7-9-2009

Symbolic planning for heterogeneous robots through composition of their motion description languages

Wenqi Zhang

Wenqi Zhang

Candidate

Mechanical Engineering

Department

This dissertation is approved, and it is acceptable in quality and form for publication on microfilm:

Approved by the Dissertation Committee:

, Chairperson

Accepted:

Dean, Graduate School

Date

Symbolic Planning for Heterogeneous Robots through Composition of their Motion Description Languages

by

Wenqi Zhang

B.S., Mechanical Engineering,
University of Tongji, 2001

DISSERTATION

Submitted in Partial Fulfillment of the
Requirements for the Degree of

Doctor of Philosophy
Engineering

The University of New Mexico

Albuquerque, New Mexico

May, 2009

Dedication

To my parents and sister who always encourage me to pursue the best education, and to my dearest husband Jun Wei for his continuous support.

Acknowledgments

I would like to thank my advisor, Professor Herbert Tanner, for his support, guidance, inspiration and for introducing me to this interesting research.

I would like to thank the dissertation committee: Professor Chaouki T. Abdallah, Professor Rafael Fierro, Professor Ron Lumia, Professor John Russell for your advisement during my studies.

Deep thanks to my friends: Andres Cortez, Zhenhua Chen and Jason Sanchez for their support.

Symbolic Planning for Heterogeneous Robots through Composition of their Motion Description Languages

by

Wenqi Zhang

ABSTRACT OF DISSERTATION

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Doctor of Philosophy
Engineering

The University of New Mexico

Albuquerque, New Mexico

May, 2009

Symbolic Planning for Heterogeneous Robots through Composition of their Motion Description Languages

by

Wenqi Zhang

B.S., Mechanical Engineering,
University of Tongji, 2001

Ph.D, Engineering, University of New Mexico, 2009

Abstract

This dissertation introduces a new formalism to define compositions of interacting heterogeneous systems, described by extended motion description languages (MDLes). The properties of the composition system are analyzed and an automatic process to generate sequential atom plan is introduced. The novelty of the formalism is in producing a composed system with a behavior that could be a superset of the union of the behaviors of its generators.

As robotic systems perform increasingly complex tasks, people resort increasingly to switching or hybrid control algorithms. A need arises for a formalism to compose different robotic behaviors and meet a final target. The significant work produced to date on various aspects of robotics arguably has not yet effectively captured the interaction between systems. Another problem in motion control is automating the process of planning and it has

been recognized that there is a gap between high level planning algorithms and low level motion control implementation. This dissertation is an attempt to address these problems.

A new composition system is given and the properties are checked. We allow systems to have additional cooperative transitions and become active only when the systems are composed with other systems appropriately. We distinguish between events associated with transitions a push-down automaton representing an MDLe can take autonomously, and events that cannot initiate transitions. Among the latter, there can be events that when *synchronized* with some of another push-down automaton, become active and do initiate transitions.

We identify MDLes as recursive systems in some basic process algebra (BPA) written in Greibach Normal Form. By identifying MDLes as a subclass of BPAs, we are able to borrow the syntax and semantics of the BPAs merge operator (instead of defining a new MDLe operator), and thus establish closeness and decidability properties for MDLe compositions.

We introduce an instance of the sliding block puzzle as a multi-robot hybrid system. We automate the process of planning and dictate how the behaviors are sequentially synthesized into plans that drive the system into a desired state.

The decidability result gives us hope to abstract the system to the point that some of the available model checkers can be used to construct motion plans. The new notion of system composition allows us to capture the interaction between systems and we realize that the whole system can do more than the sum of its parts. The framework can be used on groups of heterogeneous robotic systems to communicate and allocate tasks among themselves, and sort through possible solutions to find a plan of action without human intervention or guidance.

Contents

List of Figures

List of Tables

Glossary

Q	a finite set of states
Φ	a finite set of alphabet
δ	the transition function
q_0	the start state
F	the set of accept states
Ω	stack alphabet
V	a finite set of variables
Ψ	terminals
R	rules
S	start variable
$u(x,t)$	feedback control laws
$\xi(y,t)$	boolean functions
N	non-terminal symbols in MDLe
η	terminals in MDLe

Glossary

S	start symbol in N;
R	rules to create MDLe strings
Σ	the stack alphabet in MDLe
Z_0	start symbol in stack
$\lfloor\!\lfloor$	left merge operator
$\|$	merge operator
$\mathcal{H}$	function to generate common events
MDLe	extended Motion Description Language
LTL	Linear Temporal Logic
CTL	Computation Tree Logic
MA	Maneuver Automaton
ACP	Algebra of Communicating Processes
BPA	Basic Process Algebra
GNF	Greibach Normal Form
CFG	Context Free Grammar
CCS	Calculus of Communicating Systems
CSP	Communicating Sequential Processes
TCSP	Theoretical Communicating Sequential Processes
DPDA	Deterministic Pushdown Automaton

Chapter 1

Introduction

1.1 Motivation

The significant advances in robot control and motion planning notwithstanding, we still lack a theory that integrates features of modern control theory with automated reactive decision-making. This is due in part to the scope and difficulty of the problem, and in part to the limited expressive power of current models. The complexity issues involved in discrete planning are well known [111]. It has also been recognized that there is a gap between such discrete high level planning algorithms, and low level motion control implementation (see [69] and the references therein). The approach in [69] attempts to bridge this gap using linear temporal logic (LTL) to describe the task to be performed in a formalism similar to natural language, and then translate the logic formula into an automaton that gives rise to a hybrid controller.

A need arises for a methodology to "stitch" together different controlled robotic behaviors in a way that a final objective is ultimately met. The need for a "standard" language for motion control is becoming pressing as future control specifications need to capture these features for control to be able to meet modern engineering and societal challenges.

A standard language for motion control should be able to manage complexity; allow differential equation control interrupted by discrete logic; allow one to write reusable and robust software.

A formal language is a language that is defined by mathematical or machine processable formulas. We want to find a language along the lines of [85]

> In a definitive calculus there should be as few operators as possible, each of which embodies some distinct and intuitive idea, and which together give completely general expressive power.

We attempt to utilize an existing motion control "meta-language" [63, 80] to abstract low level controllers —implemented in any possible programming language— into elementary behaviors in a way to facilitate high level planning.

In the rest of this chapter, we give a review of the prior work in related areas.

1.2 Informal review of related formalism

We begin with some concepts in the theory of computation.

Real computers are too complicated to allow us to set up a mathematical theory of them directly. Instead we use a *computational model* to represent an idealized computer. The simplest model is called *finite state machine* or *finite automaton*. It can represent various devices. For example, an automatic door is such a device. The controller has two states: *open* and *closed*. The controller moves between these two states according to input conditions: 1) when a person is in front of the door (front); 2) when a person is at the rear of the door (rear); 3) there is no person on both sides (neither); 4) there are persons on both sides (both). When the controller is in the *open* state and receives an input "neither," it transfers to state *closed*, otherwise remains in *open*. According to different situations, we

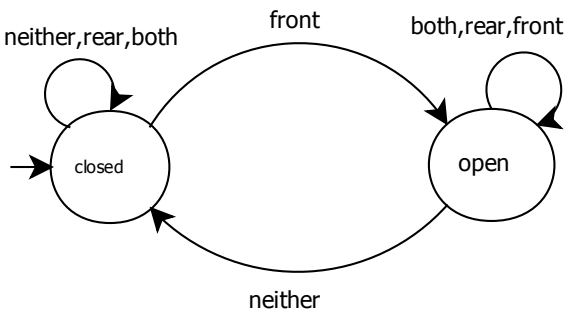

Figure 1.1: State diagram of automatic door. The two states are *open* and *closed*. Front, rear, both, neither are four different inputs. The automaton representing the automatic door change states according to inputs.

can draw a state diagram in Figure 1.1. This automaton only needs the memory to record which states the door is in to work properly. There are other devices can be represented in finite automata, such as elevators, watches *etc*, which need limited memories.

A sequence of input conditions forms a string, such as "front, both, rear, both". The group of all strings that an automaton can accept is called the *Language*. In the automatic door example, if the controller is in state *open* and what to keep in this state all the time, then input strings "front, rear, front, both", "front, rear, both, both","rear, front, both, front, both" are all acceptable strings. The set of all these strings is the language that the automatic door automaton can accept. A language is called *regular* if some finite automaton accepts it.

Grammars are the rules we need to follow to generate a language. In English languages, grammars are used to make sentences. For example, $sentence \rightarrow noun-phase+verb-phase$ is the basic grammar used in English. Similar to human languages, grammars are also used in computer science to describe languages.

Finite automata have limited memories. Pushdown automata are used when we want to represent more complex devices. A pushdown automaton has a *stack* which provides additional memory compared to a finite automaton. For example, 0^n1^n represents all the

strings that has equal number of zeros and ones, such as "000111", "0011". No finite automata can generate strings of the form 0^n1^n because there is not enough memory to remember the number of zeros and ones in the string. But an automaton with a stack can do this job: Once we read a 0, we push it onto the stack. As long as we see 1, we pop a 0 off the stack. If the input is finished exactly when the stack is empty, the automaton accepts the input string, otherwise it rejects it. This simple example shows that pushdown automata can recognize more languages than finite automata. The language that a pushdown automaton can recognize is called *context-free language*.

1.3 Prior Work

In this section, we provide a review on the prior related work in the area of symbolic motion planning, hybrid systems, process algebras, and decidability.

1.3.1 Symbolic planning

Motion and behavior planning is an essential part in control theory. Planning in robotics typically concentrates on determining a path from start position to end position in different environments.

Motion Description Languages were introduced by Roger Brockett [17–19]. The main idea came from automata theory. Brockett stated that a computer controlled device should include :

a) An interpretive language capable of describing the tasks to be done.

b) An interpreter/mechanism which accepts such descriptions and executes the task.

c) Suitable applications programs for facilitating the generation of the description of specific tasks in terms of the interpretive language.

A Motion Description Language (MDL) is an ideal language which can satisfy the above requirements. It can be expressiveness and distinguish between low level feedback rules (part b) and higher level planning (part c). The MDL-device was first introduced in [17]. Brockett described a general solution to the problem of motion control, which included these three parts: a motion description language, an interpreter mechanism, and an applications program. The language is device independent and applicable to a wide variety of systems. Experimental results show that with a suitable change of coordinate maps, this mode of programming can be supported and capable of generating a wide variety of motions. This approach is an effort that tries to elucidate the complexity of robotic systems, and translate control algorithms into robust and reusable software [78–80].

MDL subsequently evolved to MDLe (extended Motion Description Language). Manikonda, Krishnaprasad and Hendler [78] encoded and integrated aspects of modern control theory approaches with the reactive planning systems that rely on sensors and actuators. Comparing to Brockett's earlier definition, Manikonda *et al.* introduced sensor-driven trigger functions into MDL atoms. Now the basic building block of the languages (called atom) is defined as (U, ξ, T), ξ is considered as an interrupt or trigger. Another difference from Brockett's model is that input scaling is brought into the picture. A hybrid architecture planner that can be used in path planner and obstacle avoidance is also introduced.

A MDLe was being used in symbolic feedback control [62], generating symbolic feedback control sequences for navigating a sparsely described and uncertain environment. It was also used to generate software [32] that can control robots and other dynamic systems using automatically generated, high-level, symbolic control programs. The MODEbox can extract high-level control programs from observed behaviors and then produce symbolic control laws that can be executed on mobile robots to mimic the observed behavior. Hristu *et al.* described the "MDLe engine," a software tool that implemented the MDLe lan-

guage in [61, 64]. They designed a basic compiler/software foundation for writing MDLe code and provided a brief description of the MDLe syntax, implementation architecture, and functionality. Hristu and Anderson used the motion description language to represent language-based directions in directed graph [63], reducing the complexity of the map and making navigation programs robot-independent. Murry and Deno [33] developed a methodology for description of hierarchical control of robot systems in a manner of interpreted language, which was capable of describing a large class of robot systems under a variety of single level and distributed control schemes.

Other methods for performing symbolic motion and behavior planning have also been used. The Maneuver Automaton [39, 40, 42] is one of them. Maneuver automata are finite automata that produce sequences of predetermined maneuvers for unmanned vehicles. The approach can be considered an extension of [81] and [68]. A Maneuver Automaton (MA) [39] is a tuple $M = (\Sigma, Q, \delta, q_0, F)$. Σ is the maneuver alphabet, a finite collection of maneuvers. A maneuver is defined as a primitive that begins and ends at steady-state motion conditions. Motion plans are described as the concatenation of a number of well-defined motion primitives, selected from a finite library. Languages generated by maneuver automata are regular languages and a maneuver sequence can be derived from a corresponding regular grammar. Maneuver-based motion planning is a method for time-invariant dynamical control systems with symmetries, such as mobile robots and autonomous vehicles, under a variety of differential and algebraic constraints on the state and on the control inputs. Frazzoli, Dahleh and Feron [39] also checked the reachability properties of this language and give algorithms for the solution of a class problems. Most of the properties of Maneuver Automata and of the corresponding languages were analyzed in [39, 42].

Petri nets are another modeling formalism utilized in planning. The formalism was introduced in 1962 by Carl Adam Petri [99–102], at the age of 13!

A Petri Net [92, 103] is a graphical tool for the description and analysis of concurrent

processes in distributed systems. A Petri net is a directed bipartite graph in which includes space nodes, transition nodes, and directed arcs connecting places with transitions. Arcs run between places and transitions. The places from which an arc runs to a transition are called the input places of the transition; the places to which arcs run from a transition are called the output places of the transition. Petri Nets are a useful tool for describing and studying information processing systems that are characterized as being concurrent, asynchronous, distributed, parallel, nondeterministic, and/or stochastic [92]. The properties of Petri Nets that were reviewed in [92] include the reachability problem, liveness, and boundness properties.

Holt [55–57] applied Petri Nets in his project *Information System Theory of Applied Data Research*. Peterson published the first book on Petri Nets [98] in 1981, summarizing most of the developments and applications on Petri Nets. Later on, Petri Nets were applied on different systems: distributed-software systems [7, 77]; distributed-database systems [35,94]; concurrent and parallel programs [43,44]; dataflow computing systems [67,115]. Special modifications and restrictions of petri nets are added on these systems to suit to particular application.

Graph theory has also played a role in motion planning. Ehrig and Courcelle [36], [25] introduced the basic definition of Graph grammars. Klavins, Lipsky, and Ghrist [66], [65] used graph grammars to model and direct concurrent robotic self-assembly and similar self-organizing processes. They generated an acyclic graph to synthesize a binary grammar (rules involve at most two parts), and a general graph to synthesize a ternary grammar (rules involve at most three parts). With this apparatus they were able to model and direct distributed robotic assembly. They explored and analyzed the reachability and stability properties of the resulting robotic systems that were described by graph.

1.3.2 Hybrid systems

A hybrid system is a dynamic system that exhibits both continuous and discrete dynamic behavior: a system that is represented by differential equations and discrete logic rules that interact with each other. Hybrid systems have been used in a lot of applications, such as automated highway systems [60, 75, 120], air-traffic management systems [72, 76, 119], manufacturing systems [97], chemical process [38], robotics [5, 116], embedded automotive controllers [10].

Sastry and Lygeros introduced the basic definitions in hybrid systems and give a formal definition in [74], [73]. Existence of executions is an issue in hybrid systems. Since hybrid automata allow one to model a very wide variety of physical phenomena, it is possible to produce unreasonable models or one for which solutions do not exist. Hybrid automata can also exhibit multiple executions for a single initial state. Given a system represented in hybrid automaton, a need frequently arises to check if the system satisfied a certain property, such as controllability or reachability. The idea of hybrid systems has inspired a lot of work and a great deal of research has been done in these two directions.

Alur, Courcoubetis, and Halbwachs provided decidability and undecidability results for classes of linear hybrid systems in [2], and they showed that standard program analysis techniques can be adapted to linear hybrid systems. They also presented approximation techniques for dealing with systems for which iterative procedures do not converge. Henzinger [49] classified hybrid automata according to what questions about their behavior can be answered algorithmically. The classification revealed structure on mixed discrete-continuous state spaces that was previously studied on purely discrete state spaces only. Asarin, Bournez and Dand [6] suggested a novel methodology for synthesizing switching controllers for continuous and hybrid systems whose dynamics were defined by linear differential equations. They proposed an abstract algorithm that solved the problem by an iterative computation of reachable states. Decarlo and Branicky [31] gave a survey on

the major results in the (Lyapunov) stability of finite-dimensional hybrid systems and then discussed the stronger, more specialized results of switched linear (stable and unstable) systems. Davoren and Nerode [27] offered a synthetic overview of the use of logics and formal methods in the analysis of hybrid systems. McClamroch and Kolmanovsky [83] provided an overview of developments on design of hybrid controllers for continuous-time control systems that can be described by linear or nonlinear differential state equations. They introduced hybrid controllers in the form of a switching control architecture and provided a summary of control approaches that utilize this control architecture.

1.3.3 Process Algebras

Algebraic process theory was developed in the early seventies. One of the main proponents of studying the semantics of parallel programs in the early seventies was Hans Bekic. In [11] Bekic addressed the semantics of what he calls "quasi-parallel execution of processes"

> Our plan to develop an algebra of processes may be viewed as a high-level approach: we are interested in how to compose complex processes from simpler (still arbitrarily complex) ones.

Bekic used global variables in his book [12]. He contributed a number of basic ingredients to the emergence of process algebra, but offered no comprehensive theory.

A central figure in the history of process algebra is Robin Milner. Milner developed his process theory Calculus of Communicating Systems (CCS) over the years 1973 to 1980, and published his book [85] in 1980. The earliest publications of parallel composition were in [86], [87]. Following Bekic's work [11], Milner [84, 89, 90] introduced *flow graphs*, and static laws were stated for operators. In [48, 88], Milner gave the basic definition of CCS, observational equivalence and strong equivalence. Also, the Hennessy-Milner logic was introduced, which provided a logical characterization of process equivalence.

We find the first introduction of a complete process algebra in the book [85] including a set of equations and a semantical model. Milner summarized and updated his work, publishing the book *Communication and Concurrency* [91]. Divid Park followed Milner's idea and introduced the formulation of bisimulation in [96]. Plotkin did more research on the *structural operational semantics* and developed a simple and direct method for specifying the semantics of programming languages in [105] and [106].

Tony Hoare is another important contributor to process algebra. In his paper [53], he introduced the language Communicating Sequential Processes (CSP). The language has synchronous communication and is a guarded command language. To solve the problem of deadlock behavior, Theoretical CSP (TCSP) was introduced in [20]. In that language, a silent step like τ can be eliminated using two alternative composition operators. Hoare's book *Communicating Sequential Processes* [54] is a good review of CSP.

Under the guidance of Jan Bergstra, Bergstra and Klop started work on process algebra and published a paper [13], in which the term "process algebra" appeared for the first time. The word "Process" refers to behavior of a system. A system can be anything from a piece of software to a human being. The word "Algebra" implies that the approach in dealing with behavior is algebraic and axiomatic. The formal definition of Basic Process Algebra [13], [15] includes several parts: *atomic actions* $A = \{a, b, c, \ldots\}$; binary operators Δ_{BPA}; axioms E_{BPA}. Together it is denoted simply as a couple in the form

$$\mathrm{BPA} = (\Delta_{\mathrm{BPA}}, E_{\mathrm{BPA}}).$$

Process algebras offer a framework for the definition of operations on systems such as parallel composition, alternative composition (choice) and sequential composition (sequencing). Moreover, by equational reasoning , we can do verification on systems, that is, checking if a system satisfies a certain property. In [14], a process algebra was extended to account for communication thus yielding the theory Algebra of Communicating Processes (ACP).

Some other process theories can be mentioned. Research on temporal logic started in [107]. A partial order process theory was given in [82]. Following Hoare's work, Rem did more research in the direction of trace theory [110]. Hennessy introduced recursive processes, continuous algebras, communicating processes in his book [47]. De Bakker and Zucker [29, 30] used tools from metric topology to show how operations upon processes can be defined conveniently, and solved the problems encountered in the study of concurrency.

1.3.4 Decidability

System verification is an important issue in embedded control systems. A system is called decidable if we can find a computation procedure that can decide whether the system satisfies the desired properties in a finite number of steps. Decidability is a central issue in algorithmic analysis for hybrid systems.

In the context of system verification, abstraction refers to an attempt to construct finite, and computable partitions of the state space of the hybrid system, in such a way so that in the smaller system all the properties of interest are preserved. The idea is that one can check the desired property on the abstract system, and carry the conclusions over to the original (concrete) system.

There are classes of hybrid systems that can be abstracted to finite systems, such as systems represented in linear temporal logic (LTL) and computation tree logic (CTL) [37]. Bisimulation relations are typically used to construct these abstractions [27], [21], [26]. A bisimulation is a binary relation between state transition systems. Two systems are bisimular if they match each other's moves, behaving in the same way in the sense that one system simulates the other and vice-versa. Despite success in showing that in some special types of hybrid systems, namely timed automata [4], multirate automata [3], [93], rectangular automata [51], [108], O-minimal hybrid systems [70], bisimulation-based abstractions can

be decidable, this is not the case for general hybrid systems.

Patrick Cousot and Radhia Cousot introduced an abstract interpretation of programs in [26]. The abstract interpretation was used to describe computations in another universe of objects, so that the result of abstract execution gave some informations on the actual computations. Alur, Courcoubetis, Henzinger and Pei-hsin Ho [3,4] considered hybrid automata as a generalization of timed automata and presented two semidecision procedures for verifying safety properties of piecewise-linear hybrid automata. Caines and Yuan-Jun Wei [21] defined a set of dynamically consistent hybrid partition machines associated with a continuous system. Based on the properties of these abstracted machines, the definition and properties of hierarchical-hybrid control systems are presented. Nicolin, Olivero, Sifakis, and Yovine [93] introduced the forward and backward symbolic simulation, which can be used in symbolic model-checking. Henzinger, Kopke, Puri, and Varaiya [51] identified a boundary between decidability and undecidability for the reachability problem of hybrid automata. They introduced an algorithm based on the construction of a timed automaton that contained all reachability information about a given initialized rectangular automaton. The translation guaranteed the termination of symbolic procedures for the reachability analysis of initialized rectangular automata. Pappas and Sastry introduced the abstraction map and formalize the notion of abstraction of continuous systems in [95]. Lafferriere, Pappas, and Yovine [70] extended the decidability properties for classes of *linear hybrid systems*, which were introduced as hybrid systems with linear vector fields in each discrete location. Piovesan, Tanner, and Abdallah [118] introduced the notion of Finite Time Mode Abstraction to relate a hybrid automaton to a timed automaton that preserved the stability and reachability properties of the former.

Stirling [117] used concurrency theory to show the decidability of bisimulation equivalence for deterministic pushdown automaton (DPDA). DPDA, belong to a subclass of pushdown automata, which have restrictions on their basic transitions:

if $pS \xrightarrow{a} q\alpha$ and $pS \xrightarrow{a} r\beta$ then $q = r$ and $\alpha = \beta$.

if $pS \xrightarrow{a}$ and $a \in \mathbb{A}$ then $\neg(pS \xrightarrow{\epsilon})$

where p, q, r are states, $a \in \mathbb{A} \cup \epsilon$ is the automaton's alphabet and α, β are sequence of stack symbols. Moreover, it is assumed that the length of α, β is less than 3 and the ϵ transitions can only pop the stack. Stirling used the theory developed in [112] and [45] to prove the decidability of DPDA.

Model checking [23, 28, 50, 71] is the process of determining if a given property holds true in a particular system. It is widely used in verifying the correctness and stability of large, complicated pieces of software. Pioneering work in model checking of specifications expressed in some temporal logic was done by Clarke and Emerson [23] in 1981 and by Queille and Sifakis [109]. Model checking software tools are now available. HYTECH [50] is a tool for the automated analysis of embedded systems. KRONOS [28] is a software tool built with the aim of assisting designers of real-time systems to verify whether their designs meet the specified requirements. AMC [104] is a model checker for non-linear hybrid systems based on an abstraction/refinement framework.

1.4 Problem Statement

In our work we describe heterogeneous systems by the different MDLes that express their behavior. This is done to incorporate discrete logic and differential equation-based control laws.

A definition for the classical notion of system composition for discrete event system is given in [22]. To analyze the physical interaction between heterogeneous systems, and verify the properties of cooperative system, we need a formal mathematical framework. The proposed framework should be able to capture the actions that are completed by the cooperation of the single systems. It should allow automated motion and task planning for collections of heterogeneous robotic systems. To allow behavior planning, we want

an algorithm that can give us the plan automatically, which means given initial state and destination, the algorithm should give us the detailed sequence of actions step by step.

1.5 Overview of Approach

We start by assuming the existence of a set of designed behavioral primitives, and we proceed by developing a framework that dictates how these behaviors are sequentially synthesized into plans that drive the system into a desired state. In that sense, our behavioral primitive forms an alphabet of actions. A new composition system is given and the properties are checked. We distinguish between events associated with transitions a push-down automaton representing an MDLe can take autonomously, and events that cannot initiate transitions. Among the latter, there can be events that when *synchronized* with some of another push-down automaton, become active and do initiate transitions. Then we identify MDLes as recursive systems in some basic process algebra (BPA) written in Greibach Normal Form. We propose a simple context-free grammar that generates MDLes and then we use the machinery available for BPAs to formally define a composition operation for MDLes at the level of grammars. The main difference of our composition operation is the appearance in the composed system of events (transitions) not enabled in the generators: the composed system can behave in ways its generators cannot. By identifying MDLes as a subclass of BPAs, we are able to borrow the syntax and semantics of the BPAs merge operator (instead of defining a new MDLe operator), and thus establish closeness and decidability properties for MDLe compositions. We introduce an instance of the sliding block puzzle as a multi-robot hybrid system, to ensure that our formulation captures the possible interaction between heterogeneous robot systems.

1.6 Contributions

MDLes have been criticized for not capturing interaction between systems. This dissertation addresses this issue and introduces a new framework of composed MDLe. We also prove closeness and decidability properties for MDLe compositions. In the new framework, each primitives represent some distinct and intuitive idea, and together give more expressive power compared to its components. We utilize an existing motion control language to abstract low level controllers into elementary behaviors in a way to facilitate high level planning. The new formalism can be applied to different areas. For example, we can compose auditory and visual systems together to decide the accurate position of any objects; we can compose UAVS, Kheperas and allow them communicate and allocate tasks among themselves, and sort through possible solutions to find a plan of action without human intervention or guidance.

Automating the process of planning has traditionally been a problem in the realm of artificial intelligence. In the sliding block puzzle example, we automate the process of planning and the atom sequence can be generated automatically according to the initial and final state.

1.7 Organization

In Chapter 2 we briefly review the mathematical machinery used in this dissertation, in an effort to facilitate the technical discussion that follows. In Chapter 3 we offer some details on how to construct a framework for MDLe composition. Chapter 4 presents the transformation from extended Motion Description Language to Basic Process Algebra and demonstrates why in the systems we consider, language equivalence is decidable and what bisimulation properties are preserved in the composition. Chapter 5 presents an analysis of a sliding block puzzle example using the proposed framework, when the puzzle

is viewed as a multi-robot hybrid system. This example show how our framework captures physical interaction, and why the composed system can have richer behavior compared to its component subsystems. Chapter 6 summarizes the results.

Chapter 2

Technical Background

In this chapter, we briefly introduce mathematical machinery that is necessary to analyze heterogeneous cooperative systems from a formal language perspective. This chapter is organized as follows: Section 2.1 introduces the automata theory and gives the definition of finite automata and pushdown automata; Section 2.2 presents the definition of regular language and context-free language; extended Motion Description Languages (MDLe) are introduced in Section 2.3 and it is shown that MDLe is a context-free language; the axioms and operational semantics of Process Algebra are introduced in Section 2.4; Section 2.5 gives a definition of classical composition of systems.

2.1 Automata Theory

An automaton is a conceptual device that is capable of representing a language according to well defined rules. It is often used to describe the operation of machines. The discussion that follows deals with the definitions and properties of these mathematical models of computation.

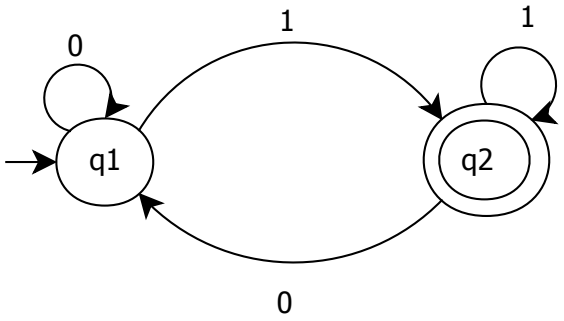

Figure 2.1: A two-state finite automaton. The machine starts in start state q_1 and proceeds according to the given input alphabet. For example, given input string 1011, the machine starts in q_1, then proceeds to q_2 after reading 1, then to q_1 when reading 0, to q_2 when reading 1, and stay in q_2 when reading 1. This string is accepted because q_2 is an accept state. It is shown that this machine accept all strings that end in 1.

Finite automata [113] can coarsely model computers with a limited amount of memory. A finite automaton has several parts. It has a set of states and rules for going from one state to another; it has an input alphabet that indicates the allowed input symbols; it has a start state and a set of accept states, the states at which the machine can proceed and successfully end. Figure 2.1 is an example of a finite automata which has two states [113].

Definition 1 ([113])**.** *A finite automaton is a 5-tuple* $(Q, \Phi, \delta, q_0, F)$*, where*

1. Q *is a finite set of* states*,*
2. Φ *is a finite set of symbols called the* alphabet*,*
3. $\delta : Q \times \Phi \rightarrow Q$ *is the* transition function*,*
4. $q_0 \in Q$ *is the* start state*, and*
5. $F \subseteq Q$ *is the* set of accept states.

Pushdown automata are similar to finite automata, but have an extra component called the *stack*. The stack provides additional memory and allows a pushdown automaton to ac-

cept strings that do not belong to a regular language. The formal definition of a pushdown automaton is similar to that of a finite automaton, except for the stack.

Definition 2 ([113]). *A pushdown automaton is a 6-tuple* $(Q, \Phi, \Omega, \delta, q_0, F)$.

1. Q *is a finite set of* states,
2. Φ *is a finite set called the* input alphabet,
3. Ω *is the* stack alphabet,
4. $\delta : Q \times \Phi \rightarrow Q$ *is the* transition function,
5. $q_0 \in Q$ *is the* start state, *and*
6. $F \subseteq Q$ *is the* set of accept states.

As we can see, automata are good computation models to represent simple discrete devices. There are also other models to describe the same behavior and this is the topic of the next section.

2.2 Expressions and Languages

Describing a finite automaton by state diagram like that at Figure 2.1 is not possible in some cases when the diagram is too big to draw or if the description depends on some unspecified parameter. In these cases, we use a formal description to specify the machine, which is called a *regular expression*. The regular expression is an algebraic description that can define exactly the same languages finite automata can describe. A regular expression consists of constants/variables and operators. These operators are: *union*, *concatenation*, and *star*. They are used in regular expressions in the following rules [59] :

- If E and F are regular expressions, then $E + F$ is a regular expression denoting the union of $L(E)$ and $L(F)$. That is, $L(E + F) = L(E) + L(F)$.
- If E and F are regular expressions, then EF is a regular expression denoting the concatenation of $L(E)$ and $L(F)$. That is, $L(EF) = L(E)L(F)$.
- If E is a regular expression, then E^* is also a regular expression. That is, $L(E^*) = (L(E))^*$.
- If E is a regular expression, then (E), a parenthesized E, is also a regular expression, denoting the same language as E. Formally; $L((E)) = L(E)$.

A regular language is usually described by a regular expression and can be accepted by a deterministic finite state machine (a finite automaton).

Definition 3 ([113]). *A language is called a regular language if some finite automaton recognizes it.*

Furthermore, it is known that:

Lemma 1 ([59]). *Every language defined by one of the finite automata is also defined by a regular expression. Every language defined by a regular expression is defined by one of the finite automata.*

A grammar is a precise description of a formal language, which is a set of strings over some alphabet. A formal grammar consists of a set of variables, a set of terminals, a set of rules for transforming strings and start variable.

Context-free grammars are more powerful than regular grammars in the sense that context-free grammars have a rule of the form $X \rightarrow \alpha$, where X is a nonterminal and α is a nonempty string of terminals and nonterminals. The languages generated by context-free grammars are called the context-free languages. They include all the regular languages.

Definition 4 ([113]). *A context-free grammar is a 4-tuple* (V, Ψ, R, S)*, where*

1. *V is a finite set of* variables*,*
2. Ψ *is a finite set, disjoint from V, of symbols called* terminals*,*
3. *R is a finite set of* rules*, with each rule being a variable and a string of variables and terminals, and*
4. $S \in V$ *is the* start variable*.*

Consider the grammar $G = (\{S\}, \{0, 1\}, R, S)$. The *variable* is S, *terminals* are $0, 1$ and *start variable* is S. The set of rules, R, is

$$S \rightarrow 0S1|\epsilon \tag{2.1}$$

This grammar can generate the language 0^n1^n for $n \geq 0$. It is known that:

Lemma 2 ([59]). *If a language is context free, then some pushdown automaton recognizes it. If a pushdown automaton recognizes some language, then the language is context free.*

In the first two sections of this chapter, the basic computation models are introduced. Now we need a suitable model for robot programming of motion control and this is done in the next section. We review the definition of such a formal language for robots and some special forms of this language.

2.3 Formal Languages for Robots

One of the important practical challenges in motion control is the implementation of theoretical tools into software that will allow the system to interact effectively with the physical world. The work on *Motion Description Language (*MDL*)* [17–19] has been an effort to

formalize a general purpose robot programming (meta) language that allows one to capture both logic and differential equations. *Extended* MDL *(*MDL*e)* is "a device-independent programming language for hybrid motion control, which allows one to compose complex, interrupt-driven control laws from a set of simple primitives, and a number of syntactic rules" [63, 80].

Every MDLe string consists of a control part, an interrupt part, and the special symbols ")", "(", and ",". Consider a robotic system, generically described in the form of the following dynamics

$$\dot{x} = f(x, u), \quad x \in \mathbb{R}^n, u \in \mathbb{R}^m \tag{2.2a}$$

$$y = h(x), \quad y \in \mathbb{R}^p, \tag{2.2b}$$

where x is the state of the system, u the control input, and y the measurable output. Let U be a finite set of feedback control laws for (2.2a),

$$u(x, t) : \mathbb{R}^n \times \mathbb{R} \to \mathbb{R}^m,$$

and B a finite set of boolean functions ξ of the output variable y and time $t \leq T \in \mathbb{R}_+$

$$\xi(y, t) : \mathbb{R}^p \times \mathbb{R} \to \{0, 1\}.$$

The basic element of MDLe is an *atom,* denoted (u, ξ), where u is a control law selected from U, and ξ is the *interrupt* selected from set B.

To *evaluate* or *run* an atom (u, ξ), means to apply the input u to (2.2a) until the interrupt function ξ evaluates to one ($\xi = 1$). An MDLe *plan* is composed of a sequence of atoms. For example, evaluating the plan $a = ((u_1, \xi_1), (u_2, \xi_2))$ means that the system state x, flows along $\dot{x} = f(x, u_1)$ until $\xi_1 = 1$, and then along $\dot{x} = f(x, u_2)$ until $\xi_2 = 1$. Plans can also be composed to generate higher order strings, as follows

$$b = ((u_3, \xi_3), a, (u_4, \xi_4)).$$

In [64] it is shown using the pumping lemma that MDLe is not a regular language, and it is suggested is context-free. Context-free languages are generated by context-free grammars. We define a context-free grammar $G = (N, \eta, R, S)$ so that it generates a motion description language $\text{MDLe} = \{(u, \xi) : u \in U, \xi \in B\}$, in the following way [64]:

- N is the finite set of non-terminal symbols E, where E is a valid *variable*;
- $\eta = \{u_i, \xi_i, (,), , \}$ is the finite set of terminals, which are the atoms of MDLe,
- S is the start symbol in N;
- R is the rules by which we create MDLe strings:

$$S \to E \tag{2.3a}$$
$$E \to EE \tag{2.3b}$$
$$E \to (u_i, \xi_i) \tag{2.3c}$$
$$E \to (E, \xi_i) \tag{2.3d}$$
$$E \to \emptyset \tag{2.3e}$$

Rule 2.3d is called "encapsulation" [64], which is essentially a while-structure, and gives MDLe its context-free character. It is known that a context-free grammar (CFG) can always be expressed in a special convenient form:

Definition 5 ([113])**.** *A context-free grammar is in* Chomsky normal form *if every rule is of the form*

$$A \to BC$$
$$A \to a,$$

where a is any terminal and A, B, and C are any variables, with the exception that B and C may not be the start variable, S. In addition, we permit the rule $S \to \emptyset$.

Another form of representation, sometimes considered a variation of the Chomsky normal form, is the Greibach normal form:

Definition 6 ([59])**.** *A context-free grammar in which every production rule is of the form $A \to a\alpha$, where A is a variable, a is a terminal, and α is a possibly empty string of variables, is said to be in* Greibach normal form *(*GNF*). If, moreover, the length of α (in symbols) does not exceed 2, we say that the context-free grammar is in restricted Greibach normal form.*

It is known that [59] every context-free grammar can be transformed into an equivalent grammar in Greibach normal form.

Pushdown automata and context-free grammars are equivalent in power [113]. Both are capable of describing the class of context-free languages. Following the description in [113], we can convert MDLe grammar into a pushdown automaton, which allow us to conveniently switch between presentations.

Definition 7. $P = (N, \eta, \Sigma, \Gamma, \delta, S, Z_0)$ *where*

- N *is the set of* states*, defined the same as* variables *in G;*
- $\eta = \{u_i, \xi_i, (,), , \}$ *is the set of* enabled events*, associated with transitions in P;*
- $\Sigma = N \cup \eta$ *is the stack alphabet;*
- $\Gamma : E \to \Gamma(E)$ *is the active event relation;*
- $\delta : E \times \eta \to E$ *is the transition function, $\delta(x, \eta) = y$ means that there is a transition labeled by event η from state x to y;*
- $S \in N$ *is the start state, defined the same as the start state in G;*
- Z_0 *is the start symbol in stack;*

Motion Description Languages are introduced in this section. An MDLe is a robot programming language that can capture both logic and interrupt-driven controll laws. It has been proved to be a context-free language and there is an equivalent pushdown automaton presentation. In next section, we introduce another computation model: Basic Process Algebra. The reason to introduce this model is that we want to borrow the syntax and semantics of the BPA merge operator and establish the properties for MDLe compositions.

2.4 Basic Process Algebra

A "process" is essentially the behavior of a dynamical system. The word "algebra" indicates that we take an algebraic/axiomatic approach when reasoning about behaviors.

A basic process algebra (BPA) [9] is a mathematical structure consisted of set of constants, $A = \{a, b, c, \ldots\}$, called *atomic actions*, a set Δ_{BPA} of two binary operators on these constants, the alternative composition $+$ and the sequential composition $\cdot$, and a set of axioms E_{BPA} that determines the properties of the operations on the atomic actions. When the set of atomic actions, A, is assumed known, a basic process algebra is denoted simply as a couple in the form

$$\mathrm{BPA} = (\Delta_{\mathrm{BPA}}, E_{\mathrm{BPA}}).$$

The set Δ_{BPA} is sometimes called "signature," while set E_{BPA} is called "equation" set (hence the symbols). The theory associated with a BPA is considered to be parameterized by the set A, which is specified according to the particular application.

The symbol $\cdot$ denotes sequential composition, and is typically omitted. We usually write xy instead of $x \cdot y$. We assume that $\cdot$ binds stronger than $+$, thus $(xy) + z = xy + z$ (brackets omitted). However, note that the brackets cannot be omitted in $x(y + z)$.

The set E_{BPA} consists of five axioms (or equations), appearing in Table 2.1. By com-

$x + y = y + x$	A1
$(x + y) + z = x + (y + z)$	A2
$x + x = x$	A3
$(x + y)z = xz + yz$	A4
$(xy)z = x(yz)$	A5

Table 2.1: The axioms of a BPA.

posing atomic actions according to Table 2.1, we produce more complex *processes*. Any such process, is an element of some algebra satisfying the axioms of BPA. All the processes that are produced in this way make up the set P. The axioms of Table 2.1 determine when two processes can be considered equal.

The axiom system of Table 2.1 is the core of a variety of more extensive process axiomatizations:

- $x \cdot y$ is the process that first executes x, and upon completion of x, process y starts.
- $x + y$ is the process that either executes x, or executes y (but not both).

A BPA does not enable us to prove that $a(cd + bd) = acd + abd$.

Note that this definition of a process algebra is purely "algebraic." Just as in the case of finite state machines, processes are identified by the set of action sequences they admit. Thus, in the above, a process is to be understood as the system that produces a given set of action sequences of the form $abaacdda \ldots$. Other authors [52] prefer to include a set Atom of atomic processes or *atoms*. The set Proc of *processes* contains all terms in the free algebra over Atom generated by sequential composition and disjunction. Then a process algebra is defined by a finite set Π of productions of the form

$$X \xrightarrow{a} P, \tag{2.4}$$

where $X \in \mathrm{Atom}$, $a \in A$, and $P \in \mathrm{Proc}$. The semantics of the above production is as follows: atomic process X performs action a and evolves into process P. The action

relations are presented in Table 2.2. In the Table, when we write $x \xrightarrow{a} y$, where x and y are processes, and a an atomic action, we mean that process x evolves into process y after the atomic action a is executed.

$a \xrightarrow{a} \surd$	R1
$a \xrightarrow{a} x' \Rightarrow x + y \xrightarrow{a} x'$ and $y + x \xrightarrow{a} x'$	R2
$x \xrightarrow{a} \surd \Rightarrow x + y \xrightarrow{a} \surd$ and $y + x \xrightarrow{a} \surd$	R3
$x \xrightarrow{a} x' \Rightarrow xy \xrightarrow{a} x'y$	R4
$x \xrightarrow{a} \surd \Rightarrow xy \xrightarrow{a} y$	R5

Table 2.2: The operational semantics of BPA.

It is said that a relation is true if and only if it can be derived from this table. The symbol $\surd$ stands for successful termination. Thus, writing $x \xrightarrow{a} \surd$ we mean that process x can terminate by executing action a.

Note the distinction between the relation operator ($\rightarrow$) and sequential composition ($\cdot$): the fact that $x \xrightarrow{a} y$ does not imply that $y = x \cdot a$, since a is an action executed as x runs, not after it is completed. The only thing that can be inferred about action a is that it is an action that process x can execute.

Now we restrict our attention to a special type of basic process algebras, with slightly finer semantics. The additional properties of this type of systems enable us to define composition more comfortably, and prove the decidability of language equivalence for the systems produced by means of composition.

Definition 8 ([8]). *A* recursive equation *over a basic process algebra is an equation of the form $X = s(x)$, where X is a variable that can take values in P and $s(x)$ is a term over the basic process algebra containing X, but no other variable.*

A set of recursive equations give rise to a specification:

Definition 9 ([8]). *A recursive specification E over a basic process algebra is a set of recursion equations over the basic process algebra.*

By this, we mean that we have a set of variables $V = \{x_0, \cdots, x_n\}$, and an equation of the form $X = s_x(V)$ with $x \in V$, where s_x is a term over the basic process algebra containing variables from the set V. The set V contains one distinguished variable called the root variable, usually the first variable in the textual presentation x_0.

A variable in V is called guarded in a given term, if it is proceeded by an atomic action:

Definition 10 ([8]). *Let s be a term over a basic process algebra, containing a variable X.*

- *An occurrence of X in s is said to be* guarded*, if s has a subterm of the form $a \cdot t$, where a is an atomic action, and t a term containing this occurrence of X; otherwise this occurrence of X in s is said to be* unguarded*.*
- *A term s is* completely guarded *if all occurrences of all variables in s are guarded. A recursive specification E is completely guarded if all right hand sides of all equations of E are completely guarded terms.*

For example, in the expression

$$s_1 = a \cdot X \cdot X + Y \cdot b + a \cdot (b + X)$$

every occurrence of X is guarded, but Y occurs unguarded. Therefore, s_1 is not completely guarded. However, the equation

$$s_2 = c(a \cdot X + Y \cdot b \cdot Y)$$

is guarded — in this case Y is guarded by c.

Definition 11 ([8]). *If a system E of recursion equations is guarded and without brackets, then each recursion equation is of the form*

$$X_i = \sum_j a_j \cdot \alpha_j,$$

where α_j is a possibly empty product (sequential composition) of atoms and variables. Now if, in addition, α_j is exclusively a product of variables, E is said to be in Greibach normal form *(*GNF*), analogous to the same definition for context-free grammars. If each α_j in E has length not exceeding 2, E is in* restricted *Greibach normal form.*

The process $x \| y$ is the process that executes process x and y in parallel. Notice that we do not assert that the first action has terminated when the second one starts. This can depend on the implementation of a process. The left merge operator, $\lfloor\!\lfloor$, describes two processes that occur in parallel, in a way similar to $\|$, but with the restriction that the first step must come from the process on the left of the expression. With the new operators, the BPA axioms are expanded as follows:

$x + y = y + x$	A1
$(x + y) + z = x + (y + z)$	A2
$x + x = x$	A3
$(x + y)z = xz + yz$	A4
$(xy)z = x(yz)$	A5
$x \| y = x \lfloor\!\lfloor y + y \lfloor\!\lfloor x$	M1
$a \lfloor\!\lfloor x = ax$	M2
$ax \lfloor\!\lfloor y = a(x \| y)$	M3
$(x + y) \lfloor\!\lfloor z = x \lfloor\!\lfloor z + y \lfloor\!\lfloor z$	M4

Table 2.3: The BPA axioms, expanded with the introduction of merge ($\|$) and left merge ($\lfloor\!\lfloor$) operators.

With the new binary operators, the action relations are enriched as shown in Table 2.4.

In this section we review the syntax and semantics of the BPA. The definition of recursive, guarded equations are given. The merge operator is introduced, which is a preparation for property checking of a composed system. The definition of parallel composition is given in next section, which inspires our work on a new framework for MDLe composition in Chapter 3.

$a \xrightarrow{a} \surd$	R1
$a \xrightarrow{a} x' \Rightarrow x + y \xrightarrow{a} x'$ and $y + x \xrightarrow{a} x'$	R2
$x \xrightarrow{a} \surd \Rightarrow x + y \xrightarrow{a} \surd$ and $y + x \xrightarrow{a} \surd$	R3
$x \xrightarrow{a} x' \Rightarrow xy \xrightarrow{a} x'y$	R4
$x \xrightarrow{a} \surd \Rightarrow xy \xrightarrow{a} y$	R5
$x \xrightarrow{a} x' \Rightarrow x\|y \xrightarrow{a} x'\|y$ and $y\|x \xrightarrow{a} y\|x'$	R6
$x \xrightarrow{a} \surd \Rightarrow x\|y \xrightarrow{a} y$ and $y\|x \xrightarrow{a} y$	R7
$x \xrightarrow{a} x' \Rightarrow x\lfloor y \xrightarrow{a} x'\|y$	R8
$x \xrightarrow{a} \surd \Rightarrow x\lfloor y \xrightarrow{a} y$	R9

Table 2.4: The action relations of BPA, expanded using the composition operators.

2.5 Classical Composition of Systems

We understand cooperative heterogeneous systems as systems that contain different kinds of hardware and software sub-systems. These sub-systems can be represented by different kind of automata; they can also run on different platforms. Ideally the sub-systems will cooperate (compose) to solve problems. Heterogeneous systems are hard to analyze for a variety of reasons [34]:

> They employ different languages and hardware/software platforms; synchronization, concurrency, and failures present difficult programming challenges; and source code may not be available, and some components in the distributed system may have different capabilities than other components.

When we say systems can be composed, we mean that some events in one automaton can be *synchronized* with some of another automaton, become active and initiate transitions, while other, not necessarily synchronized, events cause transitions in one system independently of the other. Synchronization implies a common interrupt function, a way for composed systems to know that they need to execute a certain action simultaneously. It is not necessary that the events from different automata start and terminate at the same time. If needed, two events can overlap in time and have different duration.

Parallel composition is often called synchronous composition, and this operation models one form of joint behavior of a set of systems that operate concurrently. According to [22], the parallel composition of G_1 and G_2 is defined as:

Definition 12. $G_1 \| G_2 = (V_1 \times V_2, E_1 \cup E_2, R, \Gamma_{(1\|2)}, (S_1 \times S_2))$*, where*

1. V_1 *and* V_2 *are the variable sets of single systems;*
2. E_1 *and* E_2 *are the terminal sets of single systems;*
3. *the rule set is*

$$R((v_1, v_2), e) := \begin{cases} (R_1(v_1, e), R_2(v_2, e)) & \textit{if } e \in \Gamma_1(v_1) \cap \Gamma_2(v_2) \\ (R_1(v_1, e), v_2) & \textit{if } e \in \Gamma_1(v_1) \setminus E_2 \\ (v_1, R_2(v_2, e)) & \textit{if } e \in \Gamma_2(v_2) \setminus E_1 \\ \textit{undefined}, & \textit{otherwise} \end{cases}$$

4. $\Gamma_{1\|2}(v_1, v_2) = [\Gamma_1(v_1) \cap \Gamma_2(v_2)] \cup [\Gamma_1(v_1) \setminus E_2] \cup [\Gamma_2(v_2) \setminus E_1]$*, which is the active events set for composed system;*
5. S_1 *and* S_2 *are the start variables of single systems.*

In this chapter, we reviewed the definition and basic properties of finite automata, pushdown automata, regular language, context-free languages, MDLes, basic process algebra, and the composition of systems. In the classical composition, two systems can only cooperate on common events. These events can generate transitions in their own systems irrespectively of what happens in the other, the systems resulting from the composition is more restricted than any of the components, because, in addition, these common events need to be synchronized. The composition we envision is a system with a richer behavior, which can exhibit behaviors that none of its component can. Modifying the definition to allow this is the main objective of the next chapter.

Chapter 3

System Composition

In this chapter, we set up a new framework for system composition. We separate the events of the single system into potential events and enabling events. Enabling events can be events that provide the information required to make a transition, but not the ability. Potential events can be events that provide the capability to take a transition, but not the required information. The events on which the automata can act must offer both the ability and the knowledge required to take a transition; they must be both potential and enabling. Toward this end we introduce a relation which groups all possible common events of the composed system, and a new definition of composed MDLe is given in both grammar and automaton formats. Using this framework, behaviors of the single systems can be sequentially synthesized into plans that drive the system into a desired state.

3.1 Cooperation opportunities

In the classical composition of Definition 12, a common event, that is, an event in $E_1 \cap E_2$ can only be executed if the two automata both execute it simultaneously. *Private* events are not subject to such a constraint and can be executed whenever possible. In this kind

of interconnection, a component can execute its private events without the participation of the other component, and a common event can only happen if both components can execute it. However, it could be the case that some inactive events in one automaton might be activated as a result of the physical interaction with the system represented by the other automaton. For example, a single auditory system can determine the direction to a sound producing object; a single visual system can provide the accurate position of the object only if there is line of sight. But if we compose these two systems, we can decide on the position of any object, no matter there is an occlusion or not, and regardless of its direction. The auditory system can provide the direction information to the visual system; and the visual system can establish line of sight to decide the accurate position. This is the same as how human beings decide the position of an object using ears (auditory systems) and eyes (visual systems).

To exploit this opportunity, we separate the events of each automaton into potential events and enabling events, shown in Figure 3.1. Potential and enabling events complement each other. Enabling events may provide to the other component system the opportunity to make a transition, but not the ability. They can trigger a new transition, as if giving it "the green light". Potential events represent capability on the part of the system which they are associated with, in order to take a transition, but not the opportunity. The events on which the automata can act must offer both the "ability" and the "opportunity" to take a transition; they must be both potential and enabling.

Toward this end, we introduce a way to group the potential events and enabling events of different automata, and reveal when transitions need to be synchronized. Given two automata P_1 and P_2, we define a relation $\mathcal{H}$ (Figure 3.2) on the set $(\eta_1 \cup \eta_2)$ as the collection of events on which P_1 and P_2 should be synchronized.

The set $\mathcal{H}$ contains all common events, and is defined so that it includes three different components: $\mathcal{H}_{\mathcal{I}} \cup \mathcal{H}_{\mathcal{II}} \cup \mathcal{H}_{\mathcal{III}}$

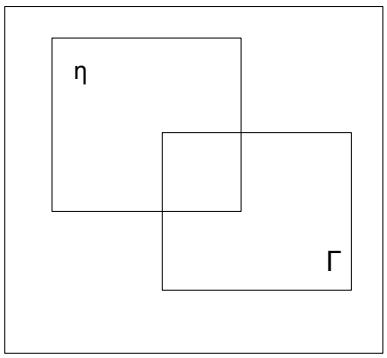

Figure 3.1: Relations between η and Γ. η is the set of events that system has power to do in potential, Γ is the set of events that system knows how to do, $\Gamma \cap \eta$ represent the events that the system can act by itself.

1. $\mathcal{H}_{\mathcal{I}} \triangleq (\Gamma_2 \cup \eta_1) \setminus (\Gamma_2 \cup \eta_2) \setminus (\Gamma_1 \cup \eta_1)$, (part I in Figure 3.3), which is the common events that system 1 can activate with "help" from system 2;

2. $\mathcal{H}_{\mathcal{II}} \triangleq (\Gamma_1 \cup \eta_2) \setminus (\Gamma_2 \cup \eta_2) \setminus (\Gamma_1 \cup \eta_1)$, (part II in Figure 3.3), which is the common events that system 2 can activate with "help" from system 1;

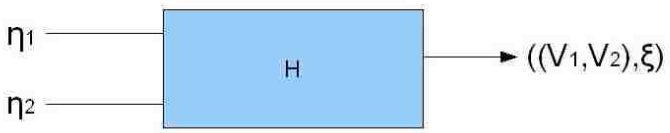

Figure 3.2: $\mathcal{H}$ is a relation, that associates two events in different automata that could be synchronized. It can be viewed also as a function taking events from two single systems are inputs, and giving the composed common events as outputs.

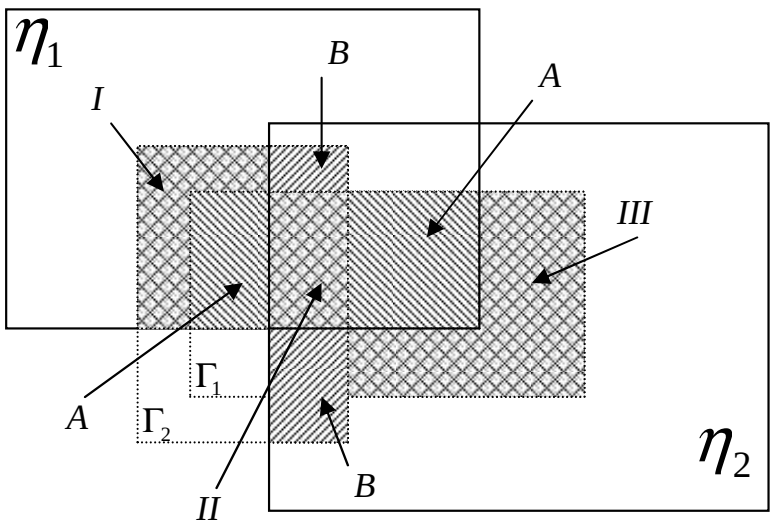

Figure 3.3: The collection of enabling and potential events. set A includes the private potential events of system 1, set B includes the private potential events of system 2, and sets I, II, III represent the common events of the composed system

3. $\mathcal{H}_{III} \triangleq (\Gamma_1 \cup \eta_2) \cap (\Gamma_2 \cup \eta_1)$, (part III in Figure 3.3), common events that both systems can activate, but they need to synchronize for taking the associated transition.

The union of $\mathcal{H}_{I}$, $\mathcal{H}_{II}$, and $\mathcal{H}_{III}$ form the set of common events for the composed system. Part (1) and part (2) are the new events generated by the composed system, which are not included in the component systems. Thus the events in $\mathcal{H}$ can generate synchronized transitions in the composed system. What remains to be seen is how the definition of composition can now be extended, and how the new synchronized transitions can be defined.

This section redefined the common events of a composed system by introducing the relation $\mathcal{H}$. In the remaining, we use $\mathcal{H}$ to denote the set of all equivalence class on $(\eta_1 \cup \eta_2)$

defined by the relation. Now we are ready to give a formal definition of composition of MDLe systems, which can capture the interaction between systems. This is introduced in next section.

3.2 Composition of MDLe systems

We state our new notion of composition in terms of grammars. These grammars determine how individual systems can take transitions. We focus our discussion on grammars that generate MDLe languages, which have been shown to be context-free. Our definition of composition is stated as follows.

Definition 13. *Consider two* MDL*es, expressed as context-free grammars* $G_1 = (N_1, \eta_1, R, S_{01})$ *and* $G_2 = (N_2, \eta_2, R, S_{02})$*, both with rule sets* R *of the form* (2.3)*. Let* S_1 *and* S_2 *be their corresponding representations as a system of guarded recursive equations, in restricted Greibach normal form over a* BPA*. The composition of* G_1 *and* G_2 *is defined as the context-free grammar* $G = (N, \eta, R, S_0)$*, where*

- $N = N_1 \times N_2$ *is the set of variables;*
- $\eta = \eta_{(1\|2)} = \eta_1 \cup \eta_2$ *is the set of terminals;*
- $S_0 = S_{01} \times S_{02}$ *is the start state of the composed system;*
- *The rule set is*

$$R(N \times \eta) := \begin{cases} (R(x,y), R(z,y)) & \text{for } x \in N_1,\ z \in N_2 \text{ and } y \in \eta, \text{ if } y \in \mathcal{H}, \\ (R(x,y), z) & \text{for } x \in N_1,\ z \in N_2 \text{ and } y \in \eta, \text{ if } y \in A, \\ (x, R(z,y)) & \text{for } x \in N_1,\ z \in N_2 \text{ and } y \in \eta, \text{ if } y \in B \\ \textit{undefined}, & \textit{otherwise.} \end{cases}$$

The transitions of the composed system still respect the grammar rules (2.3), however, the composition restricts the domain of R. The push-down automaton representing the composed system can be defined as follows:

Definition 14. $P_1 \| P_2 = (N, \eta_{(1\|2)}, \Sigma, \Gamma_{(1\|2)}, \delta, S_0, Z_0)$*, and*

- $N = N_1 \times N_2$ *is the set of states;*
- $\eta = \eta_{(1\|2)} = \eta_1 \cup \eta_2$ *is the set of enabled events;*
- $\Sigma = (N_1 \times N_2) \cup \eta_{(1\|2)}$ *is the stack;*
- $\Gamma_{(1\|2)} = \Gamma_1 \cup \Gamma_2$ *is the active event relation;*
- *The transition function is*

$$\delta(N \times \eta) := \begin{cases} (R(x,y), R(z,y)) & \textit{for } x \in N_1,\ z \in N_2 \textit{ and } y \in \eta, \textit{ if } y \in \mathcal{H}, \\ (R(x,y), z) & \textit{for } x \in N_1,\ z \in N_2 \textit{ and } y \in \eta, \textit{ if } y \in A, \\ (x, R(z,y)) & \textit{for } x \in N_1,\ z \in N_2 \textit{ and } y \in \eta, \textit{ if } y \in B \\ \textit{undefined}, & \textit{otherwise.} \end{cases}$$

- $S_0 = (S_{01} \times S_{02})$ *is the start state;*
- $Z_0 = (Z_{01} \times Z_{02})$ *is the start symbol in stack;*

Compared to the classical definition of composition, the new definition uses the same grammar rules, but now the common event set of composed system is enriched by the new components of set $\mathcal{H}$. Now common events include the events that can not generate transitions in their own automata when the latter operate individually, but can initiate transitions when the corresponding automaton is composed with another appropriate automaton.

In Chapter 3 we offer some details on how to construct a framework for MDLe composition and compare it to the classical composition. By using this new definition, the composed system can do something that the single system can not do. After giving the definition, we want to check the properties of the composed system. Chapter 4 presents the transformation from extended Motion Description Language to Basic Process Algebra and states that composition of MDLes is decidable up to bisimulation.

Chapter 4

Language Composition and Equivalence

In this chapter, we present the transformation from extended Motion Description Language to Basic Process Algebra and demonstrate why in the systems we consider, language equivalence is decidable and what bisimulation properties are preserved in the composition.

This chapter is organized as follows: Section 4.1 introduces the concept of decidability and explains why it is important; Section 4.2 lists the steps to convert from MDLe to BPA in GNF; the proof of language equivalence in a MDLe is given in Section 4.3; we prove the closeness and decidability properties separately in Section 4.4 and 4.5 and conclude that the compositions of MDLes are decidable up to bisimulation equivalence.

4.1 Complexity notions

Formal analysis of hybrid systems includes verification: checking if a hybrid system satisfies a desired specification, such as avoiding an unsafe region of the state space or moving to a designed state. Given a hybrid system, H_Σ and desired properties P, if in a finite

number of steps there is a computational procedure that can decide whether H satisfies P for H $\in H_\Sigma$ and any $P \in \boldsymbol{P}$, we say that the verification problem is *decidable*.

Purely discrete systems modeled by finite-state machines are generally decidable, since in the worst case verification can be performed by exhaustively searching the whole state space although the computation cost increases exponentially with the size of the space. However, in the case of hybrid systems, decidability is a central issue in algorithmic analysis, because of the hybrid (continuous and discrete) state space being uncountable.

In the rest of this chapter, we prove the decidability property of the composed MDLes.

4.2 From MDLe to BPA in GNF

The decidability results for language equivalence on basic process algebras in Greibach normal form carry over to MDLes. Here, we show how MDLes can be written as this special class of basic processes.

Lemma 3. MDL*e is a context-free grammar that can be written in Greibach normal form.*

Proof. We rewrite (2.3) in Chomsky normal form, an intermediate stage before we arriving at the Greibach normal form. Rewriting (2.3) in Chomsky normal form involves a sequence of steps, in which a transformation rule is applied to the set of rules written on the left to result in the rule set depicted on the right. Let us first combine rules (2.3) into a single one, using the disjunction operator |, for compactness. Then we give the resulting set of rules after each transformation.

$$
\begin{aligned}
&E \to EE \\
&E \to (u_i, \xi_i) \\
&E \to (E, \xi_i) \\
&E \to \emptyset
\end{aligned}
\qquad\qquad
E \to EE|(u_i, \xi_i)|(E, \xi_i)|\emptyset.
$$

This process consists of five steps:

Step 1: Define a new start symbol S_0 to replace S.

$$S \to E$$
$$E \to EE|(u_i, \xi_i)|(E, \xi_i)|\emptyset$$

$$S_0 \to S$$
$$S \to E$$
$$E \to EE|(u_i, \xi_i)|(E, \xi_i)|\emptyset$$

Step 2: Remove $\emptyset$ from the rules that involve variable E.

$$S_0 \to S$$
$$S \to E$$
$$E \to EE|(u_i, \xi_i)|(E, \xi_i)|\emptyset$$

$$S_0 \to S$$
$$S \to E|EE|(u_i, \xi_i)|(E, \xi_i)$$
$$E \to EE|(u_i, \xi_i)|(E, \xi_i)$$

Step 3: Eliminate the original start variable S.

$$S_0 \to S$$
$$S \to E$$
$$E \to EE|(u_i, \xi_i)|(E, \xi_i)$$

$$S_0 \to E|EE|(u_i, \xi_i)|(E, \xi_i)$$
$$E \to EE|(u_i, \xi_i)|(E, \xi_i)$$

Step 4: Eliminate the unit rules.

$$S_0 \to E|EE|(u_i, \xi_i)|(E, \xi_i)$$
$$E \to EE|(u_i, \xi_i)|(E, \xi_i)$$

$$S_0 \to EE|(u_i, \xi_i)|(E, \xi_i)$$
$$E \to EE|(u_i, \xi_i)|(E, \xi_i)$$

Step 5: Convert the remaining rules into the proper form by adding variables and rules.

$$S_0 \to EE|(u_i, \xi_i)|(E, \xi_i)$$
$$E \to EE|(u_i, \xi_i)|(E, \xi_i)$$

$$S_0 \to EE|Lu_i, \xi_i R|LE, \xi_i R$$
$$E \to EE|Lu_i, \xi_i R|LE, \xi_i R$$
$$L \to ($$
$$R \to)$$

$$
\begin{aligned}
&S_0 \to EE|Lu_i, B|A, B \\
&E \to EE|Lu_i, B|A, B \\
&L \to (\\
&R \to) \\
&A \to LE \\
&B \to \xi_i R
\end{aligned}
\qquad
\begin{aligned}
&S_0 \to EE|D, B|A, B \\
&E \to EE|D, B|A, B \\
&L \to (\\
&R \to) \\
&A \to LE \\
&B \to \xi_i R \\
&D \to Lu_i
\end{aligned}
$$

$$
\begin{aligned}
&S_0 \to EE|DF|AF \\
&E \to EE|DF|AF \\
&L \to (\\
&R \to) \\
&A \to LE \\
&B \to \xi_i R \\
&D \to Lu_i \\
&F \to , B
\end{aligned}
\qquad
\begin{aligned}
&S_0 \to EE \\
&E \to EE|DF|AF \\
&L \to (\\
&R \to) \\
&A \to LE \\
&B \to \xi_i R \\
&D \to LK \\
&F \to , B \\
&H \to , \\
&J \to \xi_i \\
&K \to u_i
\end{aligned}
\tag{4.1}
$$

Then we translate (4.1) into Greibach normal form, by first eliminating left-recursion. This process consists of three steps:

Step 1: Add a new rule $T \rightarrow E|ET$ to eliminate left-recursion $E \rightarrow EE$.

$$
\begin{array}{ll}
S_0 \rightarrow EE|DF|AF & S_0 \rightarrow EE|DF|AF|DFT|AFT \\
E \rightarrow EE|DF|AF & E \rightarrow DF|AF|DFT|AFT \\
L \rightarrow (& L \rightarrow (\\
R \rightarrow) & R \rightarrow) \\
A \rightarrow LE & A \rightarrow LE \\
B \rightarrow \xi_i R & B \rightarrow \xi_i R \\
D \rightarrow LK & D \rightarrow LK \\
F \rightarrow , B & F \rightarrow , B \\
H \rightarrow , & H \rightarrow , \\
J \rightarrow \xi_i & J \rightarrow \xi_i \\
K \rightarrow u_i & K \rightarrow u_i \\
 & T \rightarrow E|ET
\end{array}
$$

Step 2: The next step is to make all the other rules start with a terminal.

$$
\begin{array}{ll}
S_0 \rightarrow EE|DF|AF|DFT|AFT & S_0 \rightarrow (KFT|(EFT|(KF|(EF \\
E \rightarrow DF|AF|DFT|AFT & E \rightarrow (KF|(EF|(KFT|(EFT \\
L \rightarrow (& L \rightarrow (\\
R \rightarrow) & R \rightarrow) \\
A \rightarrow LE & A \rightarrow (E \\
B \rightarrow \xi_i R & B \rightarrow \xi_i R \\
D \rightarrow LK & D \rightarrow (K \\
F \rightarrow , B & F \rightarrow , B \\
H \rightarrow , & H \rightarrow , \\
J \rightarrow \xi_i & J \rightarrow \xi_i \\
K \rightarrow u_i & K \rightarrow u_i \\
T \rightarrow E|ET & T \rightarrow (KF|(EF|(KFT|(EFT
\end{array}
$$

Step 3: The final step is to convert all the rules in restricted GNF by adding rules.

$S_0 \rightarrow (KFT|(EFT|(KF|(EF$
$E \rightarrow (KF|(EF|(KFT|(EFT$
$L \rightarrow ($
$R \rightarrow)$
$A \rightarrow (E$
$B \rightarrow \xi_i R$
$D \rightarrow (K$
$F \rightarrow , B$
$H \rightarrow ,$
$J \rightarrow \xi_i$
$K \rightarrow u_i$
$T \rightarrow (KF|(EF|(KFT|(EFT$

$M \rightarrow KF$
$N \rightarrow EF$
$S_0 \rightarrow (MT|(NT|(KF|(EF$
$E \rightarrow (M|(N|(MT|(NT$
$L \rightarrow ($
$R \rightarrow)$
$A \rightarrow (E$
$B \rightarrow \xi_i R$
$D \rightarrow (K$
$F \rightarrow , B$
$H \rightarrow ,$
$J \rightarrow \xi_i$
$K \rightarrow u_i$
$T \rightarrow (M|(N|(MT|(NT$

$$
\begin{aligned}
&M \to u_i F \\
&N \to (MF|(NF \\
&S_0 \to (MT|(NT|(KF|(EF \\
&E \to (M|(N|(MT|(NT \\
&L \to (\\
&R \to) \\
&A \to (E \\
&B \to \xi_i R \\
&D \to (K \\
&F \to , B \\
&H \to , \\
&J \to \xi_i \\
&K \to u_i \\
&T \to (M|(N|(MT|(NT
\end{aligned}
\tag{4.2}
$$

□

The next Lemma states that an MDLe can be translated into a BPA in Greibach normal form [8].

Lemma 4. *The terms of an* MDLe *are a finite trace set of a normed process* p*, recursively defined by means of a guarded system of recursion equations in restricted Greibach normal form over a* BPA.

Proof. Lemma 3 allows us to express an MDLe as a CFG in Greibach normal form, which in addition satisfies the conditions of Notation 4.5 of [8]. We apply Notation 4.5 in conjunction with Proposition 5.2 of [8] to write the CFG of (4.2) as a BPA as follows.

- If S is the system represented as a CFG in Greibach normal form, let S' denote the system represented in BPA by replacing | by $+$, and $\to$ by $=$.

- Let S' be in restricted Greibach normal form over the BPA, with unique solution p. Then $\mathrm{ftr}(p)$ (the set of finite traces of p) is just the context-free language generated by S.

Applying the change of notation suggested,

$$\begin{array}{ll}
M \rightarrow u_i F & M = u_i F \\
N \rightarrow (MF | (NF & N = (MF + (NF \\
S_0 \rightarrow (MT | (NT | (KF | (EF & S_0 = (MT + (NT + (KF + (EF \\
E \rightarrow (M | (N | (MT | (NT & E = (M + (N + (MT + (NT \\
L \rightarrow (& L = (\\
R \rightarrow) & R =) \\
A \rightarrow (E & A = (E \\
B \rightarrow \xi_i R & B = \xi_i R \\
D \rightarrow (K & D = (K \\
F \rightarrow , B & F =, B \\
H \rightarrow , & H =, \\
J \rightarrow \xi_i & J = \xi_i \\
K \rightarrow u_i & K = u_i \\
T \rightarrow (M | (N | (MT | (NT & T = (M + (N + (MT + (NT
\end{array} \tag{4.3}$$

and thus we have a BPA in restricted Greibach normal form. Note that according to Definition 11, each variable string in the right hand side of (4.3) has length of at most two. By applying Proposition 5.2 of [8], to remove the parts of the system that do not contribute to the generation of the finite traces, we conclude that the BPA of (4.3) generates the strings of the original MDLe. □

4.3 Language equivalence in a MDLe

Systems of guarded recursive equations enjoy nice properties, in the sense that verifying the bisimulation equivalence is decidable [8].

Theorem 1 ([8]). *Let E_1, E_2 be normed systems of guarded recursion equations (over basic process algebras) in restricted Greibach normal form. Then the bisimulation relation $\approx$, that is whether $E_1 \approx E_1$, is decidable.*

Theorem 1 allows us to conclude that

Corollary 1. *If motion description languages are written in the form of a system of guarded recursive equations in Greibach normal form over a basic process algebra, the bisimulation relation is decidable.*

Proof. Using Lemma 3, each MDLe is written as a context-free language in Greibach Normal Form. Lemma 4 translates this representation into a system of guarded recursive equations in restricted Greibach normal form over BPA. By Theorem 1 of [8], language equivalence for systems in (guarded) restricted Greibach normal form such as the MDLes translated using Lemma 4, is decidable up to bisimularity. □

4.4 MDLes are closed under composition

A natural question that arises next, is whether the composition operator preserves bisimularity: do we lose this property when we expand the basic process algebra system by including the operators $\|$ and $\between$ to arrive at the system the semantics of which are described in Tables 2.3 and 2.4?

To answer this question, first let us check the property of composition operator $\|$. The next result establishes that operator $\between$ is closed.

Lemma 5. *An* MDL*e written as a system of guarded recursive equations in restricted Greibach normal form is closed under the left merge $\between$ operator.*

Proof. Assume that G is written as a system of guarded recursive equations in restricted Greibach form, according to (4.2). We prove the claim by taking all merge combinations of variables in this representation, and showing that the result is a system of equations that are also guarded in restricted Greibach normal form. According to Table 2.3,

$$\begin{aligned}
U \between B &= (\nu + \nu B) \between B \overset{M4,M2}{=} \nu B + (\nu B) \between B \\
&\overset{M3}{=} \nu B + \nu(B \parallel B) \overset{A3,M1}{=} \nu B + \nu(B \between B) \\
U \between S_0 &= (\nu + \nu B) \between S_0 \overset{M4,M2}{=} \nu S_0 + (\nu B) \between S_0 \\
&\overset{M3}{=} \nu S_0 + \nu(B \parallel S_0) \overset{M1}{=} \nu S_0 + \nu(B \between S_0 + S_0 \between B) \\
B \between S_0 &= (\nu + \nu BB + \nu B) \between S_0 \\
&\overset{M4,M2}{=} \nu S_0 + (\nu B) \between S_0 + (\nu BB) \between S_0 \\
&\overset{M3,A5}{=} \nu S_0 + \nu(B \between S_0 + S_0 \between B) + \nu(BB) \between S_0 \\
&\overset{M3}{=} \nu S_0 + \nu(B \between S_0 + S_0 \between B) + \nu(BB \between S_0 + S_0 \between BB).
\end{aligned}$$

Note that reversing the order of variables in the above merge operations yields the same type of expressions encountered above:

$$\begin{aligned}
B \between U &= (\nu B + \nu BB + \nu) \between U \\
&= \nu U + \nu(BB \between U + U \between BB) + \nu(B \between U + U \between B) \\
S_0 \between U &= (\nu U + \nu BU + \nu) \between U \\
&= \nu U + \nu(U \between U) + \nu(BU \between U + U \between BU) \\
S_0 \between B &= (\nu U + \nu BU + \nu) \between B \\
&= \nu B + \nu(U \between B + B \between U) + \nu(BU \between B + B \between BU).
\end{aligned}$$

All expressions above are guarded recursive equations in restricted Greibach normal form.

Since the left-merge operation $\lfloor\!\lfloor$ is closed, it follows from M1 in Table 2.3 that $\|$ is closed too. □

4.5 MDLe composition preserves bisimularity

The following Proposition gives an affirmative answer to the question of decidability of language equivalence for MDLes that are a result of composition of two MDLes.

Proposition 1. *The composition operator $\|$ preserves bisimularity. That is, if $P \approx Q$, then $P\|R \approx Q\|R$.*

Proof. Consider a relation $\mathcal{R}$ over the set of processes, such that $P\|R$ and $Q\|R$ belong to $\mathcal{R}$ whenever $P \approx Q$. We show that $\mathcal{R}$ is a bisimulation.

Case 1. Process P (or Q) executes action a. If $P \approx Q$, then $(P\|R, Q\|R) \in \mathcal{R}$. Assume that $P \xrightarrow{a} P'$. Then by action relation R6 in Table 2.4, we have

$$P \xrightarrow{a} P' \Rightarrow P\|R \xrightarrow{a} P'\|R.$$

Since $P \approx Q$, there exists Q' such that $Q \xrightarrow{a} Q'$, and $P' \approx Q'$. By definition of the relation $\mathcal{R}$, $(P'\|R, Q'\|R) \in \mathcal{R}$. Similarly, it can be shown that if $Q \xrightarrow{a} Q'$, then there exists a P', with $P' \approx Q'$ and $(P'\|R, Q'\|R) \in \mathcal{R}$.

Case 2. Process R executes action a. Recall that bisimulation is a reflexive relation, that is $R \approx R$. With this observation, this case reduces to the previous one, and $(P\|R, P\|R) \in \mathcal{R}$.

Case 3. Process P terminates after executing action a. This means that $P \xrightarrow{a} \surd$. Relation R7 of Table 2.4 implies that

$$P \xrightarrow{a} \surd \Rightarrow P\|R \xrightarrow{a} R.$$

Since $P \approx Q$, we need to have $Q \xrightarrow{a} \surd$. Thus, by R7 of Table 2.4,

$$Q \| R \xrightarrow{a} R.$$

Note that $R \approx R$ by definition and thus the processes derived with the a-transition belong to relation $\mathcal{R}$. The case where Q terminates after executing a is identical.

Case 4. Process R terminates after executing a. In other words, $R \xrightarrow{a} \surd$. Then, by R7 of Table 2.4,

$$R \xrightarrow{a} \surd \Rightarrow P \| R \xrightarrow{a} P.$$

Similarly, we have

$$R \xrightarrow{a} \surd \Rightarrow Q \| R \xrightarrow{a} Q.$$

Given that $P \approx Q$, the processes derived from $P \| R$ and $Q \| R$ when R executes a, belong to $\mathcal{R}$.

Case 5. Processes P and R are synchronously execute action a. In this case, we resort to axiom M1 of Table 2.3, and treat the transitions of P and R separately according to cases 1 and 2 above. The case where Q executes a synchronously with R is identical.

Case 6. Processes P and R terminate synchronously by executing action a. Axiom M1 of Table 2.3 allows us to treat the synchronous transition to termination as an asynchronous one. In this case, we proceed according to cases 3 and 4 discussed above.

In conclusion, for all combinations of possible transitions for $P \| R$ and $Q \| R$, we have that $P \| R \approx Q \| R$ if $P \approx Q$. □

From Proposition 1 follows our main result on compositions of motion description languages:

Corollary 2. *The compositions of* MDL*es are decidable up to bisimulation equivalence.*

Proof. The operation $'\|'$ is closed (Lemma 5) and also preserves bisimularity (Lemma 1), which means the composition of MDLes can also be written as a system of guarded recursive equations in restricted Greibach normal form over a basic process algebra. By Theorem 1, it follows that this composition is decidable. □

In this Chapter, we identify MDLes as recursive systems in some basic process algebra written in Greibach Normal Form. We use the machinery available for BPAs to formally define a composition operation for MDLes at the level of grammars. We indicate that appropriately defined MDLe grammars can be composed, and language equivalence (whether two such grammars generate the same finite traces), is decidable up to bisimulation. So far, we introduced a new framework of MDLe composition and checked the decidability property of the composed system in Chapters 3 and 4. We want to design an experiment to illustrate our method, showing how to compose two MDLes and generate a new MDLe which behaves in ways its generators can not. A sliding block puzzle example is introduced in Chapter 5, which serves as a case study.

Chapter 5

A Case Study

We introduce a sliding block puzzle example using the proposed framework, where the robot-puzzle system is viewed as a multi-agent system. The robot and block systems are written in MDLe automata introduced in Chapter 2. We consider blocks as single systems, represented in MDLe, although these single systems can only stay in the original position by themselves if they do not cooperate with other systems.

We envision the blocks as being agents and we imagine that a robot can move between them as another agent, and push them into different locations. In such a multi-agent system, none of the component subsystems (blocks and robot) can change the configuration of the puzzle. There has to be some physical interaction (robot pushing a block) for the configuration to change, and this interaction needs to be encoded in the set of possible events in the system. The classical systems composition cannot capture this interaction and is therefore unable to express the fact that the agents in this system can cooperate.

Using the definition of composition as given in Chapter 3, however, we are able to capture the possibility of blocks being pushed to new positions by the robot, and give a mathematical description of the composed multi-agent system in which this interaction is explicitly modeled. Once this is achieved, classical AI planning tools (search algorithms)

can be applied to design plans for the robot to solve the puzzle. What enables the puzzle "system" to solve itself, instead of a human designer to code a particular solution, is the automaton description facilitated by the new definition. To use the new framework, we need to know the map beforehand, which means we need to know the events of the single system. The program can be used not only for one case, but also for any pair of the start position and end position on this robot-puzzle system. The plan is generated automatically and the calculation is done by a deterministic machine in polynomial time.

This chapter includes two parts: Section 5.1 is a brief review of sliding block puzzles; Section 5.2 is the details of the special 15 block puzzle experiment.

5.1 Sliding Block Puzzles

Sliding block puzzles are essentially two-dimensional in nature. There is a number of blocks arranged in rows and columns so that there is typically one empty location and all other positions are occupied. Blocks can only slide along rows or columns to arrive a certain end-configuration. The 15 block puzzle was invented by Noyes Chapman [114] in 1880s. Fifteen blocks, numbered 1 to 15, were placed in a four by four frame and the 16th position was empty. The goal was to place all the blocks in a correct order and leave the empty square in the lower right position.

Solving a sliding block puzzle requires finding the sequence of moves and identifying the paths opened up by each move. Blocks can not be lifted off the board and can only make a move when the empty space is nearby. Martin Gardiner said in his paper [41] :

> These puzzles are very much in want of a theory. Short of trial and error, no one knows how to determine if a given state is obtainable from another given state, and if it is obtainable, no one knows how to find the minimum chain of moves for achieving the desired state.

This very special nature of sliding puzzle problems make them intriguing. We feel that if our methodology can provide a solution to such a problem (not necessarily optimal) the capability we offer would push the state of the art not only in cooperative robotics, but also in planning and artificial intelligence in general.

It has been shown that sliding-block puzzles are PSPACE-complete [46, 58]. However, under certain simplifying assumptions and for cases of such puzzles like the one we consider here (Fig. 5.1), a polynomial algorithm can be constructed to move a single block from any initial position to any final position [46].

5.2 A special case

We introduce an instance of the sliding block puzzle as a multi-robot hybrid system. We use this example as a reality check, to ensure that our formulation captures the possible interaction between heterogeneous robot systems. The sliding puzzle considered here has fifteen blocks and one empty place. Between these blocks, we imagine narrow corridors which allow a small robot to move. The robot and all the blocks are thought to be autonomous agents, each with its own MDLe. The robot can move a block to the empty space, but blocks can not move by themselves.

In the simple instance of the sliding block puzzle depicted in Figure 5.1, the goal is for the robot (initially at position 30) to move the block at position 1 to location 6. A block can do nothing by itself; any transitions within the block's MDLe may only be activated after composition with the robot agent, which can *push* a block to a different location. However, these potential transitions in the block's configuration need to be encoded in the blocks enabled event set η.

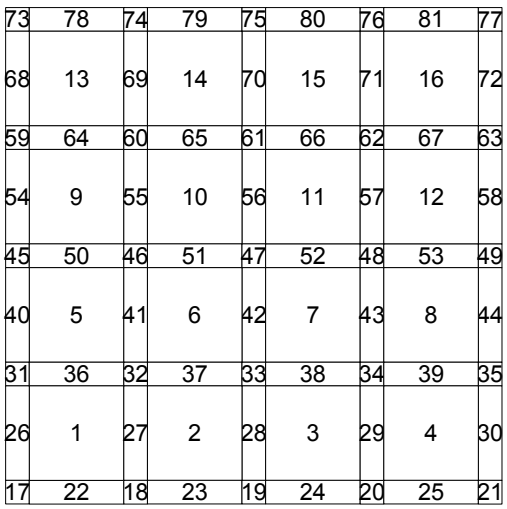

Figure 5.1: The Sliding block puzzle considered here. No.1 through No.16 represent possible positions for any block. No.17 through No.81 represent possible positions for the robot.

5.2.1 The block automaton

For a block to be able to make a transition, the destination location must be unoccupied; thus block agents need to keep track of whether nearby locations are occupied. We therefore model the state of the block as a triplet, consisting of the state of motion, its position, and the availability of an empty location in the immediate neighborhood. The block automaton is $Block = (N_b, \eta_b, V_b \cup \eta_b, \Gamma_b, \delta_b, S_{0b}, Z_{0b})$, where

1. $N_b = (N_{b1}, N_{b2}, N_{b3})$ is a state in this automaton, where

   - $N_{b1} \in \{u_1, u_2, u_3, u_4, u_5\}$ and the latter is a set of possible actions, defined as follows:

  - u_1 : be pushed east,
  - u_2 : be pushed west,
  - u_3 : be pushed north,
  - u_4 : be pushed south,
  - u_5 : stay at location;

- $N_{b2} \in \{P_1, ...P_{16}\}$, and the latter is the set of possible positions for the block;
- $N_{b3} \in \{b_1, b_2, b_3, b_4, b_5\}$, and the latter is the set of all possible configurations of whether they are empty or not. The existence and location of a nearby empty location is specified as follows:
  - b_1 : east,
  - b_2 : west,
  - b_3 : north,
  - b_4 : south,
  - b_5 : no empty space available;

2. $\eta_b = \{\nu_b | \nu_b = ((u_i, P_j, b_k), \xi)\}$, with i and k in $\{1, ..., 5\}$, and j in $\{1, ..., 16\}$, which includes all the events associated with transitions in the automaton, and ξ is the interrupt function;

3. $\Gamma_b : N_b \rightarrow 2^{\eta_b}$ is the active event relation (initially mapping to $\emptyset$);

4. $\delta_b : N_b \times \eta_b \rightarrow N_b$ is the transition function, also mapping to $\emptyset$ since the range of Γ_b is empty, suggesting that the block automaton can make no transitions on its own (except for the case of u_5).

Symbols S_{0b} and Z_{0b} correspond to the initial state and stack symbol, respectively.

5.2.2 The robot automaton

For the robot, an atom consists of the state of motion and its position. The robot can move along the rows and columns of the grid, and push against a block in order to move it. The automaton for the robot is a tuple $Robot = (N_r, \eta_r, N_r \cup \eta_r, \Gamma_r, \delta_r, S_{0r}, Z_{0r})$, where

1. $N_r = (N_{r1}, N_{r2})$ is the states of this automaton, and
   - $N_{r1} \in= \{w_1, w_2, w_3, w_4, w_5, w_6, w_7, w_8, w_9\}$ with the latter being the set of all controllers that the robot can use. These are defined as follows:
     - w_1 : push east,
     - w_2 : push west,
     - w_3 : push north,
     - w_4 : push south,
     - w_5 : stay at location,
     - w_6 : move east,
     - w_7 : move west,
     - w_8 : move north,
     - w_9 : move south,
   - $N_{r2} \in P_{17}, ...P_{81}$ with the latter being the set of all possible positions that robot can be at.
2. $\eta_r = \{\nu_r | \nu_r = ((w_i, P_j), \xi_r)\}$, with i is in $\{1, ..., 9\}$, j in $\{17, ..., 81\}$, which includes all the events associated with transitions in the automaton, and ξ_r is the interrupt function;
3. $\Gamma_r : N_r \rightarrow 2^{\eta_r}$ is the active event relation determining which events are active at each robot state;
4. $\delta_r : N_r \times \eta_r \rightarrow N_r$ is the transition function.

Figure 5.2: The Khepera II mobile Robot

Similarly, Symbols S_{0r} and Z_{0r} correspond to the initial state and stack symbol, respectively.

5.2.3 The Khepera Robot

In the experimental implementation of our planning methodology, we use the Khepera II mobile robot to move the blocks. The Khepera II [1] is a miniature mobile robot with functionality similar to larger robots used in research and education. It allows the testing of algorithms developed in simulation for trajectory planning, obstacle avoidance, etc.

The Khepera can be used on a wide range of experiments. It supports different programming environments, such as SysQuake, LabVIEW, MATLAB, and 3D WEBOTS Khepera simulator. The robot can communicate with a host computer over a wired or wireless serial port. The communication protocol implemented on Khepera is described in more detail in its user manual [1].

Khepera robots are high-performance and very small, requiring very little area for use. They can be used on an ordinary desktop. Their compact size, along with numerous extensions make them an appealing research tool. The battery on a robot that has been in use for some time can last about 15 minutes when it is fully charged. If wireless communication is used, the time is much shorter than 15 minutes. The life of the battery decreases with the

use of the robot. Controlling the robot remotely is feasible but not always reliable since it has been observed that even if communication is perfect, sometimes the robot fails to respond. Thus, the most reliable way of executing experiments with Khepera is to download the complete code on its microprocessor and have it run locally.

5.2.4 Shortest Paths

The goal of this experiment is to automatically derive a plan for the composed system of blocks and robot, so that the block moves from one initial position to a destination. There are many ways to do so. We can use a maneuver at more than 1000 steps or with less than 10 steps. We want to find an efficient way to solve this problem.

At first we tried to use abstraction bases on bisimulation relations. The idea is to abstract the 15 piece puzzle to a smaller one, a 9-block puzzle or 4-block puzzle, which is easier to check and implement. But we find that every block in this puzzle has its own properties and transfer between blocks' location appear unique. So none of them can be grouped together. An automated approach [16] to finding bisimular relations in the composed 15 block puzzle system does not yield any results. Our inability to find a bisimulation equivalence relation using the standard algorithm does not necessarily mean that the system can not be abstracted. There might be special sequences of transitions producing maneuvers which can be associated with each other, but these sequences can be invisible to the algorithm.

To plan maneuvers in the 15 piece puzzle system we use the Floyd Warshall algorithm [24]. The Floyd Warshall algorithm is an application of Dynamic Programming. It's a graph analysis algorithm which can find the shortest paths in a weighted, directed graph. The algorithm is able to compare all possible paths through the graph between each pair of vertices and find the shortest one. It is done by first computing an estimate shortest path between two vertices, then incrementally improving this estimate until it is known

to be optimal. Dijkstra's algorithm is another choice to solve the shortest-paths problem but it can only calculate a single-source at one time. In contrast, with the Floyd Warshall algorithm a single run gives shortest-paths between all pairs of vertices. The Pseudo code of Floyd Warshall algorithm is the following:

```
for k: = 0 to n  1
     for each (i,j) in (0..n- 1)
        path[i][j] = min(path[i][j], path[i][k]+path[k][j])
```

The complexity of the algorithm is $\Theta(n^3)$ and can be executed by a deterministic machine in polynomial time.

The implementation of this algorithm in the 15 puzzle problem is introduced in Section 5.2.5.

5.2.5 Planning

In Section 5.2.1 and Section 5.2.2, two single systems are introduced to model a block and a robot respectively. None of them can complete the task separately and the classical composition of systems does not offer a combined system with the capability to do so either. We need the set $\mathcal{H}$ introduced in Chapter 3 to define the common events of the composed system and synchronize these two systems to complete the task. We also want to generate an automatic process that can create plans: given the start position and end position of the robot and block, the automatic process can generate the sequence of MDLe atoms which the robot can execute to move the block to the desired location. This process includes several parts:

Step 1: The first step in the planning process is to calculate the shortest path that the block can follow to move between the two places. As introduced in Section 5.2.4, given *start position* and *end position*, we use Floyd Warshall algorithm to calculate the shortest path for block movement. We consider the 16 blocks as 16 nodes. The weight of the edge between 2 nodes is 1 if one of them can be moved to another position directly and vice versa. The weight is defined 100 if we can not complete the movement in 1 step, and 0 if the 2 nodes are the same. For example, node 6 can only be moved to node 2,5,7,10. So the weight between edges $6--2$, $6--5$, $6--7$, $6--10$ is 1, the weight of other edges ($6--1$, $6--3$, $6--4$, $6--8$, $6--9$, $6--11$, $6--12$, $6--13$, $6--14$, $6--15$, $6--16$) is 100 and the weight of edge $6--6$ is 0. According to this definition, the weight matrix can be defined as follows:

```
D=[  0   1   100 100 1   100 100 100  100  100  100  100  100  100  100  100;
     1   0   1   100 100 1   100 100  100  100  100  100  100  100  100  100;
     100 1   0   1   100 100 1   100  100  100  100  100  100  100  100  100;
     100 100 1   0   100 100 100 1    100  100  100  100  100  100  100  100;
     1   100 100 100 0   1   100 100  1    100  100  100  100  100  100  100;
     100 1   100 100 1   0   1   100  100  1    100  100  100  100  100  100;
     100 100 1   100 100 1   0   1    100  100  1    100  100  100  100  100;
     100 100 100 1   100 100 1   0    100  100  100  1    100  100  100  100;
     100 100 100 100 1   100 100 100  0    1    100  100  1    100  100  100;
     100 100 100 100 100 1   100 100  1    0    1    100  100  1    100  100;
     100 100 100 100 100 100 1   100  100  1    0    1    100  100  1    100;
     100 100 100 100 100 100 100 1    100  100  1    0    100  100  100  1  ;
     100 100 100 100 100 100 100 100  1    100  100  100  0    1    100  100;
     100 100 100 100 100 100 100 100  100  1    100  100  1    0    1    100;
     100 100 100 100 100 100 100 100  100  100  1    100  100  1    0    1  ;
     100 100 100 100 100 100 100 100  100  100  100  1    100  100  1    0  ]
```

Using the weight matrix, the shortest path between any two nodes can be calculated. The outcome is:

```
A=[ 0  1  2  3  1   2    3    4     5     6     7     8     9    10    11  12;
    2  0  2  3  1   2    3    4     5     6     7     8     9    10    11  12;
```

```
2  3  0  3  1  2   3   4   5   6   7   8   9   10  11  12;
2  3  4  0  1  2   3   4   5   6   7   8   9   10  11  12;
5  1  2  3  0  5   6   7   5   6   7   8   9   10  11  12;
2  6  2  3  6  0   6   7   5   6   7   8   9   10  11  12;
2  3  7  3  6  7   0   7   5   6   7   8   9   10  11  12;
2  3  4  8  6  7   8   0   5   6   7   8   9   10  11  12;
5  1  2  3  9  5   6   7   0   9   10  11  9   10  11  12;
2  6  2  3  6  10  6   7   10  0   10  11  9   10  11  12;
2  3  7  3  6  7   11  7   10  11  0   11  9   10  11  12;
2  3  4  8  6  7   8   12  10  11  12  0   9   10  11  12;
5  1  2  3  9  5   6   7   13  9   10  11  0   13  14  15;
2  6  2  3  6  10  6   7   10  14  10  11  14  0   14  15;
2  3  7  3  6  7   11  7   10  11  15  11  14  15  0   15;
2  3  4  8  6  7   8   12  10  11  12  16  14  15  16  0]
```

The path can be read from the outcome matrix. For example, if we want a path from node 1 to node 14, first we read $A(1, 14)$, which is 10; next read $A(1, 10)$, which is 6; then read $A(1, 2)$, which is 1. So the whole plan is $1 \rightarrow 2 \rightarrow 6 \rightarrow 10 \rightarrow 14$. This procedure is programed automatically and we only need to give *start position* and *end position* to have the whole plan.

Step 2: Include the empty block position into the plan. The empty block position is not considered in the Floyd Warshall algorithm. However in this problem, before a block moves from one node to the next node, this receiving node must be unoccupied. For example, when we want to move block A from node 1 to node 2, node 2 must be unoccupied. This is equal to move the "empty block" to node 2 and then move block A from node 1 to 2. Then the whole plan for block movement is: $E(start \rightarrow 2) \longrightarrow (1 \rightarrow 2) \longrightarrow E(1 \rightarrow 6) \longrightarrow (2 \rightarrow 6) \longrightarrow E(2 \rightarrow 10) \longrightarrow (6 \rightarrow 10) \longrightarrow E(6 \rightarrow 14) \longrightarrow E(10 \rightarrow 14)$, where $E()$ represents the path for "empty block" movement. To derive the maneuver, we employ abstraction, because for the motion of the "empty block", we can find equivalence class of the states. It is not a formal abstraction using bisimulation because the states in the equivalence class do not have the same transitions. The motivation for considering them as equivalence class is in every such set, the blocks follow the same movement to the destination. We abstract

all the possible "empty block" and target block positions into 16 groups. Figure 5.3 shows 12 groups. There are 4 other groups need to be considered separately for the special position the blocks occupy. These four groups are (underlined number represent empty block, overlined number represent the block we want to move):

- Group 13: node [$\underline{13}$ $\overline{14}$ 15] and [$\underline{14}$ 15 16]. They have the same format as group 1 but follow different movement rules.
- Group 14: node [13 $\overline{14}$ $\underline{15}$] and [14 $\overline{15}$ $\underline{16}$]. They have the same format as group 2 but follow different movement rules.
- Group 15: node [$\underline{4}$ $\overline{8}$ 12] and [$\underline{8}$ $\overline{12}$ 16]. They have the same format as group 11 but follow different movement rules.
- Group 15: node [4 $\overline{8}$ $\underline{12}$] and [8 $\overline{12}$ $\underline{16}$]. They have the same format as group 12 but follow different movement rules.

Blocks that have the same format fall into the same group. For example, [6 $\overline{7}$ $\underline{8}$] and [10 $\overline{11}$ $\underline{12}$] can be put together to group 2. In [6 $\overline{7}$ $\underline{8}$], the "empty block" is in node 8 and we need to move block from node 7 to node 6. In [10 $\overline{11}$ $\underline{12}$], the "empty block" is in node 12 and we need to move block from node 11 to node 10. In both cases, we want to move block from node i to i-1 (always consider the block we are trying to move is in node i). The elementary block motions in this group are: $i+5 \rightarrow i+1$, $i+4 \rightarrow i+5$, $i+3 \rightarrow i+4$, $i-1 \rightarrow i+3$, $i \rightarrow i-1$. $(i+5 \rightarrow i+1)$ means move block on node $i+5$ to node $i+1$ and similar to the others.

Given every group a different sequence of elementary block motions, the movement of "empty block" ($E()$) is calculated. Together with the shortest path calculated in Step 1, now we have a complete path for the block.

Step 3: For each elementary block motion, there should be a plan for the robot in order to implement it. When calculating the robot path, only the common events in set $\mathcal{H}$ are considered, which means the corresponding robot position is decided according

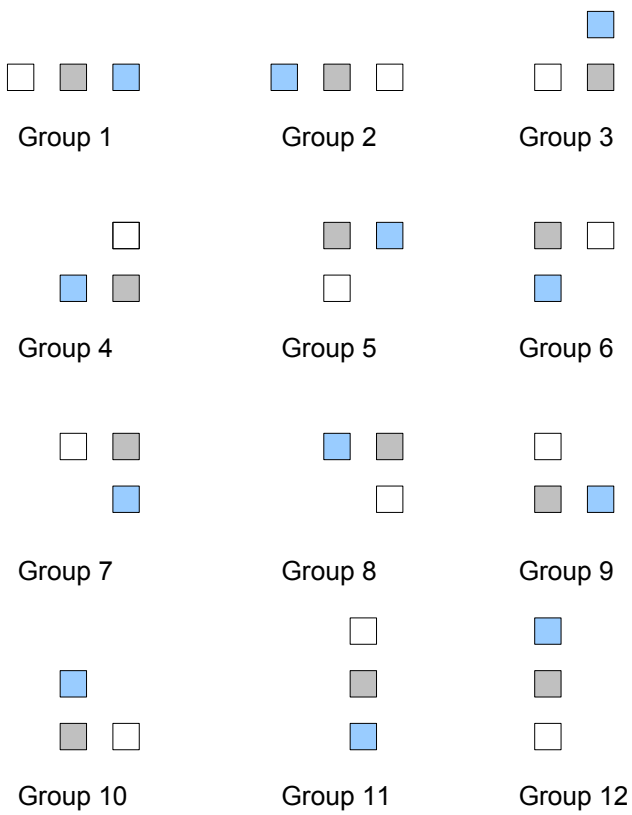

Figure 5.3: The configuration of block positions. White square represents empty block position; gray square represents the initial position of the block we want to move; blue square represents the end position of the block.

to the block path. For example, if the block path include node 3, we only consider node 28, 29, 24 in the robot path; this is because only [1 3 1;1 28], [2 3 2;2 29] and [3 3 3;2 24] are included in set $\mathcal{H}$. Set $\mathcal{H}$ limits the cases we need to consider. There is no "equivalence class" in these cases we can find and we do this transfer case by case.

Step 4: Generate atoms. In this step we transfer the sequence of position number to atom plan. Similarly to step 2, we also abstract all the movements of robots into 13 groups. In every group, the movement follows the same rule. For example, $30 \rightarrow 53$ and

$28 \rightarrow 51$ can be grouped together cause the robot follows the same routine to the next position, which can be encodes as an atom sequence: [8 1 0 1 6 4 5 1 1]. The atom number is decided by the distance the Khepera II is supposed to move. We have 9 atoms for this problem, which are (units correspond to encoder counts): $\mathrm{Atom0}$: moving 1032 units; $\mathrm{Atom1}$: turn left; $\mathrm{Atom2}$: moving 4228 units; $\mathrm{Atom3}$: turn right; $\mathrm{Atom4}$: pushing 2044 units; $\mathrm{Atom5}$: moving -2044 units; $\mathrm{Atom6}$: moving 100 units; $\mathrm{Atom7}$: moving 2144 units; $\mathrm{Atom8}$: moving 3176 units.

Step 5: Download plan to robot. After the atom plan is generated, it should be sent to the robot and the robot can follow the command to act. We program the atom plan into the language that Khepera II can read and download it to the robot.

Once the start and end position is given, the program can generate the plan sequentially according to step 1–5. After the atom plan is generated automatically, it is downloaded to the robot. The robot follows the command and pushes the block all the way to the target position.

The steps of generating atom plans are listed above. In next section, we show a detail atom plan of moving block from node 1 to node 6.

5.2.6 Example: Detail plan to move block from 1 to 6

To move a block from position 1 to position 6, starting from the configuration shown in Figure 5.1, the shortest path for the block is:

$$FF = \quad 1 \quad 2 \quad 6$$

The elementary block motion is:

$$Q = \begin{matrix} 3 & 2 & 1 & 5 & 6 & 2 \\ \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\ 4 & 3 & 2 & 1 & 5 & 6 \end{matrix}$$

The path for the robot is:

$$G = \begin{matrix} 30 & 28 & 27 & 26 & 50 & 42 & 23 \end{matrix}$$

The atom plan send to the robot is:

$$Plan = \begin{matrix} 0 & 1 & 2 & 1 & 0 & 1 & 6 & 4 & 5 \\ 1 & 0 & 1 & 7 & 1 & 0 & 1 & 6 & 4 \\ 5 & 1 & 0 & 1 & 7 & 1 & 0 & 1 & 6 \\ 4 & 5 & 1 & 8 & 3 & 0 & 3 & 6 & 4 \\ 5 & 1 & 1 & 3 & 8 & 3 & 0 & 3 & 6 \\ 4 & 5 & 3 & 8 & 1 & 0 & 1 & 6 & 4 \\ 5 \end{matrix}$$

The physical outcome of this plan is shown in Figures 5.4 and 5.5

By allowing systems to have additional cooperative transitions, they can perform some new actions when composed with appropriate others.

In this chapter, we introduce an instance of the sliding block puzzle as a multi-robot hybrid system, ensure that our formulation captures the possible interaction between heterogeneous robot systems. We automatically generate the atom plan and the robot can push the block to the designed destination automatically according to the plan.

Figure 5.4: The initial configuration of the robot and blocks.

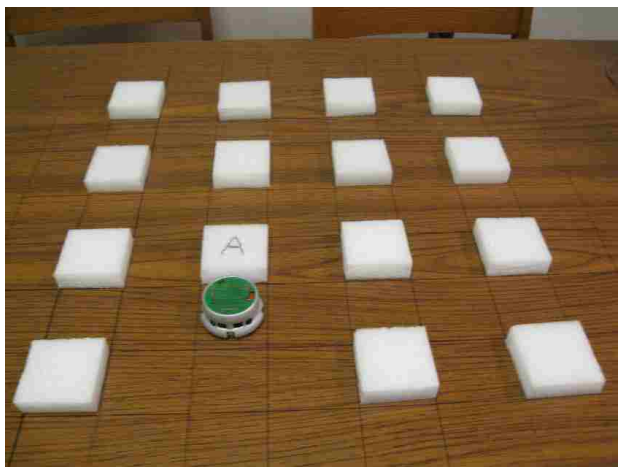

Figure 5.5: The final configuration of the robot and blocks.

Chapter 6

Conclusions and Future Work

In this chapter we conclude our results presented in this dissertation and present possible future work of these results.

We start by assuming the existence of a set of designed behavioral primitives, and we proceed by developing a framework that dictates how these behaviors are sequentially synthesized into plans that drive the system into a desired state. In that sense, our behavioral primitive forms an alphabet of actions.

A new composition system is given and the properties are checked. We distinguish between events associated with transitions a push-down automaton representing an MDLe can take autonomously, and events that cannot initiate transitions. Among the latter, there can be events that when *synchronized* with some of another push-down automaton, become active and do initiate transitions. Then we identify MDLes as recursive systems in some basic process algebra (BPA) written in Greibach Normal Form. We propose a simple context-free grammar that generates MDLes and then we use the machinery available for BPAs to formally define a composition operation for MDLes at the level of grammars. The main difference of our composition operation is the appearance in the composed system of events (transitions) not enabled in the generators: the composed system can behave in

ways its generators cannot.

By identifying MDLes as a subclass of BPAs, we are able to borrow the syntax and semantics of the BPAs merge operator (instead of defining a new MDLe operator), and thus establish closeness and decidability properties for MDLe compositions.

A sliding block puzzle as a multi-robot hybrid system is given to ensure that our formulation captures the possible interaction between heterogeneous robot systems. This example shows how our framework captures physical interaction, and why the composed system can have richer behavior compared to its component subsystems. We automate the process of planning in this example and the atom sequence can be generated automatically according to the initial and final state.

The decidability properties of composed MDLes give us hope that the composed big system can be abstracted to a point that available model checkers [50, 71, 121] can be used to construct motion plans. Research along this direction will expand the applications on model checkers, such as UPPAAL and HYTECH.

We utilize an existing motion control language to abstract low level controllers into elementary behaviors in a way to facilitate high level planning. The application of the new formalism can be extended to different areas. For example, we can compose auditory and visual systems together to decide the accurate position of any objects; we can compose UAVS, Kheperas and allow them communicate and allocate tasks among themselves, and sort through possible solutions to find a plan of action without human intervention or guidance.

Appendices

A Γ **function of block**

B Γ **function of robot**

C **H function**

D **Shortest Path**

E **Plan**

F **C code sent to the robot**

Appendix A

Γ function of blocks

Given a state of the block, this function lists all the events that can be activated from this state. If there is no such events, the output is [0 0 0].

```
function [a]=gamma(u,p,b)
if ((p==6)|(p==7)|(p==10)|(p==11))
    if u==5
        u=b;p=p;b=b;
    elseif ((u==1)&(b==1))
            u=5;p=p+1;b=2;
    elseif ((u==2)&(b==2))
            u=5;p=p-1;b=1;
    elseif ((u==3)&(b==3))
            u=5;p=p+4;b=4;
    elseif ((u==4)&(b==4))
            u=5;p=p-4;b=3;
    else
            u=0;p=0;b=0;
```

```
    end
elseif (p==1)
    if (u==5)&((b==1)|(b==3))
        u=b;p=p;b=b;
    elseif ((u==1)&(b==1))
        u=5;p=2;b=2;
    elseif ((u==3)&(b==3))
            u=5;p=5;b=4;
    else
            u=0;p=0;b=0;
    end
elseif (p==4)
    if (u==5)&((b==2)|(b==3))
        u=b;p=p;b=b;
    elseif ((u==2)&(b==2))
        u=5;p=3;b=1;
    elseif ((u==3)&(b==3))
            u=5;p=4;b=4;
    else
            u=0;p=0;b=0;
    end
elseif (p==13)
    if (u==5)&((b==1)|(b==4))
        u=b;p=p;b=b;
    elseif ((u==1)&(b==1))
        u=5;p=14;b=2;
    elseif ((u==4)&(b==4))
            u=5;p=9;b=3;
    else
            u=0;p=0;b=0;
```

```
    end
elseif (p==16)
    if (u==5)&((b==2)|(b==4))
        u=b;p=p;b=b;
    elseif ((u==2)&(b==2))
        u=5;p=15;b=1;
    elseif ((u==3)&(b==3))
            u=5;p=12;b=3;
    else
            u=0;p=0;b=0;
    end
elseif ((p==2)|(p==3))
    if (u==5)&((b==1)|(b==2)|(b==3))
        u=b;p=p;b=b;
    elseif ((u==1)&(b==1))
        u=5;p=p+1;b=2;
    elseif ((u==2)&(b==2))
            u=5;p=p-1;b=1;
    elseif ((u==3)&(b==3))
            u=5;p=p+4;b=4;
    else
            u=0;p=0;b=0;
    end
elseif ((p==14)|(p==15))
    if (u==5)&((b==1)|(b==2)|(b==4))
        u=b;p=p;b=b;
    elseif ((u==1)&(b==1))
            u=5;p=p+1;b=2;
    elseif ((u==2)&(b==2))
            u=5;p=p-1;b=1;
```

```
    elseif ((u==4)&(b==4))
            u=5;p=p-4;b=3;
    else
            u=0;p=0;b=0;
    end
elseif ((p==5)|(p==9))
    if (u==5)&((b==1)|(b==3)|(b==4))
        u=b;p=p;b=b;
    elseif ((u==1)&(b==1))
            u=5;p=p+1;b=2;
    elseif ((u==3)&(b==3))
            u=5;p=p+4;b=4;
    elseif ((u==4)&(b==4))
            u=5;p=p-4;b=3;
    else
            u=0;p=0;b=0;
    end
elseif ((p==8)|(p==12))
    if (u==5)&((b==2)|(b==3)|(b==4))
        u=b;p=p;b=b;
    elseif ((u==2)&(b==2))
            u=5;p=p-1;b=1;
    elseif ((u==3)&(b==3))
            u=5;p=p+4;b=4;
    elseif ((u==4)&(b==4))
            u=5;p=p-4;b=3;
    else
            u=0;p=0;b=0;
    end
else
```

```
   u=0;p=0;b=0;
end
a=[u,p,b];
```

Appendix B

Γ function of robot

Given a state of the robot, this function lists all the events that can be activated from this state. If there is no event that can be activated from this state, the output is [0 0;0 0].

```
function [c]=gammarobot(uu,pp)
%in south 2 corridor,can push north,move east,west
if ((pp>=22)&(pp<=25))|((pp>=36)&(pp<=39))
    if uu==5
        uu=3;pp=pp;uu1=6;pp1=pp-4;uu2=7;pp2=pp-5;
        c=[[uu,pp];[uu1,pp1];[uu2,pp2]];
    elseif uu==3
        uu=5;pp=pp+14;c=[uu,pp;0,0];
    else
        uu=0;pp=0;c=[uu,pp;0,0];
    end
%in north 2 corridor,can push south,move east,west
elseif ((pp>=64)&(pp<=67))|((pp>=78)&(pp<=81))
    if uu==5
        uu=4;pp=pp;uu1=6;pp1=pp-4;uu2=7;pp2=pp-5;
```

Appendix B. Γ function of robot

```
        c=[uu,pp;uu1,pp1;uu2,pp2];
    elseif uu==4
        uu=5;pp=pp-14;c=[uu,pp;0,0];
    else
        uu=0;pp=0;c=[uu,pp;0,0];
    end
%in middle corridor,can push south,north, move east,west
elseif ((pp>=50)&(pp<=53))
    if uu==5
      uu=3;pp=pp;uu1=4;pp1=pp;uu2=6;pp2=pp-4;uu3=7;pp3=pp-5;
      c=[uu,pp;uu1,pp1;uu2,pp2;uu3,pp3];
    elseif uu==4
        uu=5;pp=pp-14;c=[uu,pp;0,0];
    elseif uu==3
        uu=5;pp=pp+14;c=[uu,pp;0,0];
    else
        uu=0;pp=0;c=[uu,pp;0,0];
    end
%in west 2 corridor,can push east,move north,south
elseif (pp==26)|(pp==40)|(pp==54)|(pp==68)|
(pp==27)|(pp==41)|(pp==55)|(pp==69)
    if uu==5
        uu=1;pp=pp;uu1=8;pp1=pp+5;uu2=9;pp2=pp-9;
        c=[uu,pp;uu1,pp1;uu2,pp2];
    elseif uu==1
        uu=5;pp=pp+1;c=[uu,pp;0,0];
    else
        uu=0;pp=0;c=[uu,pp;0,0];
    end
%in east 2 corridor,can push west,move north,south
```

Appendix B. Γ function of robot

```
elseif (pp==29)|(pp==43)|(pp==57)|(pp==71)|
(pp==30)|(pp==44)|(pp==58)|(pp==72)
    if uu==5
        uu=2;pp=pp;uu1=8;pp1=pp+5;uu2=9;pp2=pp-9;
        c=[uu,pp;uu1,pp1;uu2,pp2];
    elseif uu==2
        uu=5;pp=pp-1;c=[uu,pp;0,0];
    else
        uu=0;pp=0;c=[uu,pp;0,0];
    end
%in middle corridor,can push east,west, move south,north,
elseif (pp==28)|(pp==42)|(pp==56)|(pp==70)
    if uu==5
      uu=1;pp=pp;uu1=2;pp1=pp;uu2=8;pp2=pp+5;uu3=9;pp3=pp-9;
      c=[uu,pp;uu1,pp1;uu2,pp2;uu3,pp3];
    elseif uu==1
        uu=5;pp=pp+1;c=[uu,pp;0,0];
    elseif uu==3
        uu=5;pp=pp-1;c=[uu,pp;0,0];
    else
        uu=0;pp=0;c=[uu,pp;0,0];
    end
%in southwest corner,can move east,north
elseif (pp==17)
    if uu==5
        uu=6;pp=pp;uu1=8;pp1=pp;
        c=[uu,pp;uu1,pp1];
    else
        uu=0;pp=0;c=[uu,pp;0,0];
    end
```

Appendix B. Γ function of robot

```
%in southeast corner,can move west,north
elseif (pp==21)
    if uu==5
        uu=7;pp=pp;uu1=8;pp1=pp;
        c=[uu,pp;uu1,pp1];
    else
        uu=0;pp=0;c=[uu,pp;0,0];
    end
%in northwest corner,can move east,south
elseif (pp==73)
    if uu==5
        uu=6;pp=pp;uu1=9;pp1=pp;
        c=[uu,pp;uu1,pp1];
    else
        uu=0;pp=0;c=[uu,pp;0,0];
    end
%in northeast corner,can move west,south
elseif (pp==77)
    if uu==5
        uu=7;pp=pp;uu1=9;pp1=pp;
        c=[uu,pp;uu1,pp1];
    else
        uu=0;pp=0;c=[uu,pp;0,0];
    end
%in south side corridor,can move east,west,north
elseif (pp==18)|(pp==19)|(pp==20)
    if uu==5
        uu=6;pp=pp;uu1=7;pp1=pp;uu2=8;pp2=pp;
        c=[uu,pp;uu1,pp1;uu2,pp2];
    else
```

Appendix B. Γ function of robot

```
        uu=0;pp=0;c=[uu,pp;0,0];
    end
%in west side corridor,can move east,south,north
elseif (pp==31)|(pp==45)|(pp==59)
    if uu==5
        uu=6;pp=pp;uu1=8;pp1=pp;uu2=9;pp2=pp;
        c=[uu,pp;uu1,pp1;uu2,pp2];
    else
        uu=0;pp=0;c=[uu,pp;0,0];
    end
%in north side corridor,can move east,west,south
elseif (pp==74)|(pp==75)|(pp==76)
    if uu==5
        uu=6;pp=pp;uu1=7;pp1=pp;uu2=9;pp2=pp;
        c=[uu,pp;uu1,pp1;uu2,pp2];
    else
        uu=0;pp=0;c=[uu,pp;0,0];
    end
%in south side corridor,can move east,west,north
elseif (pp==35)|(pp==49)|(pp==63)
    if uu==5
        uu=7;pp=pp;uu1=8;pp1=pp;uu2=9;pp2=pp;
        c=[uu,pp;uu1,pp1;uu2,pp2];
    else
        uu=0;pp=0;c=[uu,pp;0,0];
    end
%in middle corridor,can move in 4 direction
elseif (pp==32)|(pp==33)|(pp==34)|(pp==46)|(pp==47)|
(pp==48)|(pp==60)|(pp==61)|(pp==62)
    if uu==5
```

Appendix B. Γ function of robot

```
      uu=6;pp=pp;uu1=7;pp1=pp;uu2=8;pp2=pp;uu3=9;pp3=pp;
      c=[uu,pp;uu1,pp1;uu2,pp2;uu3,pp3];
    else
        uu=0;pp=0;c=[uu,pp;0,0];
    end
else
    uu=0;pp=0;
    c=[uu,pp;0,0];
end
```

Appendix C

H function

This function can generate the synchronized common events for the composed system. The input is two events, one from block system, one from robot system. If these two events can be synchronized, the output is the common event of the composed system; if not, the output is 0.

```
function [h]=hfunction(a,c)
if a(2)==1
    if (a(1)==1)&(a(3)==1)&(c(1,1)==1)&(c(1,2)==26)
        h=[u,p,b;uu,pp];
    elseif (a(1)==3)&(a(3)==3)&(c(1,1)==3)&(c(1,2)==22)
        h=[u,p,b;uu,pp];
    else
        h=0;
    end
elseif a(2)==2
    if (a(1)==1)&(a(3)==1)&(c(1,1)==1)&(c(1,2)==27)
        h=[u,p,b;uu,pp];
```

```
    elseif (a(1)==2)&(a(3)==2)&(c(2,1)==2)&(c(2,2)==28)
        h=[u,p,b;uu,pp];
    elseif (a(1)==3)&(a(3)==3)&(c(1,1)==3)&(c(1,2)==23)
        h=[u,p,b;uu,pp];
    else
        h=0;
    end
elseif a(2)==3
    if (a(1)==1)&(a(3)==1)&(c(1,1)==1)&(c(1,2)==28)
        h=[u,p,b;uu,pp];
    elseif (a(1)==2)&(a(3)==2)&(c(1,1)==2)&(c(1,2)==29)
        h=[u,p,b;uu,pp];
    elseif (a(1)==3)&(a(3)==3)&(c(1,1)==3)&(c(1,2)==24)
        h=[u,p,b;uu,pp];
    else
        h=0;
    end
elseif a(2)==4
    if (a(1)==2)&(a(3)==2)&(c(1,1)==2)&(c(1,2)==30)
        h=[u,p,b;uu,pp];
    elseif (a(1)==3)&(a(3)==3)&(c(1,1)==3)&(c(1,2)==39)
        h=[u,p,b;uu,pp];
    else
        h=0;
    end
elseif a(2)==5
    if (a(1)==1)&(a(3)==1)&(c(1,1)==1)&(c(1,2)==40)
        h=[u,p,b;uu,pp];
    elseif (a(1)==3)&(a(3)==3)&(c(1,1)==3)&(c(1,2)==36)
        h=[u,p,b;uu,pp];
```

```
    elseif (a(1)==4)&(a(3)==4)&(c(2,1)==4)&(c(2,2)==50)
        h=[u,p,b;uu,pp];
    else
        h=0;
    end
elseif a(2)==6
    if (a(1)==1)&(a(3)==1)&(c(1,1)==1)&(c(1,2)==41)
        h=[u,p,b;uu,pp];
    elseif (a(1)==2)&(a(3)==2)&(c(2,1)==2)&(c(2,2)==42)
        h=[u,p,b;uu,pp];
    elseif (a(1)==3)&(a(3)==3)&(c(1,1)==3)&(c(1,2)==37)
        h=[u,p,b;uu,pp];
    elseif (a(1)==4)&(a(3)==4)&(c(2,1)==4)&(c(2,2)==51)
        h=[u,p,b;uu,pp];
    else
        h=0;
    end
elseif a(2)==7
    if (a(1)==1)&(a(3)==1)&(c(1,1)==1)&(c(1,2)==42)
        h=[u,p,b;uu,pp];
    elseif (a(1)==2)&(a(3)==2)&(c(1,1)==2)&(c(1,2)==53)
        h=[u,p,b;uu,pp];
    elseif (a(1)==3)&(a(3)==3)&(c(1,1)==3)&(c(1,2)==38)
        h=[u,p,b;uu,pp];
    elseif (a(1)==4)&(a(3)==4)&(c(2,1)==4)&(c(2,2)==52)
        h=[u,p,b;uu,pp];
    else
        h=0;
    end
elseif a(2)==8
```

```
    if (a(1)==2)&(a(3)==2)&(c(1,1)==2)&(c(1,2)==44)
        h=[u,p,b;uu,pp];
    elseif (a(1)==3)&(a(3)==3)&(c(1,1)==3)&(c(1,2)==39)
        h=[u,p,b;uu,pp];
    elseif (a(1)==4)&(a(3)==4)&(c(2,1)==4)&(c(2,2)==53)
        h=[u,p,b;uu,pp];
    else
        h=0;
    end
elseif a(2)==9
    if (a(1)==1)&(a(3)==1)&(c(1,1)==1)&(c(1,2)==54)
        h=[u,p,b;uu,pp]1;
    elseif (a(1)==3)&(a(3)==3)&(c(1,1)==3)&(c(1,2)==50)
        h=[u,p,b;uu,pp];
    elseif (a(1)==4)&(a(3)==4)&(c(1,1)==4)&(c(1,2)==64)
        h=[u,p,b;uu,pp];
    else
        h=0;
    end
elseif a(2)==10
    if (a(1)==1)&(a(3)==1)&(c(1,1)==1)&(c(1,2)==55)
        h=[u,p,b;uu,pp];
    elseif (a(1)==2)&(a(3)==2)&(c(2,1)==2)&(c(2,2)==56)
        h=[u,p,b;uu,pp];
    elseif (a(1)==3)&(a(3)==3)&(c(1,1)==3)&(c(1,2)==51)
        h=[u,p,b;uu,pp];
    elseif (a(1)==4)&(a(3)==4)&(c(1,1)==4)&(c(1,2)==65)
        h=[u,p,b;uu,pp];
    else
        h=0;
```

```
    end
elseif a(2)==11
    if (a(1)==1)&(a(3)==1)&(c(1,1)==1)&(c(1,2)==56)
        h=[u,p,b;uu,pp];
    elseif (a(1)==2)&(a(3)==2)&(c(1,1)==2)&(c(1,2)==57)
        h=[u,p,b;uu,pp];
    elseif (a(1)==3)&(a(3)==3)&(c(1,1)==3)&(c(1,2)==52)
        h=[u,p,b;uu,pp];
    elseif (a(1)==4)&(a(3)==4)&(c(1,1)==4)&(c(1,2)==66)
        h=[u,p,b;uu,pp];
    else
        h=0;
    end
elseif a(2)==12
    if (a(1)==2)&(a(3)==2)&(c(1,1)==2)&(c(1,2)==58)
        h=[u,p,b;uu,pp];
    elseif (a(1)==3)&(a(3)==3)&(c(1,1)==3)&(c(1,2)==53)
        h=[u,p,b;uu,pp];
    elseif (a(1)==4)&(a(3)==4)&(c(1,1)==4)&(c(1,2)==67)
        h=[u,p,b;uu,pp];
    else
        h=0;
    end
elseif a(2)==13
    if (a(1)==1)&(a(3)==1)&(c(1,1)==1)&(c(1,2)==68)
        h=[u,p,b;uu,pp];
    elseif (a(1)==4)&(a(3)==4)&(c(1,1)==4)&(c(1,2)==78)
        h=[u,p,b;uu,pp];
    else
        h=0;
```

```
    end
elseif a(2)==14
    if (a(1)==1)&(a(3)==1)&(c(1,1)==1)&(c(1,2)==69)
        h=[u,p,b;uu,pp];
    elseif (a(1)==2)&(a(3)==2)&(c(2,1)==2)&(c(2,2)==70)
        h=[u,p,b;uu,pp];
    elseif (a(1)==4)&(a(3)==4)&(c(1,1)==4)&(c(1,2)==79)
        h=[u,p,b;uu,pp];
    else
        h=0;
    end
elseif a(2)==15
    if (a(1)==1)&(a(3)==1)&(c(1,1)==1)&(c(1,2)==70)
        h=[u,p,b;uu,pp];
    elseif (a(1)==2)&(a(3)==2)&(c(1,1)==2)&(c(1,2)==71)
        h=[u,p,b;uu,pp];
    elseif (a(1)==4)&(a(3)==4)&(c(1,1)==4)&(c(1,2)==80)
        h=[u,p,b;uu,pp];
    else
        h=0;
    end
elseif a(2)==16
    if (a(1)==2)&(a(3)==2)&(c(1,1)==2)&(c(1,2)==72)
        h=[u,p,b;uu,pp];
    elseif (a(1)==4)&(a(3)==4)&(c(1,1)==4)&(c(1,2)==81)
        h=[u,p,b;uu,pp];
    else
        h=0;
    end
else
```

Appendix C. H function

```
    h=0;
end
```

Appendix D

Shortest Path

This is the Floyd Warshall algorithm that can calculate the shortest path for the block.

```
n=16

D=   [0   1   100 100 1   100 100 100
      100 100 100 100 100 100 100 100;
      1   0   1   100 100 1   100 100
      100 100 100 100 100 100 100 100;
      100 1   0   1   100 100 1   100
      100 100 100 100 100 100 100 100;
      100 100 1   0   100 100 100 1
      100 100 100 100 100 100 100 100;
      1   100 100 100 0   1   100 100
      1   100 100 100 100 100 100 100;
      100 1   100 100 1   0   1   100
      100 1   100 100 100 100 100 100;
      100 100 1   100 100 1   0   1
      100 100 1   100 100 100 100 100;
```

```
      100 100 100 1   100 100 1   0
      100 100 100 1   100 100 100 100;
      100 100 100 100 1   100 100 100
      0   1   100 100 1   100 100 100;
      100 100 100 100 100 1   100 100
      1   0   1   100 100 1   100 100;
      100 100 100 100 100 100 1   100
      100 1   0   1   100 100 1   100;
      100 100 100 100 100 100 100 1
      100 100 1   0   100 100 100 1  ;
      100 100 100 100 100 100 100 100
      1   100 100 100 0   1   100 100;
      100 100 100 100 100 100 100 100
      100 1   100 100 1   0   1   100;
      100 100 100 100 100 100 100 100
      100 100 100 1   100 1   0   1  ;
      100 100 100 100 100 100 100 100
      100 100 100 1   100 100 1   0  ]
Pi= [0   1   0   0   1   0   0   0
      0   0   0   0   0   0   0   0  ;
      2   0   2   0   0   2   0   0
      0   0   0   0   0   0   0   0  ;
      0   3   0   3   0   0   3   0
      0   0   0   0   0   0   0   0  ;
      0   0   4   0   0   0   0   4
      0   0   0   0   0   0   0   0  ;
      5   0   0   0   0   5   0   0
      5   0   0   0   0   0   0   0  ;
      0   6   0   0   6   0   6   0
      0   6   0   0   0   0   0   0  ;
```

```
      0   0   7   0   0   7   0   7
      0   0   7   0   0   0   0   0  ;
      0   0   0   8   0   0   8   0
      0   0   0   8   0   0   0   0  ;
      0   0   0   0   9   0   0   0
      0   9   0   0   9   0   0   0  ;
      0   0   0   0   0   10  0   0
      10  0   10  0   0   10  0   0  ;
      0   0   0   0   0   0   11  0
      0   11  0   11  0   0   11  0  ;
      0   0   0   0   0   0   0   12
      0   0   12  0   0   0   0   12 ;
      0   0   0   0   0   0   0   0
      13  0   0   0   0   13  0   0  ;
      0   0   0   0   0   0   0   0
      0   14  0   0   14  0   14  0  ;
      0   0   0   0   0   0   0   0
      0   0   15  0   0   15  0   15 ;
      0   0   0   0   0   0   0   0
      0   0   0   16  0   0   16  0  ]
for k=1:n
    for i=1:n
        for j=1:n
            if D(i,j)<= (D(i,k)+D(k,j))
                Pi(i,j)=Pi(i,j);
            else  Pi(i,j)=Pi(k,j);
            end
            D(i,j)=min(D(i,j),D(i,k)+D(k,j));
   %         for p=1:16
   %             A(p)=D;B(p)=Pi
```

```
    %          end


          end
      end
      D
      Pi
end
```

Appendix E

Planning

The path for the robot is calculated and the atom plan is generated.

```
close all
clc
A =[

0     1     2     3     1     2     3     4
5     6     7     8     9    10    11    12;
2     0     2     3     1     2     3     4
5     6     7     8     9    10    11    12;
2     3     0     3     1     2     3     4
5     6     7     8     9    10    11    12;
2     3     4     0     1     2     3     4
5     6     7     8     9    10    11    12;
5     1     2     3     0     5     6     7
5     6     7     8     9    10    11    12;
```

```
2     6     2     3     6     0     6     7
5     6     7     8     9    10    11    12;
2     3     7     3     6     7     0     7
5     6     7     8     9    10    11    12;
2     3     4     8     6     7     8     0
5     6     7     8     9    10    11    12;
5     1     2     3     9     5     6     7
0     9    10    11     9    10    11    12;
2     6     2     3     6    10     6     7
10    0    10    11     9    10    11    12;
2     3     7     3     6     7    11     7
10   11     0    11     9    10    11    12;
2     3     4     8     6     7     8    12
10   11    12     0     9    10    11    12;
5     1     2     3     9     5     6     7
13    9    10    11     0    13    14    15;
2     6     2     3     6    10     6     7
10   14    10    11    14     0    14    15;
2     3     7     3     6     7    11     7
10   11    15    11    14    15     0    15;
2     3     4     8     6     7     8    12
10   11    12    16    14    15    16     0 ];


s = input('start position: ');
e = input('end position: ');
C = [];
i=1;
C(1) = A(s,e);
while (C(i) ~= 0)
```

```
    C(i+1)= A(s,C(i));
    i = i+1;
end
C;
d = length(C);
F =[];
F(1) = 0;
F(2) = C(d-1);
for i = 3:d
   F(i) = C(d+1-i);
end

F(d+1) = e;
for i=1:d
    FF(i)=F(i+1);
end
fprintf('Shortest path from start to end position')
FF
%move empty block next to the start position
s1 = 4;
e1 = F(2);
D = [];
i=1;
D(1) = A(s1,e1);
while (D(i) ~= 0)
    D(i+1)= A(s1,D(i));
    i = i+1;
end
D;
d1 = length(D);
```

```
if d1 == 2
    F(1) = 4;
    Q = [];
else
    F1 =[];
    F1(1) = D(d1-1);
  for i = 2:d1-1
   F1(i) = D(d1-i);
  end
  F1;
  for i = 1:d1-2
   F11(i) = F1(i+1);
  end
  for i = 1:d1-2
   F12(i) = F1(i);
  end
 Q = vertcat(F11,F12);
 F(1) = F1(d1-1);
end

F;
for i=2:d
    if  (F(i-1) == F(i+1))
        Q1 = [F(i);F(i+1)];
        Q = horzcat(Q,Q1);
    elseif  ((F(i-1) == 13) && (F(i) == 14)&&(F(i+1) == 15))
           || ((F(i-1) == 14) && (F(i) == 15)&&(F(i+1) == 16))
        B = [F(i)-5 F(i)-4 F(i)-3 F(i)+1 F(i);
             F(i)-1 F(i)-5 F(i)-4 F(i)-3 F(i)+1];
        Q = horzcat(Q,B);
```

```
elseif  ((F(i-1) == 15) && (F(i) == 14)&&(F(i+1) == 13))
       || ((F(i-1) == 16) && (F(i) == 15)&&(F(i+1) == 14))
    B = [F(i)-3 F(i)-4 F(i)-5 F(i)-1 F(i);
         F(i)+1 F(i)-3 F(i)-4 F(i)-5 F(i)-1];
    Q = horzcat(Q,B);
elseif  ((F(i-1) == 4) && (F(i) == 8)&&(F(i+1) == 12))
       || ((F(i-1) == 8) && (F(i) == 12)&&(F(i+1) == 16))
    B = [F(i)-5 F(i)-1 F(i)+3 F(i)+4 F(i);
    F(i)-4 F(i)-5 F(i)-1 F(i)+3 F(i)+4];
    Q = horzcat(Q,B);
elseif  ((F(i-1) == 12) && (F(i) == 8)&&(F(i+1) == 4))
       || ((F(i-1) == 16) && (F(i) == 12)&&(F(i+1) == 8))
    B = [F(i)+3 F(i)-1 F(i)-5 F(i)-4 F(i);
         F(i)+4 F(i)+3 F(i)-1 F(i)-5 F(i)-4];
    Q = horzcat(Q,B);
elseif ((F(i-1) == F(i)-1)&&(F(i+1) == F(i)+1))
    B = [F(i)+3 F(i)+4 F(i)+5 F(i)+1 F(i);
         F(i)-1 F(i)+3 F(i)+4 F(i)+5 F(i)+1];
    Q = horzcat(Q,B);
elseif  ((F(i-1) == F(i)+1)&&(F(i+1) == F(i)-1))
    B = [F(i)+5 F(i)+4 F(i)+3 F(i)-1 F(i);
         F(i)+1 F(i)+5 F(i)+4 F(i)+3 F(i)-1];
    Q = horzcat(Q,B);
elseif  ((F(i-1) == F(i)-1)&&(F(i+1) == F(i)+4))
    B = [F(i)+3 F(i)+4 F(i); F(i)-1 F(i)+3 F(i)+4];
    Q = horzcat(Q,B);
elseif  ((F(i-1) == F(i)+4)&&(F(i+1) == F(i)-1))
    B = [F(i)+3 F(i)-1 F(i); F(i)+4 F(i)+3 F(i)-1];
    Q = horzcat(Q,B);
elseif  ((F(i-1) == F(i)-4)&&(F(i+1) == F(i)+1))
```

```
        B = [F(i)-3 F(i)+1 F(i); F(i)-4 F(i)-3 F(i)+1];
        Q = horzcat(Q,B);
    elseif  ((F(i-1) == F(i)+1)&&(F(i+1) == F(i)-4))
        B = [F(i)-3 F(i)-4 F(i); F(i)+1 F(i)-3 F(i)-4];
        Q = horzcat(Q,B);
    elseif  ((F(i-1) == F(i)-1)&&(F(i+1) == F(i)-4))
        B = [F(i)-5 F(i)-4 F(i); F(i)-1 F(i)-5 F(i)-4];
        Q = horzcat(Q,B);
    elseif  ((F(i-1) == F(i)-4)&&(F(i+1) == F(i)-1))
        B = [F(i)-5 F(i)-1 F(i); F(i)-4 F(i)-5 F(i)-1];
        Q = horzcat(Q,B);
    elseif  ((F(i-1) == F(i)+4)&&(F(i+1) == F(i)+1))
        B = [F(i)+5 F(i)+1 F(i); F(i)+4 F(i)+5 F(i)+1];
        Q = horzcat(Q,B);
    elseif  ((F(i-1) == F(i)+1)&&(F(i+1) == F(i)+4))
        B = [F(i)+5 F(i)+4 F(i); F(i)+1 F(i)+5 F(i)+4];
        Q = horzcat(Q,B);
    elseif  ((F(i-1) == F(i)+4)&&(F(i+1) == F(i)-4))
        B = [F(i)+5 F(i)+1 F(i)-3 F(i)-4 F(i);
             F(i)+4 F(i)+5 F(i)+1 F(i)-3 F(i)-4];
        Q = horzcat(Q,B);
    elseif  ((F(i-1) == F(i)-4)&&(F(i+1) == F(i)+4))
        B = [F(i)-3 F(i)+1 F(i)+5 F(i)+4 F(i);
             F(i)-4 F(i)-3 F(i)+1 F(i)+5 F(i)+4];
        Q = horzcat(Q,B);
    else
        fprintf('ERROR')
    end
end
fprintf('Detailed movement of the block')
```

```
Q

G =[];

G(1) = 30;

%H = [];
%j=1;
j = length(Q);
for i = 1:j
   if ((Q(1,i) == 3)&&( Q(2,i) == 4))
       G(i+1) = 28;
   elseif ( (Q(1,i) == 2)&&( Q(2,i) == 3)  )
       G(i+1) = 27;
   elseif ( (Q(1,i) == 1)&&( Q(2,i) == 2)  )
       G(i+1) = 26;
   elseif ( (Q(1,i) == 7)&&( Q(2,i) == 8) )
       G(i+1) = 42;
   elseif ( (Q(1,i) == 6)&&( Q(2,i) == 7)  )
       G(i+1) = 41;
   elseif ( (Q(1,i) == 5)&&( Q(2,i) == 6)  )
       G(i+1) = 40;
   elseif ( (Q(1,i) == 11)&&( Q(2,i) == 12)  )
       G(i+1) = 56;
   elseif ( (Q(1,i) == 10)&&( Q(2,i) == 11)  )
       G(i+1) = 55;
   elseif ( (Q(1,i) == 9)&&( Q(2,i) == 10)   )
       G(i+1) = 54;
   elseif ( (Q(1,i) == 15)&&( Q(2,i) == 16)   )
       G(i+1) = 70;
```

```
elseif ( (Q(1,i) == 14)&&( Q(2,i) == 15)  )
    G(i+1) = 69;
elseif ( (Q(1,i) == 13)&&( Q(2,i) == 14) )
    G(i+1) = 68;
elseif ( (Q(1,i) == 4)&&( Q(2,i) == 3)  )
    G(i+1) = 30;
elseif ( (Q(1,i) == 3)&&( Q(2,i) == 2)  )
    G(i+1) = 29;
elseif ( (Q(1,i) == 2)&&( Q(2,i) == 1)  )
    G(i+1) = 28;
elseif ( (Q(1,i) == 8)&&( Q(2,i) == 7)  )
    G(i+1) = 44;
elseif ( (Q(1,i) == 7)&&( Q(2,i) == 6) )
    G(i+1) = 43;
elseif ( (Q(1,i) == 6)&&( Q(2,i) == 5))
    G(i+1) = 42;
elseif ( (Q(1,i) == 12)&&( Q(2,i) == 11))
    G(i+1) = 58;
elseif ( (Q(1,i) == 11)&&( Q(2,i) == 10))
    G(i+1) = 57;
elseif ( (Q(1,i) == 10)&&( Q(2,i) == 9))
    G(i+1) = 56;
elseif ( (Q(1,i) == 16)&&( Q(2,i) == 15))
    G(i+1) = 72;
elseif ( (Q(1,i) == 15)&&( Q(2,i) == 14))
    G(i+1) = 71;
elseif ( (Q(1,i) == 14)&&( Q(2,i) == 13))
    G(i+1) = 70;
elseif ( (Q(1,i) == 5)&&( Q(2,i) == 1))
    G(i+1) = 50;
```

```
elseif ( (Q(1,i) == 9)&&( Q(2,i) == 5))
    G(i+1) = 64;
elseif ( (Q(1,i) == 13)&&( Q(2,i) == 9))
    G(i+1) = 78;
elseif ( (Q(1,i) == 6)&&( Q(2,i) == 2))
    G(i+1) = 51;
elseif ( (Q(1,i) == 10)&&( Q(2,i) == 6))
    G(i+1) = 65;
elseif ( (Q(1,i) == 14)&&( Q(2,i) == 10))
    G(i+1) = 79;
elseif ( (Q(1,i) == 7)&&( Q(2,i) == 3))
    G(i+1) = 52;
elseif ( (Q(1,i) == 11)&&( Q(2,i) == 7))
    G(i+1) = 66;
elseif ( (Q(1,i) == 15)&&( Q(2,i) == 11))
    G(i+1) = 80;
elseif ( (Q(1,i) == 8)&&( Q(2,i) == 4))
    G(i+1) = 53;
elseif ( (Q(1,i) == 12)&&( Q(2,i) == 8))
    G(i+1) = 67;
elseif ( (Q(1,i) == 16)&&( Q(2,i) == 12))
    G(i+1) = 81;
elseif ( (Q(1,i) == 1)&&( Q(2,i) == 5))
    G(i+1) = 22;
elseif ( (Q(1,i) == 5)&&( Q(2,i) == 9))
    G(i+1) = 36;
elseif ( (Q(1,i) == 9)&&( Q(2,i) == 13))
    G(i+1) = 50;
elseif ( (Q(1,i) == 2)&&( Q(2,i) == 6))
    G(i+1) = 23;
```

```
   elseif ( (Q(1,i) == 6)&&( Q(2,i) == 10))
       G(i+1) = 37;
   elseif ( (Q(1,i) == 10)&&( Q(2,i) == 14))
       G(i+1) = 51;
   elseif ( (Q(1,i) == 3)&&( Q(2,i) == 7))
       G(i+1) = 24;
   elseif ( (Q(1,i) == 7)&&( Q(2,i) == 11))
       G(i+1) = 38;
   elseif ( (Q(1,i) == 11)&&( Q(2,i) == 15))
       G(i+1) = 52;
   elseif ( (Q(1,i) == 4)&&( Q(2,i) == 8))
       G(i+1) = 25;
   elseif ( (Q(1,i) == 8)&&( Q(2,i) == 12))
       G(i+1) = 39;
   elseif ( (Q(1,i) == 12)&& (Q(2,i) == 16))
       G(i+1) = 53;
   else
        fprintf('ERROR')
   end
%H(1) = B(G(1),G(2));
%while (H(j) ~= 0)
%    H(j+1)= B(B(j),B(j+1));
%    j = i+1;
%end
%  j=length(H);
end
fprintf('Path for the robot')
G
%from robot position to atom
Plan = [];
```

```
k=length(G);
for i=1:k-1
    if G(i+1) == (G(i)-2)
        H = [0 1 2 1 0 1 6 4 5 1];
        Plan = horzcat(Plan,H);
    elseif  G(i+1) == (G(i)+23)
        H = [8 1 0 1 6 4 5 1 1];
        Plan = horzcat(Plan,H);
    elseif  G(i+1) == (G(i)+24)
        H = [8 3 0 3 6 4 5 1 1];
        Plan = horzcat(Plan,H);
    elseif  G(i+1) == (G(i)-1)
        H = [0 1 7 1 0 1 6 4 5 1];
        Plan = horzcat(Plan,H);
    elseif  G(i+1) == (G(i)+1)
        H = [8 1 0 1 6 4 5 1 1];
        Plan = horzcat(Plan,H);
    elseif  G(i+1) == (G(i)-19)
        H = [8 1 0 1 6 4 5 ];
        Plan = horzcat(Plan,H);
    elseif  G(i+1) == (G(i)-18)
        H = [8 3 0 1 6 4 5 ];
        Plan = horzcat(Plan,H);
    elseif  G(i+1) == (G(i)-14)
        H = [1 0 1 7 1 0 1 6 4 5];
        Plan = horzcat(Plan,H);
    elseif  G(i+1) == (G(i)+14)
        H = [1 0 3 7 3 0 3 6 4 5 1 1];
        Plan = horzcat(Plan,H);
    elseif  G(i+1) == (G(i)-8)
```

```
        H = [3 8 3 0 3 6 4 5 3];
        Plan = horzcat(Plan,H);
    elseif  G(i+1) == (G(i)+6)
        H = [3 8 1 0 1 6 4 5 3];
        Plan = horzcat(Plan,H);
    elseif  G(i+1) == (G(i)-11)
        H = [1 8 1 0 1 6 4 5 1];
        Plan = horzcat(Plan,H);
    elseif  G(i+1) == (G(i)+3)
        H = [1 8 3 0 3 6 4 5 1];
        Plan = horzcat(Plan,H);
    else
        fprintf('ERROR')
    end
end
fprintf('Plan send to the robot')
Plan
```

Appendix F

C code sent to the robot

Atom plan generated for Khepera II robot. This code can be sent to the robot to guide the action of the robot.

```
/**********************************
 *
 * This Process  should Allow for
 * moving and pushing the block
 *
 *
 *
 *
 *
 *
 *
 *
 * June 2008
 * wenqi zhang
 *
 *
```

```
 *
 **********************************/


#include <sys/kos.h>
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

#define size 52


/**********************************
 *
 * Process 0
 * moving 1032 units forward

 **********************************/

int atom0()
{
    int p1, p2, f=99;

    uint32 mn1=0,mn2=1,maxSpeed=5,maxAccel=25;
    int KP=3000,KI=20,KD=4000;
    mot_stop();
    mot_config_profil_1m(mn1,maxSpeed,maxAccel);
    mot_config_profil_1m(mn2,maxSpeed,maxAccel);
    mot_config_position_1m(mn1,KP,KI,KD);
    mot_config_position_1m(mn2,KP,KI,KD);
```

```
   mot_put_sensors_2m(0,0);
   mot_new_position_2m(1032,1032);
   p1=mot_get_position(0);
   p2=mot_get_position(1);

   //while(p1 != 1032 || p2 != 1032){
     while ((p1 < 1029) || (p1 > 1035)
     || (p2 < 1029) || (p2 > 1035)){
     p1=mot_get_position(0);
     p2=mot_get_position(1);
   // printf("p1_%d -- p2_%d\r\n",p1,p2);
   //  printf("%d\r\n",f);
   }
   mot_stop();
   mot_config_position_1m(mn1,KP,KI,KD);
   mot_config_position_1m(mn2,KP,KI,KD);


   mot_put_sensors_2m(0,0);
   tim_suspend_task(500);
   f=100;
   printf("%d\r\n",f);
}
 /**********************************
*
* Process 1
* turn left *

**********************************/
```

Appendix F. C code sent to the robot

```
int atom1()
{
    int p1,p2, f=99;
    uint32 mn1=0,mn2=1,maxSpeed=5,maxAccel=25;
    int KP=3000,KI=20,KD=4000;
    mot_stop();

    mot_config_profil_1m(mn1,maxSpeed,maxAccel);
    mot_config_profil_1m(mn2,maxSpeed,maxAccel);
    mot_config_position_1m(mn1,KP,KI,KD);
    mot_config_position_1m(mn2,KP,KI,KD);
    mot_put_sensors_2m(0,0);

    mot_new_position_2m(520,-520);
    p1=mot_get_position(0);
    p2=mot_get_position(1);

   // while( p1 !=-520 || p2 !=520 ){
      while ((p1 < -523) || (p1 > -517)
      || (p2 < 517) || (p2 > 523)){
      p1=mot_get_position(0);
      p2=mot_get_position(1);
   //  printf("p1_%d -- p2_%d\r\n",p1,p2);
   //    printf("%d\r\n",f);
    }

    mot_stop();
    mot_config_position_1m(mn1,KP,KI,KD);
    mot_config_position_1m(mn2,KP,KI,KD);
```

```
    mot_put_sensors_2m(0,0);
    tim_suspend_task(500);
    f=100;
    printf("%d\r\n",f);
}
/**********************************
 *
 * Process 2
 * moving 4228 units forward
 *
 **********************************/

int atom2()
{
    int p1,p2, f=99;
    uint32 mn1=0,mn2=1,maxSpeed=5,maxAccel=25;
    int KP=3000,KI=20,KD=4000;
    mot_config_profil_1m(mn1,maxSpeed,maxAccel);
    mot_config_profil_1m(mn2,maxSpeed,maxAccel);
    mot_config_position_1m(mn1,KP,KI,KD);
    mot_config_position_1m(mn2,KP,KI,KD);


    mot_new_position_2m(4228,4228);
    p1=mot_get_position(0);
    p2=mot_get_position(1);

   // while( p1 !=4228 || p2 !=4228 ){
      while((p1 < 4225) || (p1 >4231)
```

```
      || (p2 < 4225) || (p2 > 4231))
      {
      p1=mot_get_position(0);
      p2=mot_get_position(1);
  //  printf("p1_%d -- p2_%d\r\n",p1,p2);
  //    printf("%d\r\n",f);
      }
    mot_stop();
    mot_config_position_1m(mn1,KP,KI,KD);
    mot_config_position_1m(mn2,KP,KI,KD);

    mot_put_sensors_2m(0,0);
    tim_suspend_task(500);
    f=100;
    printf("%d\r\n",f);

 }
 /**********************************
 *
 * Process 3
 *turn right *
 **********************************/

int atom3()
{
    int p1,p2, f=99;
    uint32 mn1=0,mn2=1,maxSpeed=5,maxAccel=25;
    int KP=3000,KI=20,KD=4000;
    mot_config_profil_1m(mn1,maxSpeed,maxAccel);
    mot_config_profil_1m(mn2,maxSpeed,maxAccel);
```

```
    mot_config_position_1m(mn1,KP,KI,KD);
    mot_config_position_1m(mn2,KP,KI,KD);



    mot_new_position_2m(-520,520);
    p1=mot_get_position(0);
    p2=mot_get_position(1);

    //while( p1 !=520|| p2 !=-520 ){
      while((p1 < 517) || (p2 > 523)
      || (p2 <-523) || (p2 > -517))
      {
      p1=mot_get_position(0);
      p2=mot_get_position(1);
  //   printf("p1_%d -- p2_%d\r\n",p1,p2);
  //    printf("%d\r\n",f);
    }

    mot_stop();
    mot_config_position_1m(mn1,KP,KI,KD);
    mot_config_position_1m(mn2,KP,KI,KD);

    mot_put_sensors_2m(0,0);
    tim_suspend_task(500);
    f=100;
    printf("%d\r\n",f);

}
```

```
/**********************************
 *
 * Process 4
 *moving 2044 units forward*
 **********************************/

int atom4()
{

    int p1,p2, f=99;
    uint32 mn1=0,mn2=1,maxSpeed=5,maxAccel=25;
    int KP=3000,KI=20,KD=4000;
    mot_config_profil_1m(mn1,maxSpeed,maxAccel);
    mot_config_profil_1m(mn2,maxSpeed,maxAccel);
    mot_config_position_1m(mn1,KP,KI,KD);
    mot_config_position_1m(mn2,KP,KI,KD);


    mot_new_position_2m(2044,2044);
    p1=mot_get_position(0);
    p2=mot_get_position(1);

   // while( p1 !=2044|| p2 !=2044 ){
   while((p1 < 2041) || (p1 >2047)
   || (p2 < 2041) || (p2 > 2047))
      {
      p1=mot_get_position(0);
      p2=mot_get_position(1);
   //  printf("p1_%d -- p2_%d\r\n",p1,p2);
```

```
   //   printf("%d\r\n",f);
    }

    mot_stop();
    mot_config_position_1m(mn1,KP,KI,KD);
    mot_config_position_1m(mn2,KP,KI,KD);

    mot_put_sensors_2m(0,0);
    tim_suspend_task(500);
    f=100;
    printf("%d\r\n",f);


}


/**********************************
 *
 * Process 5
 *moving 2044 units backward*
 **********************************/

int atom5()
{
    int p1,p2, f=99;
    uint32 mn1=0,mn2=1,maxSpeed=5,maxAccel=25;
    int KP=3000,KI=20,KD=4000;
    mot_config_profil_1m(mn1,maxSpeed,maxAccel);
    mot_config_profil_1m(mn2,maxSpeed,maxAccel);
    mot_config_position_1m(mn1,KP,KI,KD);
```

```
    mot_config_position_1m(mn2,KP,KI,KD);


    mot_new_position_2m(-2044,-2044);
    p1=mot_get_position(0);
    p2=mot_get_position(1);

   // while( p1 !=-2044|| p2 !=-2044 ){
   while((p1 < -2047) || (p1 >-2041)
   || (p2 < -2047) || (p2 > -2041))
      {
      p1=mot_get_position(0);
      p2=mot_get_position(1);
   //  printf("p1_%d -- p2_%d\r\n",p1,p2);
   //   printf("%d\r\n",f);
    }

    mot_stop();
    mot_config_position_1m(mn1,KP,KI,KD);
    mot_config_position_1m(mn2,KP,KI,KD);

    mot_put_sensors_2m(0,0);
    tim_suspend_task(500);
    f=100;
    printf("%d\r\n",f);
}


/**********************************
 *
```

```
 * Process 6
 *moving 130 units forward*
 **********************************/

int atom6()
{
    int p1,p2, f=99;
    uint32 mn1=0,mn2=1,maxSpeed=5,maxAccel=25;
    int KP=3000,KI=20,KD=4000;
    mot_config_profil_1m(mn1,maxSpeed,maxAccel);
    mot_config_profil_1m(mn2,maxSpeed,maxAccel);
    mot_config_position_1m(mn1,KP,KI,KD);
    mot_config_position_1m(mn2,KP,KI,KD);


    mot_new_position_2m(130,130);
    p1=mot_get_position(0);
    p2=mot_get_position(1);

   // while( p1 !=130|| p2 !=130 ){
   while((p1 < 127) || (p1 >133) || (p2 < 127) || (p2 > 133))
      {
      p1=mot_get_position(0);
      p2=mot_get_position(1);
    // printf("p1_%d -- p2_%d\r\n",p1,p2);
    //  printf("%d\r\n",f);
    }

    mot_stop();
    mot_config_position_1m(mn1,KP,KI,KD);
```

```
    mot_config_position_1m(mn2,KP,KI,KD);

    mot_put_sensors_2m(0,0);
    tim_suspend_task(500);
    f=100;
    printf("%d\r\n",f);

}

/**********************************
 *
 * Process 7
 *moving 2144 units forword*
 **********************************/

int atom7()
{
    int p1,p2, f=99;
    uint32 mn1=0,mn2=1,maxSpeed=5,maxAccel=25;
    int KP=3000,KI=20,KD=4000;
    mot_config_profil_1m(mn1,maxSpeed,maxAccel);
    mot_config_profil_1m(mn2,maxSpeed,maxAccel);
    mot_config_position_1m(mn1,KP,KI,KD);
    mot_config_position_1m(mn2,KP,KI,KD);


    mot_new_position_2m(2144,2144);
    p1=mot_get_position(0);
    p2=mot_get_position(1);
```

```
    //while( p1 !=2144|| p2 !=2144 ){
    while((p1 < 2141) || (p1 >2147)
    || (p2 < 2141) || (p2 > 2147))
      {
      p1=mot_get_position(0);
      p2=mot_get_position(1);
  //   printf("p1_%d -- p2_%d\r\n",p1,p2);
  //    printf("%d\r\n",f);
    }

    mot_stop();
    mot_config_position_1m(mn1,KP,KI,KD);
    mot_config_position_1m(mn2,KP,KI,KD);

    mot_put_sensors_2m(0,0);
    tim_suspend_task(500);
    f=100;
    printf("%d\r\n",f);
}

/**********************************
 *
 * Process 8
 *moving 3176 units forword*
 **********************************/

int atom8()
{
    int p1,p2, f=99;
    uint32 mn1=0,mn2=1,maxSpeed=5,maxAccel=25;
```

```
    int KP=3000,KI=20,KD=4000;
    mot_config_profil_1m(mn1,maxSpeed,maxAccel);
    mot_config_profil_1m(mn2,maxSpeed,maxAccel);
    mot_config_position_1m(mn1,KP,KI,KD);
    mot_config_position_1m(mn2,KP,KI,KD);


    mot_new_position_2m(3176,3176);
    p1=mot_get_position(0);
    p2=mot_get_position(1);

   // while( p1 !=3176|| p2 !=3176 ){
   while((p1 < 3173) || (p1 >3179)
   || (p2 < 3173) || (p2 > 3179))
      {
      p1=mot_get_position(0);
      p2=mot_get_position(1);
 //    printf("p1_%d -- p2_%d\r\n",p1,p2);
 //     printf("%d\r\n",f);
    }

    mot_stop();
    mot_config_position_1m(mn1,KP,KI,KD);
    mot_config_position_1m(mn2,KP,KI,KD);

    mot_put_sensors_2m(0,0);
    tim_suspend_task(500);
    f=100;
    printf("%d\r\n",f);
}
```

```
/**********************
*process 1
************************/

static int32 vIDProcess[1];

static void
process_1()
{ /*int receive = 9;

  for(;;)
 receive = getchar();
   scanf("%d", &receive);
   while (receive == 9)
   {
      scanf("%d", &receive);

   }
   tim_suspend_task(500);
   printf("%d",receive);*/
   int i;
   int a[size] = {0,1,2,1,0,1,6,4,5,1,0,1,7,1,0,1,6,4,5,1,0,1,7,
   1,0,1,6,4,5,1,8,3,0,3,6,4,5,1,8,3,0,3,6,4,5,1,8,3,0,3,6,4};

  /* a[0] = 0;
   a[1] = 1;
   a[2] = 4;
   a[3] = 1;
   a[4] = 0;
```

```
  a[5] = 1;
  a[6] = 0;*/

//int *pa;
//pa = &a[0];
for (i = 0; i < size; i++)
{
  if (a[i] == 0)
  {
     printf("%d\r\n",a[i]);
     atom0();
         }

  if (a[i] == 1)
  {
      printf("%d\r\n",a[i]);
      atom1();

      }

  if (a[i] == 2)
  {
      printf("%d\r\n",a[i]);
      atom2();

      }

  if (a[i] == 3)
  {
      printf("%d\r\n",a[i]);
```

```
        atom3();

        }

    if (a[i] == 4)
    {
        printf("%d\r\n",a[i]);
        atom4();


        }

    if (a[i] == 5)
    {
        printf("%d\r\n",a[i]);
        atom5();

        }

    if (a[i] == 6)
    {
        printf("%d\r\n",a[i]);
        atom6();

        }

    if (a[i] == 7)
    {
        printf("%d\r\n",a[i]);
        atom7();
```

```
        }

    if (a[i] == 8)
    {
        printf("%d\r\n",a[i]);
        atom8();

        }

   // else
   // {
   //     printf("ERROR\r\n");

   //     }

   tim_suspend_task(2000);


 }
 printf("%6s\r\n", "FINISH");

}
/**********************************
 * Main
 *
 *
 *
 *
 *
```

```
 *
*********************************/

int main(void)
{

  int32 status;

  static char prName_1[]="Process 1 : pick atom to run";


  com_reset ();
  var_reset ();
  sens_reset ();
  str_reset ();
  mot_reset ();
  tim_reset ();
  tim_suspend_task(2000);

  status = install_task (prName_1, 800, process_1);
  if (status == -1)
    exit (0);
  vIDProcess[1] = (uint32) status;

  return 0;
}
```

References

[1] K-Team S. A. *Khepera II User Manual*. K-Team, 1.1 edition, March 2002.

[2] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P. H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138:3 – 34, 1995.

[3] R. Alur, C. Courcoubetis, T. A. Henzinger, and P. H. Ho. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In A. Nerode P. Antsaklis, W. Kohn and S. Sastry, editors, *Hybrid Systems: Computation and Control*, volume 736 of *Lecture Notes in Computer Science*, pages 209–229. Springer-Verlag, 1993.

[4] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[5] R. Alur, R. Grosu, Y. Hur, V. Kumar, and I. Lee. Modular specification of hybrid systems in charon. In *Hybrid Systems: Computation and Control*, volume 1790 of *Lecture Notes In Computer Science*, pages 6–19. Springer-Verlag, 2000.

[6] E. Asarin, O. Bournez, T. Dang, O. Maler, and A. Pnueli. Effective synthesis of switching controllers for linear systems. In *Proceedings of the IEEE*, volume 88, pages 1011–1025, Jul 2000.

[7] P. Azema, G. Juanole, E. Sanchis, and Michel Montbernard. Specification and verification of distributed systems using prolog interpreted petri nets. In *ICSE '84: Proceedings of the 7th international conference on Software engineering*, pages 510–518, Piscataway, NJ, USA, 1984. IEEE Press.

[8] J. Baeten, J. Bergstra, and J. Klop. Decidability of bisimulation equivalence for process generating context-free languages. *Journal of the ACM*, 40(3):653–683, 1993.

[9] J. Baeten and W. P. Weijland. *Process Algebra*. Cambridge Tracts in Theoretical Computer Science 18. Cambridge University Press, 1991.

[10] A. Balluchi, L. Benvenuti, M. DiBenedetto, C. Pinello, and A. Sangiovanni-Vincentelli. Automotive engine control and hybrid systems: Challenges and opportunities. In *Proceedings of the IEEE*, volume 88, pages 888–912, Jul 2000.

[11] H. Bekic. Towards a mathematical theory of processes. Technical Report 25.125, IBM Laboratory Vienna, 1971.

[12] H Bekic and C. B. Jones. *Programming languages and their definition*. Springer-Verlag New York, Inc., New York, NY, USA, 1984.

[13] J. A. Bergstra and J. W. Klop. Fixed point semantics in process algebra. Technical Report IW 208, Mathematical Centre, Amsterdam, 1982.

[14] J. A. Bergstra and J. W. Klop. Process algebra for synchronous communication. *Information and Control*, 60(1–3):109–137, 1984.

[15] J. A. Bergstra and J. W. Klop. A convergence theorem in process algebra. In J. W. de Bakker and J. J. M. M. Rutten, editors, *Ten Years of Concurrency Semantics*, pages 164–195. World Scientific, 1992.

[16] A. Bouajjani, J-C. Fernandez, and N. Halbwachs. Minimal model generation. In *Proceedings of the 2nd International Workshop on Computer Aided Verification*, volume 531 of *Lecture Notes in Computer Science*, pages 197 – 203. 1991.

[17] R. W. Brockett. On the computer control of movement. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 534–540, 1988.

[18] R. W. Brockett. Formal languages for motion description and map making. In J. Bailleul, R. Brockett, and B. Donald, editors, *Robotics*, volume 41, pages 181–193. ACM, 1990.

[19] R. W. Brockett. Hybrid models for motion control systems. In JH. Trentelman and J. C. Willems, editors, *Perspectives in control*, pages 29–51. Birkhauser-Verlag, 1993.

[20] S. D. Brookes, C. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *Journal of the ACM*, 31(3):560–599, 1984.

[21] P. E. Caines and Y. J. Wei. Hierarchical hybrid control systems: A lattice theoretic formulation. *IEEE Transactions on Automatic Control*, 43(4):501–508, 1998.

[22] C. G. Cassandras and S. Lafortune. *Introduction to Discrete Event Systems*. Kluwer Academic, 2001.

[23] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Program Language System*, 8(2):244–263, 1986.

[24] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to Algorithms and Java*. McGraw-Hill Higher Education, 2002.

[25] B. Courcelle. *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics*. MIT Press, 1st edition, 1990.

[26] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[27] J. Davoren and A. Nerode. Logics for hybrid systems. In *Proceedings of the IEEE*, volume 88, pages 985–1010, Jul 2000.

[28] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool KRONOS. In *Hybrid Systems : Computation and Control*, volume 1066 of *Lecture Notes in Computer Science*, pages 208–219. Springer-Verlag, 1996.

[29] J. de Bakker and J. Zucker. Denotational semantics of concurrency. In *STOC '82: Proceedings of the fourteenth annual ACM symposium on Theory of computing*, pages 153–158, New York, NY, USA, 1982. ACM.

[30] J. de Bakker and J. Zucker. Processes and the denotational semantics of concurrency. *Information and Control*, 54:70–120, 1982.

[31] R. DeCarlo, M. Branicky, and S. Pettersson. Perspectives and results on the stability of hybrid systems. In *Proceedings of the IEEE*, volume 88, pages 1069–1082, Jul 2000.

[32] F. Delmotte, T. R. Mehta, and M. Egerstedt. Modebox: A software tool for obtaining hybrid control strategies from data. *IEEE Robotics and Automation Magazine*, 15(1):87–95, March 2008.

[33] D. Curtis Deno, Richard M. Murray, Kristofer S. J. Pister, and S. Shankar Sastry. Control primitives for robot systems. *IEEE Transactions on Systems, Man, and Cybernetics*, 22:183–193, Jan/Feb 1992.

[34] P. Devanbu and E. Wohlstadter. Evolution in distributed heterogeneous systems. In *Proceedings of the NSF Workshop on New Visions for Software Design and Productivity: Research and Applications*, 2001.

[35] J. B. Dugan and G. Ciardo. Stochastic petri net analysis of a replicated file system. *IEEE Transactions on Software Engineering*, 15:394–401, 1989.

[36] H. Ehrig. Introduction to the algebraic theory of graph grammars. In *Graph-Grammars and Their Application to Computer Science and Biology*, volume 73 of *Lecture Notes in Computer Science*, pages 1–69. Springer Berlin, 1979.

[37] E. A. Emerson. Temporal and modal logic. In Elsevier Science, editor, *Handbook of Theoretical Computer Science*, volume B, pages 995–1072. Elsevier Science, 1990.

[38] S. Engell, S. Kowalewski, C. Schulz, and O. Stursberg. Continuous discrete interactions in chemical processing plants. In *Proceedings of the IEEE*, volume 88, pages 1050–1068, Jul 2000.

[39] E. Frazzoli, M. A. Dahleh, and E. Feron. Maneuver-based motion planning for nonlinear systems with symmetries. *IEEE Transanctions on Robotics*, 21:1077–1091, 2005.

[40] Emilio Frazzoli. Explicit solutions for optimal maneuver-based motion planning. *Proceedings of 42nd IEEE Conference on Decision and Control*, 4:3372 – 3377, Dec. 2003.

[41] Martin Gardner. The hypnotic fascination of sliding-block puzzles. *Scientific American*, 210:122–130, 1964.

[42] V. Gavrilets, E. Frazzoli, B. Mettler, M. Piedmonte, and E. Feron. Aggressive maneuvering of small autonomous helicopters: A human-centered approach. *The International Journal of Robotics Research*, 20(10):795 – 807, 2001.

[43] U. Goltz and W. Reisig. CSP-programs as nets with individual tokens. *Advances in Petri Nets 1984*, pages 169–196, 1985.

[44] Ursula Goltz and Alan Mycroft. On the relationship of CCS and petri nets. In *Proceedings of the 11th Colloquium on Automata, Languages and Programming*, pages 196–208, London, UK, 1984. Springer-Verlag.

[45] M. Harrison, I. Havel, and A. Yehudai. On equivalence of grammars through transformation trees. *Theoretical Computer Science*, 9:173–205, 1979.

[46] Robert A. Hearn and Erik D. Demaine. Pspace-completeness of sliding-block puzzles and other problems through the nondeterministic constraint logic model of computation. *Theoretical Computer Science*, 343(1-2):72–96, October 2005.

[47] M. Hennessy. *Algebraic Theory of Processes*. MIT Press, 1st edition, 1988.

[48] M. Hennessy and R. Milner. On observing nondeterminism and concurrency. In J.W. de Bakker and J. van Leeuwen, editors, *Proceedings 7th International Colloquium on Automata, Languages and Programming*, volume 85 of *Lecture Notes in Computer Science*, pages 299–309. Springer Verlag, 1980.

[49] T. A. Henzinger. The theory of hybrid automata. In *Proceedings of the 11th Annual Symp Logic in Computer Science*, pages 278–292, 1996.

[50] T. A. Henzinger, P.-H. Ho, and H. Wong-Toi. A user guide to HYTECH. In E. Brinksma, W.R. Cleaveland, K.G. Larsen, T. Margaria, and B. Steffen, editors, *TACAS 95: Tools and Algorithms for the Construction and Analysis of Systems*, volume 1019 of *Lecture Notes in Computer Science*, pages 41–71. Springer-Verlag, 1995.

[51] T. A. Henzinger, P. W. Kopke, A. Puri, and P. Varaiya. What's decidable about hybrid automata. *Computer System Science*, 57:94–124, 1998.

[52] Y. Hirshfeld and M. Jerrum. Bisimulation equivalence is decidable for normed process algebra. Technical Report ECS-LFCS-98-386, School of Informatics at the University of Edinburgh, May 1998.

[53] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[54] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[55] A. Holt. Introduction to occurrence systems. In *Associative Information Techniques*, pages 175–203, New York, USA, 1971. American Elsevier.

[56] A. Holt and F. Commoner. Events and conditions: introduction. In *Record of the Project MAC conference on concurrent systems and parallel computation*, pages 3–52, New York, NY, USA, 1970. ACM.

[57] A. Holt, H. Saint, R. Shapiro, and S. Warshall. Final report of the information systems theory project. Technical Report RADCTR-68-305, New York: Griffiss Air Force Base, 1968.

[58] J. E. Hopcroft, J. T. Schwarz, and M. Sharir. On the complexity of motion planning for multiple independent objects:pspace-hardness of the 'warehouseman's problem. *International Journal of Robotics Tesearch*, 3(4):76–88, 1984.

[59] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 2007.

[60] R. Horowitz and P. Varaiya. Control design of an automated highway system. In *Proceedings of the IEEE*, volume 88, pages 913–925, Jul 2000.

[61] D. Hristu and S. Anderson. Directed graphs and motion description language for robot navigation. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 2689–2694, 2002.

[62] D. Hristu and S. Anderson. Symbolic feedback control for navigation. *IEEE Transactions on Automatic Control*, 51:926–937, June 2006.

[63] D. Hristu, P. Krishnaprasad, S. Anderson, F. Zhang, L. D'Anna, and P. Sodre. The MDLe engine: A software tool for hybrid motion control. Technical Report 2000-54, Institute for Systems Research, University of Maryland, 2000.

[64] D. Hristu-Varsakelis, M. Egerstedt, and P. Krishnaparsad. On the structural complexity of the motion description language MDLe. In *Proceedings of the 42nd IEEE Conference on Descision and Control*, pages 3360–3365, 2003.

[65] E. Klavins. Automatic synthesis of controllers for distributed assembly andformation forming. In *IEEE International Conference on Robotics and Automation*, volume 3, pages 3296–3302, 2002.

[66] E. Klavins, R. Ghrist, and D. Lipsky. A grammatical approach to self-organizing robotic systems. *IEEE Transactions on Automatic Control*, 51:949–962, 2006.

[67] Werner E. Kluge. Reduction, data flow and control flow models of computation. In *Proceedings of an Advanced Course on Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986-Part II*, pages 466–498, London, UK, 1987. Springer-Verlag.

[68] L. Kovar, M. Gleicher, and F. Pighin. Motion graphs. *ACM Transactions on Graphics (Special Issue: Proc. ACM SIGGRAPH 2002)*, 21(3):473–482, 2002.

[69] H. Kress-Gazit, G. Fainekos, and G. J. Pappas. From structured english to robot motion. In *IEEE/RSJ International Conference on Robots and Systems*, pages 2717–2722, 2007.

[70] G. Lafferriere, G. J. Pappas, and S. Yovine. A new class of decidable hybrid systems. In *Hybrid Systems : Computation and Control*, volume 1569 of *Lecture Notes in Computer Science*, pages 137–151. Springer Verlag, 1999.

[71] Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a Nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, Oct 1997.

[72] C. Livadas, J. Lygeros, and N. Lynch. High-level modeling and analysis of the traffic alert and collision avoidance system (tcas). In *Proceedings of the IEEE*, volume 88, pages 926–948, Jul 2000.

[73] J. Lygeros. Lecture notes on hybrid systems. Notes for an ENSIETA workshop, February–June 2004.

[74] J. Lygeros, D. Godbole, and S. Sastry. Simulation as a tool for hybrid system design. In *Proceeding of the 5th Annual Conference on AI, Simulation, and Planning, in High Autonomy Systems*, pages 16–22, 1994.

[75] J. Lygeros, D. N. Godbole, and S. Sastry. Verified hybrid controllers for automated vehicles. *IEEE Transactions on Automatic Control*, 43:522 – 539, 1998.

[76] J. Lygeros, G. J. Pappas, and S. Sastry. An approach to the verification of the center-tracon automation system. In *Hybrid Systems : Computation and Control*, volume 1386 of *Lecture Notes in Computer Science*, pages 289–304. Springer Verlag, 1998.

[77] D. Mandrioli, R. Zicari, C. Ghezzi, and F. Tisato. Modeling the ada task system by petri nets. *Computer Languages*, 10(1):43–61, 1985.

[78] V. Manikonda, J. Hendler, and P. Krishnaprasad. Formalizing behavior-based planning for nonholonomic robots. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 142–149, August 1995.

[79] V. Manikonda, P. Krishnaprasad, and J. Hendler. A motion description language and a hybrid architecture for motion planning with nonholonomic robots. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 2021–2028, May 1995.

[80] V. Manikonda, P. Krishnaprasad, and J. Hendler. Languages, behaviors, hybrid architectures and motion control. In J. Baillieul and J. C. Willems, editors, *Mathematical Control Theory*, pages 200–226. Springer-Verlag, 1998.

[81] A. Marigo and A. Bicchi. Steering driftless nonholonomic systems by control quanta. In *Proceedings of the 37th IEEE Conference on Decision and Control*, pages 4164–4169, Tampa, FL, USA, 1998.

[82] Mazurkiewicz. Concurrent program schemes and their interpretations. Technical Report DAIMI PB-78, Aarhus University, 1977.

[83] N. H. McClamroch and I. Kolmanovsky. Performance benefits of hybrid controller design for linear and nonlinear systems. In *Proceedings of the IEEE*, volume 88, pages 1083–1096, Jul 2000.

[84] G.J. Milne and R. Milner. Concurrent processes and their syntax. *Journal of the ACM*, 26(2):302–321, 1979.

[85] R. Milner. *A Calculus of Communicating Systems*. Springer Verlag,1980.

[86] R. Milner. An approach to the semantics of parallel programs. In *Proceedings of the Conference on Information Theory*, pages 285–301, 1973.

[87] R. Milner. A mathematical model of computing agents. In H.E. Rose and J.C. Shepherdson, editors, *Proceedings Logic Colloquium*, volume 80 of *Studies in Logic and the Foundations of Mathematics*, pages 157–174. North-Holland, 1975.

[88] R. Milner. Synthesis of communicating behaviour. In J. Winkowski, editor, *Proceedings 7th Mathmatical Foundations of Computer Science*, volume 64 of *Lecture Notes in Computer Science*, pages 71–83. Springer Verlag, 1975.

[89] R. Milner. Flowgraphs and flow algebras. *Journal of the ACM*, 26(4):794–818, 1979.

[90] R. Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, 25:267–310, 1983.

[91] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[92] T. Murata. Petri nets: Properties, analysis and applications. In *Proceedings of the IEEE*, volume 77, pages 541–580, 1989.

[93] X. Nicolin, A. Olivero, J. Sifakis, and S. Yovine. An approach to the description and analysis of hybrid automata. In *Hybrid Systems: Computation and Control*, volume 736 of *Lecture Notes in Computer Science*, pages 149–178. Springer-Verlag, 1993.

[94] M. T. Ozsu. Modeling and analysis of distributed database concurrency control algorithms using an extended petri net formalism. *IEEE Transactions on Software Engineering*, 11(10):1225–1240, 1985.

[95] G. J. Pappas and S. Sastry. Towards continuous abstractions of dynamical and control systems. In A. Nerode P. Antsaklis, W. Kohn and S. Sastry, editors, *Hybrid Systems: Computation and Control*, volume 1273 of *Lecture Notes in Computer Science*, pages 329–341. Springer-Verlag, 1997.

[96] D. M. R. Park. Concurrency and automata on infinite sequences. In P. Deussen, editor, *Proceedings 5th Graph Isomorphism Conference*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183. Springer Verlag, 1981.

[97] D. Pepyne and C. Cassandras. Optimal control of hybrid systems in manufacturing. In *Proceedings of the IEEE*, volume 88, pages 1108–1123, Jul 2000.

[98] James Lyle Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice Hall PTR, NJ, USA, 1981.

[99] C. A. Petri. Concepts of net theory. In *Mathematical Foundations of Computer Science*, pages 137–146, 1973.

[100] C. A. Petri. Concurrency. In *Advanced Course on General Net Theory of Processes and Systems*, pages 251–260, 1975.

[101] C. A. Petri. Introduction to general net theory. In *Advanced Course on General Net Theory of Processes and Systems*, pages 1–19, 1975.

[102] C. A. Petri. Tools of general net theory (abstract). In *PNPM '87: The Proceedings of the Second International Workshop on Petri Nets and Performance Models*, page 2, Washington, DC, USA, 1987. IEEE Computer Society.

[103] C. A. Petri and W. Reisig. Petri net. *Scholarpedia*, 3(4):6477, 2008.

[104] A. Platzer and E. M. Clarke. The image computation problem in hybrid systems model checking. In *Hybrid Systems: Computation and Control*, volume 4416 of *Lecture Notes in Computer Science*, pages 473–486. Springer–Verlag, 2007.

[105] Plotkin and D. Gordon. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, 1981.

[106] Plotkin and D. Gordon. The origins of structural operational semantics. *Journal of Logic and Algebraic Programming*, 60–61:3–15, 2004.

[107] Amir Pnueli. The temporal logic of programs. In *SFCS '77: Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, pages 46–57, Washington, DC, USA, 1977. IEEE Computer Society.

[108] A. Puri and P. Varaiya. Decidability of hybrid systems with rectangular differential inclusions. In *Computer Aided Verification*, pages 95–104, 1994.

[109] J. P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proceedings of the 5th Colloquium on International Symposium on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351. Springer Berlin, 1982.

[110] M. Rem. Partially ordered computations, with applications to vlsi-design. In J.W. de Bakker and J. van Leeuwen, editors, *Foundations of Computer Science IV, part 2.*, volume 159 of *Mathematical Centre Tracts*, pages 1–44. 1983.

[111] Stuart Russell and Peter Norvig. *Artificial intelligence: A modern approach*. Prentice Hall, 1st edition, 1995.

[112] Géraud Sénizergues. The equivalence problem for deterministic pushdown automata is decidable. *ICALP '97: Proceedings of the 24th International Colloquium on Automata, Languages and Programming*, pages 671–681, 1997.

[113] M. Sipser. *Introduction to the Theory of Computation*. PWS Publishing Company, 1997.

[114] J. Slocum and D. Sonnevel. *The 15 Puzzle*. Slocum Puzzle Foundation, 2006.

[115] T. Smigelski, T. Murata, and M. Sowa. A timed petri net model and simulation of a dataflow computer. In *International Workshop on Timed Petri Nets*, pages 56–63, Washington, DC, USA, 1985. IEEE Computer Society.

[116] M. Song, T. J. Tarn, and N. Xi. Integration of task scheduling, action planning, and control in robotic manufacturing. In *Proceedings of the IEEE*, volume 88, pages 1097–1107, Jul 2000.

[117] Colin Stirling. Decidability of DPDA equivalence. *Theoretical Computer Science*, 255(1–2):1–31, 2001.

[118] Herbert Tanner, Jorge Piovesan, and Chaouki T. Abdallah. Discrete asymptotic abstractions of hybrid systems. In *45th IEEE Conference on Decision and Control*, pages 917–922, San Diego, CA, USA, 2006.

[119] C. Tomlin, G. J. Pappas, and S. Sastry. Conflict resolution for air traffic management: A study in muti-agent hybrid systems. *IEEE Transactions on Automatic Control*, 43(4):509–521, 1998.

[120] P. Varaiya. Smart cars on smart roads: Problems of control. *IEEE Transactions on Automatic Control*, 38:195 – 207, 1993.

[121] S. Yovine. KRONOS: A verification tool for real-time systems. *Springer International Journal of Software Tools for Technology Transfer*, 1:123–133, 1997.