing in the WSCCS Model Simulator,

```
Define, as, agent, action, variable, End, Expression.
```

The second token class contains the strings inside the `Expression` block, and the operators. Depending on the block the token appears, one of the following token types is assigned: `operator, digits, variable, action, and agent`. Notice that `variable, action`, and `agent` tokens appear as keywords as well. Those keywords and token names are intentionally chosen to be the same.

The strings in `Define` block are identified as `variable, action`, and `agent`. However, these strings are not placed in the output sequence. Instead, they are stored in a table to help extracting the strings in the `Expression` block. Each logical element in block is fetched from table, and gets the related token name. If it does not appear in table, it can be either an operator, or digit(s). The list of operators enabled to make use of in the source text is listed in the grammar. It is worth to mention here that, there is no token as `weight`. Because, the weight value might be given as an arithmetical expression. In this case, the token that is to be identified as weight contains more than one logical unit forcing the definition of the token. The solution is to define some variables that will appear in the weight expressions.

To accomplish the token extraction task, the form of each token type must be clearly specified. The following is the list of structures of tokens by type, presented as regular expression.

- Token: `keyword`

  `Keyword = (Define + as + agent + action + variable + End + Expression)`
- Token: `agent`

  `agent = (_ + λ) letter (letter + digit)*`
- Token: `action`

  `action = # + (! + λ + _ + ^)letter(letter + digit)*`
- Token: `variable`

  `variable = (_ + λ) letter (letter + digit)*`
- Token: `digit`

  `digit = (1+2+3+4+5+6+7+8+9)(0+1+2+3+4+5+6+7+8+9)*`
- Token: `operator`

  `operator = (~+ − + ( + ) + * + | + / + . + , + \ + < + { + } + . + :)`

6.4.2 Lexical Analysis Approach

Regular expressions are convenient for specifying lexical tokens, a formalism is required that can be implemented as a computer program. For this purpose we can use finite automata. A finite automaton has a finite set of *states*; *edges* lead from one state to another, and each edge is labeled with a *symbol*. One state is the *start* state, and certain of the states are distinguished as *final* states. If there is only one move from one state to another with a certain symbol, the automaton is called a Deterministic Finite Automaton, DFA; otherwise Nondeterministic Finite Automaton, NFA (Sipser, 1996).

For each token type, a distinct DFA is designed; which are altogether serving as a lexical analyzer. Te output of a typical lexical analyzer is a stream of tokens each of which holding the type and value of token. Figure 6.4 demonstrates the token stream of only one agent.

6.4.3 WSCCS Example

The following is a source code of a model written in WSCCS ++ Editor. The example is taken from Section 5.6.1. Figure 6.2 contains some part of the output of the Lexer.

```
Define;
Active, Passive, Colony, as agent;
p,q, as variable;
End;
Expression;
Active = (p:#.Passive) ~ (q:#.Active);
Passive = 1:#.Passive;
Colony = (Active | Active )| (Passive | Passive);
End;
```

(Model 6.1)

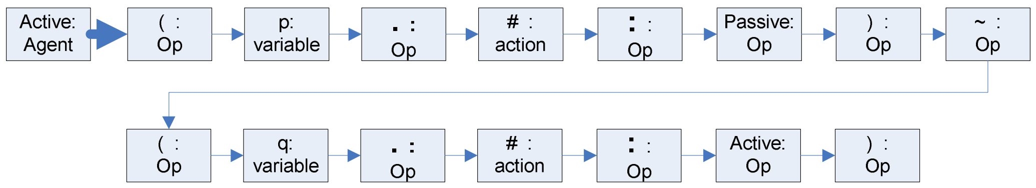


Figure 6.2 The token stream for the Active agent in Model 6.1

6.5 Syntax Analyzer—Parser for WSCCS

The syntax analyzer is usually called as *parser*. The parser receives the output of the lexical analyzer as input, and has two functions: checking its input, whether the token pattern is permitted by the grammar of language, WSCCS. It also imposes on the tokens a tree structure that is used by the subsequent phases.

Before dealing with the parsing method we applied, it is necessary to clarify what is expected as the output of the parser. It is a list of parse trees one for each agent definition. These trees have the following properties:

- Leaf nodes are labeled with terminals, i.e., agent names, action names, operators, etc. Inner nodes are marked with nonterminals, i.e., implicit or explicit agent expressions.
- The root of the tree is labeled with the starting symbol of the grammar, i.e. `E`.
- The terminals labeling the leaf nodes correspond to the input token stream, in the same order as they occur in the input.

6.5.1 Parsing Method

From the tokens, to generate the resulting structure which will be used for context handling, there are some well-known parsing techniques in the theory of formal languages. One is bottom-up parsing, and the other one is top-down parsing. Both methods involve recursive parsing of the tokens from either left to right or right to left. They differ in generating the result whether in the first step, or last step. A parser is implemented as a push-down automaton (Sipser, 1996).

Almost all programming languages, as well as most other formal languages used for describing the syntax of complex file types, can be specified using formalisms known as *context free grammars*, CFG (Sipser, 1996). Another issue related with grammars is, the need for unambiguous CFGs. Ambiguity means, given a CFG, and a string, creating two distinct derivation trees. Since the derivation tree corresponds to the abstract syntax tree that is going to be the output of the parser, ambiguous CFGs causes some sentences to be interpreted in different ways, i.e., uncertainty in precedence rules of operators, which is an undesired side effect. The formal grammar of WSCCS given in Appendix A is rearranged in such a way that no ambiguity exists.

Top-Down Parsing: Top down parsing relies on a grammar's determinism property to work. Parser begins with creating the root of the tree, which is known to be labeled with starting symbol. Then the lower nodes are constructed in pre-order, which means the top of any subtree is constructed before its children. The process continues recursively for all subtrees till the first terminal matches the leftmost child. This matching does not happen accidentally: the parser chooses the alternatives of the higher nodes precisely to obtain this match. In summary, the main task of top-down parser is to pick up the correct alternative for nonterminals.

Bottom-Up Parsing: The bottom up parsing method constructs the nodes in post order: the root of the tree constructed in the last step after all of its subtrees have been constructed.

A bottom up parser always constructs the node that is top of the first complete subtree it meets when it is scanning from left to right through the incoming token stream, where a complete subtree is a tree all of whose children have already been constructed. Tokens are considered as leaves, or from another point of view, subtrees of height 1, and the nodes representing the tokens are created as they are met. The new subtree must be a part of the complete parse tree, but an obvious problem is that the parse tree is unknown.

In summary, the main task of a bottom up parser is to predict the root of all subtrees whose children are already created repeatedly.

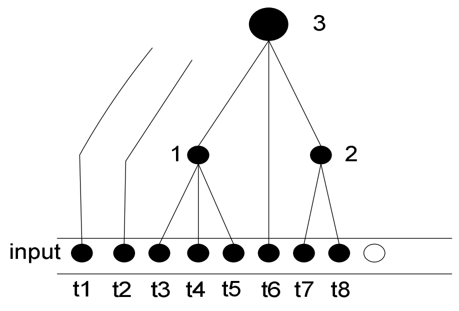


Figure 6.3 A bottom-up parser constructing its first, second, and third nodes (Grune, Bal, Jacobs, & Langendoen, 2001).

In Figure 6.3 the large dot (3) indicated the node that is created last, the black dots represents the already constructed nodes. Note that there is no unknown branch of any subtree, except the leaves t1, and t2.

6.5.2 Abstract Syntax Tree Creation within WSCCS++

A LR parser with look ahead of two has been designed for parsing the sentences, where each sentence refers to an agent, written in WSCCS grammar as designed in Appendix A.

The output of parser is a set of parse trees with general definition in 6.3. (Figure 6.4) However, as these parse trees are used in semantic analysis; some modifications have been performed in order to remove the information used in parsing, but redundant in semantic analysis. The resulting form of the tree is called as an *abstract syntax tree (AST).*

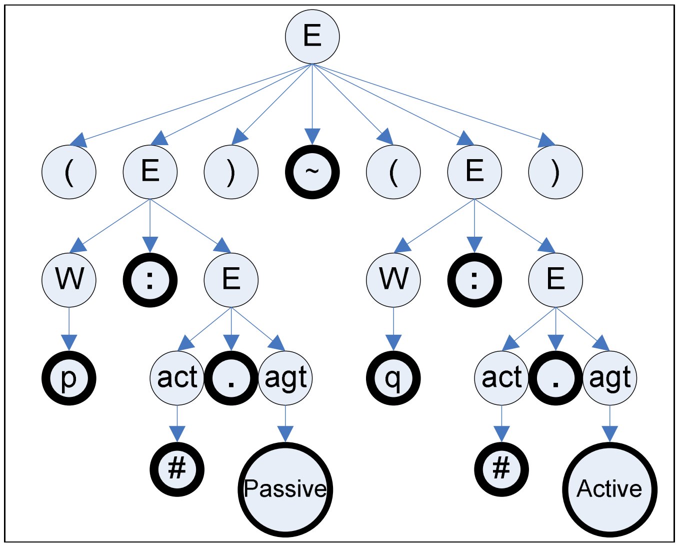


Figure 6.4 Parse tree for the Active agent in Model 6.1. The bold faced nodes correspond to the token stream, except parentheses. The inner nodes are variables of the grammar.

Figure 6.4 demonstrates an example parse tree which is the result of parsing. It is obvious that it retains all information including details that only the parser requires. After all, the crucial property of the agent Active is that, it has two weighted choices in its actions. Any other information can be considered as superfluous as semantic analysis is concerned in the further steps. On the other hand, an AST captures the syntactic structure of the expression in a much clear form. Figure 6.5 depicts the AST of the same agent as Figure 6.4.

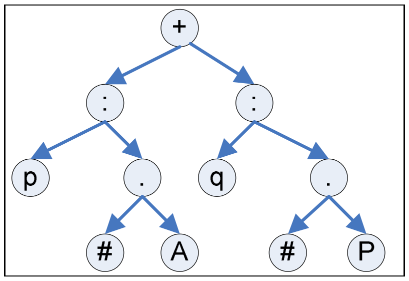


Figure 6.5 Abstract Syntax Tree of the Active agent, with A representing Active, and P representing Passive agents.

In obtaining AST, there are two options: pruning the parse tree in a way that, from root to leaves in each level, the children of a parent is examined. The child defining the semantic behavior of the parent (an operator) is moved to the upper level. The rest of the children are cut, except the ones with children. If any node has only one child (agent, action, or weight-variable), it is moved to upper level.

The second way is not to generate parse tree, but instead creating the AST while parsing. This method seems more attractive than the first one. However, it is essential to output error messages which point the user where the error is and to allow him/her to correct the mistake. Generating AST without parse tree disallows certain error checking. From this point of view, both parse trees, and abstract syntax trees are required.

6.6 Extracting Propositional Variables

The set of propositional variables is required in both creating state transition system, which appears as next step, and questioning the model, which is the main objective.

One needs to decide what the propositions should be while the available information is limited with the model itself, and the output of parser, which is a set of abstract syntax trees, one for each agent. Then, typically each agent can be treated as a propositional variable, meaning that every agent satisfies True, and no agent satisfies False.

Concerning Model 6.1, the set of propositions is $V = \{A, P\}$ (Active and Passive)

6.7 Defining the Property

There are two inputs of the WSCCS++, the model written in WSCCS language, and the property to be checked against the model. The model checking on probabilistic systems is performed via probabilistic temporal logics, and almost all temporal logics have probabilistic extensions. Hence, the specification language of the input query is a kind of branching time temporal logic with probability, time, and the notion of “action” extensions to handle probabilistic systems, called pCTL.

The user is allowed to enter the property only after parsing of the model, and propositional variable extraction phases are completed, since the property includes terms which are outputs of priori phases. To introduce the property to the system, we have provided an environment in which user do not have to type, but to select the components from existing lists. The benefits of this structure are that, the user does not have to be aware of the syntax of underlying input language, and creation of error free sentences. The operators provided by the pCTL language which is a static set, and the operands supplied by the WSCCS model which is a dynamically generated set are offered to user in a proper form, which is essentially the syntax of the pCTL. Thereafter, by choosing operators and operands from the proposed lists, the syntactically correct input property is constituted.

Although the user does not have to deal with the syntax of the property definition language directly while defining the property, still has to be familiar with the operators, and their meanings. Therefore the language of pCTL is presented first, although the semantics is left to the Model Checking section.

6.7.1 Definition of pCTL

The probabilistic computational tree logic, PCTL, has been introduced in (Hansson & Jonsson, 1989). The PCTL deals with discrete time as well. Time and probability components allow to formulate queries such as "does the model satisfies the property p within 3 steps with a probability of at least pr".

The main difference between PCTL and CTL is the quantification over paths, CTL allows universal (A) and existential (E) quantification over paths, i.e. one can state that a property should hold for all paths, or some computation paths respectively. It is not possible to state that a property should hold for a certain portion of computations, e.g. for at least 50% of the computations. In PCTL, on the other hand, arbitrary probabilities can be assigned to path formulas, thus obtaining a more general quantification over paths.

In this work, we are not only concerning whether or not a given property is hold in some states or paths, but also the global behavior of the system. That means, one might be able to ask "does the model satisfy the property p within 3 time steps with a probability pr while the system perform the action a finally (or a serious of actions)". Interestingly performing the action a in last step, or performing a serious of actions correspond to the interpreting F-Finally- and G-Globally- operators on paths respectively. Consequently, we have added the notions of "action" and "series of actions" in state formulas concerning probabilities of PCTL. The latest temporal logic is called pCTL.

The following is the definition of pCTL:

Assume V is a set of propositional variables ranged over f, and a is either an action, $a \in Act$, or a sequence of actions of form $a_0.a_1...a_n$ each $a_i \in Act$. The set of pCTL formulas is divided into path formulas and state formulas as in CTL, which are inductively defined as follows:

- Every $f \in V$ is a state formula,
- If ϕ and Ψ are state formulas then, so are
    - $\neg\phi$,
    - $\phi \wedge \Psi$,
    - $\phi \vee \Psi$,

  are state formulas as well, called "basic state formulas".
- If ϕ and Ψ are state formulas and t is a nonnegative integer or ∞, then
    - $AU^{\leq t}(\phi, \Psi)$, both Ψ will be true within t time steps and, ϕ will be true from now on, until Ψ becomes true.
    - $EU^{\leq t}(\phi, \Psi)$, either ϕ will remain true for at least t time steps, or that both Ψ become true within t time units and that ϕ will be true from now on until Ψ becomes true.

  are also path formulas, called "basic path formulas".
- If ϕ is a path formula and p is a real number with $0 \leq p \leq 1$, then
    - $[\phi]_{\geq p}$, for a given state s, ϕ holds for a path from s with a probability at least p;
    - $[\phi]_{> p}$ for a given state s, ϕ holds for a path from s with a probability greater than p;
    - $G\langle a \rangle.[\phi]_{\geq p}$, for a given state s, ϕ holds for a path from s, with sequence of action labelings $\langle a \rangle$, and with a probability at least p;
    - $G\langle a \rangle.[\phi]_{> p}$ for a given state s, ϕ holds for a path from s with sequence of action labelings $\langle a \rangle$, and with a probability greater than p;
    - $F\langle a \rangle.[\phi]_{\geq p}$, for a given state s, ϕ holds for a path from s, with the last action $\langle a \rangle$, and with a probability at least p;
    - $F\langle a \rangle.[\phi]_{> p}$ for a given state s, ϕ holds for a path from s with the last action $\langle a \rangle$, and with a probability greater than p;

are state formulas, called “probabilistic state formulas”.

For the formulas exemplified as follows we prefer to use its equal notation:

$$\left[EU^{\leq t}(\phi,\Psi)\right]_{\geq p} \equiv EU^{\leq t}_{\geq p}(\phi,\Psi)$$

6.7.2 Expressive Power of pCTL

There are temporal operators which appear in CTL, but have not mentioned in pCTL. That is, X –Next– , F–Finally–, and G–Globally–. The reason is that, they are operators concerning paths, and in pCTL path is included in the satisfaction relation itself with “time” component. Therefore, all traditional CTL operators can be written in term of U-Until- operator.

$$G^{\leq t}_{\geq p}(\phi) \equiv EU^{\leq t}_{\geq p}(\phi, false)$$

means that the formula ϕ holds during t time units with a probability of at least p.

$$F^{\leq t}_{\geq p}(\phi) \equiv AU^{\leq t}_{\geq p}(true, \phi)$$

means that the formula ϕ holds within t time units with a probability of at least p.

One may concern continuously operating system, rather than defining an upper limit on time. In this case, the following expressions are used to express the properties. Note that, addition of the universal and existential quantifiers to the left hand side, handled with the probabilities on the right hand side.

$AG(\phi) \equiv EU^{\leq\infty}_{\geq 1}(\phi, false) : AG(\phi)$ means that ϕ is always true (in all states that can reached with probability of 1),

$AF(\phi) \equiv AU^{\leq\infty}_{\geq 1}(true, \phi)$: $AF(\phi)$ means that a state where ϕ is true will eventually be reached with probability 1,

$EG(\phi) \equiv EU_{>0}^{\leq\infty}(\phi, \mathit{false})$: $EG(\phi)$ means that there is a nonzero probability for ϕ to be true always in time,

$EF(\phi) \equiv AU_{>0}^{\leq\infty}(\mathit{true}, \phi)$: $EF(\phi)$ means that there is a state where ϕ holds which can be reached with nonzero probability.

Actually, pCTL is superset of CTL, therefore all CTL operators can be written in terms of pCTL operators, with an unlimited time, and probability of 1.

6.7.3 Entering the Property

There exists a three layered construction schema in correspondence with the syntax of pCTL. That is, the probabilistic state formulas, which have the most complex form, may contain basic path formulas. And basic path formulas are composed of basic state formulas. Consequently, basic state formulas are generated from propositional variables first, Figure 6.6, then if required basic path formulas, Figure 6.7, and probabilistic formulas, Figure 6.8, are made up.

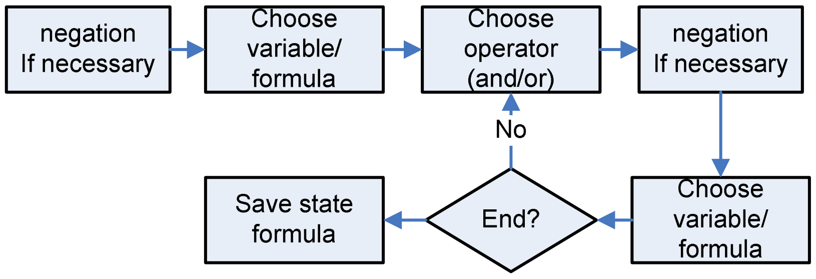


Figure 6.6 Making up basic pCTL state formulas.

By this way, basic state formulas can be formed in any complexity with the precedence rules, such that, negation has the highest priority, then conjunction, and disjunction is processed. That means, whenever conjunction operator is met in the sequence, its immediate left and right neighbors are associated as its operands. For example, the below formula

$$p \vee q \wedge \neg r \vee s$$

is interpreted as:

$$p \vee (q \wedge \neg r) \vee s$$

Similarly:

$$p \wedge \neg q \vee r = (p \wedge \neg q) \vee r$$
$$p \wedge q \wedge \neg r = (p \wedge q) \wedge \neg r$$
$$p \vee q \wedge r = p \vee (q \wedge r)$$

However, if one wants to define a formula as

$$(p \vee q) \wedge (\neg r \vee s)$$

it is also possible with a level of abstraction. Namely, both left and right hand side subformulas of "`and`" are defined, and saved, and automatically added to state formula list. Then, they can be combined with conjunction operator.

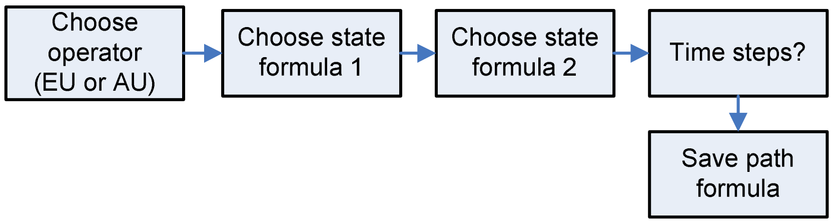


Figure 6.7 Generating the basic pCTL path formulas.

In Figure 6.7 the state formulas can be selected from a list which consists of both propositional variables, and the basic state formulas generated by Figure 6.6.

With this three layered schema, one can define as complex sentences as desired. Completing the definition of the property, it is necessary to generate abstract syntax tree to be able to apply semantic rules in further steps, which is performed by parser.

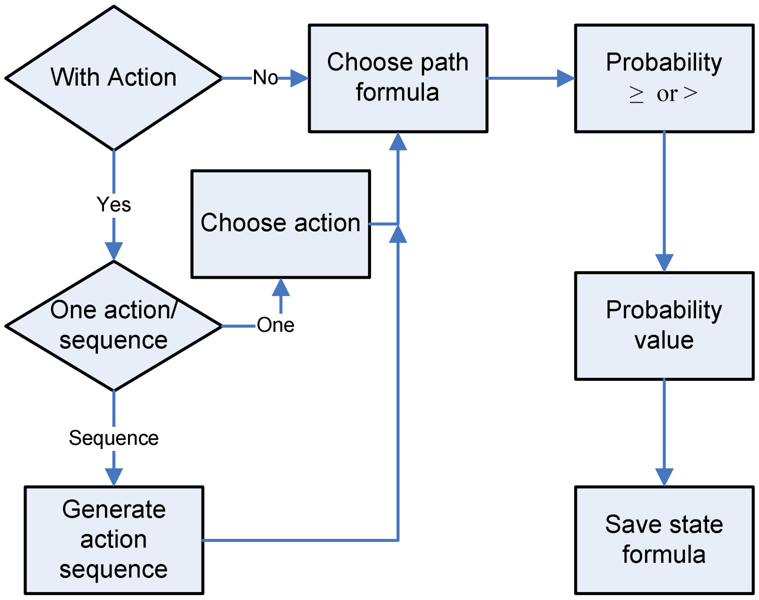


Figure 6.8 Creating probabilistic state formulas of pCTL with or without action constraints on paths.

6.8 Lexical and Syntax Analysis of the Property

The input of any parser is a sequence of tokens produced by lexical analyzer; therefore before lexing and parsing the sentence, the token types must be defined first, and the token stream consisting of token-value pairs must be generated by the lexer.

6.8.1 Tokens for pCTL

The token types of pCTL are decided according to the syntactical rules. There are two groups of token types; one concerns the operators, whereas the other is for operands. The following list contains the token types, and the value sets:

- Token = `basic_state_op`

  `basic_state_op = (not + and + or)`
- Token = `path_op`

  `path_op = (AU + EU)`
- Token = `pr_identifier`

  `pr_identifier = (≥ + > )`

- Token = `action_identifier`

  `action_identifier = (G + F)`
- Token = `basic_state_var`

  `basic_state_var = {set of propositional variables}`

  Note that, this propositional variable set is an output of WSCCS parser.
- Token = `min_time`

  `min_time: nonnegative integer value`
- Token = `action_str`

  `action_str = action(.action)*`

  where action is ranged over the set of actions is an output of WSCCS parser.
- Token = `pr_value`

  `pr_value: real number in` $[0,1]$ `interval`

Also, three abstract token types are provided in order to distinguish path, state and probabilistic state formulas constructed, `path_formula_token(PFT)`, `state_formula_token (SFT)`, `probabilistic_state_formula_token (PSFT)`. The values of these tokens are a stream of token-value pairs of path formulas, state formulas, or probabilistic path formulas respectively.

6.8.2 Lexical Analyzer –Lexer– for pCTL

The task of a lexer was defined as separating the input text into logical parts, and assigning a token type for each of those parts. The environment used for defining the pCTL sentences is not a typed one, instead offers the logical units which are already tokenized. Therefore, there is no separate unit as lexer.

6.8.3 Syntax Analyzer —Parser— for pCTL

We have followed a different parsing approach than that of WSCCS parsing. The reason is that, the input of pCTL parser is one statement at a time, where WSCCS may contain unlimited number of statements. Also, the three layered property definition schema performs almost all tasks that parser supposed to, by providing abstraction in each layer.

AST Creation for Basic State Formulas: Bottom up parsing from left to right with three units of look ahead information has been employed.

a. The idea behind subtree creation for "`not`" is, whenever "not" is met, read next token, make it a child of "not",
b. In case of reading either propositional variable or an abstract token pointing "`SFT`", a corresponding node is created or root of the "`SFT`" is taken into consideration. The subsequent token of a propositional variable can be either "`and`" or "`or`".
c. If read token is "`and`" its node is immediately created. There must be a free node which is read just before "`and`", that node is connected as a child of "`and`". Then, the next token in the stream is read. It must be either a propositional variable or a "`SFT`". This token is also connected to the "`and`" as the second child.
d. If the read token is "`or`", in order to preserve precedence order, three more tokens are read to check whether an "`and`" operator exists (Third look ahead value is used in case the first read one is "not" operator). If "`and`" exists, its corresponding subtree is created first by using rule (c), then the subtree is connected as a child of previously constructed "`or`" node.

Figure 6.9 shows creation order of nodes for the formula below. The numbers under the symbols denote the reading order of corresponding token.

$$\begin{array}{cccccccc} p & \vee & \neg & q & \wedge & r & \vee & s \\ 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \end{array}$$

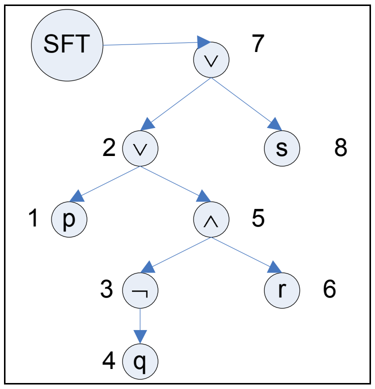


Figure 6.9 AST of given formula, the numbers next to the nodes indicate the creation order.

AST Creation for Basic Path Formulas: There are two different operators involved in path formulas EU, and AU. Both are treated as ternary operators with one operand specifying time, and the two operands specifying state formula. Therefore, AST creation for basic path formulas is relatively simple as compared to the state formulas. The important point here is the ordering of state formulas. The syntax of pCTL imposes us to keep track of whether one of the state formulas holds before the other. Therefore, any path formula of the form $AU^{\leq t}(\phi,\Psi)$ or $EU^{\leq t}(\phi,\Psi)$ result in ASTs as in Figure 6.10, with SFT being a root of the state formula subtree.

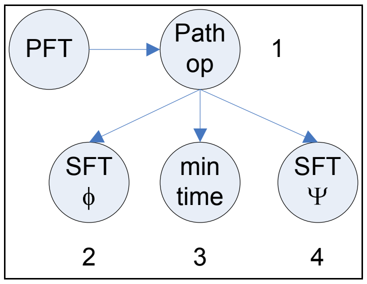
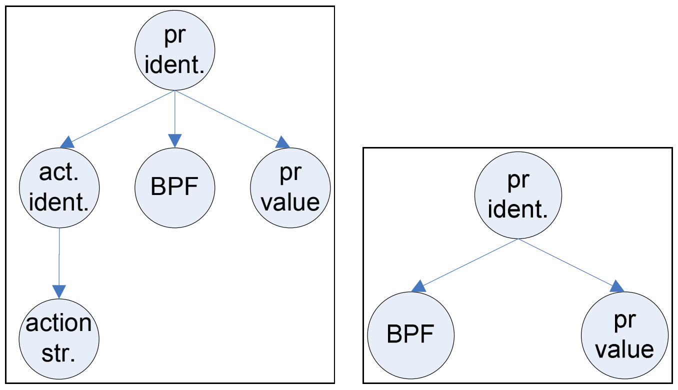


Figure 6.10 AST for path formulas, the numbers next to nodes indicate the creation order.

AST Creation for Probabilistic State Formulas: Probabilistic state formulas contain a basic path formula, a probability identifier denoting “at least” or “greater than” cases, a probability value, and if desired an action or a sequence of actions. So, there are two possible outcomes for probabilistic state formulas, the AST with action(s), and the tree without actions, Figure 6.11(a) and Figure 6.11(b) respectively.



(a) (b)

Figure 6.11 AST s for probabilistic state formulas with action or action sequence (a), and without actions (b).

6.8.4 *An Example Formula and its AST*

Let the set of propositional variables be $\{A, P\}$, the set of actions be $\{x, y\}$, and let the pCTL formula be a probabilistic path formula given as following:

$$\phi = \neg A,\ \Psi = A \vee P \quad \text{and} \qquad \mathrm{F}\langle \mathrm{x} \rangle .EU_{\geq 0.5}^{\leq 20}(\phi, \Psi)$$

The formula means that, with a probability at least 0.5, and the action to be performed in last step being x:

- either ϕ will be true during at least 20 time steps, or
- Ψ will become true within 20 time steps, and ϕ will be true before Ψ.

The corresponding AST is drawn as in Figure 6.12.

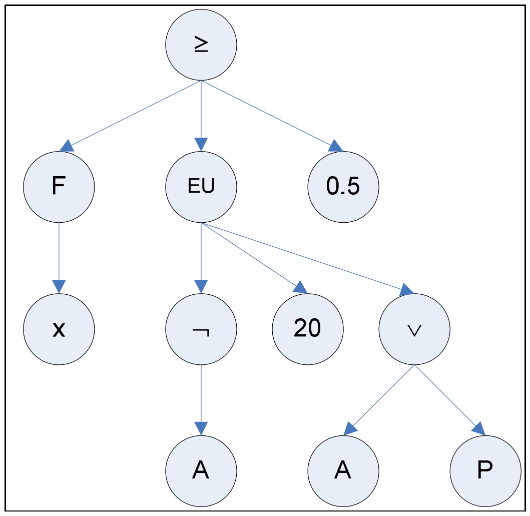


Figure 6.12 Full detailed AST of the formula.

6.9 Constructing State Transition Graph

In constructing state transition graph of models, there are three options: Kripke structures, labeled transition system, and a combination of two, labeled Kripke transition systems. The system of labeled Kripke transitions helps us in creation of state transition system in current work. Because, in WSCCS formalism, there are states whose contents must be identified in state graph clearly. The transition relation between states is described by both the actions, and the probabilities of action occurrences. We proposed the Labeled Kripke Transition System* (LKTS*), that relates the probabilities and actions with the transitions between states.

In the following subsections, the LKTS* is introduced first. After explaining construction of LKTS* graphs algebraically along with an example, the graph generator of WSCCS++ is presented.

6.9.1 Labeled Kripke Transition Systems for WSCCS++*

To obtain the LKTS of any model M, *states* of M, the set of *actions*, the transition relation, and *labeling* of M should be clearly identified.

The starting state, say s_0, is identified by the system definition, that means, the propositional variables of corresponding running agents appearing initially in the system would be true at s_0. Then the definition of each agent gives the transition relation; which is a combination of a probability, and an action to a successor state. So, we propose a different transition relation than that of previously defined LKTS as following:

The transition relation of a LKTS $M = (S, Act, \rightarrow, L)$ was defined in (Chapter 5.4.3) as:

- $\rightarrow \subseteq S \times Act \times S$, where S is set of states, and Act is a set of actions.

Instead of this transition relation, we suggested to use the following one in order to be able to show all possible transitions on a single graph, and call this transition system as LKTS*, $M = (S, Act, \rightarrow, W, L)$, where W is a set of weight expressions of the form nw^k. Then the transition relation is formed as:

- $\rightarrow \subseteq S \times (W : Act) \times S$, S is a set of states, Act is a set of actions, and W is the set of weight expressions, which is treated as probability as defined in (Chapter 4.2.2). In more common notation, we prefer to write $S \xrightarrow{W:Act} S$

Basically, the original rule is the Least Relation of WSCCS, if weight is used instead of action, it relates to Least Multi-Relation of WSCCS. What we have done is to combine the two relations into one.

6.9.2 Generating LKTS Graphs*

The necessary inputs to obtain the LKTS* graphs are, the set of propositional variables, the set of each agent definitions, the definition of LKTS* transition rule, and the sets of semantic, equational rules of WSCCS.

2.6.5.9 Composing Initial State

The global system agent definition holds the propositional variables of initial state.

Example

As an example, consider the Model 6.1 which is rewritten below. The agents were defined as[2]:

```
Active = (p:#.Passive) ~ (q:#.Active);
Passive = 1:#.Passive;
Colony = (Active | Active )| (Passive | Passive);
```

The set of propositional variables for this model was extracted as:

$$V = \{A, P\}$$

with A representing $Active$ and P representing $Passive$ agents.

The last requirement is the global system agent. Assume that "`Colony`" is specified as global agent. The definition of "`Colony`" contains the concurrently running agents $\{Active, Active, Passive, Passive\}$.Thus, the labeling of initial state s_0 is formed as following.

$$L(s0) = \{A, A, P, P\}$$

2.6.5.10 Determining the State Space

Once the initial state is composed of from propositional variables, which is essentially the list of running agents when the system is started to execute, the rest of the states –agents– can be obtained by applying the equational rules of WSCCS on running agents. In other words, the list of agents that might appear at any time in the system is determined by calculating all possible transitions.

[2] Some novel operators of WSCCS are replaced with some other operators in WSCCS++ editor, so that the user can easily type, and to remove overlapping semantics.

$+$ operator in weighted choice is replaced with ~

$\times$ operator in parallel composition is replaced with |

The set of Equational Rules helps algebraically to figure out the set of applicable actions and/or the set of agents as outcomes. The example demonstrates how the rules are applied by hand, and the evolution of states.

The following steps are used to compose the neighborhood of a single state, which is involved in local model checking. However, if global model checking is to be performed, then the steps repeated until no new state is found, and results in global state space of the system.

Step 1: Replace the agents with the definitions:

$$Active \mid Active \mid Passive \mid Passive = (p:\#.Active + q:\#.Passive)\times$$
$$(p:\#.Active + q:\#.Passive)\times(1:\#.Passive)\times(1:\#.Passive)$$

Step 2: Applying Exp4 Rule given in Appendix A, which explains parallel composition of agents with weighted action choice rules, all possible behaviors of the system in one time step are obtained.

$$= p*p*1*1:(\#.Active\times\#.Active\times\#.Passive\times\#.Passive)+$$
$$p*q*1*1:(\#.Active\times\#.Passive\times\#.Passive\times\#.Passive)+$$
$$q*p*1*1:(\#.Passive\times\#.Active\times\#.Passive\times\#.Passive)+$$
$$q*q*1*1:(\#.Passive\times\#.Passive\times\#.Passive\times\#.Passive)$$

Step 3: There exist 4 choices, it is necessary to calculate the probability of choosing each branch by using multiplication / addition rules of weight expressions.

$$= p^2:(\#.Active\times\#.Active\times\#.Passive\times\#.Passive)+$$
$$(p*q):(\#.Active\times\#.Passive\times\#.Passive\times\#.Passive)+$$
$$(q*p):(\#.Passive\times\#.Active\times\#.Passive\times\#.Passive)+$$
$$q^2:(\#.Passive\times\#.Passive\times\#.Passive\times\#.Passive)$$

Step 4: Next step is to evaluate the action sequence to be performed by each branch, which is defined in Exp1:

$$\begin{aligned}&= p^2 : \#\#\#\#.(Active \times Active \times Passive \times Passive) + \\ &(p*q) : \#\#\#\#.(Active \times Passive \times Passive \times Passive) + \\ &(q*p) : \#\#\#\#.(Passive \times Active \times Passive \times Passive) + \\ &q^2 : \#\#\#\#.(Passive \times Passive \times Passive \times Passive)\end{aligned}$$

Step 5: By definition of $\#$ action, performing $\#$ actions concurrently by a number of agents yields in $\#$ action again.

$$\begin{aligned}&= p^2 : \#.(Active \times Active \times Passive \times Passive) + \\ &(p*q) : \#.(Active \times Passive \times Passive \times Passive) + \\ &(q*p) : \#.(Passive \times Active \times Passive \times Passive) + \\ &q^2 : \#.(Passive \times Passive \times Passive \times Passive)\end{aligned}$$

Step 6: $p*q = q*p$ by the rules of multiplying weights, and rule Exp3 is applied on the agents following the probabilities in branches, which says the parallel composition operator is commutative.

$$\begin{aligned}&= p^2 : \#.(Active \times Active \times Passive \times Passive) + \\ &(p*q) : \#.(Active \times Passive \times Passive \times Passive) + \\ &(p*q) : \#.(Active \times Passive \times Passive \times Passive) + \\ &q^2 : \#.(Passive \times Passive \times Passive \times Passive)\end{aligned}$$

Step 7: There are two weighted choices having same action to be performed, and same set of resulting agents. Applying the weight addition rule we obtain the complete set of possible transitions for the initial configuration of system.

$$\begin{aligned}&= p^2 : \#.(Active \times Active \times Passive \times Passive) + \\ &2(p*q) : \#.(Active \times Passive \times Passive \times Passive) + \\ &q^2 : \#.(Passive \times Passive \times Passive \times Passive)\end{aligned}$$

Step 8: In the last step, the new states, if any, are constructed by extracting the list of agents from the list of transitions that the current system may perform. The operational semantic rules of WSCCS serve for extracting agents from transitions. Using W1 rule, which deals with weighted choice, followed by A1 rule that is used to evaluate the result of action performing, we come up with the subsequent agents: Let the whole transition be T:

$$\begin{aligned}T = p^2 :\#.(Active \times Active \times Passive \times Passive) + \\ 2(p*q):\#.(Active \times Passive \times Passive \times Passive) + \\ q^2 :\#.(Passive \times Passive \times Passive \times Passive)\end{aligned}$$

Applying W1 and A1 yields:

$$T \xrightarrow{p^2/(p+q)^2:\#} (Active \times Active \times Passive \times Passive)$$
$$T \xrightarrow{2pq/(p+q)^2:\#} (Active \times Passive \times Passive \times Passive)$$
$$T \xrightarrow{q^2/(p+q)^2:\#} (Passive \times Passive \times Passive \times Passive)$$

The result is nothing but the transition rules of LKTS* graph structure with T representing the initial configuration; except, instead of the propositional variables, the agents appearing on transitions. Now, one can easily substitute the agents with their corresponding propositional variables, where T is substituted with initial state, s_0, as below:

$$s_0 \xrightarrow{p^2/(p+q)^2:\#} (A \times A \times P \times P)$$
$$s_0 \xrightarrow{2pq/(p+q)^2:\#} (A \times P \times P \times P)$$
$$s_0 \xrightarrow{q^2/(p+q)^2:\#} (P \times P \times P \times P)$$

We have obtained 2 new states. The labeling of states, and the transitions:

$$L(s_1) = \{A, P, P, P\}$$
$$L(s_2) = \{P, P, P, P\}$$

$$s_0 \xrightarrow{p^2/(p+q)^2:\#} s_0$$
$$s_0 \xrightarrow{2*p*q/(p+q)^2:\#} s_1$$
$$s_0 \xrightarrow{q^2/(p+q)^2:\#} s_2$$

Repeating the process until all possible states are determined, the following complete set of states and transitions are reached:

$$S = \{s_0, s_1, s_2\}$$

$$s_0 \xrightarrow{p^2/(p+q)^2:\#} s_0 \qquad s_1 \xrightarrow{p/(p+q):\#} s_1$$
$$s_0 \xrightarrow{2*p*q/(p+q)^2:\#} s_1 \qquad s_1 \xrightarrow{q/(p+q):\#} s_2$$
$$s_0 \xrightarrow{q^2/(p+q)^2:\#} s_2 \qquad s_2 \xrightarrow{1:\#} s_2$$

The LKTS* transition graph can be drawn as Figure 6.13.

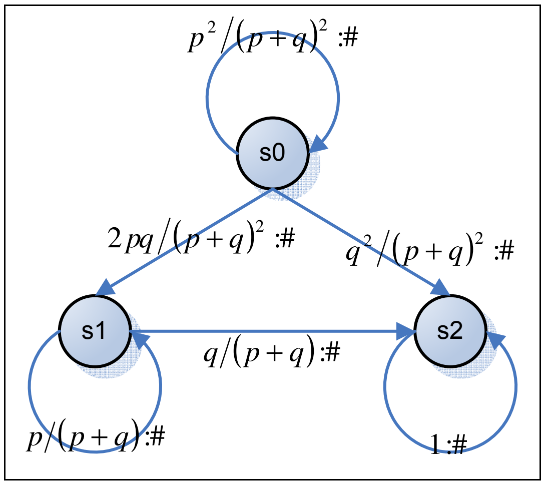


Figure 6.13 The LKTS* transition graph belonging to the Model 6.1

6.9.3 Graph Generator of WSCCS++

The inputs of the graph generator are a set of propositional variables, a set of abstract syntax trees –one for each agent definition such that one is identified as global system agent. The required rule sets are operations on weights, equational and operational semantics of WSCCS, and the definition of LKTS*. The output is a graph structure which is stored as a matrix with states in its rows and columns, and the pair $w:act$ as values whenever a transition exists between a pair of states.

According to the design of WSCCS++, each agent has equal importance by definition, which means, the global system agent has no different notion in input text. However, it is specified by the user while submitting the property. This global system agent corresponds to initial state of transition graph.

The labeling of initial state as well as other states is performed as following:

```
Labeling the states:
Check the abstract syntax tree of agent
If the root is parallel composition operator
    Find all its children,
    Associate them with propositional variables
    Add propositional variable into current state's labels.
Otherwise, add the propositional variable of agent under question
```

The challenging tasks in generating state transition graphs are how to apply the equational rules then semantic rules on given agent definitions, and in which order to apply. The ordering of applying the rules mainly decided by each agent's AST. That is, the operator on top of AST points the equational rule to be carried out.

Fundamentally, the example presented in 6.9.2.2 gives the main idea on implementing the rules. Therefore, we'll use the same steps and figure out the realization schema for the rules used in example.

The basic idea during all steps is to make up the computation branches in of the normal forms, which are, action performing as in $a.P$, or weighted action performing as $w:a.P$.

Step 1: Regardless of the autonomous behavior of each agent, the global agent (in case of local model checking, the agent specified by the user) is examined first. It comprises of a list of agents acting in parallel with parallel composition operator on top of its AST. So, for each agent, a copy of its AST is created, corresponding to the task of substituting the agent names with their definitions. The result is a number of agents in the environment, acting concurrently.

Steps 2 - 3: The following algorithm executes Exp4 Rule, which assumes all agents are to perform a weighted choice, hence have (+) on root of corresponding ASTs.

```
Check the root node of each AST in the environment.
For all $AST_i$ with root being (+)
      find the number of its children, say $num\_AST_i$
Endif
For any AST having (+) on the root, say $AST_j$
      Find $k_j$, as $k_j = \prod_i num\_AST_i : i \in N$ and $i \neq j$, where N is
      the total number of (+) rooted ASTs in the environment
      Create $k_j$ number of copies of each child of $AST_i$
EndFor
```

The Exp2 rule can be attached to this procedure with the following subroutine:

```
For any AST having (.) on its root, say $AST_j$
    Find $k_j$, as $k_j = \prod_i num\_AST_i : i \in N$, where N is the number of
    the total number of (+) rooted ASTs in the environment
    Create $k_j$ number of copies of each child of $AST_j$
EndFor
```

The total number of copies of children is calculated as:

n: total number of agents

m_i: number of children for agent i

$copy\_children = \left(\prod_i m_i\right)^n$ where i is ranging over n

Furthermore, for each AST_j with m_j children, there exist m_i number of sets each of which has k_j number of same subtrees, resulting in $\left(\prod_i m_i\right)$ number of new parallel acting agents.

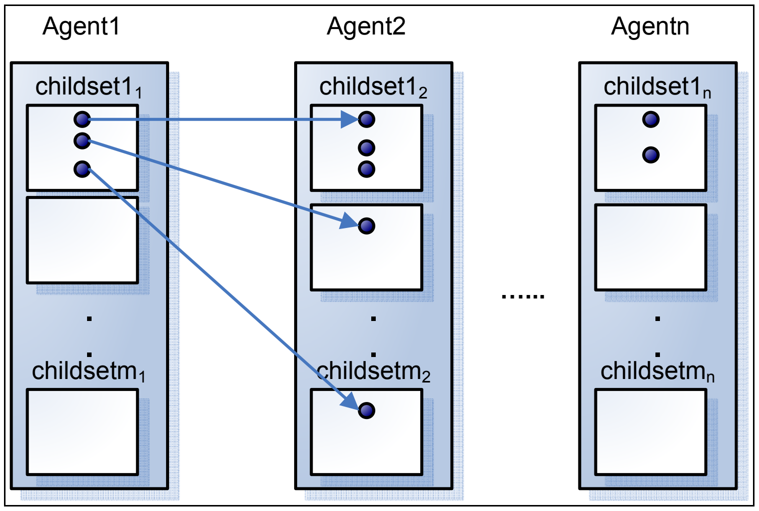
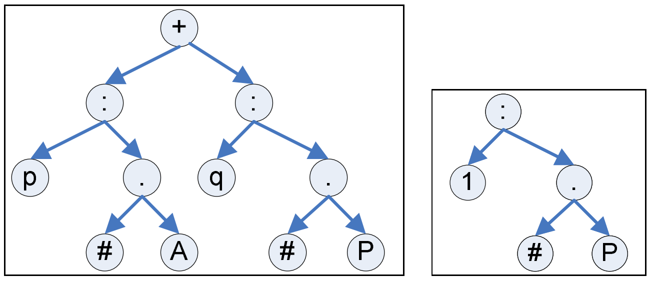


Figure 6.14 Current situation of working environment, and carrying out Cartesian product.

Making up new parallel acting agents is nothing but computing Cartesian products of all agents' subsets such that each product is generating a new parallel acting agent (Figure 6.14). In order to clarify the recently constructed agents, the temporary AST of one computation branch is shown in Figure 6.16, where in Figure 6.15 the ASTs for Active and Passive agents has been redrawn in (a) and (b) respectively.



(a) (b)

Figure 6.15 (a) AST of Active agent. (b) AST of Passive agents.

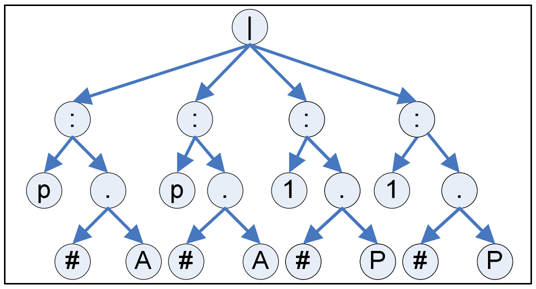


Figure 6.16 One computation branch of working example.

To complete the evaluation of Exp 4 and/or Exp2, all recently generated computation branches are connected with a weighted choice operator (+), indicating that, initializing with n agents, each of which has m_i choices in its actions, there exists $\prod_{i=1}^{n} m_i$ distinct computation paths, and each computation path consists of n components. However, (+) operator expects all of its children to be weight indicators (:) on top. Therefore, the computation branches as in Figure 6.16 should be reformed.

All weights of any branch need to be multiplied which later on gives the probability of choosing that path, before making them a whole.

```
For all recently generated (|) ASTs, define a probability pr,
    For all subtrees
           Take weight on left child, multiply with pr
           Connect right subtree to (|)
           Destroy the subtree
    EndFor
    Create a node with weight indicator (:) on root
    Create a node for with pr, connect to (:) as left child
    Connect (|) as right child
EndFor
```

Creating a (+) rooted AST, connecting all (:) AST as children finalizes steps 2 & 3.

Steps 4-5-6: Evaluating the global behavior of each computation branch:

```
For all ( : ) subtrees of (+) rooted AST obtained earlier,
   Move to right child, with (|) on top.
   Create a node that will contain action sequence
   For all subtrees of  ( | )
        Take action from left child
        Add to action sequence
        Connect right child to ( | )
        Destroy subtree
   End for
   Evaluate the result of action sequence – corresponds to Step 5
   Create a new node for action performing (.) on top
        Connect action sequence as left child
        Connect (|) subtree as left child
   Connect (.) subtree as right child of (:)
```

Steps 4-5-6 yield in AST of the form in Figure 6.17. Only one of the computation branches shown in detail.

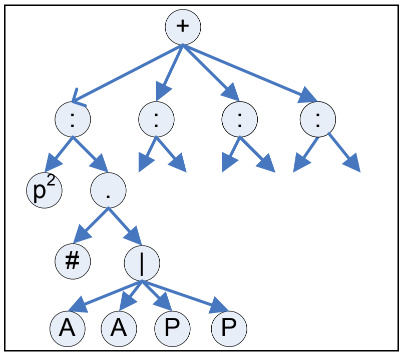


Figure 6.17 Current AST of the global system.

Steps 7-8: Discovering new states of LKTS*, and computing their transitions relations is nothing but traversing the AST.

```
For all subtrees of (+)
      Take right child as weight
      Move to left child
      Take its left child as action sequence
  Move to right child which is  (|)
  Call state labeling routine.
  If a new state is detected, create a transition from current
      state, and assign the weight: action pair as transition label.
End For
```

The generation of complete LKTS* graph ends with repeatedly applying the steps until no new state is detected.

6.10 Model Checking in WSCCS++

Model checking in its general definition considers whether or not a model satisfies a given property. However, there are three typical questions usually considered in model checking:

- Safety properties: can be checked on finite traces, and concern questions such as whether the system in invariant / deadlock free. That is, if a model is guaranteed to be safe, then “something bad never happens”.
- Liveness properties: can be checked on infinite traces, the questions of whether the system is fair, or responds some actions are considered as liveness properties. In general, if a model satisfies liveness property, then “something good will eventually happen”.
- Fairness properties: can be checked on infinite traces, and require that certain states be reached or certain conditions happen repeatedly. A fairness condition consisting of a set of states imposes that only the transitions that include that set of states be checked against a property.

Basically model checking is nothing but an algorithm such that against a given property, traversing the tree, i.e. temporal logic based model checking, or moving on transition graph and constructing a path, i.e. automata based model checking. Temporal logic based model checking has been employed in WSCCS++, since automata based techniques suffer from state space explosion.

In traditional model checking approaches the queries that might be submitted to the model are exact queries, such as the one stated in liveness property, "something bad never happens". Although the meaning of "bad" changes by the system, the treatment is similar. The answer is typically "yes", or "no". Therefore, the temporal logics mentioned in Chapter 5.6 to accomplish the processing of the query on a tree do not take into account the probabilities.

In a general probabilistic system the questions are formed in following manner: (i) the property to be checked, and (ii) the probability of holding the property. In other words:

- What is the probability of ϕ is satisfied by M? The answer will be a certain real value in $[0,1]$,
- Does M satisfies ϕ with a probability p? The answer will be "yes" or "no" with a certain real value or interval in $[0,1]$.

6.10.1 Paths, Calculating Probabilities on Paths

pCTL formulas are interpreted over LKTS* structures in current work. A specified initial state is associated with the structure. In order to interpret the formulas of pCTL, it is necessary to specify the *path*, and *probability measure of a path.*

Given a LKTS* structure $M = (S, Act, \rightarrow, W, L)$, a path π from a state s_0 in a structure is defined as an infinite sequence of states

$$s_0 \xrightarrow{w_0:a} s_1 \xrightarrow{w1:a} \ldots \xrightarrow{w_{n-1}:a} s_n \xrightarrow{w_n:a} \ldots$$

with s_0 being the first state of the path, and a_i ranging over Act. The nth state s_n of π is denoted $\pi[n]$, and a prefix of π of length n is denoted $\pi\uparrow n$, i.e.,

$$\pi\uparrow n = s_0 \xrightarrow{w_0:a} s_1 \xrightarrow{w1:a} \ldots \xrightarrow{w_{n-1}:a} s_n$$

For each LKTS* structure, and for each state s_0, we define the probability measure pr_m on every path initialized with s_0.

$$pr_m\left(\pi\uparrow n = s_0 \xrightarrow{w_0:a} s_1 \xrightarrow{w1:a} \ldots \xrightarrow{w_{n-1}:a} s_n\right) = \prod_{i=0}^{n-1} w_i$$

where multiplication holds the rules of WSCCS weight expression multiplication (because they are treated as probabilities as previously defined).

The serious of actions for $\pi\uparrow n, acs(\pi\uparrow n)$, is nothing but the sequential composition of the actions as in WSCCS.

$$acs(\pi\uparrow n) = a_0.a_1 \ldots a_{n-1}$$

2.6.5.11 The Satisfaction Relation, Semantics of pCTL

The truth value of a pCTL formula ϕ for a LKTS* structure M is given as

$$M, s \models \phi$$

which intuitively means that , the state formula ϕ is true at state s. However, in order to define the satisfaction relation for states, it is helpful to use another satisfaction relation which is actually concerns paths:

$$M, \pi \models\!\equiv \phi$$

which means that, the path π satisfies the path formula ϕ in structure M. The relations $M, s \models \phi$ and $M, \pi \models\equiv \phi$ are inductively defined as follows:

1. $M \models \phi \equiv M, s^i \models \phi$ with s^i is the initial state of structure M.
2. $M, s \models f$ iff $f \in L(s)$,
3. $M, s \models \neg\phi$ iff $M, s \not\models \phi$ in other words $\neg(M, s \models \phi)$,
4. $M, s \models \phi \wedge \Psi$ iff $M, s \models \phi \wedge M, s \models \Psi$,
5. $M, s \models \phi \vee \Psi$ iff $M, s \models \phi \vee M, s \models \Psi$,
6. $M, s \models \phi \Rightarrow \Psi$ iff $M, s \models \neg\phi \vee M, s \models \Psi$,
7. $M, \pi \models\equiv AU^{\leq t}(\phi, \Psi)$ iff there exists an $i \leq t$ such that $M, \pi[i] \models \Psi$ and $\forall j : 0 \leq j < i : (M, \pi[i] \models \phi)$,
8. $M, \pi \models\equiv EU^{\leq t}(\phi, \Psi)$ iff either $M, \pi \models\equiv AU^{\leq t}(\phi, \Psi)$ or, $\forall j : 0 \leq j \leq t : (M, \pi[j] \models \phi)$
9. $M, s \models [\phi]_{\geq p}$ iff the pr_m measure of the set of paths π starting in s for which $M, \pi \models\equiv \phi$ is at least p.
10. $M, s \models [\phi]_{> p}$ iff the pr_m measure of the set of paths π starting in s for which $M, \pi \models\equiv \phi$ is greater than p.
11. $M, s \models G\langle a \rangle.[\phi]_{\geq p}$ iff the pr_m measure is at least p for the set of paths π starting in s for which $M, \pi \models\equiv \phi$, and $acs(\pi) = \langle a \rangle$.
12. $M, s \models G\langle a \rangle.[\phi]_{> p}$ iff the pr_m measure is greater than p for the set of paths π starting in s for which $M, \pi \models\equiv \phi$, and $acs(\pi) = \langle a \rangle$.
13. $M, s \models F\langle a \rangle.[\phi]_{\geq p}$ iff the pr_m measure is at least p for the set of paths π starting in s for which $M, \pi \models\equiv \phi$, and $acs(\pi)[s_{last}] = \langle a \rangle$, where $acs(\pi)[s_{last}]$ indexes the last action in the path formula π.
14. $M, s \models F\langle a \rangle.[\phi]_{> p}$ iff the pr_m measure is greater than p for the set of paths π starting in s for which $M, \pi \models\equiv \phi$, and $acs(\pi)[s_{last}] = \langle a \rangle$.

6.10.2 WSCCS Example

Working on Model 6.1, we have obtained LKTS* graph, and its unfolded computation tree. The following examples will be processed on the same example.

Now, we attempt to formulate some queries, however, since the model is so primitive in the sense that there is action other than $\#$, and once an agent turns out to $Passive$ there is no chance to become $Active$ again, the types of queries is very limited.

- Is it possible to have a state having 4 $Active$ agents?

  First formulate the propositional formula ϕ to define 4 active agents in same state such that:

  $$\phi = A \wedge A \wedge A \wedge A$$

  Deciding the path quantifier to be used is given explicitly in verbal form, "is there any state on any path", corresponding to existential quantifier with "Finally" operator. Thus:

  $$EF(\phi) \equiv AU^{\leq\infty}(true, \phi)$$

  And the last requirement is to decide the probability which is interpreted as "greater than zero".

  $$AU_{>0}^{\leq\infty}(true, \phi)$$

  Then the satisfaction query is:

  $$M, s_0 \models AU_{>0}^{\leq\infty}(A \wedge A \wedge A \wedge A)$$

This relation corresponds to the semantic rule 10. In order to find the result, we suggest an algorithm that first checks the state space, which is finite.

$$L(s0) = \{A, A, P, P\}$$
$$L(s_1) = \{A, P, P, P\}$$
$$L(s_2) = \{P, P, P, P\}$$

Since there is no state containing 4 A propositions, the answer for $M, s_0 \models AU_{>0}^{\leq\infty}\left(A \wedge A \wedge A \wedge A\right)$ will be “No”.

6.11 Conclusion

There exist several modeling and reasoning software tools for different kinds of modeling languages, as well as process algebraic languages. However, none of the known process algebra tools gives support for WSCCS. This chapter introduces the WSCCS++, which is a tool developed for modeling in WSCCS and reasoning models written in WSCCS language.

The theoretical background given in Chapter 5 is extended in a way that, probabilities of action occurrences and paths can be questioned as model checking relations.

CHAPTER SEVEN

WSCCS CASE STUDIES

7.1 Conflicting Actions

Consider the below A and B agents, and the colony definition. Agent A may perform actions either a or b, where performing a is prioritized over performing b. Agent B performs the complementary actions !a, or !b, such that performing !b is prioritized over performing !a. Whenever these two agents working in parallel, as in `Colony` agent, they are expected to communicate over the channels either a-!a or b-!b. But, this is not the case, because of the prioritization of the actions.

$$A \overset{def}{\equiv} w : a.X + 1 : b.Y$$

$$B \overset{def}{\equiv} 1 : !a.Z + w : !b.T$$

$$Colony \overset{def}{\equiv} A \times B < Act - \{a,b\}$$

At first glance, `Colony` definition may seem to be conflicting, since one of the agents desires to perform one action, whilst the other agent desires to perform the other action's complement. The followings are all of the possible transitions of $A \times B$:

$$\begin{aligned} A \times B &\overset{def}{\equiv} (w : a.X + 1 : b.Y) \times (1 : !a.Z + w : !b.T) \\ &= (w : a.X) \times (1 : !a.Z) + (w : a.X) \times (w : !b.T) + \\ &\quad (1 : b.Y) \times (1 : !a.Z) + (1 : b.Y) \times (w : !b.T) \\ &= w : \#.X \times Z + w^2 : a.!b.X \times T + 1 : b.!a.Y \times Z + w : \#.Y \times T \end{aligned}$$

Applying the restriction operator, $< Act - \{a,b\}$, which means the terms containing neither action a, nor action b, we obtain the resulting transitions.

$$A \times B < Act - \{a,b\} = w : \#.X \times Z < Act - \{a,b\} + w : \#.Y \times T < Act - \{a,b\}$$

7.2 Two Ant Colonies in Life Game

In this scenario, it is assumed that there are two colonies of ants, A and B, playing the game of "stay alive". The winner colony may be awarded by some resource. Each colony may contain different types and number of ants. The idea is to observe how the system and its components, ants, evolve in time. In other more abstract words, which team wins the game. Although several design choices are possible, it is assumed that, there is a leader, and a medical expert and a number of worker ants in each team at the beginning. The behavior of each agent is listed below.

- Leader: It controls, and coordinates the colony. It has the ability to fight, and distributes this ability among teammates. Or it might be shot by the other team's "Fighter". Then it becomes a "Wounded Ant", or it remains as it is. Definitions the Leaders of colony A, and B:

  $$LeaderA \overset{def}{\equiv} 4w : giveFightA.LeaderA + 1 : !shootB.WoundedA + 5 : \#.LeaderA$$

  $$LeaderB \overset{def}{\equiv} 4w : giveFightB.LeaderB + 1 : !shootA.WoundedB + 5 : \#.LeaderB$$

  There are two "giveFight" actions with A and B suffixes. It was necessary to distinguish who would be given the ability. If there were no such suffixes, each team member might be benefited. Note that the three behaviors given in verbal description are defined in the same order. Since the primary job of leaders is to distribute its abilities among its teams', this action is prioritized.

- Medical Expert: It applies treatment to the teammates when they are wounded. It cannot fight, thus it is not able to shoot. However, it might be shot, or just walks around doing nothing.

  $$MedA \overset{def}{\equiv} w : treatA.MedA + 1 : !shootB.WoundedA + 1 : \#.MedA$$

  $$MedB \overset{def}{\equiv} w : treatB.MedB + 1 : !shootA.WoundedB + 1 : \#.MedB$$

- Worker: It does not have any extra ability as long as fighting capability is not given to him by the team "Leader". Then it becomes another agent, "Fighter".

Or, the "Worker" might be shot by the other team's "Fighter". Then it becomes a "Wounded Ant".

$$Worker A \overset{def}{\equiv} 4:!giveFightA.FighterA + 4:!shootB.WoundedA + 2:\#.Worker A$$

$$Worker B \overset{def}{\equiv} 4:!giveFightB.FighterB + 4:!shootA.WoundedB + 2:\#.Worker B$$

- Fighter: It can shoot an ant of the other team. If the target is "Worker", or "Leader", or "Medical Expert" it succeeds, and causes them to get wounded.

$$FighterA \overset{def}{\equiv} w:shootA.FighterA + 1:!shootB.WoundedA + 1:\#.FighterA$$

$$FighterB \overset{def}{\equiv} w:shootB.FighterB + 1:!shootA.WoundedB + 1:\#.FighterB$$

- Wounded Ant: Once the ant is shot by other team's "Fighter" ant, it becomes a "Wounded Ant", and waits for the treatment from the team's "Medical Expert".

$$WoundedA \overset{def}{\equiv} 1:\#.WoundedA + 1:!treatA.Worker A$$

$$WoundedB \overset{def}{\equiv} 1:\#.WoundedB + 1:!treatB.Worker B$$

Defining all agents individually, it is crucial to describe the concurrently running agents, and the communication between them if exists. Since there is no restriction on number of running agents of any type, and of any colony, only the interacting agent behaviors will be identified. The following lists such agent activities.

- "Leader" distributes its fighting ability,

  $LeaderA \times Worker A < \{\#\}$, $LeaderB \times Worker B < \{\#\}$

- "Fighter" of a colony shoots,

  $FighterA \times LeaderA < \{\#\}$, $FighterB \times LeaderB < \{\#\}$
  $FighterA \times Worker A < \{\#\}$, $FighterB \times Worker B < \{\#\}$
  $FighterA \times MedA < \{\#\}$, $FighterB \times MedB < \{\#\}$
  $FighterA \times FighterB < \{\#\}$

  Note that in the latter case, it is not clear whether "FighterA" or "FighterB" will die. It is a nondeterministic choice.

- "Medical Expert" applies treatment,

  $MedA \times WoundedA < \{\#\}$, $MedB \times WoundedB < \{\#\}$

$TeamA$ and $TeamB$ defines the initial team configurations. It is assumed that there are three "Worker" ants, a "Leader", and a "Medical Expert" in each team at the beginning. $Colony$ is the whole system definition which consists of two teams given.

$$TeamA \equiv LeaderA \times Worker\,A \times Worker\,A \times Worker\,A \times MedA$$
$$TeamB \equiv LeaderB \times Worker\,B \times Worker\,B \times Worker\,B \times MedB$$
$$Colony = TeamA \times TeamB < \{\#\}$$

7.2.1 WSCCS++ Demo

The following Figure 7.1 is the main entrance of the WSCCS++, and Figure 7.2 is screenshot of the model definition editor with the described model written after compilation.

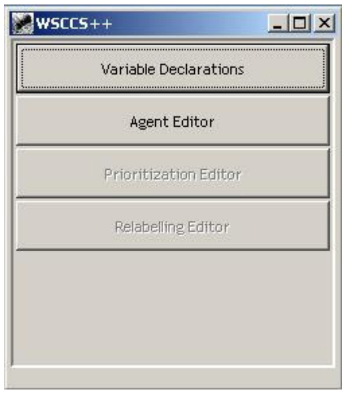
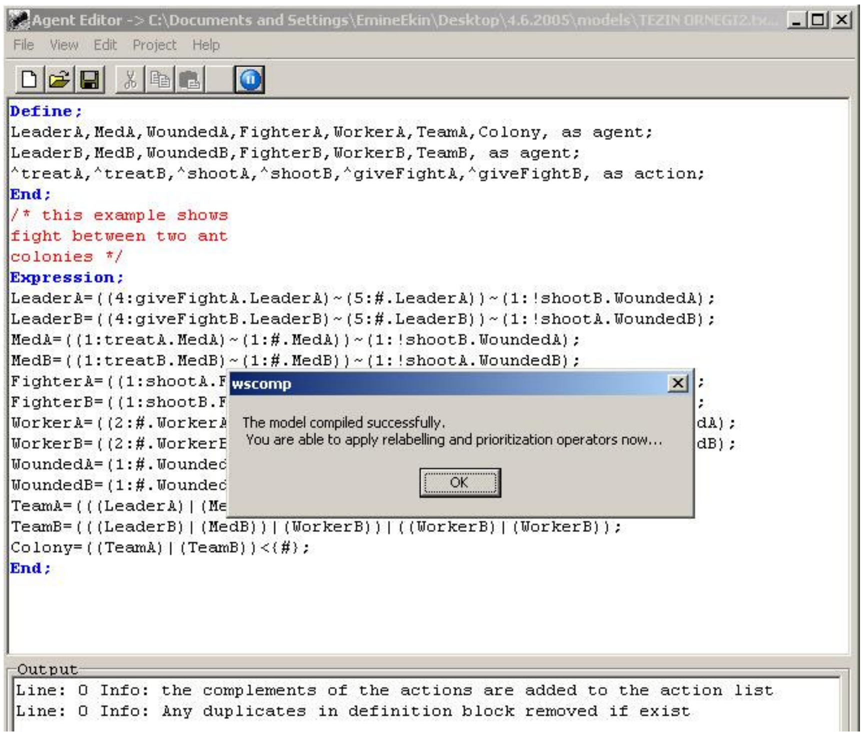


Figure 7.1 Main Window of the WSCCS++.


Agent Editor -> C:\Documents and Settings\EmineEkin\Desktop\4.6.2005\models\TEZIN ORNEGI2.tx...
File View Edit Project Help
Output
OK

```
Define;
LeaderA,MedA,WoundedA,FighterA,WorkerA,TeamA,Colony, as agent;
LeaderB,MedB,WoundedB,FighterB,WorkerB,TeamB, as agent;
^treatA,^treatB,^shootA,^shootB,^giveFightA,^giveFightB, as action;
End;
/* this example shows
fight between two ant
colonies */
Expression;
LeaderA=((4:giveFightA.LeaderA)~(5:#.LeaderA))~(1:!shootB.WoundedA);
LeaderB=((4:giveFightB.LeaderB)~(5:#.LeaderB))~(1:!shootA.WoundedB);
MedA=((1:treatA.MedA)~(1:#.MedA))~(1:!shootB.WoundedA);
MedB=((1:treatB.MedB)~(1:#.MedB))~(1:!shootA.WoundedB);
FighterA=((1:shootA.F                                        ;
FighterB=((1:shootB.F                                        ;
WorkerA=((2:#.WorkerA                                        dA);
WorkerB=((2:#.WorkerB                                        dB);
WoundedA=(1:#.Wounded
WoundedB=(1:#.Wounded
TeamA=(((LeaderA)|(Me
TeamB=(((LeaderB)|(MedB))|(WorkerB))|((WorkerB)|(WorkerB));
Colony=((TeamA)|(TeamB))<(#);
End;
```

```
wscomp
The model compiled successfully.
You are able to apply relabelling and prioritization operators now...
```

```
Line: 0 Info: the complements of the actions are added to the action list
Line: 0 Info: Any duplicates in definition block removed if exist
```

Figure 7.2 Model definition, and the result of compilation in WSCCS++.

7.2.2 *Generating LKTS* Graph*

2.6.5.12 *Extract Set of Propositional Variables*

Given the definitions of agents, extract the set of propositional variables first, such that, each agent has a corresponding variable described as:

$LeaderA \rightarrow \mathrm{LA},\ LeaderB \rightarrow \mathrm{LB},$

$MedA \rightarrow \mathrm{MA},\ MedB \rightarrow \mathrm{MB},$

$Wor\ker A \rightarrow \mathrm{WA},\ Wor\ker B \rightarrow \mathrm{WB},$

$FighterA \rightarrow \mathrm{FA},\ FighterB \rightarrow \mathrm{FB}$

$WoundedA \rightarrow \mathrm{WNA},\ WoundedB \rightarrow \mathrm{WNB}$

$TeamA \rightarrow \mathrm{TA},\ TeamB \rightarrow \mathrm{TB}$

Notice that, TA and TB are abstract variables since both $TeamA$ and $TeamB$ are abstractions for a number of parallel acting agents, called teams. Therefore, they can be reduced to set of conjunctions of related propositional variables:

$TA = LA \wedge WA \wedge WA \wedge WA \wedge MA$,

$TB = LB \wedge WB \wedge WB \wedge WB \wedge MB$

2.6.5.13 Form the Initial State from Sys Agent

$$Colony \overset{def}{\equiv} TeamA \times TeamB < \{\#\}$$

Substituting the $TeamA$ and $TeamB$ with actually running agents:

$$Colony = \begin{pmatrix} LeaderA \times Worker\,A \times Worker\,A \times Worker\,A \times MedA \times \\ LeaderB \times Worker\,B \times Worker\,B \times Worker\,B \times MedB \end{pmatrix} < \{\#\}$$

$Colony$ relates to initial state s_0, therefore, the labeling function of s_0, is as follows:

$$L(s_0) = \{LA, LA, WA, WA, WA, MA, LB, WB, WB, WB, MB\}$$

2.6.5.14 Discovering Successors of Initial State

Replace each agent appearing in $Colony$ with its definition:

$$Colony = \begin{pmatrix} 4w : giveFightA.LeaderA + 1 : !shootB.WoundedA + 5 : \#.LeaderA \times \\ 4 : !giveFightA.FighterA + 4 : !shootB.WoundedA + 2 : \#.Worker\,A \times \\ 4 : !giveFightA.FighterA + 4 : !shootB.WoundedA + 2 : \#.Worker\,A \times \\ 4 : !giveFightA.FighterA + 4 : !shootB.WoundedA + 2 : \#.Worker\,A \times \\ w : treatA.MedA + 1 : !shootB.WoundedA + 1 : \#.MedA \times \\ 4w : giveFightB.LeaderB + 1 : !shootA.WoundedB + 5 : \#.LeaderB \times \\ 4 : !giveFightB.FighterB + 4 : !shootA.WoundedB + 2 : \#.Worker\,B \times \\ 4 : !giveFightB.FighterB + 4 : !shootA.WoundedB + 2 : \#.Worker\,B \times \\ 4 : !giveFightB.FighterB + 4 : !shootA.WoundedB + 2 : \#.Worker\,B \times \\ w : treatB.MedB + 1 : !shootA.WoundedA + 1 : \#.MedB \end{pmatrix} < \{\#\}$$

Apply Exp4 rule of Equational rule set, since the computation tree too large, only one initial branch of it will be demonstrated here, although full result is shown later.

$$
\begin{aligned}
Colony_1 &= 4^8 w^4 : giveFightA.LeaderA \times !giveFightA.FighterA \times \\
&\quad !giveFightA.FighterA \times !giveFightA.FighterA \times \\
&\quad treatA.MedA \times giveFightB.LeaderB \times \\
&\quad !giveFightB.FighterB \times !giveFightB.FighterB \times \\
&\quad !giveFightB.FighterB \times treatB.MedB \\
&= 4^8 w^4 : \begin{pmatrix} giveFightA \times !giveFightA \times !giveFightA \times !giveFightA \\ \times treatA \times giveFightB \times !giveFightB \times !giveFightB \times \\ !giveFightB \times treatB \end{pmatrix}. \\
&\quad \begin{pmatrix} LeaderA \times FighterA \times FighterA \times FighterA \times MedA \times \\ LeaderB \times FighterB \times FighterB \times FighterB \times MedB \end{pmatrix}
\end{aligned}
$$

Although this branch is highly prioritized, and very likely be chosen to be performed, this is not the case. See the action happenings which needs to evaluate # action.

The parallel occurrence of actions yields the following result:

$$
\begin{aligned}
&\Rightarrow giveFightA \times !giveFightA \times !giveFightA \times !giveFightA \\
&\times treatA \times giveFightB \times !giveFightB \times !giveFightB \times \\
&!giveFightB \times treatB \\
&= \# \times treatA \times !giveFightA \times !giveFightA \times \# \times \\
&\quad treatB \times !giveFightB \times !giveFightB
\end{aligned}
$$

Since the resulting global behavior is not equal to the one given in restriction set, this branch of cannot be performed, and is not included in state space.

The following is the complete one step computation branches of $Colony$ agent.

$$
\begin{aligned}
&Colony = \\
&4^6 w^2 * 9 : \#. \begin{pmatrix} LeaderA \times FighterA \times Wor\ker A \times Wor\ker A \times MedA \times LeaderB \\ \times FighterB \times Wor\ker B \times Wor\ker B \times MedB \end{pmatrix} + \\
&4^4 w * 30 : \#. \begin{pmatrix} LeaderA \times FighterA \times Wor\ker A \times Wor\ker A \times MedA \times LeaderB \\ \times Wor\ker B \times Wor\ker B \times Wor\ker B \times MedB \end{pmatrix} + \\
&4^4 w * 30 : \#. \begin{pmatrix} LeaderA \times Wor\ker A \times Wor\ker A \times Wor\ker A \times MedA \times LeaderB \\ \times FighterB \times Wor\ker B \times Wor\ker B \times MedB \end{pmatrix} + \\
&1600 : \#. \begin{pmatrix} LeaderA \times Wor\ker A \times Wor\ker A \times Wor\ker A \times MedA \times LeaderB \\ \times Wor\ker B \times Wor\ker B \times Wor\ker B \times MedB \end{pmatrix} +
\end{aligned}
$$

From the set of possible evolutions above, only the one with the highest priority may occur (Chapter 4.3.3), which has the weight value of $4^6 w^2 * 9$.

Hence, there is only one new state and its labeling evolved from one step transition is:

$L(s_1) = \{LA, FA, WA, WA, MA, LB, FB, WB, WB, MB\}$

$s_0 \xrightarrow{1:\#} s_1$

In order to demonstrate, how solution is constructed in case of equal priorities, one more step is established from s_1. Among a large number potential of transitions having priorities, the following set has captured since each member has the highest priority.

$4^7 w^4 : (\#).\begin{pmatrix} LeaderA \times FighterA \times FighterA \times WoundedA \times MedA \times \\ LeaderB \times FighterB \times FighterB \times WoundedB \times MedB \end{pmatrix}$ + //this path will be called s$_2$

$4^6 w^4 * 2 : (\#)\begin{pmatrix} LeaderA \times FighterA \times FighterA \times WoundedA \times MedA \times \\ LeaderB \times FighterB \times FighterB \times Wor\ker B \times WoundedB \end{pmatrix}$ + //this path will be called s$_3$

$4^6 w^4 * 2 : (\#).\begin{pmatrix} LeaderA \times FighterA \times FighterA \times Wor\ker A \times WoundedA \times \\ LeaderB \times FighterB \times FighterB \times WoundedB \times MedB \end{pmatrix}$ + //this path will be called s$_4$

$4^6 w^4 : (\#).\begin{pmatrix} LeaderA \times FighterA \times FighterA \times Wor\ker A \times WoundedA \times \\ LeaderB \times FighterB \times FighterB \times Wor\ker B \times WoundedB \end{pmatrix}$ + //this path will be called s$_5$

Notice that, each weighted choice has priority of 4. Then the choice between them becomes a probabilistic process such that relative frequencies lead to the probabilities.

Recently set up states, labelings, and LKTS* transitions:

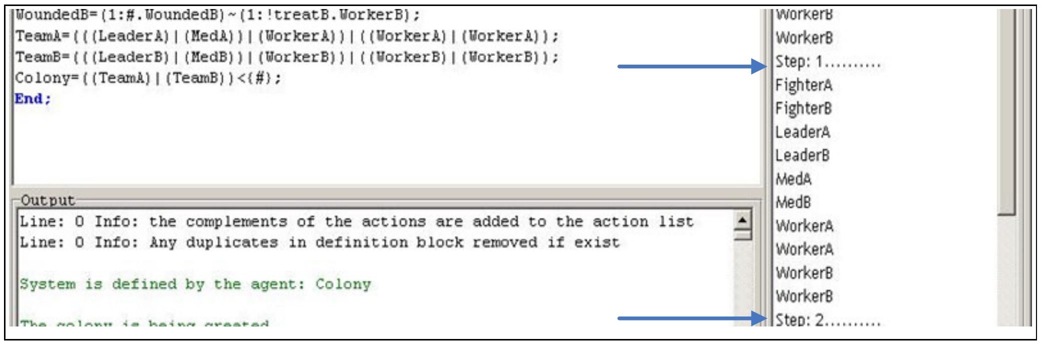
$L(s_2) = \{LA, FA, FA, WNA, MA, LB, FB, FB, WNB, MB\}$
$L(s_3) = \{LA, FA, FA, WNA, MA, LB, FB, FB, WB, WNB\}$
$L(s_4) = \{LA, FA, FA, WA, WNA, LB, FB, FB, WNB, MB\}$
$L(s_5) = \{LA, FA, FA, WA, WNA, LB, FB, FB, WB, WNB\}$

$s_1 \xrightarrow{4/9:\#} s_2$
$s_1 \xrightarrow{2/9:\#} s_3$
$s_1 \xrightarrow{2/9:\#} s_4$
$s_1 \xrightarrow{1/9:\#} s_5$

The other successor states are determined in the same way, which is not reasonable performing by hand, and constructs a basis for why an automation tool is required.

2.6.5.15 Visualizing State Space in WSCCS++

Usually, generating and visualizing the entire state space is not a reasonable task because of the state space explosion. However, WSCCS offers the user to view the initial state space, and step by step transitions on demand. The Figure 7.3 is such an example screenshot.



```
WoundedB=(1:#.WoundedB)~(1:!treatB.WorkerB);
TeamA=(((LeaderA)|(MedA))|(WorkerA))|((WorkerA)|(WorkerA));
TeamB=(((LeaderB)|(MedB))|(WorkerB))|((WorkerB)|(WorkerB));
Colony=((TeamA)|(TeamB))<(#);
End;
```

```
Output
Line: 0 Info: the complements of the actions are added to the action list
Line: 0 Info: Any duplicates in definition block removed if exist

System is defined by the agent: Colony

The colony is being created
```

```
WorkerB
WorkerB
Step: 1..........
FighterA
FighterB
LeaderA
LeaderB
MedA
MedB
WorkerA
WorkerA
WorkerB
WorkerB
Step: 2..........
```

Figure 7.3 Viewing state spaces for each step on demand.

7.2.3 Formulating the Property

Formulating model checking questions is achieved by expressing the property verbally first:

"Is there any path on which team B wins?"

From the designer's of view, winning of a team is decided by the all members of other teams' get wounded. Therefore, the question is actually, "is there any path, such that all members of team A get wounded before team B"

The propositional formulas are:

$\phi = LB \vee MB \vee WB \vee FB$ --there is a member of team B, who is not wounded.

$\Psi = WNA \wedge \neg(LA \wedge MA \wedge FA \wedge WA)$ -- there is a wounded A, and no other member of team A exists.

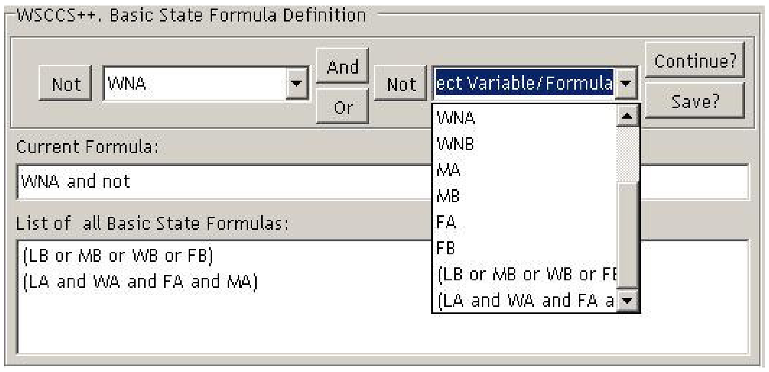
Assigning number of time steps that the property is required to be satisfied, say 3, and a probability value, we obtain the complete formula as:

$$AU^{3}_{\geq 0.8}(\phi, \Psi)$$

Notice that, the property does not include an action or action sequence, the reason behind is, the global system is restricted to perform only # action at any time,

2.6.5.16 Submitting the Property to WSCCS++ Model Checking Component

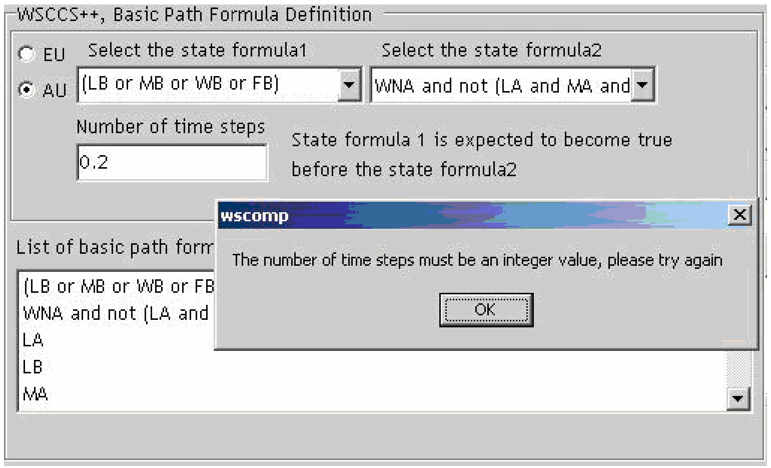
As mentioned in Chapter 6, WSCCS++ offers an environment to describe the property in which the user does not have to type anything, but selects both operators, and associated operands from the presented lists. Though the figures 7.4 to 7.6, the property definition editor screenshots has been shown.



WNA and not

(LB or MB or WB or FB)
(LA and WA and FA and MA)

Figure 7.4 Defining basic state formulas in WSCCS++.



WSCCS++, Basic Path Formula Definition

( ) EU (•) AU

Select the state formula1: (LB or MB or WB or FB)

Select the state formula2: WNA and not (LA and MA and

Number of time steps: 0.2

State formula 1 is expected to become true before the state formula2

List of basic path form

(LB or MB or WB or FB

WNA and not (LA and

LA

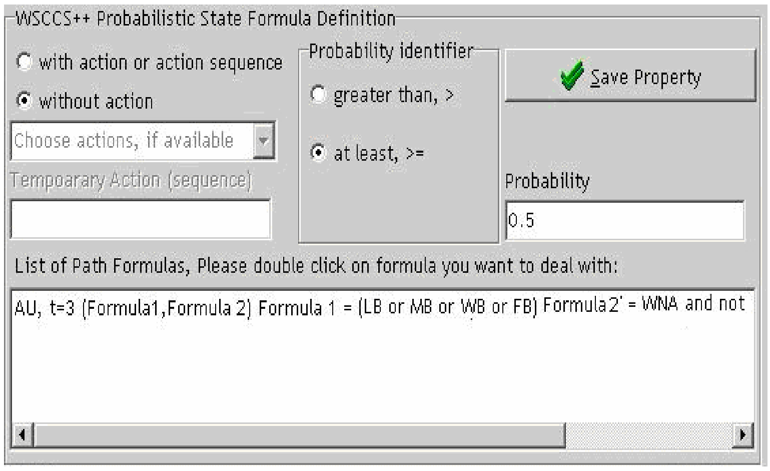
LB

MA

wscomp

The number of time steps must be an integer value, please try again

Figure 7.5 Describing basic path formulas which are based on state formulas. In case of noninteger number of time steps, an error message is displayed.



WSCCS++ Probabilistic State Formula Definition

( ) with action or action sequence

(•) without action

Choose actions, if available

Tempoarary Action (sequence)

Probability identifier

( ) greater than, >

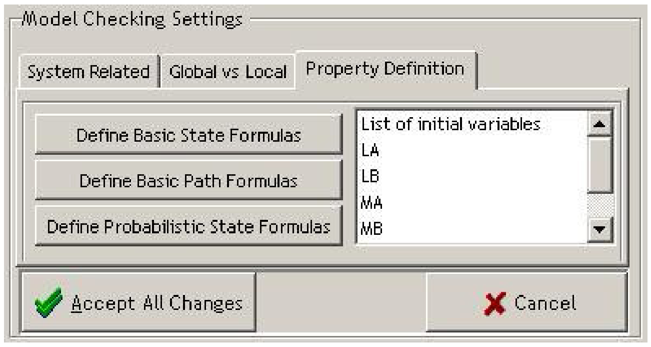
(•) at least, >=

Probability: 0.5

List of Path Formulas, Please double click on formula you want to deal with:

AU, t=3 (Formula1,Formula 2) Formula 1 = (LB or MB or WB or FB) Formula2' = WNA and not

Figure 7.6 Entering probabilistic path formulas.



Model Checking Settings

System Related
Global vs Local
Property Definition
Define Basic State Formulas
Define Basic Path Formulas
Define Probabilistic State Formulas

List of initial variables

LA

LB

MA

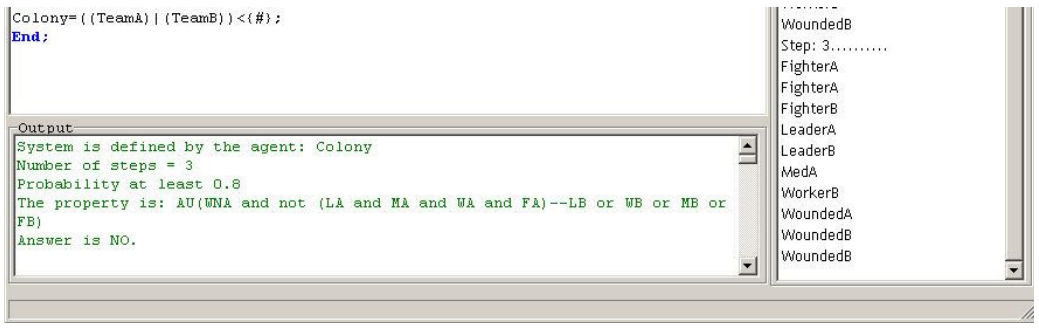
MB

Accept All Changes
Cancel

Figure 7.7 Model checking control panel.

7.2.4 Executing the Property

Given the definition of whole system, and the property, the following result is obtained. Notice that we have chosen to see the entire state space of 3 steps using the control panel.



```
Colony=((TeamA)|(TeamB))<(#);
End;
```

Output

```
System is defined by the agent: Colony
Number of steps = 3
Probability at least 0.8
The property is: AU(WNA and not (LA and MA and WA and FA)--LB or WB or MB or
FB)
Answer is NO.
```

```
WoundedB
Step: 3..........
FighterA
FighterA
FighterB
LeaderA
LeaderB
MedA
WorkerB
WoundedA
WoundedB
WoundedB
```

Figure 7.8 The result of the property processed by WSCCS++.

CHAPTER EIGHT

CONCLUSIONS, CONTRIBUTIONS, FUTURE WORK

8.1 Summary & Contributions Inline

Most of the topics discussed throughout the dissertation are formal treatments ranging from modeling an ant colony behavior to parsing a sentence written in a temporal logic language.

Ant colony optimization algorithms (ACO) for solving combinatorial optimization problems involve the idea behind the ability of ants constructing shortest paths. We have used the same inspiration in order to solve thee different problems and figuring out different results:

(i) Traveling Salesman Problem: To understand the relationship between the parameters of ant colony optimization algorithm,

(ii) Single Machine Total Tardiness Problem: To compare and contrast the efficiency of the ACO with some well-known heuristics in the area.

(iii) 2D projected map problem: The problem was a real world problem, which can be defined as finding minimum distance path on a three dimensional landscape, and an application from pipeline design. However, we have used the 2D projection of the map in order to test ACO algorithms robustness on a new type of problem. We have proposed several modifications on novel ACO algorithms in order to improve the performance.

Starting from Chapter 4, the treatment on ACO has been changed. The main goal was, answering how we could deepen the understanding of the relationship between the members of a colony and the colony itself, which later on asserted as *modeling* and *verification.* Formal modeling is a very popular topic in theoretical computer science and has very diverse applications including biological systems. Therefore, we have moved to the formal modeling of ant colony behavior instead of using it. We have chosen one of recently developed process algebra, WSCCS, in order to establish formal models of ant colonies, since it has shown to be a very successful language in modeling biological systems.

Having a formal at hand brings no insight if one cannot examine the model. Therefore, we have again made a move to the notion of verification. Verification concerns proving or disproving the correctness of an abstract model against its formal specifications and is also one of the popular subjects in the area, both from software engineering side, and theoretical side.

Model checking which is a verification approach involves the overall behavior of model such that it is converted to a transition system, whose states represent the global states of the distributed system, and whose transition relation gives the possible evolutions of the system. Then it can be checked whether such transition system is a model of a temporal logic formula.

Although there is no considerable modification on novel syntax and semantics of WSCCS, to be able to benefit from every aspect of WSCCS we have proposed several extensions on formalisms involved in model checking as following:

(i) Utilization of Probabilistic Computation Tree Logic (PCTL) for the formulas of model checking, broadening its syntax and semantics so that one can formulate the queries including actions, yielding a new branching time logic, pCTL.

(ii) Made use of Labeled Kripke Transition Systems (LKTS), on which pCTL is to be interpreted; extending the definition of transition relation to cover the notion of probability, new transition system is LKTS*.

Incorporating all mentioned formalisms, a complete modeling and verification schema has been constructed.

The need for automated tools of verification has been started to arise when the systems getting more complex, which means, in turn a large increase in state space. Such systems are almost impossible be managed by hand because of likely errors. Therefore, we have offered a software tool, called WSCCS++, that lets the user to design his/her own model, to compile the model according to the syntactical rules of WSCCS, and perform model checking on the model as proposed.

8.2 Future Works

In correspondence with the two directions in dealing with the ant colony behavior in dissertation, there are different paths in future works as well. First one concerns ACO algorithms. Extending ACO algorithms for solving some interesting problems, for instance, Multimedia Information Retrieval problems, or improving ACO algorithms for solving big-sized combinatorial problems can be enumerated as possible future forks related with ACO algorithms.

The second path concerns modeling ant colony behavior. Modeling biological systems and attempting to extract some useful behavioral information is such an interesting area that one needs to be close both the biological face, and formal face-computer science. Thus, this dissertation can be expanded in both aspects. For instance, honey bees and termites are also known to be social insects. Modeling termites with the offered WSCCS++ tool would be an appealing exercise.

On the other hand, from the formal point of view, the model checking and/or verification approaches might be improved in order to decrease time requirements to handle large state spaces.

REFERENCES

Abdul-Razaq T.S., Potts C.N., & Van Wassenhove L.N. (1990). A survey of algorithms for the single machine total weighted tardiness scheduling problem. *Discrete Applied Mathematics*, 26: 235-253, 1990.

Alidaee B. & Ramakrishnan K.R. (1996). A computational experiment of COVERT-AU class of rules for single machine tardiness scheduling problem. *Computers and Industrial Engineering*, vol.30, no. 2, pp.201-209.

Andersen H., Stirling C., & Winskel G. (1994). A compositional proof system for the modal mu-calculus. *In Proceedings of 9th LICS*. IEEE Computer Society Press.

Baker K.R. (1974). *Introduction to sequencing and scheduling*, New York: John Wiley.

Bevier R.W., Hunt W.A., Moore J.S., & Young W.D. (1989). An approach to systems verification. *Journal of Automated Reasoning*, 5(4):411-428.

Biberstein O., Buchs D., & Guelfi N. (2001). Object-oriented nets with algebraic specifications: The CO-OPN/2 formalism, *Advances in Petri Nets on Object-Orientation*, G. Agha and F. De Cindio and G. Rozenberg (Eds.), LNCS: 2001, Springer Verlag, pp. 70-127.

Bilgen B., & Özkarahan I. (2002). Simulated annealing technique for single machine scheduling problem, *Proceedings of The 30th International Conference on Computers and Industrial Engineering*, pp. 95-100, Tinos Island, Greece.

Bolognesi, T. & Brinksma E. (1989). *Introduction to the ISO specification language LOTOS*, in: P. H. J. van Eijk, C. A. Vissers & M. Diaz, editors, The Formal Description Technique LOTOS, Elsevier Science Publishers North-Holland, pp. 23-73.

Bonabeau E., Dorigo M., & Theraulaz G. (1999). *Swarm intelligence: From natural to artificial systems* New York: Oxford University Press.

Boyer R. S., & Moore J. S. (1979). *A Computational Logic*, New York: Academic Press.

Boyer R. S., & Moore J. S. (1988). *A Computational Logic handbook*, New York: Academic Press.

Bradfield J., & Stirling C. (2001). Modal logics and mu-calculi: An introduction. In J. Bergstra, A. Ponse, and S. Smolka, editors, Handbook of Process Algebra. Elsevier.

Bullnheimer B., Hartl R.F., & Strauss C. (1999). Applying the ant system to the vehicle routing problem. In: Voss S., Martello S., Osman I.H., Roucairol C. (eds.), Meta-Heuristics: Advances and Trends in Local Search Paradigms for Optimization, Kluwer:Boston.

Bullnheimer B., Hartl R.F., & Strauss C. (1999). A new rank based version of the ant system: A computational study. *Central European Journal for Operations Research and Economics,* 7(1),25-38.

Burch J. R., Clarke E. M., McMillan K. L, Dill D. L., & Hwang.L.J. (1990). Symbolic model checking: 10^{20} states and beyond. In 5th Annual IEEE Symposium on Logic in Computer Science, pp. 428-439, Philadelphia, PA,

Burkart O., & Steffen B. (1994). Pushdown processes: Parallel composition and model checking. *CONCUR'94*, LNCS: 836. pp 98-113.

Burks A. (ed.) (1970). *Essays on cellular automata*. University of Illinois Press, Urbana, Illinois.

CelLab for Windows (n.d) Retrieved Feb 7, 2006 from http://www.fourmilab.ch/cellab/manual/clabwin.html

Clarke E.M., & Emerson E.A. (1981). Design and synthesis of synchronization skeletons using branching time temporal logic. *Proceedings of IBM Logics of Programs Workshop*, LNCS vol. 131, pp. 52-71.

Clarke E.M., & Grumberg O. (1987). Research on automatic verification of finite state concurrent systems. *Ann. Rev. Computer Science* (2):269-290.

Clarke E.M., & Kurshan R. (1996). Computer aided verification. *IEEE Spectrum* 33,6 pp.61-67.

Clarke E.M., Burch J., Long D., McMillan K.,& Dill D. (1994). Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 13, No. 4, pp. 401-424.

Clarke E.M., Emerson E.A., & Sistla A.P. (1986). Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems.* 8(2): 244-263.

Cleaveland R. & Sims S. (1996). The NCSU concurrency workbench, *Proceedings of the 8th International Conference on Computer Aided Verification*. LNCS vol. 1102.

Cleaveland R., Klein M., & Steffen B. (1992). Faster model checking for the modal mu-calculus. *Proceedings of Computer Aided Verification (CAV'92)*, Bochmann G.V. & Probst D.K. eds. LNCS vol. 663. pp. 410-422

Cleaveland R., Parrow J., & Stefen B. (1993). The concurrency workbench: A semantics based tool for the verification of concurrent systems. *ACM Transactions on Programming Languages and Systems* (15) 1. pp. 36-72.

CLIPS: A NASA Developed Expert System Tool, (1987). NASA Technical Briefs.

Clocksin W.F., & Mellish C.S. (2003). *Programming in prolog : Using the ISO standard* (5th ed.) Heidelberg: Springer.

Colmerauer A., & Roussel P. (1992). The birth of Prolog. Retrieved Feb 1, 2006 from http://www.lim.univ-mrs.fr/~colmer/ArchivesPublications/HistoireProlog/

Colorni A., Dorigo M., & Maniezzo V. (1992). Distributed optimization by ant colonies., F.Varela & P.Bourgine (Eds.) *Proceedings of the First European Conference on Artificial Life*, (134-142). Paris, France: Elsevier Publishing.

Colorni A., Dorigo M., Maniezzo V., & Trubian M. (1994). Ant system for job-shop scheduling. *JORBEL - Belgian Journal of Operations Research, Statistics and Computer Science*, *34*(1):39-53.

Constable R.L. (1986). *Implementing mathematics with the NURPL proof development system,* Prentice-Hall.

Cormen T., Leiserson C., & Rivest R. (1990). *Introduction to algorithms*. MIT Press.

Costa D., & Hertz A. (1997). Ants can color graphs. *Journal of the Operational Research Society*, *48*, 295-305.

CPN-AMI: Home Page, (n.d.) Retrieved Feb 7, 2006 from http://www-src.lip6.fr/logiciels/mars/CPNAMI/ .

Cpntools (n.d) Retrieved Feb 7, 2006 from http://wiki.daimi.au.dk/cpntools/cpntools.wiki .

Crauwels H.A.J., Potts C.N., & Van Wassenhove L.N. (1998). Local search heuristics for the single machine total weighted tardiness scheduling problems. *INFORMS Journal On Computing*, 10(3): 341-350.

CSML and MCB (n.d) Retrieved Feb 2, 2006 from http://www.cs.cmu.edu/~modelcheck/csml.html

CV: Introduction (n.d) Retrieved Feb 2, 2006 from http://www.cs.cmu.edu/~cmuvhdl/

Della Croce F., Tadei R., Baracco P., & Grosso A. (1996). A new decomposition approach for the single machine total tardiness scheduling problem, *Journal of the Operations Research Society*, 49. pp. 1101-1106.

Deneubourg J.-L., Aron S., Goss S., & Pasteels J. M. (1990). The self-organizing exploratory patterns of the Argentine ant. *Journal of Insect Behaviour 3*: 159-168.

Di Caro G., & Dorigo M. (1998). Ant colonies for adaptive routing in packet-switched communication networks. *Proceedings of PPSN-V, Fifth International Conference on Parallel Problem Solving from Nature*, LNCS, vol. 1498. pp. 673-698.

Di Caro G., & Dorigo M. (1998). AntNet: Distributed stigmergetic control for communication networks. *Journal of Artificial Intelligence Research*, 9:317-365.

Dorigo M., & Gambardella L.M. (1997). Ant colonies for the traveling salesman problem. *BioSystems*, *43*:73-81.

Dorigo M., & Gambardella L.M. (1997). Ant colony system: A cooperative learning approach to the traveling salesman problem. *IEEE Transactions on Evolutionary Computation,* 1(1):53-66.

Dorigo M. (1992) Optimization, learning, and algorithms, (In Italian) Politecnico di Milano, Italy, Ph.D. Thesis.

Dorigo M., Maniezzo V., & Colorni A. (1991). Positive feedback as a search strategy. Dip. Eletronica, Politecnico di Milano, Tech. Rep. 91-016.

Dorigo M., Maniezzo V., & Colorni A. (1996). The ant system: Optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man, and Cybernetics – Part B*, *26*(1): 29-42.

Dowek G., Felty A., Herbelin H, & Huet G. (1993). The COQ proof assistant user's guide. Technical Report 154, Inria-Rocquencourt.

Ekin E., & Yakhno T. (2001). A case study of adapting ant system to optimization problems. *Proceedings of TAINN'2001, Turkish Symposium on Artificial Intelligence and Neural Networks.* TRNC.

Elseaidy W., Cleaveland R., & Baugh J. (1996). Modeling and verifying active structural control systems. *Proceedings of the 1994 Real Time Systems Symposium*.

Emerson E.A. (1990). Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, volume B: Formal Models and Semantics,* chapter 14, pages 996–1072. Elsevier Science.

Emerson E.A.,& Halpern J.Y. (1986). 'Sometimes' and 'not never' revisited: On branching time versus linear time temporal logic. *JACM*, vol. 33, no.1, pp.151-178.

EvoCellLab-KyberWiki (n.d) Retrieved Feb 7, 2006 from http://kybkreis.org/wiki/index.php/EvoCell

Ferber J. (1999). *Multi agent systems an introduction to distributed artificial intelligence* (J. Ferber, trans.) New York: Addison Wesley. (Original work published 1995)

Fernandez J.C., Garavel H., Kerbrat A., Mateescu R., Mounier L., & Sighireanu M. (1996). CADP (CÆSAR/ALDEBARAN development package): A protocol validation and verification toolbox. *Proceedings of the 8th International Conference on Computer-Aided Verification*, LNCS vol. 1102.

Finin T., Weber J., Wiederhold G., Genesereth M., Genesereth M., McKay D., McGuire J., Pelavin R., Shapiro ., & Beck C. (1993). Specification of the KQML agent-communication language plus example agent policies and architectures. The DARPA Knowledge Sharing Initiative External Interfaces Working Group

Fisher M.L. (1976). A dual algorithm for the one-machine scheduling problem. *Mathematical Programming*, 11. pp. 293, 297.

Fogel D. (1993). Applying evolutionary programming to selected traveling salesman problems, *Cybernetics and Systems: An International Journal,* vol. 24, pp. 27–36.

Fredlund L. (1994). The timing and probability workbench: A tool for analyzing timed processes. Technical Report, Department of Computer Systems, Uppsala University.

Fry T.D., Vicens L., Macleod K., & Fernandez S. (1989). A heuristic solution procedure to minimize $\overline{T}$ on a single machine, *Journal of the Operational Research Society,* 40. pp. 293-297.

Gambardella L. M. & Dorigo M. (1997). HAS-SOP: An hybrid ant system for the sequential ordering problem. Technical Report. No. IDSIA 97-11, IDSIA, Lugano, Switzerland.

Gambardella L. M., Taillard E., & Dorigo M. (1999). Ant colonies for the quadratic assignment problem. *Journal of the Operational Research Society*, *50*:167-176.

Gambardella L.M., & Dorigo M. (1995). Ant-Q: A reinforcement learning approach to the traveling salesman problem. *Proceedings of ML-95, Twelfth International Conference on Machine Learning*, Morgan Kauffman, 252-260.

Gambardella L.M., & Dorigo M. (1996). Solving symmetric and asymmetric TSPs by ant colonies. *Proceedings of IEEE International Conference on Evolutionary Computation (ICEC'96)* pp. 622-627. IEEE Press 1996.

Gardner, M. (1970). The fantastic combinations of John Conway's new solitaire game "life". *Scientific American*, vol. 223, no. 4, pp 120-123.

Gardner, M. (1971). On cellular automata, self-reproduction, the garden of Eden and the game "life". *Scientific American*, vol. 224, no. 2, pp 112-117.

Garey M.R., & Johnson D.S. (1979). *Computers and intractability: A guide to the theory of NP-Completeness*, San Francisco, CA: Freeman.

Gerth R., Peled D., Vardi M.Y., & Wolper P. (1995). Simple on-the-fly automatic verification of linear temporal logic. *Proceedings of IFIP/WG 6.1 Symposium On Protocol Specification, Testing and Verification.*

Gilmore P.C. (1960). A Proof Method for Quantification Theory: Its justification and realization. *IBM Journal of Research and Development*, 4:28-35.

Glover, F. & Laguna, M. (1996). *Tabu search.* Dordrecht, Netherlands: Kluwer.

Gordon M. J. C., & Melham T. F. (1993). editors. *Introduction to HOL: A theorem proving environment for higher-order logic.* Cambridge University Press, Cambridge, UK.

Gordon M. J., Milner A. J., & Wadsworth C. P. (1979). Edinburgh LCF: A mechanized logic of computation, LNCS vol. 78.

Goss S., Aron S., Deneubourg J.-L.,& Pasteels J. M. (1989). Self-organized shortcuts in the Argentine Ant *Naturwissenchaften 76* 579-581.

Grune D., Bal H.E., Jacobs C.J.H., & Langendoen K.G. (2001). *Modern compiler design.* London: John Wiley & Sons.

Hansson H. A., & Jonsson B. (1989). A framework for reasoning about time and reliability. *Proceedings of 10th IEEE Real -Time Systems Symposium*, pp. 102–111, Santa Monica, Ca.,IEEE Computer Society Press.

Har'el Z., & Kurshan R.P. (1990). Software for analytical development of communications protocols. AT&T Bell Laboratories Technical Journal 69, 1, 45-59.

Hennessy M.C.B., & Milner R. (1985). Algebraic laws for nondeterminism and concurrency. *Journal of the ACM*, 32, pp. 137-161.

Hillston J. (1996). A compositional approach to performance modeling. Ph.D. Thesis, Cambridge University Press, Cambridge.

Hoare C.A.R. (1978). Communicating sequential processes, *Communications ACM* 21 (8) 666–677.

Holzmann G. (1991). *Design and validation of computer protocols*. New Jersey: Prentice-Hall, Englewood Cliffs.

Hunt W.A. (1987). The mechanical verification of a microprocessor design. Technical Report 6, Computational Logic Incorporated, Austin, TX.

INA: Integrated Net Analyzer Version 2.2 (Description, Manual, Download, …)(n.d) Retrieved Feb 7, 2006 from http://www.informatik.hu-berlin.de/~starke/ina.html

Java Pathfinder (n.d) Retrieved Feb , 2006 from http://javapathfinder.sourceforge.net

John Conway's Game of Life (n.d) Retrieved Feb 7, 2006 from http://www.bitstorm.org/gameoflife/

Kasami T. (1965). An efficient recognition and syntax-analysis algorithm for context-free languages. Scientific report AFCRL-65-758, Air Force Cambridge Research Lab, Bedford, MA.

Kaufmann M., & Moore J.S. (1994). Design goals for ACL2. Technical Report 101, Computational Logic, Inc, Austin, Texas.

Kaufmann M., & Moore J.S. (1996). ACL2: An industrial strength version of NQTHM. *Proceedings of COMPASS'96, The Eleventh Annual Conference on Computer Assurence*, pp. 23-24, Gaithersburg, MD.

Kaviola R. (1995). On modal mu-calculus and Büchi tree automata. *Inf. Proc. Letters* 54, 17-22.

Kowalski R.A. (1988). The early years of logic programming. *Communications of the ACM*, 31(1):38-42.

Kozen D. (1983). Results on the propositional mu-calculus. *Theoretical Computer Science*, 27, pp. 333-354.

Kripke S. (1963). Semantical analysis of modal logic. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, 9:67-96.

Kurshan R.P. (1994). *Computer aided verification coordinating processes*, Princeton University Press.

Lamport L. (1980). Sometimes is sometimes "not never"—on the temporal logic of programs, *Proceedings of 7th Annual Symposium on Principles of Programming Languages*, pp. 174-185.

Lamport L. (1983). What good is temporal logic. *Information Processing* 83:657-668.

Lamport L. (1986). A simple approach to specifying concurrent systems. Technical Report, DEC.

Larsen K.G., & Skou A. (1989). Bisimulation through probabilistic testing, *Proceedings of POPL'1989.*

Lawler E.L. (1977). A pseudopolynomial algorithm for sequencing jobs to minimize total tardiness. *Annals of Discrete Mathematics*, 1. pp 331-342.

Letz R., Schumann J., & Bayerl S. (1992). SETHEO: A high-performance theorem prover. *Journal of Automated Reasoning*, 8(2):183-212.

Lin S. (1965). Computer solutions of the traveling salesman problem. *Bell Systems Journal*, vol. 44, pp. 2245–2269.

Maniezzo V. & Colorni A. (1999). The ant system applied to the quadratic assignment problem. *IEEE Transactions in Knowledge and Data Engineering*, Volume 11 Issue 5 pages 769-778.

Maniezzo V., Colorni A., & Dorigo M. (1994). The ant system applied to the quadratic assignment problem. Tech. Rep. IRIDIA/94-28, Université Libre de Bruxelles, Belgium.

Maziero C.A (2000). *The ARP Tool,* Retrieved Feb 7, 2006 , from http://www.ppgia.pucpr.br/~maziero/diversos/petri/arp.html.

Milner R. (1989). *Communication and concurrency* Prentice Hall PTR.

Milner R. (1983). Calculi for synchrony and asynchrony, *Theoretical Computer Science* 25 267–310.

Modern cellular automata-live color cellular automata (n.d) Retrieved Feb 7, 2006 from http://www.collidoscope.com/modernca/

Morton T.E., & Pentico D.W. (1993). *Heuristic scheduling systems with applications to production systems and project management*. NY: John Wiley and Sons.

Morton T.E., Rachamadugu K.R., & Vepsalainen A.P.J. (1984). Accurate myopic heuristics for tardiness scheduling. Carnegie Mellon University, Working Paper, W.P., 36-83-84.

Müller-Olm M., Schmidt D., & Steffen B. (1999). Model-checking a tutorial introduction. Cortesi a. & File G. Editors, SAS'99, LNCS vol. 1694, pp. 330-354.

Osman I. H. & Kelly, J. P. (Eds.) (1996). *Meta-heuristics: theory and applications.* Dordrecht, Netherlands: Kluwer.

Panwalkar S.S., Smith M. L., & Koulamas C.P. (1993). A heuristic for the single machine tardiness problem. *European Journal of Operational Research* 70, 304-310.

Papadimitriou C.H., & Steiglitz K. (1982). *Combinatorial optimization algorithms and complexity*. New Jersey: Prentice-Hall.

Paulson L. C. (1988). Isabelle: The next seven hundred theorem provers. In E. Lusk and R. Overbeek, editors. *Proceedings of 9th International Conference on Automated Deduction (CADE)*, *Argonne, IL*, LNCS vol. 310, pp. 772-773.

Perrier J.Y., Sipper M., & Zahnd J. (1996). Toward a viable, self-reproducing universal computer. *Physica D*, vol. 97, pp 335-352.

Peterson L. (1981). *Petri net theory and the modelling of systems*, Prentice-Hall, Englewood Cliffs, New Jersey.

Petri C.A. (1962). Kommunikation mit automaten, (Communicating with automata)(in German) Ph.D Thesis, Technical University Darmstadt, Germany.

Pnueli A. (1985a). Applications of temporal logic to the specification and verification of reactive systems: A survey of current trends. LNCS vol. 224. 510-584.

Pnueli A. (1985b). Linear and branching structures in the semantics and logics of reactive systems. *Proceedings of the 12th ICALP*, pp. 15-32.

Pnueli A. (1977). The temporal logic of programs. *Proceedings of the 18th IEEE Smposium on the Foundations of Computer Science (FOCS-77)*, pp. 46–57. IEEE Computer Society Press.

Pnueli A. (1981). A temporal logic of concurrent programs. *Theoretical Computer Science*, 13:45-60.

Potts C.N., & Van Wassenhove L.N. (1982). A decomposition algorithm for the single machine total tardiness problem. *Operations Research Letters* 11 177-181.

Potts C.N., & Van Wassenhove L.N. (1991). Single machine tardiness sequencing heuristics, *IIE Transactions*, 23, pp. 346-354.

Prawitz D., Prawitz H., & Voghera N. (1960). A mechanical proof procedure and its realization in an electronic computer. *Journal of the ACM*, 7:102-128.

Prior A. N. (1957). *Time and modality*, Oxford: Clarendon Press.

Prior A. N. (1967). *Past, present and future*, Oxford: Clarendon Press.

Prior A. N. (1969) *Papers on time and tense*, Oxford: Clarendon Press

QPN (n.d) Retrieved Feb 7, 2006 from http://ls4-www.informatik.uni-dortmund.de/QPN/ .

Queille J., & Sifakis J. (1982). Specification and verification of concurrent systems in CÆSAR. In Proceedings of Fifth ISP.

Reinelt G. (1991). TSPLIB A traveling salesman problem library. *ORSA Journal on Computing*, 3:376- 384.

Rescher N., & Urquhart A. (1971). *Temporal logic*. Springer-Verlag, Wien, New York.

Robinson J. A. (1965). A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23-41.

Roscoe A. (1994). Model checking CSP. A.Roscoe (Ed.) *A Classical Mind: Essays in Honour of C.A.R. Hoare*, Prentice Hall.

Schoonderwoerd R., Holland O., Bruten J., & Rothkrantz L. (1997). Ant-based load balancing in telecommunications networks. *Adaptive Behavior*, 5(2):169-207.

Shankar N. (1993). Abstract datatypes in PVS. Technical Report SRI-CSL- 93-9, Computer Science Laboratory, SRI International, Menlo Park, CA.

Shankar N. (1994). Metamathematics, machines, and Gödel's proof. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge, UK.

Shankar N., Owre S., & Rushby J. M. (1993). The PVS proof checker: A Reference Manual. Computer Science Laboratory, SRI International, Menlo Park, CA.

Sims S. (1999). *The process algebra compiler user's manual*. Reactive Systems, Inc.

Sipser M. (1996). *Introduction to the theory of computation*, Course Technology.

Smith A. (1969). Cellular automata theory. Technical Report 2, Stanford Electronic Lab., Stanford University.

Stickel M. E. (1986). A prolog technology theorem prover. J. H. Siekmann. (Ed.), *8th International Conference on Automated Deduction (CADE),* LNCS:230*,* pp. 573- 587, Oxford, England: Springer-Verlag.

Stirling C., & Walker D. (1989). Local model checking in the modal mu-calculus. J. Diaz, & F. Orejas (Ed.) *Proceedings of TAPSOFT'89* LNCS: 351, pp. 369-383.

Stirling C. (1995). Local model checking games. *Proceedings of Concur'95* LNCS vol. 962, pp. 1-11.

Stützle T., & Hoos H. (1997). Improvements on the ant system: Introducing MAX-MIN system. *Proceedings of the International Conference on Artificial Neural Networks and Neural Networks*, Springer Verlag, pp. 245-249.

Stützle T., & Hoos H. (1998). $\mathcal{MAX}$-$\mathcal{MIN}$ ant system and local search for combinatorial optimization problems. S. Voss, S. Martello, I.H. Osman, & C. Roucairol, (Ed.), *Meta-Heuristics: Advances and trends in Local search paradigms for optimization*, pp. 313-329. Boston: Kluwer.

Sumpter D.J.T. (2000). From bee to society: An agent based investigation of honey bee colonies. *PhD Thesis*. University of Manchester, Institute Science and Technology, Manchester.

SyMP: Symbolic Model prover (n.d.) Retrieved Feb 2, 2006 from http://www.cs.cmu.edu/~modelcheck/symp.html

Szwarc W., & Mukhopadhyay S. (1996). Decomposition of the single machine total tardiness problem, *Operations Research Letters* 19, pp. 243-250.

The SMV System (n.d) Retrieved Feb 2, 2006 from http://www.cs.cmu.edu/~modelcheck/smv.html

Toffoli T., & Margolus N. (1987). *Cellular automata machines*. The MIT Press, Cambridge, Massachusetts.

Tofts C. (1994). Processes with probabilities, priority and time. *Formal Aspects of Computing, 6*. (5), 536-564.

van Benthem Jutting L. S. (1979). Checking Landau's 'Grundlagen' in the Automath System. Technical report, Mathematical Centre, Amsterdam, Mathematical Centre Tracts.

Vardi M.Y., & Wolper P. (1986). An automata-theoretic approach to automatic program verification. *In Proceedings of Logics in Computer Science*.

von Neumann, J. (1966). Theory of self-reproducing automata. University of Illinois Press, Illinois. Edited and completed by A. W. Burks.

Wang H. (1960a). Proving theorems by pattern recognition. *Communications of the ACM*, 4(3):229-243.

Watkins C.J.C.H., & Dayan P. (1992). Q-Learning. *Machine Learning*, 8:279-292.

Wilkerson L.J., & Irwin J.D. (1971). An improved algorithm for scheduling independent tasks, *AIIE Transactions*, 3, pp. 239-245.

WinPeSim (n.d) Retrieved Feb 7, 2006 from http://www.winpesim.de/

Wolfram S., (1985) Cryptography with Cellular Automata, *CRYPTO'85*.

Wolfram S. (1983). Statistical mechanics of cellular automata. *Reviews of Modern Physics*, vol. 55, no. 3, pp 601-644.

Wolfram S. (1984). Universality and complexity in cellular automata. *Physica D*, vol. 10, pp 1 - 35.

Wolfram S. (1984a). Cellular automata as models of complexity. *Nature*, vol. 311, pp 419-424.

Wolfram S. (1984b). Universality and complexity in cellular automata. *Physica D*, vol. 10, pp 1-35.

Wolfram S. (2002). *A new kind of science, Champaign*. IL: Wolfram Media.

Wos L., Overbeek R., Lusk E., & Boyle J. (1992). *Automated reasoning: Introduction and applications.*, New York: McGraw-Hill, Second edition.

Yakhno T., & Ekin E. (2002). Ant systems: Another alternative for optimization problems? Proceedings of ADVIS’2002, LNCS vol. 2457, pp 324-326.

Younger D.H. (1967). Recognition and parsing of context-free languages in time n^3. *Information and Control* 10(2): 189–208.

APPENDIX A

THE LANGUAGE OF WSCCS

Any process algebra as well as WSCCS consists of essentially three parts to be defined.

- Syntax, usually given as context free grammar, to define the agents, and the actions they perform. The basic features of WSCCS, weighing the actions, parallel agent composition, and communication are all described in syntactical rules.
- Semantics, defines the series of actions those are valid for any agent. By involving transitional semantics, WSCCS allows us to consider the models as finite automaton systems, that is we can draw transition graphs of each agent.
- Congruence and the equational theory, help on deciding whether or not two agents are equivalent in their behaviors, also checking how much similar agents they are, using algebraic context or syntactical checking respectively. Congruence involves the term bisimilarity, which is defined via an equivalence relation over two state transition systems.

A.1. WSCCS Syntax

The novel syntax presented in (Tofts, 1994) has been slightly modified.

- *Act* refers to the finite of atomic action symbols, with # being the unit action. For any $a_i \in Act$, $!a_i$ refers to the complementary action of a_i, which forms the basis of the communication. In case of communication, the resulting action is the unit action, #. Let L be any subset of the Act such that $L \subseteq Act$ and $\# \in L$.

  $Act = \{a_1, a_2, .., a_m\}$
- W stands for the set of weights, w_i denotes each element of W. $W = \{w_1, w_2, .., w_t\}$ with each $w_i = nw^k$:

- $n \in Z^{+}$ is the relative frequency of occurrence of an action with which the weight is associated,
- $k \in N$. w^{k} gives the action's priority.

If an agent is able to perform more than one action at any time, the actions whose priorities are equal to the highest priority may occur. The probability of actions with equal weights is then determined by their relative frequency. nw^{0} is denoted by n.

The following is the set of rules for calculating the weights in case of addition and multiplication with the assumption $k > k'$.

$$nw^{k} + mw^{k'} = mw^{k'} + nw^{k} = nw^{k} \quad \text{(Eq.A.1)}$$

$$nw^{k} + mw^{k} = mw^{k} + nw^{k} = \left(n+m\right) w^{k} \quad \text{(Eq.A.2)}$$

$$nw^{k} * mw^{k'} = mw^{k'} * nw^{k} = (nm) w^{k+k'} \quad \text{(Eq.A.3)}$$

- Agt denoting a set of agent variables, $A \in Agt$, $Agt = \{A_1, A_2, .., A_l\}$,
- $f : Act \rightarrow Act$, is the partial renaming function such that $f\left(\#\right) = \#$ and, $f\left(!a\right) = !f\left(a\right)$.
- I is an arbitrary indexing set over N, $I = \{1..n\}$,
- E is an agent expression which is formed by the following grammar:

The following is the WSCCS formal grammar

$G_{WSCCS} = \left(V, T, E, P\right)$ with

- V is a finite set of variables, $V = \{E, L, w, f, X\}$
- T is a finite set of terminal symbols, $T = Agt \cup Act \cup \{0,1,2,3,4,5,6,7,8,9\} \cup \{0\} \cup \{.,:,+,\times,<,\Theta,(,),[,], fix, -, *, /\}$
- $E \in V$ is the starting symbol,

- P is the finite set of production or substitution rules.

$$
\begin{aligned}
&E \to 0 \,|\, Agt \,|\, Act.E \,|\, E \times E \,| \\
&E \to E < L \,|\, prio(E) \,|\, f(E) \,| \\
&E \to \sum_{i \in I} w_i : E \,\Big|\, fix_j\left(\{X_i = E_i\} : i, j \in I\right) \\
&L \to \bigcup_{i \in I} a_i \\
&Agt \to A_1 \,|\, A_2 \,|\, ... \,|\, A_l \\
&Act \to a_1 \,|\, a_2 \,|\, ... \,|\, a_m
\end{aligned}
$$

Figure A.1 WSCCS formal grammar

A.2 WSCCS Operational Semantics

The semantics of any formal language allows unambiguous interpretation of a series of commands written in mentioned language. Transitional semantics of WSCCS, described in (Tofts, 1994) results the agents to be represented as finite state machines, which are later converted to transition graphs. In any transition graph, agents appear at nodes, and weights and actions appear on labels of each edge. Then, the congruence of two agents is investigated over the related transition graphs.

The semantics of WSCCS is presented in natural deduction style, where the conclusion is a transition relation that is true provided by the hypothesis is true. The hypothesis may contain one or more transition relations, and they belong to the WSCCS semantics rule set.

Before listing the whole WSCCS semantics, it is necessary to define the transition relations involved. There are two types of transitional relations; the least relation, and the least multi-relation, for the transitions caused by actions, and for the transitions caused by weights respectively.

- Least relation: $\to \subseteq Agt \times Act \times Agt$ written as: $A \xrightarrow{a} B$
- Least multi-relation: $\mapsto \subseteq bag\left(Agt \times W \times Agt\right)$ written as: $A \overset{w}{\mapsto} B$

  This is a weak relation, thus the relations of this kind may result in different outcomes, such as:

$$1:A+1:A+2:B$$

According to the least multi-relation, there are three possible transitions. However, by the addition rule of weights, the result should be equal to the

$$2:A+2:B.$$

Therefore, the following definition of weighted transition is given to have a consistent semantics.

An agent A can make a weight based transition to B if

$$A\overset{w}{\longmapsto}B=\sum\left\{w_i\,\middle|\,A\overset{w_i}{\longmapsto}B\right\}$$

Together with the operators defined in formal WSCCS grammar, there is another predicate involved in semantics.

$does_L(E)$: It is a least relation, such that evaluates true if the agent E can perform the actions from the set L. Usually, it is used to generate the behavior of restriction operator, "<".

$\dfrac{}{a.E \xrightarrow{a} E}$	A1	$\dfrac{}{\sum\{w_i : E_i : i \in I\} \overset{w}{\mapsto} E_i}$	W1
$\dfrac{E \xrightarrow{a} E' \times F \xrightarrow{b} F'}{E \times F \xrightarrow{ab} E' \times F'}$	A2	$\dfrac{E \overset{w}{\mapsto} E' \times F \overset{v}{\mapsto} F'}{E \times F \overset{wv}{\mapsto} E' \times F'}$	W2
$\dfrac{E \xrightarrow{a} E' \times F \overset{w}{\mapsto} F'}{E \times F \overset{w}{\mapsto} E \times F'}$	AW1	$\dfrac{E \overset{w}{\mapsto} E' \times F \xrightarrow{a} F'}{E \times F \overset{w}{\mapsto} E' \times F}$	AW2
$\dfrac{E \xrightarrow{a} E', a \in L}{does_L(E)}$	A3	$\dfrac{E \overset{w}{\mapsto} E', does_L(E')}{does_L(E)}$	W3
$\dfrac{E \xrightarrow{a} E', a \in L}{E < L \xrightarrow{a} E' < L}$	A4	$\dfrac{E \overset{w}{\mapsto} E', does_L(E)}{E < L \overset{w}{\mapsto} E' < L}$	W4
$\dfrac{E \xrightarrow{a} E'}{f(E) \xrightarrow{f(a)} f(E')}$	A5	$\dfrac{E \overset{w}{\mapsto} E'}{f(E) \overset{w}{\mapsto} f(E')}$	W5
$\dfrac{E_k\{fix_i(F)/X_{i:i\in I}\} \xrightarrow{a} E'}{fix_k(F) \xrightarrow{a} E'}$	A6	$\dfrac{E_k\{fix_i(F)/X_{i:i\in I}\} \overset{w}{\mapsto} E'}{fix_k(F) \overset{w}{\mapsto} E'}$	W6
$\dfrac{E \xrightarrow{a} E'}{prio(E) \xrightarrow{a} prio(E')}$	A7	$\dfrac{E \overset{nw^k}{\mapsto} E', \exists k', \forall E''\left((k \geq k') \wedge E \overset{nw^{k'}}{\mapsto} E''\right)}{prio(E) \overset{n}{\mapsto} prio(E')}$	W7

Figure A.2 Operational semantics of WSCCS. All of the rules respect to the operators, whose behaviors are described in syntax, except A6 and W6. The notation $E\{E_i / X_i : i \in I\}$ is used to state that substituting in E all occurrences of each X_i with the related E_i.

A.3 Congruence and Equational Theory

As an abstract term, congruence means, similarity between two objects. In WSCCS, congruence refers to the equivalence relation between two automata (the semantics allow us to consider the models as finite state machines), which is called bisimilarity.

A.3.1 Congruence via Bisimilarity

Two kinds of bisimulation will be described, direct bisimulation and relative bisimulation. The bisimulations described here are based on the accumulation techniques of (Larsen & Skou, 1989).

2.6.5.17 A.3.1.1 Direct Bisimulation

Roughly speaking, if any two agents are said to be direct bisimulation equivalent, that means they perform exactly the same set of transitions both weight based, and action based, and the resulting agents of those transitions are identical as well.

The following definitions are given by (Tofts, 1994).

Definition A.1: P is an agent, and S is being a set of agents,

- $P \overset{w}{\mapsto} S$ such that $w = \sum \left\{ w_i : P \overset{w_i}{\mapsto} Q, \exists Q \in S \right\}$; agent P is said to perform a weight based transition with weight w, to a set of agents S, if some agents in S, are the resulting agents of this transition(least multi relation).
- $P \overset{a}{\rightarrow} S \Leftrightarrow \exists Q \in S \wedge P \overset{a}{\rightarrow} Q$; agent P performs an action based transition to set of agents S, with action a, if and only if there exists some elements of S which are resulting agents of the transition (least relation).

Definition A.2: An equivalence relation $R \subseteq \Pr \times \Pr$, Pr is the equivalence classes of P with respect to R, is a *direct bisimulation* if $(P,Q) \in R$ implies for all $S \in \Pr/R$ that:

- $(\forall w \in W)\left(P \overset{w}{\mapsto} S \Leftrightarrow Q \overset{w}{\mapsto} S \right)$; if for all weight values, the agent P and Q are performing the weight based transitions, and yielding the same set of agents for the same weight values,
- $(\forall a \in Act)\left(P \overset{a}{\rightarrow} S \Leftrightarrow Q \overset{a}{\rightarrow} S \right)$**;** if for all elements of action set, the agents P and Q are performing action based transitions, and resulting in same set of

agents for the same actions,then, two agents, P and Q are said to be direct bisimulation equivalent, and written as $P \overset{d}{\sim} Q$, if an equivalence relation R exists between them.

2.6.5.18 A.3.1.2 Relative Bisimulation

Since direct bisimulation gives a strong equivalence if exists, relative frequency notion cannot be covered, but instead total frequency is considered. However, we want the following two agents are evaluated to be same.

$1:\#.A+2:\#.B$ and $3:\#.A+6:\#.B$

Thus, it is required to loosen the definition of equality, which shall be called *relative bisimulation*.

Definition A.3: An equivalence relation $R \subseteq \text{Pr} \times \text{Pr}$, Pr is the equivalence classes of P with respect to R, is a *relative bisimulation* if $(P,Q) \in R$ implies that:

- $(\exists c_1, c_2 \in Z^+)$ such that $(\forall S \in \text{Pr}/R)$ and $(\forall w, v \in W)$,

  $P \overset{w}{\mapsto} S \Leftrightarrow Q \overset{v}{\mapsto} S \wedge (c_1 w = c_2 v)$

- $(\forall S \in \text{Pr}/R)$ **and** $(\forall a \in Act)$, $P \overset{a}{\rightarrow} S \Leftrightarrow Q \overset{a}{\rightarrow} S$

Then, two agents, P and Q are said to be *relative bisimulation equivalent*, and written as $P \overset{r}{\sim} Q$, if there exist a relative bisimulation between them.

A.3.2 Equational Theory

Instead of proving that all agents in an equivalence class, either relative bisimulation, or abstract bisimulation, making the same transitions, one can compare the definitions of agents at syntactical level, by using equational theory. That is, equational theory is used to decide equality of agents in term of syntax.

Most of the symbols used in list of equational rules were introduced before, the additional terms are defined in following:

-
- $d_L\ (E)$ in `Res2` that does the same job as $does_L\ (E)$ in semantics, i.e., checks whether an agent can perform an action or not. It is recursively defined as following:
  - $d_L(E) \Leftrightarrow a \in L$
  - $(\exists k \in I) \wedge d_L(E_k) \Rightarrow d_L\left(\sum_{i\in I} E_i\right)$
- $\max_w W$ in Prio2 is the maximum power of w occurring in W, a set of weights.
- $\mathcal{N}(w_i)$ in Prio2 is a projection from weights such that each w_i is of the form nw^k by definition :

  $\mathcal{N}\left(nw^k\right) \overset{def}{=} nw^0$

Equational Rules of WSCCS

- Sum: $\sum_{i\in I} w_i : E_i = \sum_{j\in J} v_j : E_j \begin{cases} \text{there is surjection} f : I \to J \text{ with} \\ v_j = \sum\{w_i \mid i \in I \wedge f(i) = j\}, \\ \text{and for all } i \text{ with} f(i) = j \text{ then } E_i = \mathrm{E_j} \end{cases}$
- Exp1: $a.E \times b.F = ab.(E \times F)$
- Exp2: $a.E \times \sum_{j\in J} v_j : F_j = \sum_{j\in J} v_j : (a.E \times F_j)$
- Exp3: $Q \times P = P \times Q$
- Exp4: $\left(\sum_{i\in I} w_i : E_i\right) \times \left(\sum_{j\in J} v_j : F_j\right) = \sum_{(i,j)\in I\times J} v_i w_j : \left(E_i \times F_j\right)$

- Res1: $(a.E) < L = \begin{cases} a.(E < L) & \text{if } a \in L \\ 0 & \text{otherwise} \end{cases}$
- Res2: $\left(\sum_{i\in I} w_i : E_i\right) < L = \sum_{j\in J} w_j : \left(E_j < L\right)$ where $J = \left\{ i \in I \mid d_A\left(E_i\right)\right\}$
- Prio1: $prio(a.E) = a.prio(E)$
- Prio2: $prio\left(\sum_{i\in I} w_i : E_i\right) = \sum_{j\in J} \mathcal{N}\left(w_j\right).prio\left(E_j\right)$ where

  $J = \left\{ i \in I \mid \exists n \text{ and } w_i = nw^{\max_w(\{w_i\})} \right\}$
- Ren: $\sum_{i\in I} w_i : E_i = \sum_{i\in I} nw_i : E_i$ where $n \in \mathcal{P}$