Network-Level Control of Collaborative UAVs

by

Joshua Alan Love

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Engineering - Mechanical Engineering

in the

GRADUATE DIVISION
of the
UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:
Professor J. Karl Hedrick, Chair
Professor Francesco Borrelli
Professor Raja Sengupta

Fall 2011

Network-Level Control of Collaborative UAVs

Abstract

Network-Level Control of Collaborative UAVs

by

Joshua Alan Love
Doctor of Philosophy in Engineering - Mechanical Engineering

University of California, Berkeley

Professor J. Karl Hedrick, Chair

This dissertation addresses how a single human operator can interactively control a network of collaborative unmanned aerial vehicles (UAVs). It provides a model of computation for network-level controllers. These network-level controllers enable a single human operator to monitor, order, and supervise the progress of an entire network of autonomous collaborative UAVs. As the human operator observes the network's progress, they may make changes by applying runtime patches to the network-level controller. The resulting network-level controller can be analyzed to verify that it meets required conditions.

UAV networks produce significantly more information than any single human can concentrate on. To ease this cognitive burden, the human operator should interact with the UAVs at an appropriately high level of abstraction. They should focus on monitoring what is being done, deciding what should be done, and determining in what order. Other lower-level decisions can be automated by the network. To this end, Petri nets are used as a theoretical basis for stating network-level controllers. Petri nets have a graphical description that is extremely intuitive. They also convey exactly what is being done, what will be done, and in what order using a network-focused perspective.

Developing a novel Petri net-based model of computation for network-level control is the main contribution of this work. This includes forming the syntax and semantics for network-level controllers. Other related contributions include identifying invariant and analyzable properties of the network-level controllers, proofs of their correctness, and interpretations of their meanings. These allow a human operator to understand and assert that a proposed controller is indeed correct. A runtime patching language to enable modifications by the human operator is another contribution.

The theoretical concepts behind network-level controllers provide a mathematical basis for the more concrete Collaborative Sensing Language (CSL). This formal XML-based language allows a precise specification of network-level controllers for UAVs. This dissertation also describes an implementation that enabled CSL for the UAV fleet at the Center for Collaborative Control of Unmanned Vehicles (C3UV). CSL and its implementation are also original contributions.

Previous related research has focused on the detailed off-line specification of individual reactive UAV behaviors. This creates a fixed preprogrammed network which produces a specific but predetermined behavior. The various existing alternatives are not well suited for interactive control of a network by a single human operator. They either suffer from no graphical description (process algebras), excessive information (hybrid systems), an individual component and not network focus (networks of finite state machines), a fixed dimension network, or a static specification that cannot be affected on-line. The novelty and usefulness of this dissertation lies in its ability to address these issues.

Professor J. Karl Hedrick
Dissertation Committee Chair

Dedicated to my parents, Randy and Dixie Love, for their continual and unwavering support and the belief that I can do anything I set my mind to doing, even when it sounds ridiculous.

Contents

List of Figures

List of Tables

Symbols

Variables

$\mathcal{R}$	The set of real numbers
$\mathcal{Z}$	The set of integers $\{... - 1, 0, 1...\}$
$\mathcal{N}$	The set of natural numbers $\{0, 1, 2...\}$
$\mathcal{N}_+$	The set of positive integers $\{1, 2...\}$

Acknowledgments

This dissertation would not have been possible without the guidance and patience of my co-advisors Professors J. Karl Hedrick and Raja Sengupta. They believed in and encouraged my abilities in areas ranging well beyond my expertise and comfort zone. Through these experiences I have gained more than knowledge about unfamiliar topics; I have gained the confidence necessary to master new topics and situations. Special thanks to Professor Sengupta for acting as the technical advisor on my disseration. The hours of feedback helped produce a more complete and cohesive dissertation.

I would also like to thank the other members of the Center for Collaborative Control of Unmanned Vehicles (C3UV), the Vehicle Dynamics Lab (VDL), and members of the Civil Systems program. The many conversations about related and unrelated topics have helped stimulate my interests and expose me to many different perspectives that I may have otherwise missed.

Additionally, this dissertation and my research were financially supported by a UC Berkeley Graduate Fellowship and then a National Defense Science and Engineering Graduate Fellowship (NDSEG). These fellowships allowed me the freedom to pursue this ‘non-traditional’ line of research.

Finally, I would like to thank the many fine teachers and professors that have poured years of effort into my education. Hopefully I have, and will continue to, seize the opportunities you opened before me.

Chapter 1

Introduction

1.1 Motivation

Unmanned aerial vehicles (UAVs) seem like a very recent military development. In the beginning of the 21st century they gained substantial press due to their heavy use in Operation Enduring Freedom in Afghanistan and Operation Iraqi Freedom in Iraq. Thanks to the continual news coverage of these conflicts and the release of UAV video streams, UAV technologies were thrust into the public consciousness. These remotely piloted vehicles allowed the United States to exercise its air superiority with less risk and longer endurance than manned aircraft. The pilots controlling the drones over Afghanistan could be stationed in an air conditioned building within the safety of the continental United States. The pilots could even switch in and out so that UAVs with 24+ hour endurances could be remotely operated with 8 hour shifts.

While the technology was new to the general public, it was envisioned at least as early as 1917 when the United States pursued building an unmanned flying bomb for World War I. In World War II Nazi Germany developed the V-1 Rocket, which did not gather intelligence information, but was one of the first successful unmanned aerial vehicles in production. U.S. research continued in waves and during the Vietnam Conflict UAVs were secretly flown over China, Laos, North Vietnam, and other South Asian countries. These UAVs performed tasks similar to 'modern' UAVs: collected photographs, took electronic measurements, made damage assessments, and dropped propaganda leaflets [52]. In 2000 the global military UAV market was estimated at $2 billion and predicted to grow to $42 billion by 2008. With the U.S. then consuming a third of this market, it was anticipated to spend $13.5 billion in 2008 [84]. More recently it was estimated that from 2010 to 2015 the U.S. will spend at least $62 billion on UAV technology (roughly $12.5 billion per year) [18].

This substantial investment not only provided cutting-edge military systems that help keep U.S. service members safe, but also spurred on research and development. Similarly to how NASA's Space Race investment helped produce integrated circuits and computers, UAV research required sophisticated electronics, more accurate and compact sensors, reliable and fast wireless communications, autopilots, and embedded computation. The progress made on UAVs since 1917 has matured these technologies to the point where new non-military applications are being developed and conceived at an amazing rate. These commercial off-the-shelf autopilots, sensors, actuators, and even complete UAVs allow an organization to take any idea and quickly turn it into a testable reality.

One of the first non-military UAV applications was envisioned by the U.S. Department of Homeland Security. They now use UAVs to conduct border patrol enforcement and drug trafficking interdiction [38]. The UAVs can stay aloft for hours and monitor miles of the border, tracking illegal entrants as necessary. As UAV use within the government became more common, the potential uses of UAVs were expanded to include: measuring and monitoring volcanic activity [148], measuring the health of rangelands and crops and general ecological health [157, 182], taking population counts of animal species [152], providing a mobile communications backbone that could be used in the recovery after national disasters [160], monitoring traffic congestion [170], and measuring chemical weapons or pollutants in the atmosphere [108]. These are just a few of the many public and private organizations and applications that are currently being developed.

For many existing and future applications, individual UAVs may hold significant advantages over manned aircraft. They are just as mobile, are more easily packed and transported, are often significantly cheaper, can have longer endurance, and do not put a human pilot at risk. These qualities almost assure the continued propagation of UAV technology for specific applications where a single manned aircraft can be replaced. However, it is unlikely that UAV technology will stop with single UAV applications.

As the applications grow in complexity to require coverage of larger areas or time-critical completions, it may be necessary to utilize several UAVs as a collaborative network. Despite border patrol UAVs flying at high altitudes and having powerful optics, more than one is necessary to provide coverage of the border at any moment. With only a single UAV patrolling the border, the time separation between observing any one crossing of the border would be hours. As more UAVs are added to the patrol this can be brought down significantly. Similarly, using several UAVs to form a communications backbone will provide emergency communications to a much larger area than a single UAV could provide. Most of the individual UAV applications can be extended to collaborative systems that either perform the objectives faster or over a larger area, thus expanding their usefulness and impact.

Part of the allure of UAV networks is an increase in autonomy. Both individual UAVs and networks of UAVs come in various levels of automation. Many of the original UAVs were entirely automated (e.g. World War II's Nazi V-1 Rocket). They were given a target at start-up and launched. It was later that advanced circuitry and communications made remote operation possible. Currently, the MQ-1 Predator is a remotely piloted vehicle with minimal automation augmenting the remote pilot's commands. These UAVs do not intend to fully automate the system, but merely remove the human operator from danger. For large missions that require multiple Predators, the number of pilots required is directly proportional to the number of UAVs; if 4 Predators are needed then so are 4 pilots. The pilots can communicate with each other verbally and make collective decisions as if they were still embedded in the physical aircraft. There are many intermediate levels of UAV automation between fully automated with almost no human interaction and remotely piloted with almost no automation. One of the potential benefits of automation is reducing the manpower necessary to conduct UAV missions. Automated systems also make decisions faster and can choose optimal solutions that may be hard for a human to identify. However, automated systems still respond poorly to unexpected situations.

Depending on the application, different levels of autonomy are appropriate. Volcano monitoring could easily be fully autonomous with the path and objectives specified before the flight begins. Offensive military applications (bombardments) will likely remain remotely operated due to the high risks and collateral damage that could occur from any 'bug' in the code.

This dissertation intends to address UAV networks that fall between fully autonomous and remotely operated. This work provides a method for a single human operator to supervise a collaborative network of UAVs. As the network executes its tasks, the human operator can interactively modify the tasks and the order they are to be performed. Since the human operator is interactively modifying the network's intended behavior it is clearly not a fully autonomous system. Neither is it a system of remotely operated vehicles; flying a single UAV manually is so difficult that flying several manually at the same time is practically impossible. Thus, the system described is an example of mixed-initiative automation where the human interaction enters at the highest and most abstract levels of control. The human operator supervises what is being done, what will be done, and in what order while the UAVs executes these tasks autonomously.

1.2 Contributions

The contributions made in this dissertation are:

- This dissertation presents the syntax and semantics for a novel model of computation that specifies network-level controllers. These concepts allow both a rigorous mathematical definition along with an abstracted graphical interpretation. The simplicity and focus of the graphical interpretation allows a human operator to understand the network-level controller at an appropriate level.

- Invariant and analyzable properties of the network-level controllers are identified and interpreted based on the developed syntax and semantics. These properties allow a human operator to check that the network-level controller specified is indeed 'correct'.

- The network-level controller concept is extended with runtime patching to allow the human operator to drastically alter the network-level controller during operation in a manner that is guaranteed to be correct.

- The Collaborative Sensing Language (CSL) is developed as an XML-based domain specific language to specify network-level controllers for Intelligence Surveillance and Reconnaissance (ISR) applications by UAVs.

- An implementation of the Collaborative Sensing Language is described that allows one to specify and execute CSL network-level controllers.

1.3 Greater Applicability

While this dissertation focuses on the network-level control of a network of collaborative UAVs performing ISR tasks, the concepts discussed could be useful in other areas. Obviously the type of tasks performed by the UAV network could be extended or modified from ISR to another domain of interest. UAVs could be used to drop off medical supplies in emergencies or fight forest fires. In the motivation above several alternative UAV applications were discussed, but this list will grow as UAVs continue to prove themselves capable and economical solutions.

There is no reason why the concepts provided for network-level control must be confined to UAV networks. Unmanned ground vehicles (UGVs), unmanned surface vehicles (USVs), and unmanned underwater vehicles (UUVs) provide additional types of networks that could be controlled in this manner. A team of collaborative UUVs could have been very beneficial to measure and monitor the situation at the Deepwater Horizon oil spill. They could have potentially measured and tracked the oil patches at different depths of the ocean in an organized and automated manner.

Additionally, network-level control could be useful in applications that do not involve mobile robots. If a system of resources has some allocation layer to assign tasks to individual resources, network-level control could be implemented on top of this to provide a way for a human operator to supervise the network at a high level of abstraction.

1.4 Contents of Dissertation

Chapter 2 will discuss various existing modeling formalisms. It provides both the background material necessary to understand this dissertation and the research most relevant and related to this dissertation. It is organized by individual modeling formalism. Each section discusses the purely theoretical definitions of a formalism and then presents some concrete applications and research that utilize that specific modeling formalism.

Chapter 3 will present the theoretical concepts behind network-level control. It will discuss why Petri nets are the appropriate foundation for the new model of computation. It will present the syntax and semantics for network-level controllers.

Chapter 4 will present several invariant properties of network-level controllers. These properties will be results of the network-level controller semantics and will apply to all controllers. Several analyzable properties of network-level controllers will also be discussed. These properties (reachability, boundedness, liveness, deadlock) will depend on the individual network-level controller.

Chapter 5 will present a runtime patching language for network-level controllers. It will give a BNF syntax and a structural operational semantics to describe how a human operator can make modifications to the network-level controller.

Chapter 6 will connect the runtime patching language and the network-level controller. It will explain how a network-level controller being modified by runtime patches operates. It will also discuss invariant and analyzable properties for such controllers.

Chapter 7 presents the XML-based Collaborative Sensing Language (CSL). CSL was the precursor to the concepts of the previous chapters. It produces network-level controllers for ISR UAV applications at the Center for Collaborative Control of Unmanned Vehicles (C_3UV).

Finally, Chapter 8 will summarize the benefits and detriments of this approach to controlling a network. It will make suggestions for future research in this area.

Chapter 2

Modeling Systems of Systems

Systems of systems is a concept synonymous with complexity. Individual systems, themselves, are often quite complex. Be they electrical, fluid, thermal, mechanical, or information systems their complexities continue to provide many active areas of research. The different types of individual systems can be represented by different types of models and controlled using different types of control techniques. Part of the art of control is knowing which tools are appropriate for which occasions. The notion of taking these already complex components and combining them into a larger system compounds the existing complexities with new complexities due to the components' interactions. Systems of systems addresses these interactions and provides a useful set of modeling formalisms, analysis tools, and control techniques to aid in the development of these ever-growing systems. Obviously, this includes networks composed from individual UAVs.

One commonality of all engineered systems is time. A system's behavior evolves as time progresses. No matter how complex, the systems considered cannot violate time's forward progression. Since one cannot go back in time to correct previous actions, control must be deliberately and carefully crafted based on a model that can faithfully predict future behavior. Each modeling formalism to be discussed will have some notion of time, but these notions will often be different.

The most intuitive model of time is the continuous-time model used in modern control [69]. In this model, time comes from the set of real numbers, and between every two time instances there is another time instance, see figure 2.1a. Behaviors of continuous-time systems are represented as trajectories, which are curves describing the *state* of the system changing with time.

As digital processing became more and more prevalent, discrete-time models provided a theoretical basis for additional control tools that integrated computers with analog components [139]. Digital computers can only operate at specific clock speeds. Regardless of how fast they operate, they are by definition and design not analog. To bridge this difference, computers can use analog-to-digital samplers to convert the

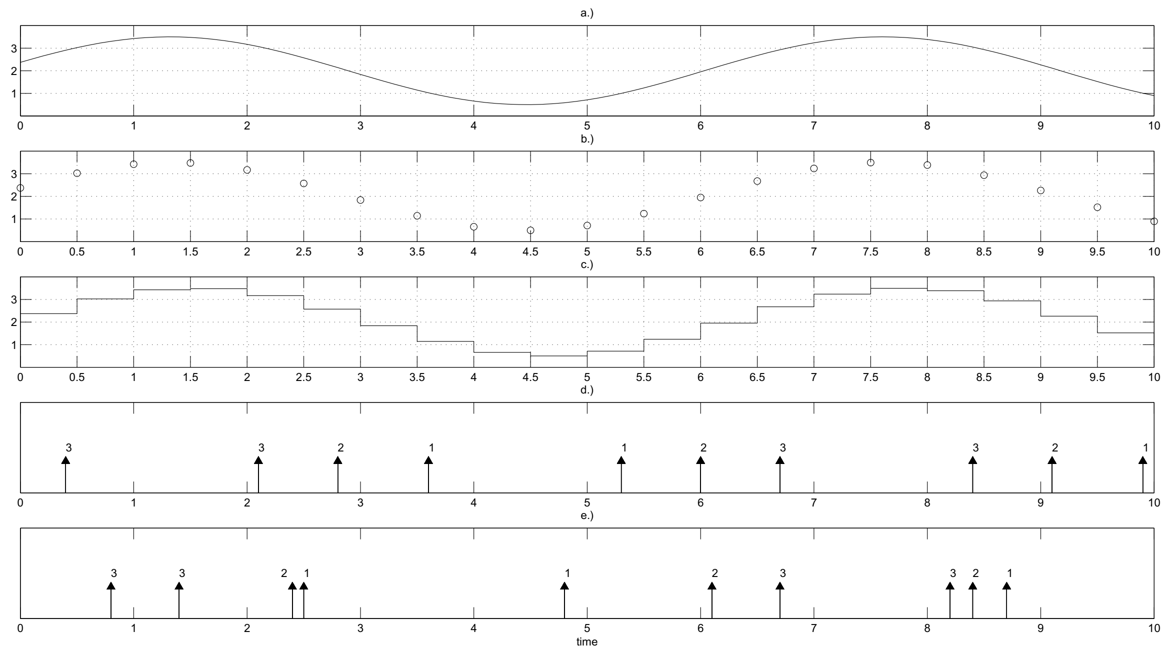


Figure 2.1: a) the behavior of a continuous-time model represented as a curve. b) a sampled sequence generated by sampling a) at a rate of $\Delta t = 0.5$. c) a continuous-time signal generated from a zero-order hold D/A converter and the sampled sequence from b). d) an event trace from a discrete-event system recording when a) goes through values 1,2,3. e) an event trace that is equivalent in 'logic-time' to d), but with entirely different spacing in 'continuous-time'.

continuous-time trajectories of analog signals into digital sequences of sampled values, see figure 2.1b. This abstraction allows one to work with the discrete-time notion of time, where every time instance k is followed by an instance $k + 1$ which is always some Δt later. This periodic view of time is closely related to the operation of CPUs, which can execute blocks of control code at rather constant, but still finite rates. To form a closed loop system, computers also utilize digital-to-analog converters to produce analog output (e.g. a zero-order hold D/A converter, figure 2.1c). These converters take a digital sequence and produce a continuous-time trajectory under some assumptions (e.g. a zero-order hold assumes a stair-step pattern between samples). By comparing figure 2.1a and c it is obvious that some information abstracted by the sampling in b) is lost and cannot be reproduced, but an appropriate assumption from selecting a zero-order hold (stair-steps) can produce a 'similar' signal in c) that approximates a). It is important to note that different modeling formalisms retain different subsets of information, this allows them to retain only the information important to what that modeling formalism is intended to describe and analyze.

The logical-time model in discrete-event systems (DES) is similar to the discrete-time model, except that it is not assumed that the time instances are evenly spaced. Discrete-event systems allow one to compare the order of sequences of events, but not the spacing between events. For instance, figure 2.1d shows an abstraction of a) where the discrete events are generated when the system's value is 1, 2, or 3. The trace of events is $3, 3, 2, 1, 1, 2, 3, 3, 2, 1$, but the spacing is not recorded by the model (it is shown in figure 2.1 for illustration). A trace of events that is identical in logical-time is shown in e), meaning they are the same events in the same order. However, the two behaviors appear very different when the time of occurrence is considered. In a logical-time model, that spacing information and the actual continuous-time time of occurrence is abstracted away.

The modeling formalisms covered in the remainder of this Chapter can be used to describe, model, and analyze systems of systems; specifically, they could be used to discuss networks of UAVs. The formalisms allow simple component models to be combined to form larger systems in the same formalism (e.g. individual UAVs modeled as finite state machines can be combined to produce a finite state machine model for a network of UAVs). An important part of each formalism are the limitations on how individual components can interact with each other.

Process algebras are discrete-event system formalisms whose behaviors are described by traces of instantaneous actions in a logical-time model (they are described by sequences of ordered actions). Any individual process algebra model describes which actions can occur and in what order. Similarly, finite state machines also describe instantaneous events that cause the machine to transition from one state to the next, also constraining event orderings. Hybrid systems, specifically hybrid automata, combine finite state machines with continuous-time dynamics. This, unlike pure finite state machines or process algebras, creates a model that can describe both continuous-time and discrete-event behaviors. Finally, Petri nets are another alternative to describing interacting systems under a discrete-event notion of time. Petri nets have a fundamentally different representation of concurrency than process algebras and finite state machines. These different modeling formalisms focus on different aspects of the system's behavior. They differ on representations of individual components, the ways of combining individual components into larger systems, and their properties that can be analyzed. These variations affect which formalism is most appropriate for any specific application.

2.1 Process Algebras

Process algebras are intended to describe the sequence of instantaneous actions a process can take. The process' syntactic definition, along with its semantics [185], determine exactly which traces (sequences) of actions are possible. When a process is allowed to have output and input actions, several communicating processes can then be composed into a larger process. In this way, small systems represented in the process algebra can be combined into a larger communicating system of systems in the same process algebra, figure 2.2.

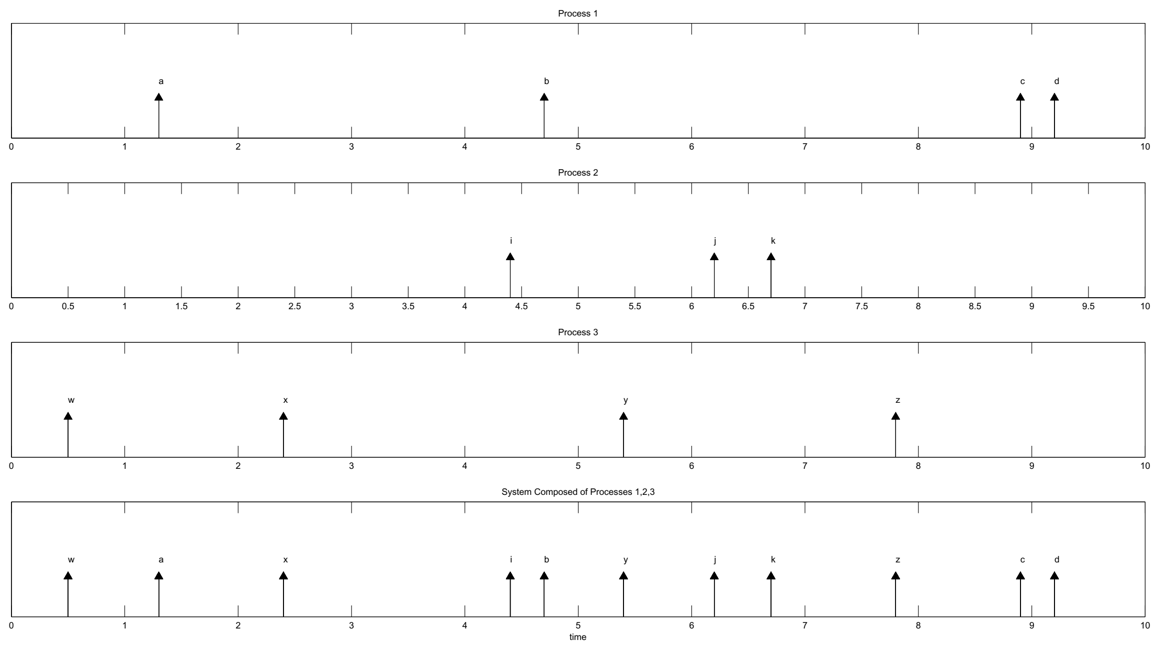


Figure 2.2: Process 1 performing a, b, c, d. Process 2 performing i, j, k. Process 3 performing w, x, y, z. A larger system composed of all three processes performing $w, a, x, i, b, y, j, k, z, c, d$.

The differing details about output, input, and composition allow for multiple views and representations within the field of process algebra. All variants are closely related by common notions while retaining unique aspects. Each process algebra also has accompanying analysis tools and techniques that can be utilized for verification purposes.

These formalisms are theoretical in nature and intended as design tools. They lack many of the features that would make them more practical for common programming use (variables, functions, arithmetic). These convenient features would only complicate and muddy the theory, which is intended to describe the interactions of processes rather than the internal machine's detailed behavior. For example, often a process will have to internally choose between several alternative actions. These formalisms do not specify how this is done, but in a practical implementation this would be based on some programmed criterion.

In real-life applications, a process already designed and coded in some language (e.g. C++ or Java) could have its irrelevant activities abstracted away to create an approximate model of its behavior in a particular process algebra. This could then be analyzed with other processes to validate a design. Equally appropriate, the design could be performed in the process algebra domain. This would create a specification model that the 'real' code would have to remain faithful to in its implementation.

The next four sub-sections describe the most prominent process algebra formalisms. They are presented in chronological order; as research progressed the core notions became clearer, more mathematically precise, and more expressive.

2.1.1 Communicating Sequential Processes (CSP)

In 1978 Hoare presented a paper that "suggests that input and output are basic primitives of programming and that parallel composition of communicating sequential processes is a fundamental program structuring method". Up to this point, input and output were informally added onto existing programming languages with no theoretical justification or explanation. He created what he called a "programming language fragment" which "should not be regarded as suitable for use as a programming language" [89]. This fragment was named Communicating Sequential Processes (CSP) and along with Milner's Calculus of Communicating Systems (CCS), presented next, began the field of process algebra.

Hoare was specifically targeting CSP as a theoretical basis for programming multiprocessor computers. He foresaw that to fully utilize these multiple core machines, individual processors must be able to communicate and synchronize with each other in order to work on a single task in unison. His proposed language provided a way of describing these interactions for a "fixed network of processors connected by input/output channels" [89]. One of CSP's main goals was to prove whether a fixed network could deadlock, resulting in all processors waiting on each other and making no progress forever. While CSP is theoretical in nature, it was highly influential and provides the core functionalities for the parallel programming language Occam [11].

The original syntax of CSP is shown in Table 2.1 using a Backus-Naur Form (BNF). The language uses a *command_list* to represent the system of processes being modeled.

The *simple_commands* include the expected: input, output, as well as assignment. One critical contribution of CSP is that processes block on communication. This means that an *output_command* can only happen if an *input_command* is also ready, and an *input_command* can only happen if an *output_command* is also ready. Whichever process, the sender or the receiver, is ready first must wait until the other process is ready for communication. This causes the processes to *synchronize* on communication. Instead of having the output and input of information being two separate events, communication is a single event undertaken by two processes at the same moment. Additionally, the sender process is listening to exactly one explicitly named receiver process, and vice versa. Only two processes are ever involved in any

Table 2.1: Syntax of Communicating Sequential Processes (CSP) [89]

		Commands
command	::=	*simple_command* $\mid$ *structured_command*
simple_command	::=	*null_command* $\mid$ *assignment_command* $\mid$ *input_command* $\mid$ *output_command*
structured_command	::=	*alternative_command* $\mid$ *repetitive_command* $\mid$ *parallel_command*
null_command	::=	skip
command_list	::=	$\{$*declaration*; $\mid$ *command*;$\}^*$ *command*
		Parallel Commands
parallel_command	::=	[*process* $\{\|$ *process*$\}^*$]
process	::=	*process_label command_list*
process_label	::=	*empty* $\mid$ *identifier* :: $\mid$ *identifier* (*label_subscript*$\{$, *label_subscript*$\}^*$ ::
label_subscript	::=	*integer_constant* $\mid$ *range*
integer_constant	::=	*numeral* $\mid$ *bound_variable*
bound_variable	::=	*identifier*
range	::=	*bound_variable*:*lower_bound*..*upper_bound*
lower_bound	::=	*integer_constant*
upper_bound	::=	*integer_constant*
		Assignment Commands
assignment_command	::=	*target_variable* := *expression*
expression	::=	*simple_expression* $\mid$ *structured_expression*
structured_expression	::=	*constructor*(*expression_list*)
constructor	::=	*identifier* $\mid$ *empty*
expression_list	::=	*empty* $\mid$ *expression*$\{$, *expression*$\}^*$
target_variable	::=	*simple_variable* $\mid$ *structured_target*
structured_target	::=	*constructor*(*target_variable_list*)
target_variable_list	::=	*empty* $\mid$ *target_variable*$\{$, *target_variable*$\}^*$
		Input and Output Commands
input_command	::=	*source*?*target_variable*
output_command	::=	*destination*!*expression*
source	::=	*process_name*
destination	::=	*process_name*
process_name	::=	*identifier* $\mid$ *identifier*(*subscripts*)
subscripts	::=	*integer_expression*$\{$, *integer_expression*$\}^*$
		Alternative and Repetitive Commands
repetitive_command	::=	$\star$*alternative_command*
alternative_command	::=	[*guarded_command* $\{ \Box$ *guarded_command*$\}^*$]
guarded_command	::=	*guard* $\rightarrow$ *command_list* $\mid$ (*range*$\{$,*range*$\}^*$)*guard* $\rightarrow$ *command_list*
guard	::=	*guard_list* $\mid$ *guard_list*;*input_command* $\mid$ *input_command*
guard_list	::=	*guard_element*$\{$;*guard_element*$\}^*$
guard_element	::=	*boolean_expression* $\mid$ *declaration*

single communication event, unlike other process algebras that can have multi-way communication events.

The *parallel_command* allows the language to take two individual processes and combine them into a larger composite process. That larger process may or may not involve communication between the sub-processes. It is possible that the two processes do not interact with each other. The *alternative_command* allows for a non-deterministic choice to be made. This command allows the process to choose from several potential processes to continue on as. The *repetitive_command* simply repeats execution of the included *alternative_command* until it cannot be continued any further.

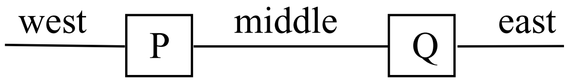


Figure 2.3: A CSP buffer holding 2 entries: the first in P, the second in Q.

Figure 2.3 shows a CSP-based buffer that holds two entries (adapted from [89]). The process definitions are:

$$P :: *[west?c \rightarrow middle!c] \tag{2.1}$$

and

$$Q :: *[middle?d \rightarrow east!d] \tag{2.2}$$

Process P repeatedly accepts input from channel *west* and stores the value input to local variable c, which it then outputs on channel *middle*. Process Q repeatedly receives input on channel *middle*, stores that value as it's local variable d, then outputs on channel *east*. Output on channel *middle* by P must occur exactly when input on *middle* occurs by Q. Figure 2.3 shows the channels connecting the overall system, $P \parallel Q$, but does not give any details about the defined behaviors of P or Q.

[89, 90] contain a more thorough description of each syntactic construct. [43] introduces a denotational style of *semantics* (semantics represent the 'meaning' of the process' syntactic definition). The semantic meaning is described using the *failures* of the system. These failures characterize any model in CSP. A failure is a tuple (s, X): s is a trace of actions that is possible in the model, X is the set of actions that can be refused after the trace s has been executed. This type of semantics (a set of failures) records more information than the set of traces accepted; it also allows one to consider if a given trace can be continued. If a behavior s progresses to the point where X contains all possible events, then no event can occur and deadlock ensues. This type of reasoning is used extensively in the Failures-Divergence Refinement tool, FDR2 [5].

FDR2 allows one to check for deadlock, livelock, and to compare a refinement of processes. A process Q refines a process P if all of the traces and refusals of Q are also traces and refusals of P (written $P \sqsubseteq Q$, if traces(Q) $\subseteq$ traces(P) and refusals(Q) $\subseteq$ refusals(P), called reverse inclusion) [68]. In this manner, a simpler specification process can be created and the actual complex process can checked to be a refinement of the specification. This means that only behaviors in the specification are in the actual process, with several specification behaviors having possibly been removed.

CSP's notation was modified based on concepts from CCS for Hoare's book [90]. The concepts are very similar with a few additions, notably the distinction between internal choice where the process must non-deterministically determine for itself which path to take and external choice where the processes' environment chooses which path to take. Additionally the *repetitive_command* was based on and supplemented with a more generic recursion construct.

CSP is excellent for designing algorithms for a specific number of components in a specific communication configuration. However, the formalism includes neither replication (creating new processes on-the-fly) nor a re-configuration of the communication connections between processes. This makes its applicability questionable in some situations (such as open network with a non-fixed number of UAVs and tasks). Where it is practical, it can provide a capable design tool assuming the properties of interest can be expressed through deadlock, livelock, or a refinement.

In 2007 [20] briefly mentioned the possibility of using CSP to model and certify autonomous systems such as individual military UAVs. This was merely an observation of one option among a list of many, but CSP could be relevant for designing multiprocessor code for UAVs. This could be for vision processing or local optimization algorithms (e.g. path planning) on UAVs with multiple CPUs. At the very least, it would allow developers to verify that the processors will not deadlock due to a flaw in the communication scheme between CPUs. In [150] CSP was used to design and verify a communication scheme between a single UAV and its ground station. There were multiple alternative communication channels (3G, WiFi, Satellite). As the higher bandwidth but lower range channel degraded and failed, the communication scheme allowed the UAV to switch over to a longer range but lower bandwidth channel in a verified manner (switch from WiFi to 3G to Satellite). This was possible because the communication channels and their configuration were known and fixed by design.

CSP could potentially be used for modeling and specifying network-level controllers, but it would have several significant drawbacks. First, the network would have to be of a fixed dimension and configuration, no UAVs or tasks would be entering and exiting the system. Second, CSP's notation is not 'human friendly' and has no equivalent graphical representation. The density and complexity of CSP statements would make it very difficult for even experts to use it in an interactive on-line manner. Third, all properties to be verified would either have to be deadlock, livelock, or refinements of other CSP models. While CSP could potentially be quite useful in UAV applications, it does not seem appropriate for the network-level control application.

2.1.2 Calculus of Communicating Systems (CCS)

At the same time Hoare was developing CSP, Robin Milner was developing the Calculus of Communicating Systems (CCS) [131]. Both were investigating the behavior of a system of communicating processes, but from different perspectives. Milner sought to make his work as mathematically rigorous and concise as possible, while Hoare focused on providing a language fragment that could be practically useful for common applications. Milner's opinion was that "In a definitive calculus there should be as few operators or combinators as possible, each of which embodies some distinct and intuitive idea, and which together give completely general expressive power"[132]. Milner was hoping to produce 'the' concurrency theory from which everything else stemmed. Unlike CSP whose semantics were originally described informally, Milner also provided operational semantics to give a rigorous mathematical definition to his syntax.

CCS's syntax is deceptively simple:

$$E ::= X \ | \ a.E \ | \ \sum \bar{E} \ | \ E|E \ | \ E \setminus N \ | \ fix_i \bar{X}\bar{E} \ | \ E[\phi] \tag{2.3}$$

Obviously CCS's syntax in equation 2.3 is significantly more compact than CSP's in table 2.1. Any CCS system is defined as an expression, E. Each E can contain named variables, X, which are statically bound to expressions. This means that variables X are merely short-hand for longer CCS expressions and not program variables that are actively over-written during execution.

Table 2.2: Semantics of Calculus of Communicating Systems (CCS)

$$\text{Action}\ \frac{}{a.E \xrightarrow{a} E} \qquad \text{Sum}\ \frac{E_i \xrightarrow{a} E'}{\sum \bar{E} \xrightarrow{a} E'}$$

$$\text{Composition-1}\ \frac{E \xrightarrow{a} E'}{E|F \xrightarrow{a} E'|F} \qquad \text{Composition-2}\ \frac{F \xrightarrow{a} F'}{E|F \xrightarrow{a} E|F'}$$

$$\text{Composition-3}\ \frac{E \xrightarrow{a} E' \quad F \xrightarrow{\bar{a}} F'}{E|F \xrightarrow{1} E'|F'}$$

$$\text{Restriction}\ \frac{E \xrightarrow{a} E' \ a \notin N}{E \setminus N \xrightarrow{a} E' \setminus N} \qquad \text{Recursion}\ \frac{E_i\{fix\bar{X}\bar{E}/\bar{X}\} \xrightarrow{a} E'}{fix_i\bar{X}\bar{E} \xrightarrow{a} E'}$$

The operational semantics of CCS are shown in table 2.2. The action operator, $a.E$, allows an action a to be prefixed to any expression E. For example $a.b.c.0$ is the process that can execute action a then action b then action c then finally become the special 'inaction' process 0. The summation operation allows a process to non-deterministically choose between potential futures. Typically the summation is written as: $a.E + b.F$, meaning the process either behaves like $a.E$ or $b.F$. The composition operation allows two CCS processes to be composed into a larger system with the two acting in parallel ($E \mid F$ is the parallel composition of E and F). It is possible for either E or F to take individual actions which affect only themselves (Composition-1 & Composition-2). It is also possible that the two engage in complementary actions signifying synchronous communication between the two, where a and $\bar{a}$ are complementary (Composition-3). The result of the complementary actions a and $\bar{a}$ is an unobservable single internal action, called 1, which is hidden from its environment. Finally, the restriction operation $E \setminus N$ allows a set of events N to be prevented from occurring by process E. This allows one to force output communication only to occur when input communication also occurs. For example $a.0 \mid \bar{a}.0 \ \setminus \{a, \bar{a}\}$ will only allow action 1 to occur. Since input a and output $\bar{a}$ cannot occur individually, the only way for the two processes to progress is for them to communicate together executing the allowed action 1 and then becoming $0 \mid 0$. Recursion, $fix_i \bar{X}\bar{E}$, is the standard mutual recursion and $E[\phi]$ is simply a renaming of the variables in E using a function ϕ.

Consider again the two-place CSP buffer from figure 2.3. This could be equivalently coded in CCS. Let the set of characters to possibly be transmitted be $\{0, 1\}$.

The CCS definition of P is:

$$P \triangleq fixX.(west_0.\overline{middle_0}.X + west_1.\overline{middle_1}.X) \tag{2.4}$$

So when P experiences an input event $west_0$ it will then experience an output event $\overline{middle_0}$, thereby passing from the input on channel $west$ to the output on channel $middle$ the value 0. Similarly if P experiences an input event $west_1$ it will then experience an output event $\overline{middle_1}$ before continuing on.

The CCS definition of Q is:

$$Q \triangleq fixX.(middle_0.\overline{east_0}.X + middle_1.\overline{east_1}.X) \tag{2.5}$$

If Q experiences an input event $middle_0$ it will then experience an output event $\overline{east_0}$, and similarly for the other messages.

CCS's method of storing which message was input is allowing multiple possible initial events and then choosing a specific path when that specific input occurs. After the matching output follows the input, the process recurses and continues again offering the choice between 0 and 1.

The two-place CCS buffer is:

$$(P \mid Q) \setminus \{middle_0, middle_1, \overline{middle_0}, \overline{middle_1}\} \tag{2.6}$$

$$\begin{aligned}(fixX.(west_0.\overline{middle_0}.X + west_1.\overline{middle_1}.X) \mid fixX.(middle_0.\overline{east_0}.X + middle_1.\overline{east_1}.X))\\ \setminus\{middle_0, middle_1, \overline{middle_0}, \overline{middle_1}\}\end{aligned} \tag{2.7}$$

Equation 2.7 substitutes in the values of variables P and Q to show the full definition. It behaves similarly to the CSP version. Due to the restrictions on the *middle* events, the output of the first buffer can only occur when the input of the second is ready and vice versa.

One drawback of CCS can already be seen. Comparing the buffer specification in equation 2.7 with CSP's, CCS's is made of more 'primitive' operations causing it to end up requiring a larger and more complicated looking specification. Experts and non-experts will struggle with the meanings of more complex combinations of recursion, choice, and restriction.

Much like CSP had the toolkit of FDR2, CCS has the Concurrency Workbench of Edinburgh [4, 171]. This tool allows the user to walk through a simulation a CCS system, generate random traces of the system, or check for different congruences between systems. If the overall CCS system produced has a finite dimension, the Concurrency Workbench also allows modal mu-calculus statements to be verified against the model [42, 172]. The modal mu-calculus contains other more familiar temporal calculi like LTL and CTL.

While Milner's CCS is more fundamental and primitive than CSP, for network-level control it shares the same detriments. First, it too specifies a pre-determined network configuration. Like CSP it also has no equivalent graphical representation. While CSP was certainly not 'human friendly' for on-line interaction, CCS's syntax is even more difficult to comprehend and lengthier due to its fundamental nature.

However, CCS was not intended for these purposes (network-level control). Milner was searching for a Turing machine-like representation of concurrency. Just as coding simple operations in Turing machines seems very inefficient and impractical when compared with coding in C++, coding in CCS will also require much more complexity and require much larger specifications than coding in concurrency representations that are not based on such fundamental primitives. CCS's contributions are theoretical justifications instead of practical code for implementations.

2.1.3 Algebra of Communicating Processes (ACP)

After Milner and Hoare produced apparently competing notions for networks of processes, Bergstra and Klop observed the many similarities between CSP and CCS. In [37] they produced a "theory of concurrency, along the lines of an algebraic approach." They recognized that both the programming language fragment CSP and the calculus CCS could be posed as algebras, thus leading to the field of process algebra (which has also been called process calculi due to CCS's use of the term calculus).

They would pose their rules as an axiom system defined over the domain of processes. The axioms would explain the different allowed ways of combining smaller processes into larger processes. They also made sure to point out that with different axiom systems come different notions of processes interacting. CSP and CCS were merely different ways of describing similar axiom systems. In [37, 35] they discuss several closely related axiom systems: BPA, the Basic Process Algebra axiom system for describing only single processes; PA, the Process Algebra axiom system for describing concurrent merging processes; ACP, the Algebra of Communicating Processes axiom system for describing concurrent and communicating processes; AMP, the Algebra for Mutual exclusions of tight regions in Processes; ACMP, a combination of ACP and AMP that has both tight regions and communication between concurrent processes; ASP, the Algebra of Synchronous Processes which forces all individual processes to operate fully synchronously.

Table 2.3: Axiom System for ACP [37]

$x + y = y + x$	A1
$x + (y + z) = (x + y) + z$	A2
$x + x = x$	A3
$(x + y) * z = x * z + y * z$	A4
$(x * y) * z = x * (y * z)$	A5
$x + \delta = x$	A6
$\delta * x = \delta$	A7
$a \mid b = b \mid a$	C1
$(a \mid b) \mid c = a \mid (b \mid c)$	C2
$\delta \mid a = \delta$	C3
$x \parallel y = x \lfloor\!\lfloor y \ + \ y \lfloor\!\lfloor x \ + \ x \mid y$	CM1
$a \lfloor\!\lfloor x = a * x$	CM2
$(a * x) \lfloor\!\lfloor y = a * (x \parallel y)$	CM3
$(x + y) \lfloor\!\lfloor z = x \lfloor\!\lfloor z \ + \ y \lfloor\!\lfloor z$	CM4
$(a * x) \mid b = (a \mid b) * x$	CM5
$a \mid (b * x) = (a \mid b) * x$	CM6
$(a * x) \mid (b * y) = (a \mid b) * (x \parallel y)$	CM7
$(x + y) \mid z = x \mid z \ + \ y \mid z$	CM8
$x \mid (y + z) = x \mid y \ + \ x \mid z$	CM9
$\partial_H(a) = a$ if $a \notin H$	D1
$\partial_H(a) = \delta$ if $a \in H$	D2
$\partial_H(x + y) = \partial_H(x) + \partial_H(y)$	D3
$\partial_H(x * y) = \partial_H(x) * \partial_H(y)$	D4

Table 2.3 shows the axiom system for ACP. Notationally a, b, c stand for arbitrary events with δ being the failure/deadlock event and x, y, z stand for arbitrary ACP processes. ACP allows the following operations on processes: alternative composition (or choice) with $+$; sequential composition with $*$; composite communication events with $\mid$ that combine individual events into a larger unique composite event, $a \mid (b \mid c)$ is a unique event representing a multi-way communication event made from and distinct from events a, b, c; parallel composition with $\|$ and $\llfloor$; and encapsulation (restriction) with ∂_H.

The axiom system provides an algebra of processes which can be manipulated and simplified using the various axioms. It makes statements like:

- A1: "the choice between doing x or y is the same as between doing y or x"
- A7: "failure followed by any process is still failure"
- C1: "the composite communication event $a \mid b$ is the same as the composite communication event $b \mid a$"
- CM1: "x in parallel with y can either have x execute first, or y execute first, or x and y can execute a communication event together"

While Bergstra and Klop suggested that different process algebras have different axiom systems, their BPA was the subset of ACP containing A1-A5 and their PA was the subset of ACP containing A1-A5 and CM1-CM4 [37]. Similarly ACMP is an extension of ACP containing the missing axioms that appear in AMP. This highlighted the fact that the addition/removal of specific rules could still produce a consistent process algebra, just one that described different types of systems.

The purpose of ACP and its variants was to help unify and focus the field of process algebra. In addition it provided a process algebra with multi-way communication and showed with ASP how synchronously executing processes could be realized with asynchronously executing processes; the opposite was shown by Milner with SCCS illustrating that asynchronous processes could be realized with synchronous processes [132].

Like CSP and CCS, ACP has a toolkit designed to support its use. Because ACP lacks recursion, it has a finite state space. This allows the toolkit mCRL2 [8, 173] to calculate the finite labeled transition system and perform model checking algorithms, provide simulation support, or view the labeled transition system itself.

Also like CSP and CCS, ACP shares the same properties that make it inappropriate for the type of interactive network-level control sought in this dissertation (ACP has a fixed communication configuration, no graphical representation, and is difficult to comprehend interactively).

2.1.4 π-Calculus

As networking and wireless communications became more prevalent, the limitations of assuming a fixed communication configuration became more obvious. Hoare and Milner identified that this could be a potential problem limiting the usefulness of both CSP and CCS. However, they preferred to focus their initial efforts on the simpler, but still frequently occurring, fixed configuration networks [131, 89]. After more than a decade of CCS research, in 1989 Milner moved on to address "systems in which one can naturally express processes which have changing structure" [135]. This work, the π-calculus, was used for applications such as modeling mobile cellular phones moving throughout a telecommunications network.

The π-calculus models the changing configuration of communicating processes that can pass a communication link from one process to another. To do this "communication links are identified by names, and computation is represented purely as the communication of names across links" [135]. For example, consider three people with email addresses. Persons 1 and 2 have each other's email addresses. Persons 2 and 3 have each other's email addresses. Person 2 can send Person 3 the email address for Person 1. This can create a new communication link that could eventually be used (Person 3 can now email Person 1). This 'link mobility' is not present in either CCS or CSP [135]. It is important to note that the mobility the π-calculus is referring to is the movement of abstract communication links and not the physical motion of the processes as they move through a physical space [133].

$$\pi ::= x(y) \ \mid \ \bar{x}\langle y\rangle \ \mid \ \tau \tag{2.8}$$

$$P ::= \sum_{i\in I} \pi_i.P_i \ \mid \ P_1|P_2 \ \mid \ (new\ a)P \ \mid \ !P \tag{2.9}$$

Milner's π-calculus syntax is quite similar to his CCS syntax [133]. Equation 2.8 shows the three different types of atomic actions. The first, $x(y)$, represents receiving an incoming communication on channel x and storing what is transmitted to the bound variable y. This means that y will be replaced by the content of the communication. The second, $\bar{x}\langle y\rangle$, represents sending an outgoing communication on channel x where the content transmitted is y. The third, τ, is the *unobservable action* that can occur. τ can simply be something not modeled, or can represent a process that is actually several processes making a private communication.

Equation 2.9 shows the four ways of constructing a composite process from simpler processes. The first option, $\sum_{i\in I} \pi_i.P_i$, is the non-deterministic choice between several alternative processes. An example of how it is most often written is: $a(b).P_1+\bar{m}\langle n\rangle.P_2$, where the overall process can either receive input for b on a and continue as P_1 or output n on m and continue as P_2. The second option, $P_1|P_2$, is the parallel composition of two processes. The third option, $(new\ a)P$, is the creation of a new variable a that is not bound by any other processes (it is like a local variable declaration). Finally, $!P$, is the replication operator that allows multiple copies of P to be created.

Table 2.4: Semantics of the π-calculus [133]

$$\text{Tau}\frac{}{\tau.P + M \to P}$$

$$\text{React}\frac{}{(x(y).P + M)|(\bar{x}\langle z\rangle.Q + N) \to \{z/y\}P|Q}$$

$$\text{Par}\frac{P \to P'}{P|Q \to P'|Q}$$

$$\text{Res}\frac{P \to P'}{(new\ x)P \to (new\ x)P'}$$

$$\text{Struct}\frac{P \to P'}{Q \to Q'}\text{if } P \equiv Q \text{ and } P' \equiv Q'$$

Table 2.4 shows the semantics of the π-calculus. The Tau rule shows that if a silent action occurs the process continues as that silent action's continuation, P, while possibly throwing away alternative paths, M. The *React* rule shows two processes communicating on x. The outputting process simply continues on as Q while the inputting process also continues on, but does so replacing the bound variable y with the message content z to become $\{z/y\}P$. The *Par* rule shows that if P can independently evolve to P', it can do so without affecting Q. This happens if P is actually several processes composed together that interact privately. *Res* shows the restriction that is created by using a new variable x. Finally, *Struct* is the rule that explains $!P$, but first one has to note that $!P \equiv P|!P$. This means that $!P$ is as many copies of P in parallel as you need.

Milner's π-calculus is actually an algebra with an axiom system, similar to ACP's in Table 2.3. For detail see [134, 146].

While the buffer example for CSP and CCS could be reproduced, and is in [133], it does not illustrate the mobility of links that is the purpose for the π-calculus. Figure 2.4 shows the first π-calculus example from Chapter 9 of Milner's book [133]. There are three processes: P, Q, R. Processes P and R share a private link z that no one else knows. Processes P and Q share a link x that other processes may also use. Process $P = \bar{x}\langle z\rangle.P'$, meaning that it outputs z on x. Process $Q = x(y).Q'$ meaning that it accepts an input y on channel x. After the communication on x, Q becomes $\{z/y\}Q'$ which contains and can use the restricted link z. This could be given a more concrete interpretation as Person P emailing Person Q the email address of Person

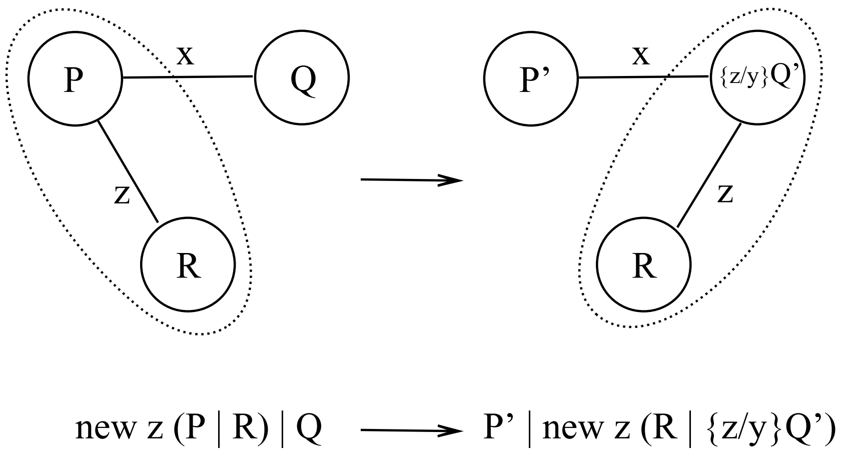


Figure 2.4: A simple example of the mobility of π-calculus links.

R.

The π-calculus's toolkit primarily consists of the Spatial Logic Model Checker (SLMC) [13, 179]. This tool allows one to specify a π-calculus process and check it against a SLMC logic statement [45, 46]. The process can be proved to satisfy the statement. Other logic languages such as LTL, CTL, and the modal mu-calculus specify behavioral patterns that processes are verified against. SLMC includes these types of predicates allowing a user to verify that certain interaction patterns are met (e.g. output then input repeatedly). SLMC extends this to include 'spatial' patterns [44]. This allows one to verify that a process is composed of at least 3 processes, or that only one process at a time can ever output on a specific name (channel). These types of properties cannot be observed purely from the system's behavior (traces of actions). This allows a user to form statements evaluating how the structure of the processes evolves as well as how the behavior evolves.

SLMC's development is a recent addition to the π-calculus literature and has allowed its developers to model and verify security protocols [175]. Private keys are modeled as restricted channels that can potentially be passed around to grant access. Secure systems are evaluated to make sure they always properly handle these secret keys. Even prior to SLMC and the ability to verify π-calculus processes, the potential use of the π-calculus for modeling and controlling systems of changing structure was pointed out. In [71] it is proposed that this could be useful for networks of AUVs or UAVs, but the thought is not elaborated on or further pursued. NASA recognized it as a potential option for future modeling of the communications of swarms of pico-class spacecraft [85]. It does not appear that the π-calculus has yet been successfully utilized to model, develop, and control networks of UAVs/AUVs/UGVs with changing communication configurations. However, it was an instrumental influence on the development of Dynamic Networks of Hybrid Automata (DNHA) which is the the modeling formalism developed for the specification and simulation language SHIFT. SHIFT combines concepts from hybrid systems with concepts from the π-calculus to create networks of hybrid systems where individual processes (hyrbrid automata) can enter and exit the system as well as be re-configured [2]. SHIFT was utilized to model and simulate automated highways where cars enter, exit, and change position. Interestingly, SHIFT occurred well before SLMC and so does not leverage any of its verification procedures, simply using the π-calculus concepts of links. SHIFT and DNHA will be discussed in more detail in Section 2.3.2.

While the π-calculus does allow specification and analysis of systems that change communication structure, it still retains several properties that make it unsuitable for specifying network-level controllers for our purpose. There is still no graphical equivalent of a process's definition. The 'flow graphs' like figure 2.4 simply show the current connections of the system, they do nothing to explain the processes' potential behaviors. To understand how the system may behave, a user would still need to observe and analyze the textual definition of the component processes. The π-calculus, like CCS, is also intended to be 'primitive'. This causes long definitions to be needed for simple tasks. Even if syntactic-sugar (abbreviations) were added, the specification of network-level controllers using multiple nested levels of least fixed-points, parallel compositions, choices, and new restricted variables would make understanding and specifying the controllers on-line difficult for the most experienced experts.

The π-calculus, like the other process algebras, is a terrific theoretical tool to be used off-line by experienced experts who have the time required to iterate on theoretical designs, specifications, and analyses. However, this dissertation's goal is to provide a set of tools to be used by non-experts in an on-line manner under time-critical circumstances. These considerations make process algebras, in general, inappropriate for our purposes.

2.2 Finite State Machines

Finite state machines are also discrete-event system formalisms [47]. They are an older, more mature, and more frequently used modeling formalism than process algebras. They have been used to model individual sequential programs and are closely related to regular languages, both of which appear frequently in computer science [167]. They can describe a software component that has a discrete number of states and changes from one state to another during the occurrence of an event. This closely mirrors the execution of a digital computer program whose state is fixed until the next discrete action is executed (e.g. the evaluation of a command), after which the program is in a new state [174, 31].

A traditional finite state machine (FSM) describes the discrete internal behavior of an individual component, whereas the process algebras focus purely on the interactions between networked components. As networking and embedded applications became more prominent, FSMs were extended to include input and output to allow a large system to be described as several interacting component systems. In this way small FSM systems can be composed together to produce a larger system of systems, which itself is represented as a FSM [60]. Like the many process algebras, there are multiple approaches to FSM input/output that allow different notions of communication [113].

Finite state machines, like process algebras, have theoretical properties that can be analyzed and verified to prove the correctness of a design. However, unlike process algebras, they also have an equivalent graphical description that eases human comprehension.

2.2.1 Individual Finite State Machines

The finite state machine modeling formalism provides ways to: describe an individual FSM as a model, determine a FSM's potential behaviors from the model, and analyze the properties of a FSM model.

Syntax

Any FSM model can be represented either syntactically or graphically. The two representations are equivalent [47]. In the future the term 'model' may be dropped, but it is understood that the actual physical machine is the FSM and the syntactic or graphic description is the FSM model.

Definition 1 *An individual finite state machine can be modeled as a tuple* $M = (X, E, f, x_0, X_m)$ *where:*

- X*: is the finite set of states that* M *can be in,*
- E*: is the finite set of events that can occur in* M *(called* M*'s alphabet),*
- f*: is the transition function (*$f : X \times E \rightharpoonup X$*),*
- x_0*: is the initial state,*
- X_m*: is the set of marked states.*

Alternatively, the model's definition can be represented graphically. The graphical representation is called a *state transition diagram.* Figure 2.5 is the state transition diagram for example 1 given below. The finite set of states, X, can be represented with a finite set of circular nodes. There is one unique labeled circular node corresponding to each unique state. The finite set of events, E, are used to label the transitions of the transition function, f. The transition function is represented as directed edges drawn from one node to another node. For every pair in $X \times E$ where f is defined, there is an edge in the state transition diagram labeled with the event from the domain. The edge goes from the state in the domain to the state in the co-domain. The initial state, x_0, is represented as an edge to x_0 that has no origin. The marked states, X_m, have a second circle around the node to distinguish them from un-marked states.

Example 1 *Let FSM* $M_1 = (X, E, f, x_0, X_m)$ *where:*

- $X = \{one, two, three\}$
- $E = \{a, b, c\}$
- f *is defined as:*

	a	*b*	*c*
one	*two*		
two	*two*	*three*	
three	*two*		*one*

- $x_0 = one$
- $X_m = \{three\}$

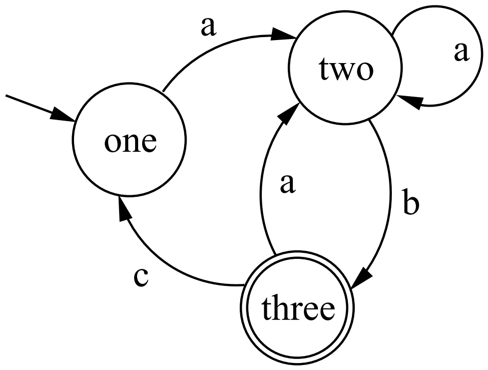


Figure 2.5: A state transition diagram equivalent to example 1 for FSM M_1

Example 1 and figure 2.5 demonstrate the equivalence of the two representations. Both forms show that the transition function, f, is only partially defined representing that certain events are not possible in certain states (e.g. M_1 cannot experience event c in state *two*). This occurs often in physical machines, where certain events are not possible in certain states (e.g. a vending machine will not give change, an event, if in the current state you have not already made a deposit).

Semantics

The above syntactic and graphic representations are enough to specify a FSM model, but they do not describe how a FSM model 'runs'. One can intuitively guess how a FSM behaves, but it is better to give a rigorous semantic definition to pin down exactly what the syntax means. The process algebras already discussed often use operational semantics to describe how a process executes in a step-by-step operational manner. FSM behavior is typically given in terms of a language [167]. In this sense the language denotes all the possible behaviors of the FSM (the language is a form of denotational semantics).

Definition 2 *A trace/string/word is a sequence of 0 or more events from a finite alphabet of events.*

Informally, a string *abcd* represents a behavior where event *a* occurred, followed by event *b*, followed by event *c*, followed by event *d*. The *empty string* is of length 0 and is often written as ϵ. It represents a behavior where no events have yet occurred. If a string represents a behavior of a FSM, that string's events must be drawn from the alphabet of the FSM, the set E in definition 1. A word can contain a finite or infinite number of events. Infinite words are often referred to as ω-words.

Example 2 *The following are all words over the alphabet* $\{a, b, c\}$*:*

- ϵ*: the empty string*
- *aba: a then b then a*
- *ccccc...: an infinite number of c's.*

Definition 3 *A language is a set of words over a common alphabet.*

A language can be finite if it contains a finite set of words. A language can also be infinite if it contains an infinite set of words. Like there is an empty string of length 0, there is also an *empty language* that contains 0 strings. It is referred to as the *null language* and represented as $\varnothing$. Individual strings are used to describe any one potential behavior of a FSM, but languages are used to describe the set of all potential behaviors of a FSM.

Example 3 *The following are all languages over the alphabet* $\{a, b, c\}$*:*

- $\{\}$*: the empty language, written shorthand as* $\varnothing$*.*
- $\{\epsilon, aba, cccc...\}$*: the language with the three strings from example 2.*

- $\{\epsilon, a, aa, aaa, aaaa, aaaaa, ...\}$*: the infinite language with every string of length* $n \in \mathcal{N}$ *made of only a's.*

The set of all possible strings that the FSM can generate from its initial state is called the *generated language* of the FSM. In order to determine this language the transition function f must be extended from events to strings in an obvious manner.

Definition 4 *The extended transition function ($f^* : X \times E^* \rightharpoonup X$) is based on the original transition function ($f : X \times E \rightharpoonup X$). It takes in a state and a string (an element of E^*) and may produce a destination state that is the result of executing the string from the given state according to:*

- $f^*(x, \epsilon) := x$
- $f^*(x, se) := f(f^*(x, s), e) \;\; for \;\; s \in E^* \;\; and \;\; e \in E$

The extended transition function states that executing no event, ϵ, keeps the FSM in the same state. It also states that the execution of a string happens incrementally, the FSM executes the first event to end up in a new state then executes the next event in the sequence according to f. Using definition 4, a formal definition of the language generated can be given.

Definition 5 *The language generated by a FSM M is:*

$$\mathcal{L}(M) := \{s \in E^* \;:\; f^*(x_0, s) \;\; is \;\; defined\}$$

The language generated by a FSM is the set of all strings of events that can be executed starting from the initial state. This is the simplest and most obvious language used to describe the behavior of a FSM, but a second language is often used as well. The *language marked* by a FSM is the subset of the language generated by the FSM that ends in a marked state. Marking a state is used to denote successful completion of some 'task'. In this way checking that a FSM can always end in a marked state means proving that the FSM can always complete its tasks.

Definition 6 *The language marked by a FSM M is:*

$$\mathcal{L}_m(M) := \{s \in \mathcal{L}(M) \;:\; f^*(x_0, s) \in X_m\}$$

The languages marked and generated by a FSM can be automatically computed from the syntactic/graphic representations of the FSM. It can also be proved if all states in the FSM can be reached from the initial state by some string. If this is not true the *accessibility operation* will eliminate all un-reachable states. Similarly it can be proved if all states in the FSM can be continued with some string to end in a marked state (if the machine can be run to successful completion). If this is not true the *co-accessibility operation* will eliminate all states that cannot reach a marked state. Algorithms to check these properties and perform these operations can be found in [47].

The languages generated and marked by FSMs are referred to as *regular languages*. They can be represented by enumerating the set of all potential strings, but for infinite languages this is not feasible. Regular expressions are a form of 'short-hand' that can be used to represent an infinite language. Additionally, the FSM model itself can be viewed as a representation of the regular language since it either generates or marks the language [167]. These representations are equivalent (graphical FSM models, syntactic FSM models, sets of strings representing a regular language, regular expressions representing a language).

Example 4 *The FSM modeled in example 1 and shown in figure 2.5 is equivalently represented as the regular expression* $M_1 := a(a^*b(a+ca))^*$.

Alternative Automata Formalisms

Finite state machines are a specific type of automaton. Automata, in general, do not have to have a finite set of states. Infinite state automata can represent any language, but do so by requiring infinite memory. By limiting automata to a finite number of states, FSMs become representable with a finite amount of memory (but are limited to representing regular languages) [47].

FSMs are also referred to as finite state automata. These automata come in two varieties: deterministic and non-deterministic. In deterministic finite state automata (DFA) the resulting state is uniquely produced by the transition function from any state/event pair. In non-deterministic finite state automata (NFA) a state/event pair can create a set of possible successor states. In NFA the specific successor state is chosen randomly from the set of possibilities [167].

Besides deterministic vs. non-deterministic and finite vs. infinite, whether the transition function is a partially or fully defined function separates different types of automata models. In classical DFA theory, often used for computer parsers, it is assumed that the transition function is fully defined. This makes practical sense since a parser reads a user generated text file that may potentially include any character in the alphabet at any point in the file. The parser must know how to handle that character, including possibly throwing an error for 'unacceptable' characters. In contrast, physical machines often have limitations preventing certain events from occurring while in a specific state. This is why the FSMs used to describe physical machines typically have only partially defined transition functions [47].

Additional types of automata are used for only considering infinite length strings. These ω-automata only generate ω-words. These modeling formalisms are used when a machine is expected to never terminate [184]. If the machine never terminates, the traditional interpretation of marked states is useless. Instead, additional conditions are required of all of the generated ω-words. These conditions typically relate how often the marked states must be visited during the infinite execution. For example, Büchi acceptance is a condition requiring at least one of the marked states be visited infinitely often during the execution. A Büchi Automaton is an ω-automata that satisfies the Büchi acceptance condition. Other acceptance conditions include: Muller, Rabin, Street [63].

Despite the many flavors of automata, they all retain the same central characteristics. There are discrete states and the machine switches from one state to another during an instantaneous event. The new state is defined by the transition function. If the events are recorded in a sequence they produce a word in a language that represents all possible behaviors of the machine.

Properties of Individual FSMs

All of the different types of automata have properties that can be analyzed. Here, FSMs will be specifically discussed. The properties of accessibility and co-accessibility have already been mentioned.

Possibly the most fundamental property is whether a FSM's language is included inside another language. If language A is included inside language B, then all strings in language A are also in language B. This allows language B to be a minimally-restrictive specification of some desired FSM condition. Language A is then used to describe the actual FSM. If language A is included inside language B, then all of the behaviors of the actual FSM satisfy the condition. Cassandras and Lafortune [47] use this language inclusion property and the algorithm for its verification extensively for supervisory control, discussed in section 2.2.3.

Another property to be evaluated is if the initial state can be verified from the FSM output. The *state verification* problem has an algorithm to determine if the FSM can be state verified. It also has a constructive algorithm for determining a test that performs the state verification if possible. This allows an on-line verification of the initial state to be performed [114].

A related problem is *state identification*. State identification assumes an unknown initial state. It determines if it is possible to find the initial state from the machine's output. If it is possible, it creates the appropriate test procedure [114]. This is similar to the concept of observability in modern control. First observability is checked as a property, then an observer is created that can be run on-line to perform state identification after a certain upper bounded number of observations.

These types of properties can be evaluated with automated academic tools. UMDES is a set of C libraries from the University of Michigan that can be used to evaluate these properties [16, 111]. DESUMA utilizes UMDES and provides a graphical interaction to make checking these properties simpler [3, 158]. Supremica provides an alternative graphical FSM modeling and verification environment [14, 19]. It performs many of the same operations but appears simpler and more exhaustive.

Additionally, since FSMs have a finite state space and are essentially a labeled transition system, they can be translated to a Promela description (which is based on Kripke Systems [103]) and have LTL [151] properties verified using the SPIN model checker [10]. Alternatively the SMV model checker [9] can be used to verify CTL properties [50].

2.2.2 Systems of Finite State Machines

Individual FSM models are good for modeling and analyzing individual machines, but what actually is an 'individual machine'? Many systems are actually several distinct smaller systems working together in unison. Finding an overall model of a complex system may be quite difficult. It is often much simpler to model each individual component machine as a separate FSM and then somehow combine them to form the overall system model. Typically this overall model is an *equivalent FSM*, meaning it is an individual FSM that is equal in behavior to the entire composed system [29]. This is similar to process algebras where large networks are modeled by the compositions of smaller component processes, each of which may itself be a composition of yet smaller processes. Like process algebras, the many different ways that FSMs could potentially communicate produce many different notions of FSM composition.

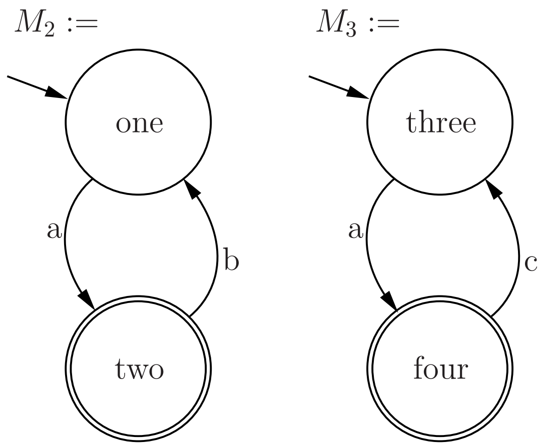


Figure 2.6: A system composed of FSM M_2 and FSM M_3

The simplest form of composition is *interleaving*. It is used to take a set of FSMs and let them run entirely independently from each other. This means that no machines communicate with synchronized events. No machines can restrict other machines' behaviors. Each machine acts entirely on its own and the overall system allows every possible interleaving of events.

Definition 7 *The interleaved composition of FSMs M_i and M_j is:*

$$M_i|||M_j := acc(X_i \times X_j, E_i \cup E_j, f, (x_{0,i}, x_{0,j}), X_{m,i} \times X_{m,j})$$

$$f((x_i, x_j), e) := \begin{cases} (f_i(x_i, e), x_j) & if\ f_i(x_i, e)\ is\ defined \\ (x_i, f_j(x_j, e)) & if\ f_j(x_j, e)\ is\ defined \\ undefined & otherwise \end{cases}$$

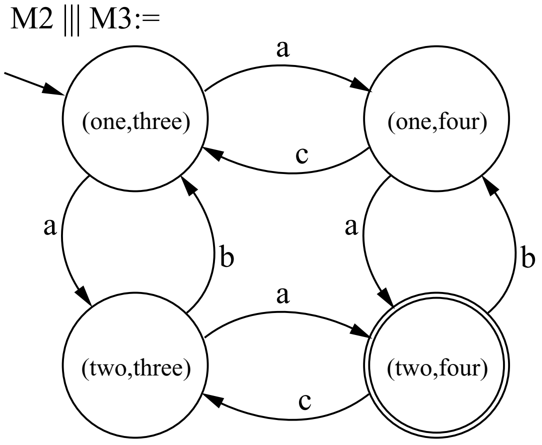


Figure 2.7: A system composed of FSM M_2 interleaved with FSM M_3

Definition 7 shows how to form the FSM equivalent to the interleaved composition of two FSMs. Figure 2.6 shows two example FSMs and figure 2.7 shows the FSM equivalent to M_2 interleaved with M_3. The states of $M_2|||M_3$ are ordered pairs from $M_2 \times M_3$ (meaning the state of the overall system is composed of the states of the individual components). The accessibility operation in the definition removes any unreachable states from the resulting FSM. Since those states weren't reachable, removing them will not affect the system's behavior. The events that can occur are all the events in either M_2 or M_3. The initial state is the pair corresponding to the initial state of each component FSM. The marked states correspond to all states where both components are marked. Finally, the transition relation is formed from the component transition relations. Any event only causes one component to change its individual state. It seems odd that in state $(one, three)$ event a can occur in either M_2 or M_3 causing different resulting states. However, since the FSMs evolve independently they cannot synchronize on this event, and which FSM evolves depends on which one individually experiences the event.

This oddity is removed by using *parallel composition* instead of interleaved composition. In parallel composition, events common to both machine's alphabets can only occur if they happen simultaneously in each machine. This synchronous execution of common events is typically used to represent communication of some form. Figure 2.8 shows the parallel composition for M_2 and M_3.

Definition 8 *The parallel composition of FSMs M_i and M_j is:*

$$M_i||M_j := acc(X_i \times X_j, E_i \cup E_j, f, (x_{0,i}, x_{0,j}), X_{m,i} \times X_{m,j})$$

$$f((x_i, x_j), e) := \begin{cases} (f_i(x_i, e), f_j(x_j, e)) & if\ f_i(x_i, e)\ and\ f_j(x_j, e)\ are\ defined \\ (f_i(x_i, e), x_j) & if\ f_i(x_i, e)\ is\ defined\ \ and\ \ e \notin E_j \\ (x_i, f_j(x_j, e)) & if\ f_j(x_j, e)\ is\ defined\ \ and\ \ e \notin E_i \\ undefined & otherwise \end{cases}$$

Definition 8 shows that parallel composition is similar to interleaved composition except for common events. If an event is only in one FSM's alphabet, that machine can experience the event whenever it is ready to. When an event is in both FSM's alphabets, both machines must be in a place to experience the event. This synchronization event does not indicate which machine 'caused' the event or which machine is output/input. That information can be additionally associated to events, but is not directly in the modeling formalism (e.g. for a specific application event a could be regarded as M_2 causing an output to be sent to M_3 which is waiting for the input). Figure 2.8 shows that the a-transitions from figure 2.7 are replaced with a single synchronized transition.

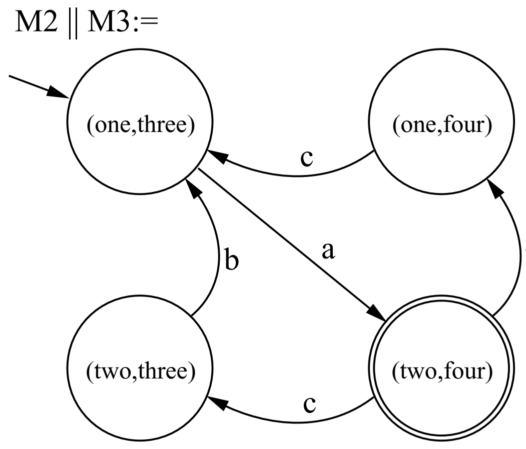


Figure 2.8: A system composed of FSM M_2 in parallel with FSM M_3

Interleaving has the least amount of synchronization (none), parallel composition has an intermediate amount of synchronization based on the common overlap of the alphabets, and *product composition* has the largest amount of synchronization by requiring every event to occur in each component FSM.

Definition 9 *The product composition of FSMs M_i and M_j is:*

$$M_i \times M_j := acc(X_i \times X_j, E_i \cup E_j, f, (x_{0,i}, x_{0,j}), X_{m,i} \times X_{m,j})$$

$$f((x_i, x_j), e) := \begin{cases} (f_i(x_i, e), f_j(x_j, e)) & if\ f_i(x_i, e)\ and\ f_j(x_j, e)\ are\ defined \\ undefined & otherwise \end{cases}$$

Definition 9 shows why product composition is sometimes referred to as 'totally synchronous'. If an event is only in one FSM's alphabet, it cannot possibly occur in the other FSM. This makes it impossible for that event to occur in the product composition of the two (see figure 2.10 and notice the absence of events b and c). Figure 2.9 shows that, before taking the accessibility operation, the product composition has several states that cannot be reached. Figure 2.10 shows the finished product composition.

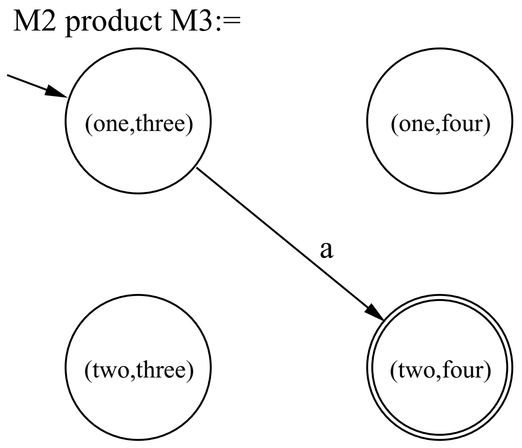


Figure 2.9: A system composed of FSM M_2 and FSM M_3 under product composition, before taking the accessibility operation

The definitions of interleaved, parallel, and product composition are adapted from [47] where additional descriptions can be found. There the associative and commutative properties of the operators are also discussed.

Tools like the aforementioned UMDES, DESUMA, and Supremica have built-in support for creating the equivalent FSMs from component FSMs and the different methods of composition. The equivalent FSM can then be analyzed like any other individual FSM. In this manner the overall system of systems can be checked for deadlock, livelock, language inclusion, and LTL/CTL properties.

There are a number of refined FSM modeling formalisms which choose specific methods of representing input/output events. These models give a specific syntax to individual events and provide additional rules explaining how different machines are

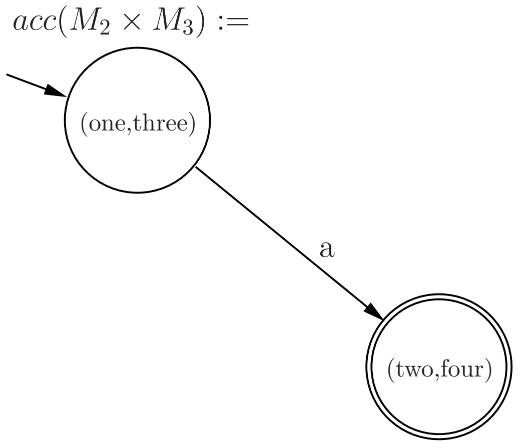


Figure 2.10: A system composed of FSM M_2 and FSM M_3 under product composition, after taking the accessibility operation to remove un-reachable states

allowed to synchronize on input/output events. The oldest examples are the Mealy and Moore machines.

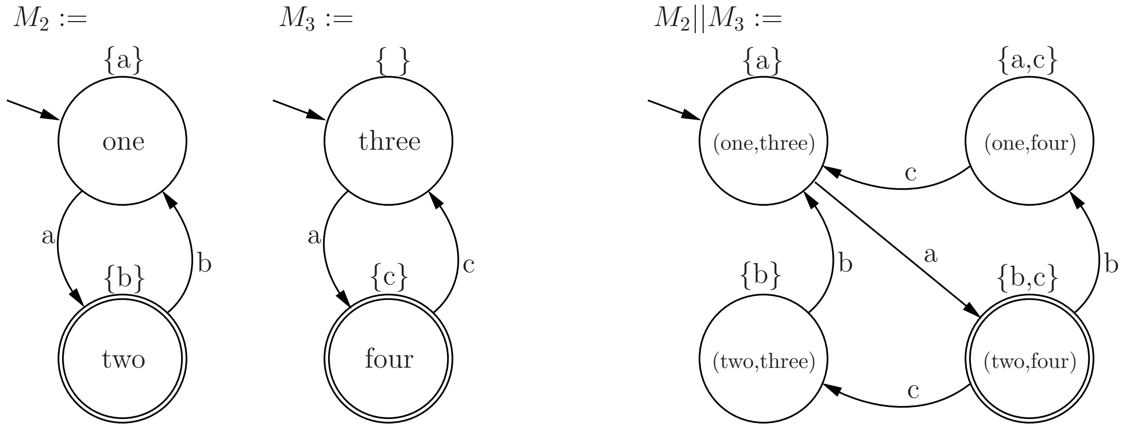


Figure 2.11: A system composed of Moore machines M_2 and M_3 under parallel composition

Moore machines can transition from state to state by observing an input event. When a Moore machine transitions to a new state it may produce an output associated with being in the new state. Moore machines are automata with state outputs. Figure 2.11 shows an example.

Mealy machines have transitions made of an (input, output) event pair. Whenever a transition occurs the machine receives the input event and produces the output event. Mealy machines can be converted to Moore machines and vice versa. The

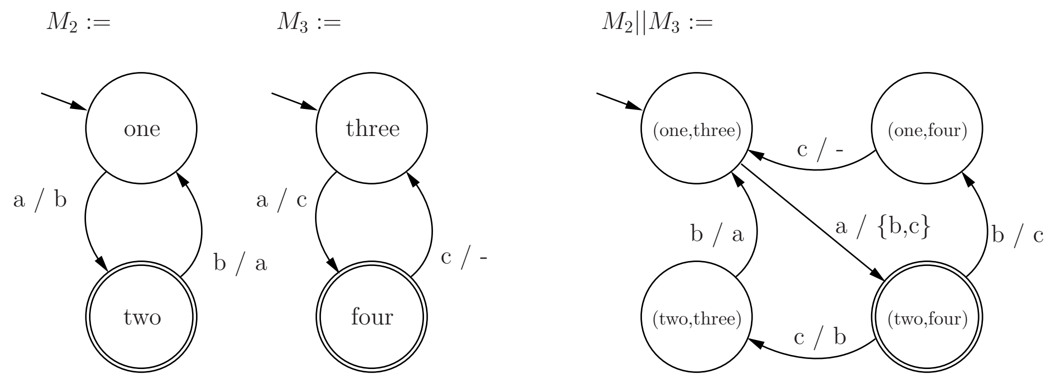


Figure 2.12: A system composed of Mealy machines M_2 and M_3 under parallel composition

second part of the Mealy machine event is the same as the output marking on the Moore machine's destination state. Both receive an input and produce an output, but the output is associated with measuring the state in Moore machines while the output is associated with producing an output communication message in Mealy machines [47, 156].

Lynch's I/O automata are similar to Mealy and Moore machines except that any transition contains only a single input, output, or internal event [126]. The I/O automata's alphabet is partitioned into these three sets (input, output, internal) and form the *signature* of the machine. The signature represents the machine's interface with its environment. An individual machine's internal events do not appear in any other machine's alphabet and may be taken whenever the machine wishes. Any individual communication event can only be output by a single machine and that machine causes the event to occur. Every other machine with that event in its alphabet must accept it as input at any state. This models one machine broadcasting an output event and every other machine listening having to transition appropriately.

Lynch extended I/O automata to Dynamic I/O Automata (DIOA) in [26]. Here the signature of the automata was made to vary with state. This allowed some of the inputs to be disabled in some of the states. It also allowed a machine to transition into a state where it had no events in its signature, meaning that it could never input, output, or internally transition. This effectively removes the automata from the system since it can no longer interact. The set of output events was also augmented with a *create*(A) action where an automata could call a specific and pre-defined DIOA A into existance, thus creating it. The standard notion of an equivalent machine had to be modified since the number of components in a DIOA system varies during execution. Lynch created a *configuration automata* which is itself an automaton, but where each state is made of the set of machines currently in existence and their individual states. So if a component automata makes an internal transition, the global configuration also makes a transition but will not change which components exist. If one component automata transitions to a state with an empty signature, the global configuration will remove that 'dead' automata. If one component executes a *create*(A) action, that component will transition and the automata A will be augmented to the configuration. DIOA does allow a *dynamic dimension system*, but all of the components have to be predefined and according to Lynch "I/O automata could be used for this purpose, with the addition of some extra structure (special Boolean flags)" [26]. It is not like the π-calculus that can replicate arbitrarily many copies from an individual process definition. The global state space contains every combination of every predefined automata in every possible state, which is extremely large but finite and predefined.

There are several other types of communicating FSMs. Shaw's communicating real-time state machines (CRSMs) are FSMs with events written using a CSP-style notation [166]. Lee discusses how to combine FSMs with multiple other concurrency models [113, 74]. Harel's Statecharts are similar to the I/O automata concepts, but additionally add hierarchy [80, 81]. Unlike most of the purely academic formalisms, Statecharts have been used for implementations [183] and are the basis for the popular Matlab Stateflow toolbox (they are Matlab's FSM formalism).

2.2.3 Supervisory Control

In traditional control applications there are typically physical components called *plants* whose definitions and behaviors cannot be modified. To affect the plant's behavior additional component systems, called *controllers*, are added and connected to the plant. The interaction of the two components alters the overall system's behavior to be more desirable, figure 2.13.

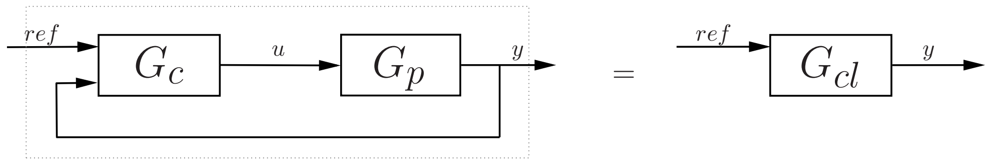


Figure 2.13: The controller G_c connected to the plant G_p in feedback is equivalent to the closed-loop system G_{cl}

This separation of controller and plant does not typically appear in computer science (e.g. it is not in process algebras). Most CS research deals with digital computer programs whose source files are immediately available. A physical system does not have a source file. To understand its behavior, it must be modeled using first principles [140] or system identification techniques [117]. Computer programs can also be easily changed to achieve almost any desired behavior. This is possible because computers are essentially *Universal Turing Machines* which can execute any Turing machine specified. Physical systems have fixed definitions based on physics. One cannot choose to negate the effect of gravity as one would add a negative sign to a C++ equation. Physical systems are more like compiled executable code with a fixed interface. One knows the inputs and outputs but does not have access to its source code and definitely does not have the ability to modify it.

This difference in approaches caused traditional automata theory from computer science to ignore control issues; after all there rarely was anything resembling a fixed plant. However, Ramadge and Wonham developed *supervisory control* techniques based on FSMs. They enabled the control of DES's containing physical as well as digital components [187, 186]. Several interacting component systems, each modeled as a FSM, can be composed together to form a plant (typically using parallel composition). Since this plant contains physical components, abstractly modeled with FSMs, it should be considered fixed. To alter the overall system's behavior, another controller FSM can be composed in parallel. This *supervisory controller* restricts the behavior of the overall system so that it becomes 'acceptable'.

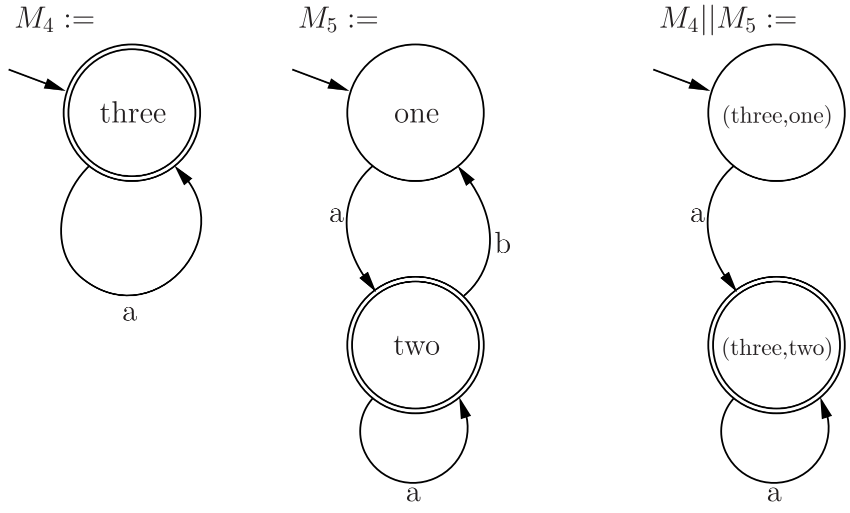


Figure 2.14: The controller M_4 connected to the plant M_5 in parallel is equivalent to the "closed-loop" system $M_4||M_5$

Since the supervisory controller is implemented with a FSM composed in parallel to the plant, the control interaction must be through the events common to their alphabets. For example, figure 2.14 shows a controller M_4 and a plant M_5, both defined over the common alphabet $\{a, b\}$. Any event in both the controller M_4 and the plant M_5 can only occur if it happens concurrently in both FSMs. In this way the controller can 'enable' and 'disable' events by either having or not having transitions for that event in the current state. Since M_4 has no event b, it disables the event b

and prevents M_5 from ever executing it. This type of control action produces, at each moment, a set of possible events the plant can then choose to execute. The controller does not normally 'force' a specific event to occur, but it could enable only a single event, leaving the plant to choose when to execute it [76]. The controller and plant could also enter a state where no events were enabled causing deadlock to occur.

With this method, controllable events can be enabled and disabled by the controlling FSM. Some events in the plant, however, are not controllable. If an event is not controllable, obviously the controller cannot disable it. This creates a requirement that all uncontrollable events must be enabled at every state. The controller may still observe the uncontrollable event which can cause a state change in the controller FSM. This state change can result in a new set of enabled/disabled events. The alphabet of events can be partitioned into either controllable or uncontrollable events.

Similar to controllability, there is a concept of observability of events [154]. Some events in the plant occur, but cannot be sensed. These events are unobservable. Obviously the controller FSM cannot react to unobservable events. Like controllable/uncontrollable, any FSM's alphabet can be partitioned into observable/unobservable [49].

A controllable and observable event can be disabled or enabled by the controller. If it is enabled and occurs the controller will sense it. A controllable and unobservable event can still be disabled or enabled by the controller, but the controller will not be able to sense it when it occurs. An uncontrollable and observable event must always be enabled by the controller. The controller can still change state based on the event occurring. Finally, an uncontrollable and unobservable event cannot be disabled and cannot be sensed when it occurs.

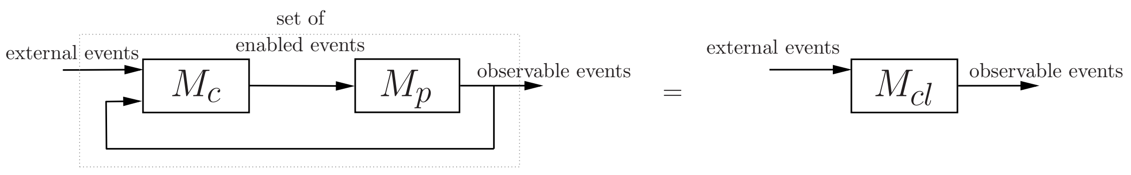


Figure 2.15: The controller M_c connected to the plant M_p in parallel is essentially a feedback control loop.

As shown in figure 2.15, a practical physical implementation of supervisory control will have sensors in the plant to measure the state and detect any of the observable events. When an enabled event occurs in the plant it causes a transition in the plant. If the event is observable it is immediately communicated to the supervisory controller. The observable events from the plant are essentially inputs to the controller. The controller immediately takes a transition with the same event to a new state. In this new state there are a new set of outgoing transitions which are the new 'enabled events'. This set of enabled events is communicated back to the plant which then enables those events. The plant may then execute one of those new enabled events whenever it pleases. Again, an enabled event may occur and possibly be observed and communicated to the controller. Additionally, there may be external events in M_c that are not in M_p; these function similarly to a reference input.

Ramadge-Wonham supervisory control introduced more than just how to create a feedback control system with FSMs. It included a synthesis procedure that takes a plant model and a desired system behavior specification and synthesizes a controller. When that controller is coupled with the plant it produces exactly the same language as the specification. The procedure either produces the desired controller, or fails, indicating that it is impossible to generate such a controller.

The overall design process begins with modeling the plant as a FSM. This can either be done using first principles or system identification tools for FSMs [114]. The plant components could be individually modeled and then composed to form the overall plant. Once there is a FSM describing the uncontrolled system (plant), a specification representing the desired closed-loop behavior must be created. The desired behavior could be represented as a language, but it is more convenient to represent it with another FSM. Common techniques for creating this specification FSM are in [47, 187] and tend to use the product composition of FSMs for representing several individual requirements. Once one has the plant and the specification FSMs, the algorithms found in [153, 155, 47] allow one to compute the supervisory controller that exactly produces the closed-loop behavior of the specification, if it exists. There are conditions which depend on the observability and controllability of the plant's events and the desired specification behavior that determine if finding a supervisory controller is possible [47, 49].

Often the desired closed-loop behavior cannot be achieved with the given partitioning of controllability and observability. One option is to change the design of the plant by adding additional sensors and/or actuators to modify which events are observable and which events are controllable. This is not always possible or economical. The other option is to modify the desired behavior so that it is achievable. Several operations defined using the specification FSM make this easier. The *supremal controllable sublanguage* of a language $\mathcal{L}$ is written $\mathcal{L}^{\uparrow C}$ [187]. It represents the largest controllable language inside $\mathcal{L}$. So while $\mathcal{L}$ may not be controllable, $\mathcal{L}^{\uparrow C}$ is guaranteed to be controllable. This operation can take the original specification FSM and compute a new specification FSM that is guaranteed to be controllable, but will contain fewer possible behaviors than the original. If the system is severely defective the new closed-loop system may have the null language, meaning that there are no behaviors in the original specification that can be achieved by the plant.

Instead of removing behaviors from the specification FSM, the *infimal prefix-closed controllable superlanguage* ($\mathcal{L}^{\downarrow C}$) adds as few additional behaviors as necessary to form a new language that is guaranteed to be controllable. With the infimal prefix-closed controllable superlanguage a designer must check that the new behaviors are tolerable. With the supremal controllable sublanguage a designer must check that all necessary old behaviors still exist. Cassandras and Lafortune discuss similar operations related to the concept of observability [47].

Ramadge-Wonham supervisory control has been used in many theoretical decentralized control problems where the plant can be modeled ahead of time and does not change configuration during execution [161]. The design and controller synthesis are done off-line and automatically produce the desired controller. The literature provides many simple theoretical distributed examples, but there does not appear to be much use of Ramadge-Wonham supervisory control within UAV research or the more general mobile robotics. It is quite likely that it has been influential without being directly applied. The only example found also notes the inexplicably few practical applications found in literature: “Despite numerous theoretical contributions to the field, only a few applications of the RW method have been reported” [116]. In [116] a system composed of two reactive wheeled robots with proximity sensors is modeled as the parallel composition of two FSMs. These robots are then composed with a supervisory controller that guarantees they don’t collide with each other. While this application was almost as simple as possible, the notions from supervisory control could still end up being quite useful and influential in more general networked mobile robotics research.

2.2.4 Applications of FSMs to UAVs

Finite state machines are a popular modeling formalism and have appeared regularly in UAV literature. One issue with using a FSM to model a UAV is the discrete nature of a FSM’s dynamics and the continuous nature of a UAV’s dynamics. Something in the system must be appropriately discretized in order to make a FSM model useful. One common approach is to discretize the environment. A second common approach is to discretize the behaviors a UAV can perform into a set of maneuvers.

Language over a Discretized Environment

If the UAV’s environment is a fixed predefined space, it can be discretized into a set of triangular regions, figure 2.16. Each region is a state in a FSM describing the environment. Any regions adjacent to each other have transitions connecting the two. In this way a string of the FSM records a path through the environment from one region to another, always requiring the next region to be adjacent to the current region. The language of the environment FSM is the set of all possible paths through the discretized environment.

In [110, 34] CTL or LTL statements are used to specify a desired robot motion. The system then searches off-line for a single string that satisfies the requested statement. When one string is found it is used as the planned robot path (a discretized

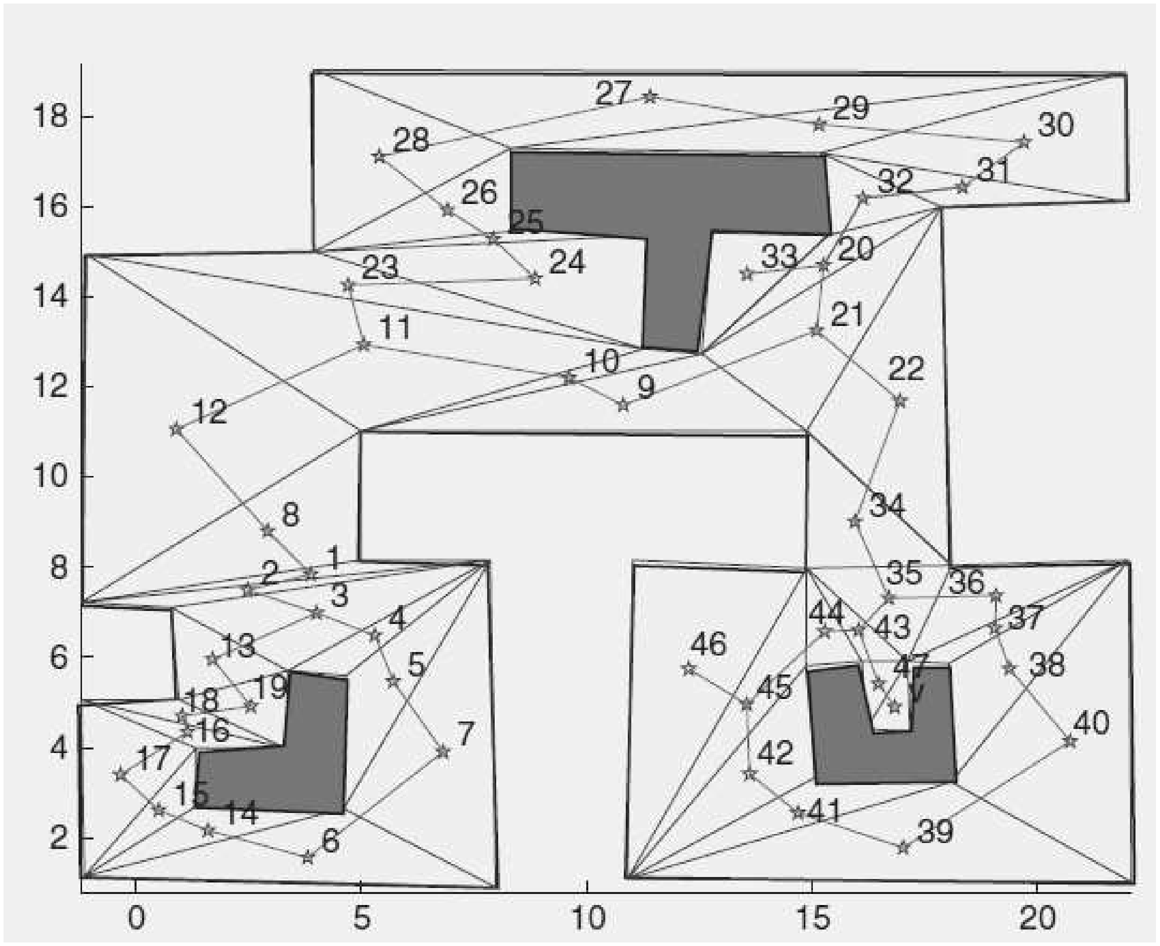


Figure 2.16: A path through a static a priori known environment which is discretized into triangular regions [34].

series of regions to be visited). This plan is then used to generate a reactive behavior that runs fully autonomously. The application described moves a wheeled robot through an indoor research laboratory environment. The robot is assumed to be fully actuated. The low-level controllers depend on this fully actuated assumption to guarantee that they can move from one region to any specified adjacent region. The authors note that this is not an appropriate assumption for more complex dynamics such as vehicles with non-holonomic constraints (e.g. UAVs). They note: "it is restricted to static, a priori known environments and simple robot dynamics". For network-level control neither of these assumptions is appropriate. Additionally, computing a compiled reactive behavior in an off-line manner is not the form of mixed-initiative interaction desired.

Language over Discretized Maneuvers

FSMs can also be used to describe the several different low-level control modes that a UAV can switch between. This is closely related to the hybrid systems to be discussed in section 2.3. If a UAV is represented as a FSM and not a hybrid system, the low-level continuous-time dynamics cannot be modeled but the linguistic switching between behaviors can. One major reason for using FSMs instead of hybrid systems is that analyzing FSM properties can be done much faster than analyzing hybrid system properties. If the property of interest is preventing deadlock, preventing livelock, ensuring a specific pattern of modal behavior, or any other linguistic property, it is best checked as a FSM and not a hybrid system.

Several groups have used *maneuver automata* to represent the potential behaviors of a mobile robot. An automated helicopter's behavior is specified in [34, 70]. The maneuver automata models a library of aggressive acrobatic maneuvers. A string from this maneuver automata can be selected off-line and used to create and compile the fully automated reactive behavior's implementation. Other research directly runs the maneuver automata without specifying a specific string to be executed [100]. This is closer to how a plant from supervisory control would operate. [100] uses a maneuver automata to specify the behavior of a UAV in a military battlefield. There is no explicit controller composed with the maneuver automata, instead the switching occurs based solely on the evolving probability of target locations.

MissionLab also implements FSM-based reactive behaviors. It is an implementation of the Configuration Description Language (CDL) defined at Georgia Tech [127, 64]. MissionLab does not directly define a maneuver automata for a mobile robot, but allows every component of the robot to be represented as a FSM that can be composed with other FSMs both inside the same robot and inside other robots. MissionLab's computer science based approach does not focus on the low-level control aspects, but ignores modeling and controlling these dynamics. The many FSMs communicate through common events. MissionLab uses this architecture to create a *Societal Agent* from many interacting sub-agents. The system's overall behavior emerges from the composition of the many sub-agents. The system is reactive; its sub-agents are specified in the CDL file that is then compiled and downloaded onto the physical robots where it is executed until the user chooses to abort. There is no explicit notion of plant or controller. There is no notion of a specified desired behavior. There is no human operator interaction during execution. If the human operator wants something specifically done, they must understand how to create a set of FSMs that will accomplish exactly what they want, then run that compiled program in an open-loop manner.

Finite state machines have many desirable characteristics that may make them useful for network-level controllers. They have equivalent graphical and textual definitions. This makes it easier for a human operator to understand on-the-fly what any individual machine is doing and will do. They can be composed together to form large systems from smaller sub-systems. There is a well developed notion of plant and controller, thanks to Ramadge and Wonham. There are relatively efficient verification techniques.

One detriment of FSMs is the state explosion problem. FSM composition is a combinatorial process that produces an extremely large number of states. A system of 5 components each having 10 states produces up to $10 \times 10 \times 10 \times 10 \times 10 = 100,000$ states in the equivalent FSM. This makes model checking slow, but also creates large equivalent systems. It would be difficult to present a FSM with 100,000 states to a human operator and have them understand any part of it. Representing concurrent sub-systems in this manner makes viewing and understanding the entire system's state complicated. This is one argument for using an alternative representation of concurrency, such as the one in Petri nets, presented in section 2.4.

2.3 Hybrid Systems

In engineering, hybrid systems result from combinations of discrete-event systems and continuous-time systems. Some engineered systems can be accurately represented with continuous-time models, such as those used in classical control. Some systems can be accurately represented with discrete-event models, such as those used for FSMs. Interestingly, some systems cannot be described by a discrete-event model, a continuous-time model, or separate discrete-event models and continuous-time models. These systems can only be accurately represented by models that include interdependent continuous-time and discrete-event dynamics. These hybrid systems include the expressiveness, complexities, and challenges of both parents and should not be used if either of the simpler parent models would suffice.

Embedded computing is a common example where hybrid systems can be used to describe an analog environment that is controlled by an embedded computer which samples connected sensors and produces commands for connected actuators [115]. The environment evolves in continuous-time while the computer is inherently a discrete-event system. Mode switches in the control logic also add to the discrete-event behavior.

Hybrid systems, such as embedded systems, have a state that is unsurprisingly comprised of discrete states and continuous states. In continuous-time models the states flow and form trajectories that are continuous functions of time, figure 2.17. In discrete-event models the states form trajectories that are piecewise constant functions of time that change instantaneously during events. In hybrid models the hybrid states can contain both types of behaviors, continuous and piecewise constant. Additionally, the interaction between the discrete-event and continuous-time dynamics also produces continuous states whose trajectories are piecewise continuous.

There are several competing types of hybrid systems formalisms. Each describes a model of computation that produces hybrid behaviors. Some process algebras, like

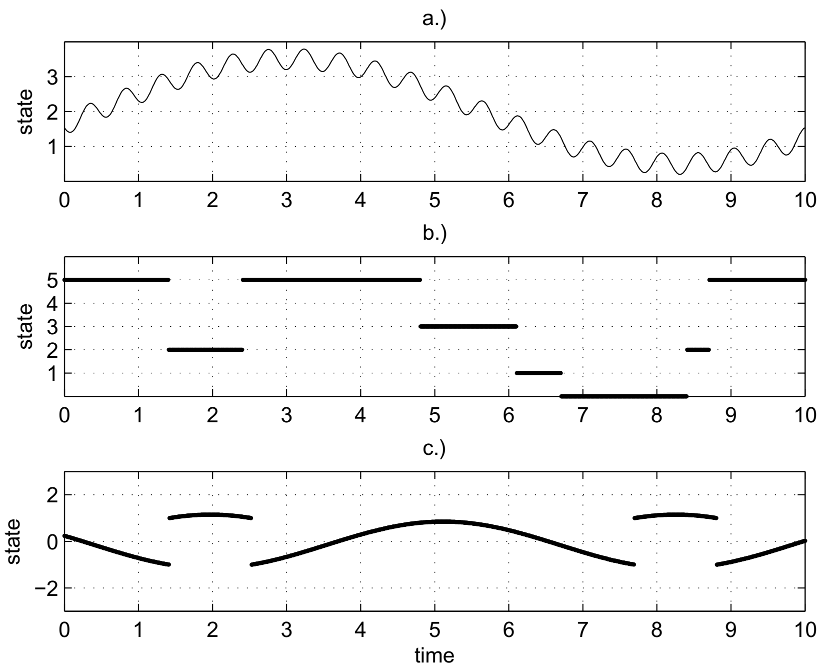


Figure 2.17: Hybrid systems have state trajectories that include: a. the continuous trajectories found in modern control, b. the piecewise constant trajectories found in FSMs, c. piecewise continuous trajectories not found in modern control or FSMs

ACP, have been extended with simple continuous-time dynamics to produce hybrid process algebras [36, 54, 159]. Embedded graph grammars add continuous-time dynamics to graph grammars (graph grammars are graphs whose edges are allowed to change according to a predetermined set of rules: the system's grammar). The embedded graph grammar produces graph nodes who change their connected edges while wandering through an analog environment [129]. The two previous types of hybrid modeling formalisms are discussed by researchers who developed them as extensions to their previous work on process algebras or graph grammars [105, 130].

Hybrid automata, on the other hand, have become more popular, more thoroughly investigated, and are still a very active area of research. Hybrid automata are the combination of finite state machines and continuous-time dynamics (e.g. non-linear differential equations). They have become the de facto modeling formalism for hybrid systems and are used in many applications and implemented by many tools. Like the FSMs from section 2.2, hybrid automata include many different and specialized formalisms.

2.3.1 Individual Hybrid Automata

The hybrid automata model described in [123] is similar syntactically and semantically to most other models found in literature and will be the basis for the terminology used in this section. Different researchers coming from different perspectives use varying terminology when referring to the same parts the hybrid automata model (e.g. the discrete state is also called the mode or the location of the automata). The terminology may vary but the ideas, behaviors, and examples are typically consistent.

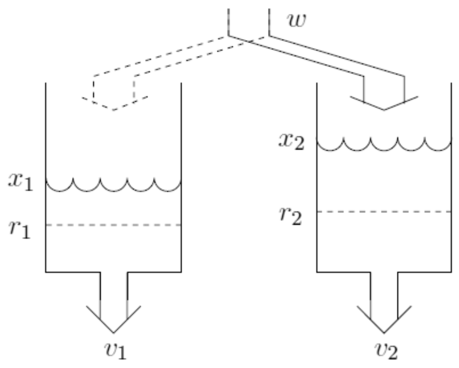


Figure 2.18: Two liquid filled tanks from example 5.

Example 5 *A frequently presented hybrid automata example involves two tanks holding liquids, figure 2.18. The tanks have liquid levels x_1 and x_2 that vary based on constant outflows v_1 and v_2 as well as an inflow from a hose that can be instantaneously switched between tanks, w. The intended behavior of the system is for each tank to remain at the provided reference levels r_1 and r_2 [123].*

Syntax

In order to specify a hybrid automaton, a mathematical set-based syntax similar to that in definition 1 for FSMs is most often used.

Definition 10 *An individual hybrid automaton can be modeled as a tuple $H = (Q, X, Init, f, Inv, E, G, R)$ where:*

- *Q is the finite set of discrete states (variables, modes, locations),*
- *X is the finite set of continuous states,*
- *$Init \subseteq Q \times X$ is the set of initial states,*
- *$f : Q \times X \rightarrow X$ produces the derivative of the continuous variables,*

- $Inv : Q \to 2^X$ *is the invariant describing when continuous evolution is possible,*
- $E \subseteq Q \times Q$ *is the set of discrete transitions between states,*
- $G : E \to 2^X$ *is the guard function determining if a discrete transition is enabled,*
- $R : E \times X \to 2^X$ *is the reset relation reassigning continuous states after a discrete transition.*

In hybrid automaton models for describing networks, to be discussed in section 2.3.2, the discrete transitions between states are additionally given explicit event labels from a specified alphabet ($E \subseteq Q \times \Sigma \times Q$). These events are then used for synchronization just as in FSMs and can be further partitioned into input, output, and internal events ($\Sigma = in \cup out \cup int$).

Example 6 *The liquid tanks in example 5 can be given a syntactic description as* $H_{tanks} = (Q_{tanks}, X_{tanks}, Init_{tanks}, f_{tanks}, Inv_{tanks}, E_{tanks}, G_{tanks}, R_{tanks})$ *where:*

- $Q_{tanks} = \{q_1, q_2\}$
- $X_{tanks} = \Re^2$
- $Init_{tanks} = \{q_1, q_2\} \times \{x \in \Re^2 | x_1 \geq r_1 \wedge x_2 \geq r_2\}$
- $f_{tanks}(q_1, x) = \begin{bmatrix} w - v_1 \\ -v_2 \end{bmatrix}$, *and* $f_{tanks}(q_2, x) = \begin{bmatrix} -v_1 \\ w - v_2 \end{bmatrix}$
- $Inv_{tanks}(q_1) = \{x \in \Re^2 | x_2 \geq r_2\}$, *and* $Inv_{tanks}(q_2) = \{x \in \Re^2 | x_1 \geq r_1\}$
- $E_{tanks} = \{(q_1, q_2), (q_2, q_1)\}$
- $G_{tanks}(q_1, q_2) = \{x \in \Re^2 | x_2 \leq r_2\}, G_{tanks}(q_2, q_1) = \{x \in \Re^2 | x_1 \leq r_1\}$
- $R_{tanks}(q_1, q_2, x) = R_{tanks}(q_2, q_1, x) = \{x\}$

Much like FSMs have an equivalent graphical representation, hybrid automata can be represented graphically. Figure 2.19 shows the hybrid automata equivalent to the syntax in example 6. The discrete states are that the hose is either filling tank 1 in q_1 or filling tank 2 in q_2. Graphically, the discrete states (circular nodes) contain a label at the top for identifying the mode, the continuous dynamics in the middle, and the invariant in effect shown at the bottom. The guards and resets are shown next to the discrete transitions which are represented as directed arcs (directed edges) between modes. The arcs used to denote the possible initial states are also annotated with the conditions necessary upon initialization.

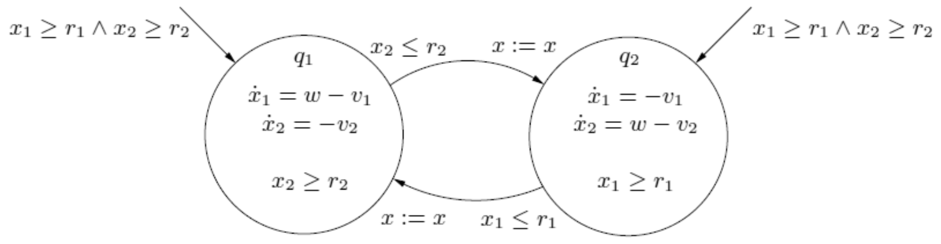


Figure 2.19: The hybrid automaton representing the two tanks of liquid specified in example 6.

Semantics

Specifying the behavior of a hybrid automaton based on its syntactic definition is somewhat more complicated than in the case of FSMs or differential equations. Hybrid automaton behavior involves two types of evolutions: continuous flows and discrete jumps. This requires a new description of time than includes both continuous flows and ordered instantaneous events.

Definition 11 *A hybrid time set τ is composed of a sequence of intervals I_i of continuous time, $\tau = \{I_i\}_{i=0}^{N}$. It is possible for an infinite number of intervals to occur, $N = \infty$. The intervals must satisfy:*

- $I_i = [\tau_i, \tau_i'] \quad \forall i < N,$
- *if* $N < \infty$ *then either* $I_N = [\tau_N, \tau_N']$ *or* $I_N = [\tau_N, \tau_N'),$
- $\tau_i \leq (\tau_i' = \tau_{i+1}) \quad \forall i.$

The hybrid time sets are used as the model of time for specifying the hybrid trajectories.

Definition 12 *A hybrid trajectory is a triple (τ, q, x) where:*

- *τ is a hybrid time set according to definition 11,*
- *q is a series of functions q_i mapping intervals to discrete states ($q_i : I_i \rightarrow Q$),*
- *x is a series of functions x_i mapping intervals to continuous states ($x_i : I_i \rightarrow X$).*

Figure 2.20 shows a hybrid trajectory with a hybrid time set $\tau = \{[0, 2], [2, 3], [3, 3.5]\}$. Multiple single point intervals can occur in a row at the same continuous time indicating the ordered execution of instantaneous events (e.g. a hybrid time set $\{[0, 1], [1, 1], [1, 1], [1, 4], [4, \infty)\}$ would have three events occur at $t = 1$ in the hybrid trajectory to cause the jumps).

There are many hybrid trajectories, not all of them are hybrid behaviors of a given hybrid automaton.

Definition 13 *A hybrid trajectory (τ, q, x) is a behavior (execution) of a hybrid automaton H if:*

- *Correctly Initialized: $(q_0(0), x_0(0)) \in Init$,*
- *Correct Discrete Behavior: for all intervals i,*
  - $(q_i(\tau_i'), q_{i+1}(\tau_{i+1})) \in E$,
  - $x_i(\tau_i') \in G(q_i(\tau_i'), q_{i+1}(\tau_{i+1}))$,
  - $x_{i+1}(\tau_{i+1}) \in R(q_i(\tau_i'), q_{i+1}(\tau_{i+1}), x_i(\tau_i'))$,
- *Correct Continuous Behavior: for all intervals i,*
  - $q_i : I_i \to Q$ *is constant over* $t \in I_i$,
  - $x_i : I_i \to X$ *satisfies the differential equation,* $\frac{dx_i}{dt} = f(q_i(t), x_i(t))\ \forall t \in I_i$,
  - $x_i(t) \in Inv(q_i(t))$ *for all* $t \in I_i$.

Definition 13 starts by requiring a hybrid behavior to start in an acceptable initial condition. The second requirement is that discrete-event jumps can only occur where transitions exist, where their guards are enabled, and where they correctly execute the continuous state reassignment. The third requirement is that during continuous evolution the discrete state stays constant, the continuous state satisfies the stated dynamics, and the invariant always remains true.

Figure 2.20 shows a hybrid behavior of the hybrid automaton in figure 2.19. One can see that the initial discrete state is q_1 and the initial continuous state is $x_1 = 0$, $x_2 = 1$. The inflow of liquid in tank 1 causes x_1 to increase while the outflow from tank 2 causes x_2 to decrease. At $t = 2$ the guard for switching to filling tank 2 becomes true and the invariant for staying in tank 1 becomes false. This forces the discrete state to switch to q_2.

[123] contains a more thorough exploration and explanation of the example's behavior. It also discusses conditions for the existence of a hybrid behavior for a given hybrid automaton. It is possible that a hybrid automaton's syntax specifies a 'defective' automaton that accepts no behaviors. Additionally the uniqueness of any behaviors is discussed.

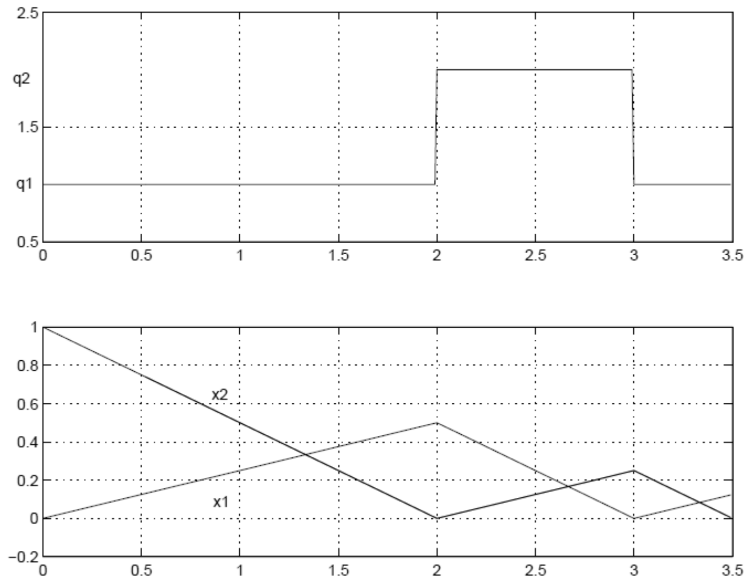


Figure 2.20: Hybrid behavior of example 5.

Non-determinism can arise due to multiple transitions being enabled at the same moment. Alternatively, if a transition is enabled while the invariant still allows the system to evolve continuously, the automaton must non-deterministically choose between transitioning and flowing.

Finally, [101] discusses the Zeno behavior that can rarely result from overly abstracting models. Zeno behavior is a mathematical curiosity whereby the system switches an infinite number of times in a finite duration. This results in the system 'stalling at the Zeno time'. It is argued that this never happens in reality, but the mathematical oddity causes problems for numerical simulations using the hybrid automaton's semantics.

Provable Properties

While simulating hybrid automata provides a useful analysis technique, explicitly proving other guaranteed properties is often necessary and more enlightening than running several simulations. One commonly desired property is stability. Individual discrete states (modes) of the hybrid automaton can be separately analyzed for continuous-time properties such as stability, but the discrete transitions (jumps) can cause systems composed of only stable modes to become unstable and systems of only unstable modes to become stable. This results in stability being a composite property of the entire hybrid automaton that cannot be proved in isolation [123]. Where it can be proved, it typically requires a manually crafted Lyapunov function that depends on the discrete state as well as the continuous state. However, there are some special cases and applications that allow for automated verification techniques [28, 27].

Calculating the reachable states to guarantee that the system avoids an unsafe region is another common area of interest, often called the 'safety problem'. This is sometimes proved manually as in [163] where two simple UAVs are given a simple collision avoidance controller. More often, the systems to be analyzed are complex and automated methods for computing the reachable states is desired. Other LTL and CTL properties besides safety can also be stated, and potentially verified automatically.

Unfortunately, the labeled transition systems for hybrid automata have an infinite number of states. FSMs and most process algebras can be model checked because they have finite transition systems that can be exhaustively searched. Once continuous variables are included this becomes more complex. A continuous variable can itself take on an infinite number of values; these points cannot all be checked individually in finite time. This makes it necessary to somehow symbolically represent a section of states as an equivalence class.

In [22] Alur et al. describe how this can be done to produce a finite discrete abstraction of a hybrid automaton that preserves LTL or CTL properties. This equivalent FSM can then be model checked automatically for these LTL or CTL properties. Unfortunately, the classes of hybrid automata where this works are limited. Alur et al. describe how this works for timed automata, multi-rate automata, and rectangular automata and then give arguments for why this set cannot be expanded.

Timed automata are simply hybrid automata with all continuous variables being clocks. All clocks progress at the same rate, 1. All transitions can only compare the clocks to constants for guards and can only reinitialize or leave alone any individual clock. This can also be viewed as adding simple timing to FSMs. While the model may be rather restricted, thanks to a property preserving finite discrete abstraction, tools like KRONOS and UPPAAL can automatically model check timed automata [17, 33].

Multi-rate automata are similar to timed automata except that their clocks all progress at different fixed rates. By redefining the units one can scale all of the clocks to execute at rate=1, thus converting a multi-rate automaton into a timed automaton.

Rectangular automata are the least restricted of these modeling formalisms. Rectangular automata allow the continuous variables to be given a rectangular range in which they execute (meaning $\dot{x} \in [4, 6]$ allows the rate anywhere between 4 and 6). Additionally all invariants, guards, and resets have to be rectangular specifications. Rectangular automata can be model checked by HyTech [82, 83, 21].

Additional research is being done on specific types of hybrid automata that can be given these property preserving finite discrete abstractions, but another current research direction is the possibility is using approximation techniques. If one can over-approximate the reachable set, checking that the over-approximation does not intersect the unsafe set is sufficient to guarantee safety. In this way tools like the level set methods described in [136, 123] can still verify some properties of the system. These tools currently only work on systems of small dimension and still take a significant amount of time to compute.

While the limited types of hybrid automata that can be model checked don't include the non-linear dynamics that characterize UAVs, they could still be useful for giving precise specifications to the hybrid UAV systems. Hybrid automata provide a common language and terminology that enable accurate simulations and allow concise discussions to occur. For example, the system architecture of the Australian Center for Field Robotics is specified with a hybrid automaton that describes how their system is anticipated to behave [51, 79]. Anyone with knowledge of hybrid automata can understand its expected behavior despite the fact that it cannot be automatically checked for stability or automatically checked to satisfy LTL/CTL properties.

2.3.2 Networks of Hybrid Automata

Since many individual hybrid automata cannot be automatically verified, it is obviously true that networks of hybrid automata may suffer from limited automated analysis tools as well. Restricted models like timed automata, multi-rate automata, and rectangular automata can have models for interacting networks. The tools like UPPAAL mentioned in section 2.3.1 typically enable networks of automata to be automatically converted into an equivalent automaton and then verified just as individual automata.

More expressive hybrid automata formalisms may not allow automated verification, but they do provide a language for specifying models for interacting networks of realistic hybrid automata. These specifications, along with their semantics, allow sophisticated simulators to be designed and developed to explore network behaviors.

Hybrid I/O Automata

Nancy Lynch augmented her I/O automata with continuous dynamics to create Hybrid I/O Automata (HIOA) [125]. Like I/O automata, the discrete transitions have associated broadcast-like events which can be used for synchronization between automata. These events are partitioned into input, output, and internal events. Internal events of an automaton are not allowed to occur in any other automaton. Each output event is only allowed to be emitted by one automaton, a single-writer assumption. Input events in the signature of any automaton must always be enabled. Whenever an output event occurs, all HIOA with that event as an input must synchronously transition.

In addition to the I/O automata-like behavior, HIOA has continuous states which are also partitioned into input, output, and internal. Like all input events must be enabled, HIOA assumes all continuous input states must be enabled (read) at all times. So as I/O automata change modes they are not allowed to change accepted events or accepted continuous input signals. Continuous output is also forced to be written by a single HIOA. Interestingly, in [124], the continuous flows and discrete transitions are lumped together in a single transition relation D, which takes the automaton from one HIOA state to the next. How this is done and how the relation is specified is not defined, merely that it transitions from a predecessor to a successor state. It seems to ignore any detail about guards, invariants, resets, or differential equations.

HIOA are used in several theoretical examples including modeling DNA replication [107] and automated platooning of cars on highways [61, 122]. Some properties are manually inductively proved based on the HIOA models, but it does not appear that a HIOA simulator has yet been created.

CHARON

CHARON is a modeling formalism from the University of Pennsylvania that also allows the parallel composition of interacting hybrid automata to form networks. CHARON adds hierarchy to the model to allow modes to be refined into sub-modes, figure 2.21. This hierarchy allows systems to be abstracted/refined into simpler/more-complex models. This makes analysis easier for the more abstract system descriptions. The hierarchical system could be 'flattened', but the hierarchy makes comprehension and analysis simpler [67].

Unlike HIOA, CHARON does not have any events labeling discrete transitions. All coordination is done through shared global variables. This means hybrid automata do not synchronize on transitions [23]. This helps to allow a compositional semantics in which individual component automata's semantics depend only on the traces of shared global variables. The shared variables are partitioned into input and output variables, but no distinction of a single writer for outputs seems to be present (but is likely understood).

The stated purpose of CHARON is to provide a modeling formalism that includes hierarchy and a compositional semantics [67, 24]. It does this as well as provides a hierarchical hybrid automaton specification language and simulation environment. Due to the compositional nature of the CHARON model and semantics, it is postulated that a parallel implementation of the simulation environment would speed up simulations. Much of the work on CHARON has involved efficiently simulating a network of hybrid automata [23].

The compositional modeling language has been used to describe several applications at U. Penn. It has been used to analyze a fixed network of fully autonomous UAVs [78, 32]. It has also been suggested for use in general platform-independent robot modeling [168, 92, 93]. Current work is being done on allowing CHARON models to automatically generate embedded software for execution [25].

If a system can be modeled by a fixed configuration network that doesn't require

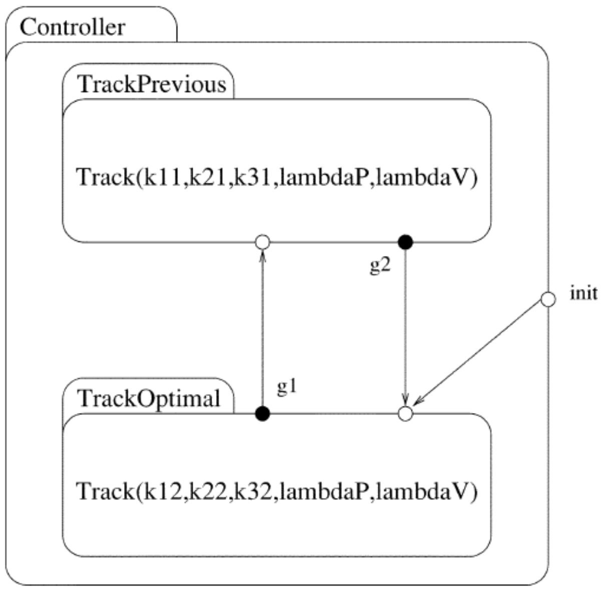

Controller
TrackPrevious
Track(k11,k21,k31,lambdaP,lambdaV)
g2
init
g1
TrackOptimal
Track(k12,k22,k32,lambdaP,lambdaV)

Figure 2.21: Example of a hierarchical hybrid automaton from CHARON [23]. The top-level mode *Controller* is composed of sub-modes *TrackPrevious* and *TrackOptimal*.

explicit synchronization events, CHARON provides a very useful and well developed toolset for graphical specification and simulation.

Dynamic Networks of Hybrid Automata (DNHA)

UC Berkeley and the California research group PATH [1] created the SHIFT language to specify and simulate Dynamic Networks of Hybrid Automata (DNHA)[2]. SHIFT and its simulator were used to create SmartAHS, which is a framework to compare alternative design concepts for automated highway systems [55, 77].

To properly model an automated highway SHIFT needed a dynamic network, one where the set of hybrid automata and their interconnections to each other can change during execution. This is necessary because any automated highway will naturally have cars entering and exiting during normal operation. This additional complexity made using previously existing hybrid automata specification languages and their simulators impossible since they assumed the world was in a static configuration, such as HIOA and CHARON still do[58].

The DNHA model is an extension of standard hybrid automata that aims to support dynamic reconfiguration of the network. The world is at any moment a set of hybrid automata. Each hybrid automaton is of a predefined type. Types are generic hybrid automata that can be used to create different *instances* with different initial conditions matching that type. In addition to the continuous and discrete variables that appear in standard hybrid automata, DNHA can have reference variables that point to other automata. These variables are typed and can only point to automata of that type. This is very similar to C++'s reference variables that allow one object to refer to another. These 'link' variables determine the *configuration* of the world. An automaton X containing a linked reference to another automaton Y has access to Y's events and output variables (DNHA also partitions the variables into state, input, and output variables). Through passing and re-assigning reference variables the configuration of the world and its behavior can change [59].

Using these reference variables DNHA can perform asynchronous communication. For example, a first automaton can set the value of one of its output variables. After that value has been set, a second linked automaton would then be able to read that value and react accordingly. The first automaton must finish setting the value before the second automaton can read it. Alternatively, DNHA can perform synchronous communication. Each transition can be labeled with events; automata can require that linked automata transition synchronously with them on the same event. This causes the linked automata to transition together (synchronously) based on the same event.

A DNHA network could be dynamic only due to changes in the linking configuration of a fixed set of automata, but this is extended by adding the capability to create/delete hybrid automata in the system. The new automata can be created from the predefined types. This is done by instantiating a new instance with parameters formed from expressions of variables already in the system. When an old automaton decides to create a new automaton instance, the new should be either passed a reference variable in the initialization by the old, so that the new can interact with someone, or the old automaton should record the new as a reference variable so that it can eventually let someone interact with the new. Otherwise, no hybrid automaton will ever have a link to the new automaton. (This is similar to a memory leak in object-oriented programming where memory is allocated and any reference to that memory is lost).

The dynamic creation/deletion of automata introduces uncertainty about how many automata will be present at one time. This mirrors the reality of the IVHS where the number of vehicles entering or exiting a stretch of highway over a given time cannot be predicted. To accommodate this uncertainty, DNHA allows variables to refer to sets of references. For example, these set-valued variables could contain references to *all cars*, or *all semis*, or *all emergency vehicles*. These set-valued variables can then be quantified over with an existential quantifier on the transition's guards. The transition's actions can then bind a specific car/semi/ambulance in a set-valued variable to a single-reference variable and then respond accordingly (e.g. if there is an ambulance, increase the speed of the ambulance and decrease the speeds of all cars).

A simplified IVHS example was presented in [56]. In the example a road-side monitor is informed when cars enter its section of road and it can optionally change some of the cars' speeds until they exit the monitored section of road. The example is shown in figure 2.22. Figure 2.23 shows that the 'Source' simulates cars that enter a section of road. Each 'Particle' (or car) has its own associated control and dynamics. Finally, the 'Monitor' can increase the speeds of cars in the second half of the section of road and can use the exit speeds of leaving particles to update the speeds of the other particles.

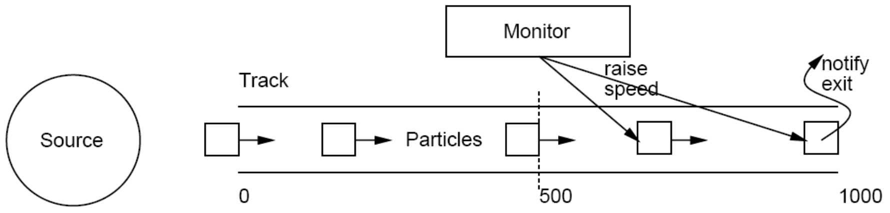


Figure 2.22: Simplified Automated Highway Example from [56].

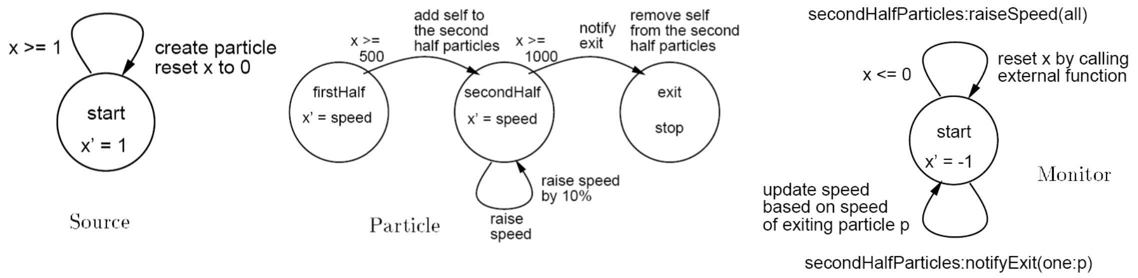

x >= 1
create particle reset x to 0
start
x' = 1
Source
add self to the second half particles
x >= 500
firstHalf
x' = speed
x >= 1000
notify exit
remove self from the second half particles
secondHalf
x' = speed
exit
stop
raise speed by 10%
raise speed
Particle
secondHalfParticles:raiseSpeed(all)
x <= 0
reset x by calling external function
start
x' = -1
Monitor
update speed based on speed of exiting particle p
secondHalfParticles:notifyExit(one:p)

Figure 2.23: SHIFT Automata for Simplified Automated Highway from [56].

DNHA's extensions to standard hybrid automata (reference variables, dynamic creation/deletion, set-valued variables, existential quantifiers) allow the details necessary for modeling dynamic reconfigurable networks of hybrid automata like an automated highway system. SHIFT's run-time system implements a compiler that translates an entire SHIFT DNHA world specification into an equivalent C format that is then compiled with a SHIFT library to produce the simulation executable. The different automata in the world are composed to form a world automaton which is then simulated at a single rate until a discrete event occurs that either changes the world or its configuration [56]. SHIFT's only available analysis tool is a thorough simulation of the reconfigurable world. For the complete SHIFT syntax and an informal discussion of the semantics see [57].

While DNHA and SHIFT were initially conceived for IVHS, they have also been used for describing the control of a mobile offshore base (MOB). This base would allow the military to assemble a floating air-strip, significantly larger than an aircraft carrier, out on the open sea from several individual barge-like vehicles [72, 73]. DNHA have also been used to describe a detailed, highly-coordinated, pre-planned, wave-based attack of UAVs on ground-based defenses [169, 39, 41, 40].

DNHA adds the additional features required to specify and simulate complex systems that include reconfiguration of the network. These features are not found in other hybrid automata formalisms, but have been proposed to be added to CHARON to create a Reconfigurable CHARON (R-CHARON) in [109].

Conclusions

Hybrid automata provide models that could include both the discrete and continuous behaviors of UAVs. The non-linear dynamics of UAVs make timed, multi-rate, and rectangular automata less than ideal for describing the position and orientation dynamics. These restricted models could however be used to verify that all on-board processes can execute at the correct rates based on their interdependent communication requirements. The complex dynamics of a UAV can be described with the more general hybrid automata models, but the analysis and verification cannot be automated and executed on-line. Even the approximation techniques require significant computation.

While there is still value in using hybrid automata to specify and simulate individual and networks of UAVs, they may not be appropriate here for specifying network-level controllers. First, the lack of automated verification techniques would provide little on-line feedback to the human operator. Second, the model itself is somewhat complicated with the interactions between the continuous and discrete dynamics. Even with the graphical representation, presenting and understanding the how the continuous state is expected to behave based on the stated continuous dynamics, guards, and resets would be difficult in an on-line setting. Understanding the interconnected behaviors of multiple coordinating hybrid automata would further complicate the situation. Additionally, asking a human operator to craft hybrid automata supervisors in an on-line manner would likely lead to systems that are both unstable and violate safety requirements since neither of these can be checked on-line.

Hybrid automata models can be extremely useful in the off-line detailed analysis and verification of UAVs. They allow a more complete perspective on the entire system which is quite useful for designers. However, this completeness is inappropriate and not needed for on-line network-level control by non-experts.

2.4 Petri Nets

Petri nets are a well established alternative to finite state machines. They were introduced in the 1960s by Carl Petri to add asynchronous communication to automata representing physical computing machines [149]. Like finite state machines, Petri nets are discrete-event models that have equivalent graphical and textual descriptions. Unlike FSMs, Petri nets have a distributed state representation that is beneficial to describing and understanding concurrency.

This distributed representation was quickly leveraged to describe flexible manufacturing systems [180, 65]. Once industrial engineers became accustomed to the techniques, they applied them not just to a company's physical assembly lines but also to the workflow processes of the front office employees [164, 176]. This allowed engineers to verify that the assembly line would not deadlock waiting for parts and that the front office would not deadlock waiting for completed paperwork. More recently this has been extended to multi-business processes thanks to the internet and web services (e.g. BPEL [119, 147]). This allows multiple vendors to communicate and coordinate their complex interdependent processes in an automated on-line manner.

2.4.1 Models

Petri nets have been augmented in various ways, but the original and most common Petri net concept refers to Place-Transition nets (P/T nets) [47, 137].

Definition 14 *A Petri net (P/T net) can be modeled as a tuple* $PN = (P, T, F, W, M_0)$ *where:*

- P *is the finite set of places,*
- T *is the finite set of transitions with* $P \cap T = \varnothing$ *and* $P \cup T \neq \varnothing$*,*
- $F \subseteq (P \times T) \cup (T \times P)$ *is the flow relation (arcs connecting places to transitions),*
- $W : F \rightarrow \mathcal{N}$ *is the weight of each arc in* F*,*
- M_0*:* $P \rightarrow \mathcal{N}$ *is the initial marking of tokens for each place in* P*,*

the Petri net's structure is $N = (P, T, F, W)$*.*

Some Petri net properties can be evaluated based on the Petri net's structure, independent of the initial marking. The marking represents the Petri net's state and is what evolves during execution.

Definition 15 *The marking* M *of a Petri net is the number of tokens in each place,* $M : P \rightarrow \mathcal{N}$*.*

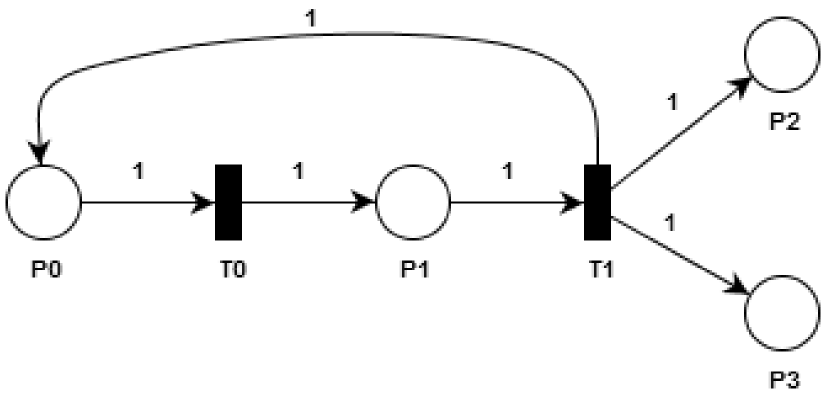


Figure 2.24: Petri net structure from example 7.

Each place $p_i \in P$ is graphically represented as a circle, shown in figure 2.24. The transitions $t_i \in T$ are graphically represented as bars. The flow relation F is graphically presented as directed arcs connecting places to transitions or transitions to places. The weights W annotate the arcs to indicate how many tokens are moved by each arc.

Example 7 *Figure 2.24 is the graphical equivalent to the Petri net structure* $N = (P, T, F, W)$ *where:*

- $P = \{p_0, p_1, p_2, p_3\}$,
- $T = \{t_0, t_1\}$,
- $F = \{(p_0, t_0), (t_0, p_1), (p_1, t_1), (t_1, p_0), (t_1, p_2), (t_1, p_3)\}$,
- $W : F \to 1$.

The distributed nature of the transitions and places make it important to know which places connect to which transitions and how. While this information is contained in the flow relation F, the pre-set and post-set operations help simplify this discussion.

Definition 16 *The pre-set of x is:* $^\bullet x = \{y \in P \cup T \mid (y, x) \in F\}$.

Definition 17 *The post-set of x is:* $x^\bullet = \{z \in P \cup T \mid (x, z) \in F\}$.

The pre-set of any transition ${}^\bullet t$ is the set of all places that have directed arcs from the place to transition t. The pre-set of any place ${}^\bullet p$ is the set of all transitions that have directed arcs from the transition to place p. The post-set of any transition $t^\bullet$ is the set of all places that have directed arcs from t to the place. The post-set of any place $p^\bullet$ is the set of all transitions that have directed arcs from p to the transition.

Example 7 and figure 2.24 have ${}^\bullet t_0 = \{p_0\}$, ${}^\bullet t_1 = \{p_1\}$, $t_0^\bullet = \{p_1\}$, $t_1^\bullet = \{p_0, p_2, p_3\}$ for the transitions. The places have pre-sets and post-sets as well which should be obvious by inspection.

While the Petri net structure N can be structurally analyzed without any marking (discussed in section 2.4.3), the model cannot execute without an initial state M_0. Figure 2.25 shows a Petri net behavior for N that results from an initial marking $M_0 = [M_0(p_0), M_0(p_1), M_0(p_2), M_0(p_3)] = [1, 0, 0, 0]$, where vector notation is adopted in the obvious order. The marking is graphically represented by drawing the tokens as filled circles inside the places. If the marking of a place is 0 there will be no tokens drawn. If the marking of a place is n there will be n individual tokens drawn.

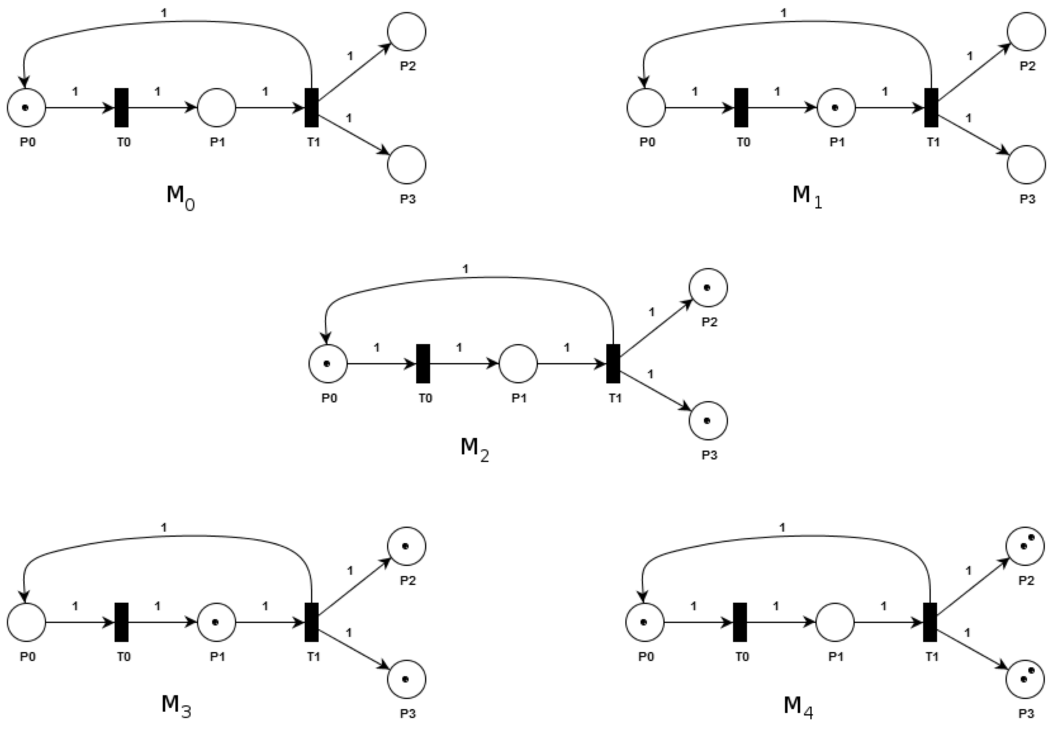


Figure 2.25: Example Petri net behavior (trajectory) from example 7.
$M_0 = [1, 0, 0, 0]$; $M_1 = [0, 1, 0, 0]$; $M_2 = [1, 0, 1, 1]$; $M_3 = [0, 1, 1, 1]$; $M_4 = [1, 0, 2, 2]$

Petri nets execute by transitions ’firing’ and changing the marking appropriately. A transition must be enabled for it to fire.

Definition 18 *A transition t_i is enabled if:* $\forall p_j \in ({}^\bullet t_i) \mid M(p_j) \geq W(p_j, t_i)$.

For a transition to be enabled, all arcs pointing to the transition must come from places who are marked with as many or more tokens than that arc's weight requires. A transition being enabled does not guarantee that it will ever fire; being enabled is a necessary but not sufficient condition to guarantee firing. When a transition t_i is enabled and does fire, it then removes tokens from ${}^\bullet t_i$ and adds tokens to $t_i^\bullet$ according to the directed arc's weights.

Definition 19 *A transition t_i firing causes the marking to change:* $M_i \xrightarrow{t_i} M_{i+1}$. *where* $M_{i+1}(p_j) = M_i(p_j) - W(p_j, t_i) + W(t_i, p_j)$.

Definition 20 *A firing sequence is a sequence of transitions:* $\sigma = t_1 t_2 t_3 ....$

Definition 21 *A state sequence is a sequence of markings* $M = M_0 M_1 M_2 ....$

Definition 22 *A firing sequence* $\sigma = t_1 t_2 t_3 ... t_k$ *is a behavior of a Petri net* PN *if:*

- *there exists a state sequence* $M = M_0 M_1 M_2 ... M_k$,
- M_0 *is the initial marking of* PN,
- $\forall i \in [0, k-1] \mid M_i \xrightarrow{t_{i+1}} M_{i+1}$,

σ *being a behavior of* PN *can be written compactly as* $M_0 \xrightarrow{\sigma} M_k$.

Figure 2.25 shows the Petri net behavior generated by the firing sequence $\sigma = t_0 t_1 t_0 t_1$ causing the state sequence $M = M_0 M_1 M_2 M_3 M_4$.

Petri nets can be easily extended to labeled Petri nets by adding an alphabet of events Σ and a labeling function l.

Definition 23 *A labeled Petri net can be modeled as a tuple* $PN = (P, T, F, W, M_0, \Sigma, l)$ *where:*

- Σ *is the alphabet of events,*
- l*:* $T \to \Sigma$ *is the labeling function assigning events to transitions,*

everything else is identical to P/T nets.

Labeled Petri nets behave identical to P/T nets but have additional information associated to each transition. The net's firing sequence, along with the labeling function, define strings in the language of the labeled Petri net.

Definition 24 *A trace/string/word of a labeled Petri net is the mapping a behavior's firing sequence to an equal length sequence of events via the labeling function* l.

The language generated by the labeled Petri net is the set of all strings mapped from any behavior.

There are several special subclasses of P/T nets that further restrict the structure. A net is called 'ordinary' if all of the arcs have a weight of 1.

Definition 25 *A state machine is an ordinary Petri net where each transition has exactly one incoming and one outgoing arc (* $\forall t \in T \mid |{}^{\bullet}t| = |t^{\bullet}| = 1$ *).*

As the name suggests, a state machine Petri net is similar to a FSM. Every FSM can be converted into a language-equivalent state machine Petri net and vice versa. This means that Petri nets can represent every regular language (first represent the language as a FSM, then convert it to an equivalent state machine Petri net). There are non-state machine Petri nets that cannot be represented as regular languages. In fact, general P/T nets represent their own class of context-sensitive languages. If 'inhibitor arcs' are added to the model the expressiveness increases to that of Turing machines, but also makes many properties undecidable [137].

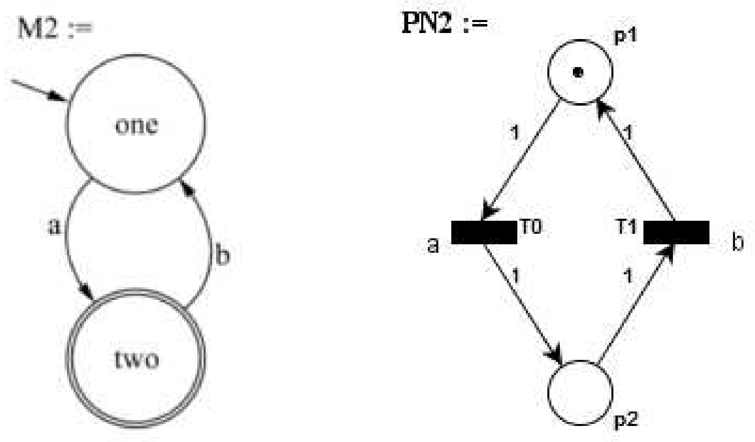


Figure 2.26: Language-equivalent FSM $M2$ and state machine Petri net $PN2$.

Similar to the one incoming/outgoing restriction on transitions for state machines, marked graphs have a one incoming/outgoing arc restriction on each place.

Definition 26 *A marked graph is an ordinary Petri net where each place has exactly one incoming and one outgoing arc (* $\forall p \in P \mid |{}^{\bullet}p| = |p^{\bullet}| = 1$ *).*

State machines are typically used to represent the internal behavior of individual processes. In figure 2.27 the state machine can execute either $T0$ or $T2$, but once it makes a choice it evolves to another state still marked by a single token. Since the

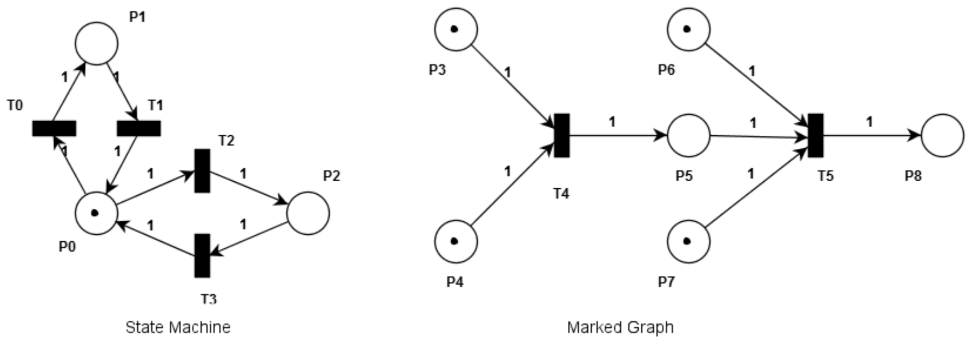


Figure 2.27: Example State Machine and Marked Graph.

number of arcs into and out of a transition are always 1, the number of tokens stays fixed, representing that the number of active processes is a constant.

Alternatively, marked graphs are used to represent the synchronization of multiple processes. Figure 2.27 shows that $P3$ and $P4$ both have to be ready for $T4$ to occur. Once $T4$ occurs $P5$ is ready and along with $P6$ and $P7$ enables $T5$ to occur. The places in the marked graphs always have one incoming/outgoing arc representing that these processes have no choice about what happens next. This is often used to model assembly lines where multiple parts are required at the same time to create a new assembly. These parts do not have a choice as to how they progress along the line. Once they are deposited in a bin, they remain there until the next event in the manufacturing process occurs to move them further along the assembly line.

State machines and marked graphs are both subclasses of free-choice nets.

Definition 27 *A free-choice net is an ordinary Petri net where all outgoing arcs from a place are either unique outgoing arcs from that place or are unique incoming arcs to the destination transition (* $\forall p \in P \mid\ |p^\bullet| \leq 1 \text{ or } {}^\bullet(p^\bullet) = \{p\}$ *).*

Free-choice nets are themselves a subclass of extended-free-choice nets, which are a subclass of asymmetric choice nets. As the model class becomes more restricted from general P/T nets towards marked graphs or state machines, the number of provable properties increases and the analysis algorithms become more efficient. This suggests that engineers should use the most restricted modeling formalism possible to gain as much provability and efficiency as possible.

Just as P/T nets and their more restricted subclasses can be labeled or not, they can also be given capacity constraints. These constraints prevent transitions from ‘over filling’ any place with tokens. These finite capacity nets can be transformed into language-equivalent P/T nets. The procedure outlined in [137] adds a set of complementary places and arcs that enforce the capacity bounds. After the transformation, the P/T net can be analyzed by standard P/T net techniques with no regard to the capacity constraints.

Definition 28 *A finite capacity Petri net can be modeled as a tuple* $PN = (P, T, F, W, M_0, K)$ *where:*

- K*:* $P \to \mathcal{N}$ *is the capacity (bound) of each place,*

everything else is identical to P/T nets.

The previous types of Petri nets were all subclasses or minor additions to the original concept. There have also been several extensions that drastically change the expressiveness and detail of the models. Coloured Petri nets (CP nets) resulted from an observation that certain structures were often repeated in large Petri net models [98]. The many identical structures represented different processes with the same behavior. For example, consider an automobile assembly line that assembles cars and trucks. While the hoods for the cars and trucks are physically different and cannot be interchanged, the assembly processes may be identical. CP nets allow the two hood assembly processes to be merged into a single hood assembly process where the tokens are additionally assigned colors. The colors are simply a typing mechanism to allow a car hood token to be distinguished from a truck hood token. These colors then allow the CP net to appropriately route the hoods after assembly. This would allow the car and truck assembly processes to share a common hood assembly process, then diverge back onto individual tracks.

An important property of the original CP nets is that they assumed a finite color set. This allowed the CP net to be transformed into a much larger P/T net. This unfolded P/T net can be analyzed for properties like structural invariants [96]. This unfolding is to be expected since CP nets' main use is to fold large P/T nets into more compact representations that are easier for engineers to understand [97]. The folded CP nets are easier for humans to understand while the unfolded P/T nets are easier for automated analysis.

The idea of attaching additional information to the tokens did not stop with finite color sets. The colors became more and more complicated tuples of information until arbitrary object oriented structures were added [99]. These High-Level Petri nets now treat tokens as typed objects with possibly infinite state spaces [112]. This greatly reduces the amount of applicable automated analysis. This is similar to how the addition of continuous variables make hybrid systems significantly more difficult to analyze than FSMs. Some simple High-Level net properties can be analyzed under strict limiting assumptions.

There have also been timed [66, 181] and hybrid [189] extensions to Petri nets. Much like timed and hybrid automata, these extensions complicate and reduce automated analysis in exchange for additional expressiveness.

2.4.2 Compositional Modeling

Petri nets are somewhat different than process algebras, FSMs, and hybrid automata in how their networks are composed. A network of FSMs is formed by specifying a set of individual FSMs and which composition operators to use. An equivalent FSM which models the entire network is the end result. The specification of the composition operator indirectly determines how the events are combined, thus indirectly affecting the resulting network model. Combining subnets into an overall Petri net requires a more direct specification of interaction.

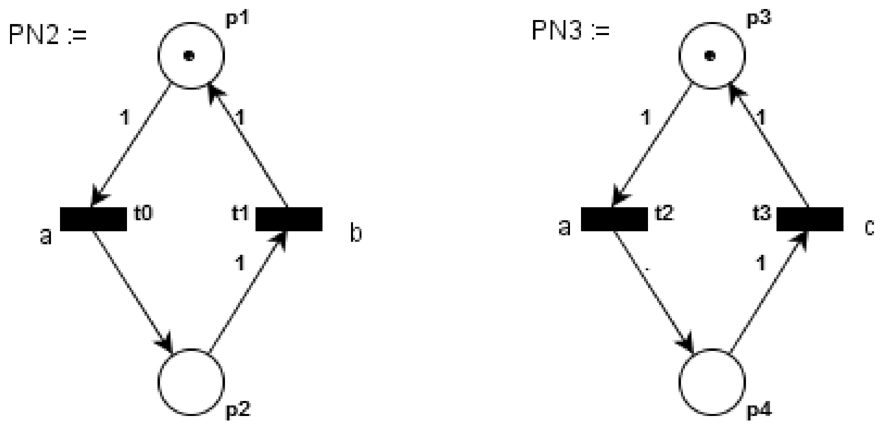


Figure 2.28: Petri nets $PN2$ and $PN3$ model components of a larger network.

For example, figure 2.28 shows state machines $PN2$ and $PN3$ which represent component subsystems of a larger network. These machines are the Petri net equivalent to FSMs $M2$ and $M3$ in figure 2.6.

The individual component Petri net models can be most easily combined by taking the union of the nets to create a network of two entirely independent interleaved components. Figure 2.29 shows the resulting net $PN4$ compared to the interleaving of $M2$ and $M3$. The combination of FSMs creates a FSM resulting in a state that is graphically represented as a single location. The combination of Petri nets creates a Petri net which results in a state that is graphically distributed throughout the net. This allows the behavior of $PN2$ to be viewed in isolation of $PN3$. With extremely large systems this distributed state representation makes it easy to focus on local interactions without worrying about the entire network's behavior.

The Petri net composition method of transition fusion takes several transitions and fuses them into one transition. All arcs connected to any of the previous transitions end up connected to the resulting transition. Figure 2.30 illustrates how transition $t1$ in $PN2$ can be fused with $t2$ in $PN3$ to create $t4$ in $PN5$. It is obvious that both the Petri net and the parallel composition FSM produce the same behavior. Transition fusion is how synchronous communication is represented in Petri nets. The resulting

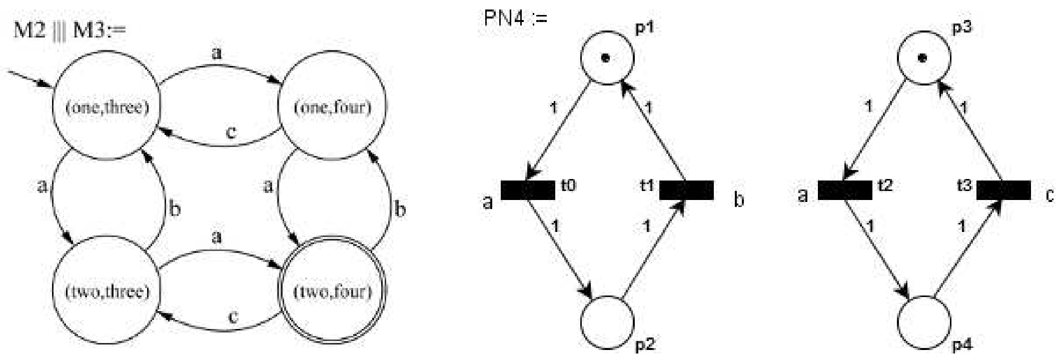


Figure 2.29: The Petri net $PN4$ is language-equivalent to $M2$ interleaved with $M3$.

FSM was created automatically by the specification of the parallel composition operation, while the Petri net's composition was manually specified. While this requires more involvement by the developer, it also allows the developer to customize each communication instead of assuming that all communications are either synchronous or asynchronous[138, 118]. Figure 2.31 shows how asynchronous communication can be added with place and arc addition. The 'send' event of transition $t1$ occurs before the 'receive' event of transition $t2$. Large networks composed from individual components may utilize both synchronous and asynchronous communication where appropriate.

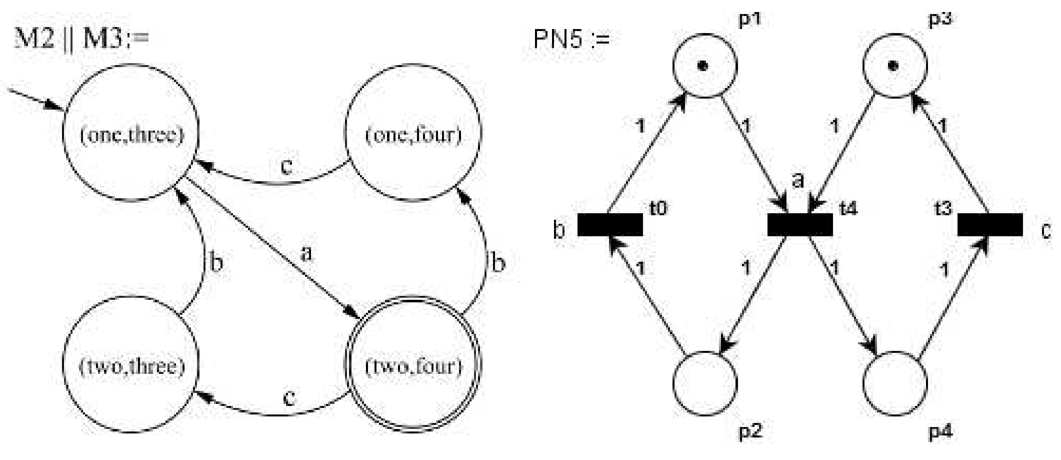


Figure 2.30: The Petri net $PN5$ uses transition fusion which is similar to the parallel composition of $M2$ and $M3$.

In many situations the subnets must share resources (e.g. assembly lines). This is modeled by another form of composition: place fusion [65, 180]. Place fusion takes a set of places and merges them together. Any arc connected to the original places is connected to the new fused place. For example, consider an assembly line where two

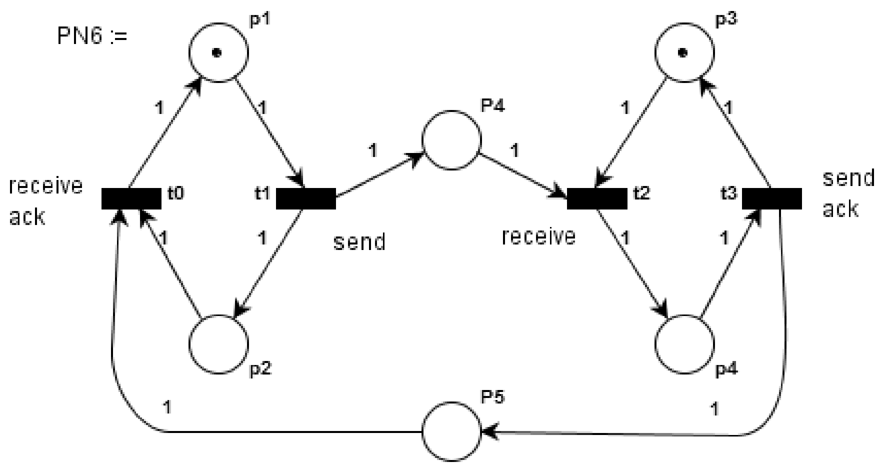


Figure 2.31: Asynchronous communication in a Petri net.

workers (call them Bill and Ted) must share a welder, figure 2.32. Sometimes Bill doesn't need the welder (represented by a token in $p1$ of $PN7$). When Bill does need the welder he takes it and uses it (represented by moving a token to $p2$ of $PN7$) until he doesn't need it and puts it back. Ted works the same way, represented by $PN8$. $PN9$ shows a model where they share a single welder, represented by the fusion of $p1$ and $p3$ to form $p5$.

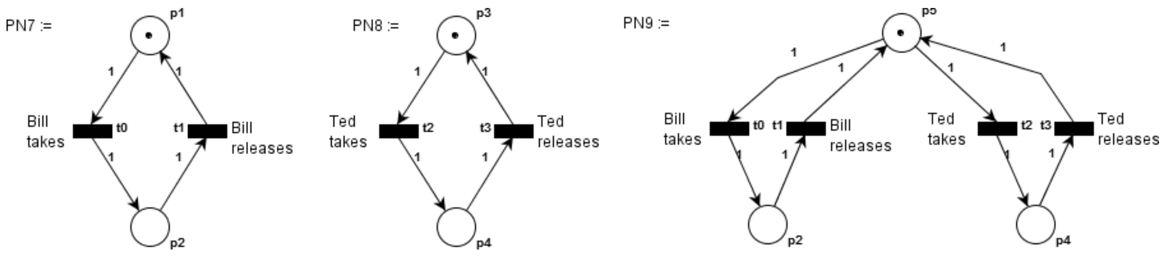


Figure 2.32: Representing limited resources with place fusion.

While the composition of components is essential, the ability to create hierarchical models is important as well. Petri nets do have a notion of abstraction and refinement for both transitions and places. This refinement replaces simple places or transitions with expanded subnets. A transition can be refined (expanded) by a subnet that has a single source transition and a single sink transition. Here, upon firing the sink transition the subnet should be empty. This corresponds to one incoming transition fire resulting in only one outgoing transition fire. This represents breaking down an event at a high-level of abstraction into its sequence of sub-events at a lower-level of abstraction. An example is shown in figure 2.33. Similarly places can be refined by a subnet that has a single source place and a single sink place, figure 2.34 [176].

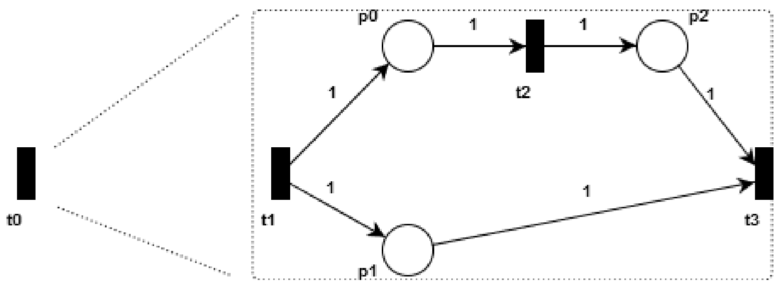


Figure 2.33: Hierarchy through transition refinement. Replace $t0$ with an appropriate subnet.

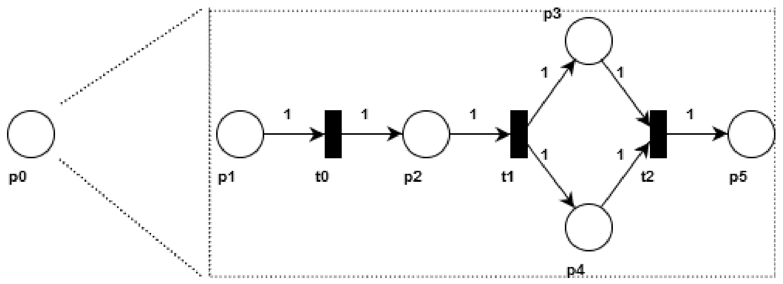


Figure 2.34: Hierarchy through place refinement. Replace $p0$ with an appropriate subnet.

These are substitution-based forms of hierarchy and will always result in a single-level Petri net. There has been one proposal for a non-substitution hierarchy where the tokens of a Petri net are themselves a Petri net. These 'Petri nets in Petri nets' or PN^2 never seemed to gain any traction beyond their author [87, 88].

2.4.3 Analysis

Like the previous modeling formalisms, Petri nets can be analyzed to better understand the network and its potential behaviors. Since most of the formalisms discussed can be transformed into P/T nets or are subclasses of P/T nets, the properties to be analyzed will be discussed in the P/T net context.

It is convenient to use matrix equations to describe the behavior of Petri nets. The behavior can be written in a form similar to the algebraic difference equations of discrete-time control.

$$M_k = M_{k-1} + A^T u_k \tag{2.10}$$

$M_k \in \mathcal{N}^m$ is the marking (state) of the net with m being the number of places in the net. M_0 is the initial marking.

$u_k \in \mathcal{N}^n$ is the firing vector with n being the number of transitions in the net. The firing vector represents which transition occurs and is all 0's except for the transition firing, which is represented as a 1 (e.g. $[0, 0, 1, 0]^T$ is a firing vector where the third of four transitions is firing).

$A \in \mathcal{Z}^{n \times m}$ is the incidence matrix of the Petri net and represents the affect of firing a transition on the marking of the net. When $A^T u_k$ is added to M_{k-1} it represents the tokens removed from places by the firing as well as the tokens added to places by the firing. The incidence matrix A contains information about the arcs between places and transitions as well as their weights. If there is no arc between a place p_j and a transition t_i assume $W(p_j, t_i) = W(t_i, p_j) = 0$.

$$A(i, j) = W(t_i, p_j) - W(p_j, t_i) \quad \forall t_i \in T,\ \forall p_j \in P \tag{2.11}$$

From equation 2.10 one can easily see that by recursive substitution:

$$M_k = M_{k-1} + A^T u_k = M_{k-2} + A^T u_{k-1} + A^T u_k = M_{k-3} + A^T u_{k-2} + A^T u_{k-1} + A^T u_k ... \tag{2.12}$$

In order for a specific marking M_d to be reached there must be a firing sequence such that:

$$M_d = M_0 + A^T \sum_{k=0}^{d} u_k \tag{2.13}$$

Meaning that there must exist a firing sequence that starts from the initial state M_0 and follows the firing vector sequence (that matches the firing sequence) $u_1 u_2 u_3 ...$ to bring the net to M_d. If $x = \sum_{k=0}^{d} u_k$ is the firing count vector, where the order is not recorded, a necessary but not sufficient condition for reachability is that $M_d - M_0 = A^T x$. So given M_0, M_d, A if a non-negative solution x does not exist the Petri net is guaranteed to never reach M_d. Unfortunately, if a non-negative solution x does exist the reachability of M_d is not guaranteed.

Some properties can be evaluated independent of the initial marking. These structural properties depend only on the incidence matrix A. These include T-Invariants which correspond to potential transition firing sequences that move the Petri net from a marking M through intermediate markings and back to M. These T-Invariants allow for infinite cycles.

Definition 29 *A T-Invariant is a solution x to $A^T x = 0$.*
It allows $M_d - M_0 = 0 = A^T x$.

P-Invariants are another structural property. These invariants describe a conservation of tokens through all possible reachable markings [137].

Definition 30 *A P-Invariant is a solution y to $Ay = 0$.*

While determining potential cycles of transitions and conservations of tokens are important, determining if a given marking M_d is guaranteed to be reachable is also very important. A matrix algebra method to prove that M_d was not reachable was already mentioned, but if the state space of the Petri net is finite, all potentially reachable markings can be computed [128]. This can be used to create a labeled transition system on which model verification techniques can be used. This can show if M_d is or is not reachable as well as prove any other LTL/CTL properties. An easy way to guarantee a finite state space is to add capacity constraints to the Petri net. If the system is actually unbounded, the algorithms such as mentioned in [137] will not terminate.

This potential non-termination problem lead to the development of the coverability graph as an alternative to the reachability graph. Each marking in a reachability graph is a vector in $\mathcal{N}^m$. The coverability graph extends this to vectors of $\{\mathcal{N} \cup \omega\}^m$. The symbol ω is used to represent 'unbounded' or 'infinite'. If ω appears in a marking of the coverability graph it indicates that there is an unbounded place (e.g. the coverability marking $[5, 4, \omega, 0]^T$ indicates that place 3 can become unbounded through some cyclic behavior). The coverability algorithm in [137] is guaranteed to terminate on bounded and unbounded Petri nets. Checking for the absence of ω in the coverability graph proves that the Petri net is bounded. If the net is bounded the coverability graph is exactly the same as the reachability graph and can be analyzed by the above reachability algorithms.

One common term that is slightly different in Petri net literature is 'safe'. In Petri nets 'safe' refers to a net that has markings consisting of only 0's and 1's. This property can be checked by evaluating all states in the coverability graph. If only 0's and 1's appear, the net is considered safe. This evolved from networks of electronic buffers and registers where safety refers to never over-flowing. The register can only have 0 or 1 values stored at a time.

The coverability graph also proves is any transition is 'dead'. If a transition t_i does not appear anywhere in the coverability graph, it will never occur and is thus termed 'dead'. If from every reachable marking a transition t_j can always later occur that transition is termed 'live'. If every transition can always potentially occur at each reachable marking the Petri net itself is referred to as 'live'.

Petri net analysis algorithms have been implemented by several toolkits of varying sophistication. The Low-Level Petri Net Analyzer (LoLA) provides a set of C++ libraries for creating and analyzing the reachability and coverability graphs of P/T nets [7]. The Integrated Net Analyzer (INA) does the same thing for P/T nets as well as Colored nets (which can be transformed into P/T nets) based on the programming language MODULA-2 [6]. The graphical tool TINA does the same, but provides a convenient user interface as well as on-the-fly model checking to hasten the analysis [15]. The most convenient and thorough tool is the Platform Independent Petri net Editor [12]. This provides all of the analysis tools for P/T nets as well as some for deterministically timed Petri nets. It allows the simulation of the system as well as analysis and exporting figures (such as all the Petri nets shown in this dissertation).

2.4.4 Applications

Petri nets have been utilized in many applications, but the area where they initially gained prominence was in modeling assembly lines such as flexible manufacturing systems [65]. The tokens in places were used to represent parts in bins or scarce resources needed for assembly such as robots and workers. The transitions represented small incremental steps in the assembly processes. For any one subassembly certain parts would be required as well as certain tools and workers, these requirements were encoded in the arcs connecting the net. These models could then be evaluated for deadlock to guarantee that the line would never freeze [188]. The bounds on the places could also be computed to appropriately size bins or buffers. Finally, if the physical processes' behavior needed to be modified because it had undesirable behaviors, additional supervisory controllers could be added. Language-based supervision similar to the Ramadge-Wonham methods proved somewhat difficult because of the distributed nature of the state [91]. However, state-based supervision by P-Invariants provided a capable method of adding control. This would add control places and control arcs in order to create additional P-Invariants that would modify the net's behavior to be acceptable [94, 95, 165].

These assembly line techniques were then adapted to apply to the workflow of offices [164]. Instead of dealing with physical parts, the model would specify the dependencies and movement of forms and paperwork. Again, the system would be analyzed for deadlock as well as bounds. Supervisory control would be added to achieve the desired network behavior [176].

As the internet grew in size, companies began to depend on it heavily for communication and coordination with other suppliers and customers. This sparked the expansion of workflow modeling from intra-company to inter-company. The Business Process Execution Language (BPEL) allows different companies to communicate and coordinate automatically using web services [119, 147]. BPEL can be given a Petri net interpretation and analyzed [62, 178, 86]. This allows a company to evaluate its entire workflow including its suppliers and customers to guarantee and implement provably correct automated interactions [143, 118].

Petri nets have also been used in traffic modeling. The places of the net represent physical segments of road and the number of tokens represent how many cars are on that road segment. These models typically include hybrid behaviors and so hybrid Petri nets are often used to simulate the city-grid behavior [104, 177].

Petri nets have also recently been used to model embedded networked systems [53, 102]. The Petri net model is used primarily as an alternative to communicating FSMs. The benefits of using Petri nets instead of FSMs included the previously mentioned ability to represent both synchronous and asynchronous communications.

Finally, and most directly related to this dissertation, Petri nets have seen very limited use in describing mobile robotic systems. In [106] they were used to describe the communication protocol behavior between a fixed set of agents. They have also been used to give an interpretation of the Predator/Prey problem as a Petri net [48]. The places corresponded to a discretized environment. The Predator and Prey were colored tokens in the places corresponding to their physical locations. A Fuzzy Petri net was used in [121] to make the strike/no-strike decision for aggressive military UAVs.

In [145] a control architecture for an AUV inspecting a dam used Petri nets to specify which control processes to execute in which order during a predefined mission. In this way the sensor processes were not run until the appropriate time, then turned off. Other processes controlling thrusters and actuators were turned on and off likewise. Single AUV control was also addressed in [141, 142]. They addressed more general single AUV missions by similarly turning on/off processes from a predefined library. This style of single vehicle control was also utilized for a fully autonomous and predefined behavior UAV in [30].

While the above research deals with mobile robotics, it deals with individual robots and not communicating networks of mobile robots. Additionally, the behaviors are predetermined and there is not the separation of plant and controller that is sought for network-level control. There is no concept of a task since each robot's behavior is explicitly predetermined.

The research most similar to the concept of network-level controllers is the very recent development of Petri Net Plans [144, 190]. This work intends to enable the multi-robot teams from the RoboCup competitions to perform complex soccer maneuvers like passing a soccer ball. Each robot's individual behavior is defined as a Petri net. The Petri nets are then combined and synchronized to form a centralized plan. The plan is then broken up and distributed to the individual robots. This research does coordinate multiple robots, but the roles of each robot appear to be fixed and the network's behavior predetermined.

Chapter 3

Network-Level Controllers

An individual UAV's behavior should be specified as a plant model that allows all potentially desirable behaviors to be executed. A supervisory controller (task) composed with the plant then determines the exact behavior of the individual UAV during execution. In order to change the UAV's execution behavior one modifies this task instead of changing the UAV's compiled plant behavior. In this way an interpreter can be created that accepts task specifications. These interpreted specifications can be easily changed on-line instead of requiring the fixed-behavior UAV to stop or shutdown during recompilation of its behavior.

With the separation of task and UAV, a network of several UAVs can be combined with several task specifications. The process of pairing tasks to UAVs can become automated task-allocation based on optimization criteria. The network's human operator must then only control which tasks are inserted into the network and when they are inserted.

Control of the network can be further simplified and automated by a Petri net-based network-level controller. Petri nets are utilized because the fluctuation in the number of tokens mirrors the fluctuation in the number of tasks. Additionally, the Petri net formalism allows an intuitive graphical representation of the tasks in the system as well as the concurrency relationships and orderings between tasks.

3.1 UAVs and Tasks

Two fundamental concepts for the system are UAVs and tasks. Both tasks and UAVs have some 'structure' and 'information' associated with them. This data is partitioned into types, definitions, and states. This partitioning can be understood through an analogy with spring-mass-damper systems, figure 3.1.

The types of systems shown in figure 3.1 all behave differently. Models can be created that predict the behavior of each type. While for these simple cases the models are based on similar modeling techniques, they result in different dynamic interactions

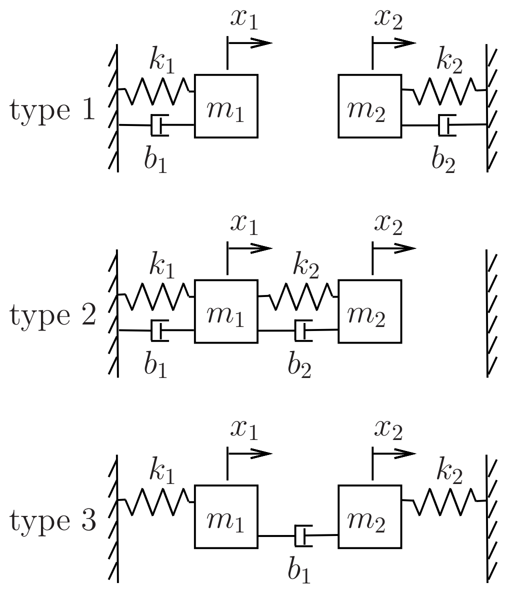


Figure 3.1: Types of spring-mass-damper systems.

of springs, masses, and dampers. It is impossible to predict the behavior of a type 1 system with algorithms coded for a type 2 system. The type of a spring-mass-damper system and its models are created during the modeling and development of that system. The type of an individual system is fixed during any execution because it does not make sense to physically switch from a type 3 to a type 2 system.

The algorithms and models for a specific type of system can be created utilizing variables abstracting the real system's parameter values. This allows the algorithms to be re-used repeatedly. In order to model a specific concrete system, these parameters must be provided (e.g. a 'real' type 1 system needs values for $m_1, b_1, k_1, m_2, b_2, k_2$). The system type determines which parameters need to be filled by an appropriate spring-mass-damper definition. Different definitions of the same type behave 'similarly' but not identically. The parameters that define a system are assumed fixed during execution like the system's type.

The spring-mass-damper state variables are changing continuously during execution under the constraints imposed by the system's type and definition parameters. These values are expected to flow automatically starting at an appropriate initial state.

Consider a network made of several spring-mass-damper systems that enter and exit during an execution. Each spring-mass-damper system has a state, definition, and type. If the set of types in the network is considered fixed then the developers can create the models and analysis algorithms needed for the execution of every potential system in the network. If arbitrary types of new systems were allowed, new models and analysis algorithms would need to be generated and incorporated into the network on-the-fly. This may be interesting, but would add significant complications not directly related to the content of this dissertation. When a new definition of an expected type is created and communicated, the network then understands and can predict how systems based on this definition will behave. That definition only needs to be communicated once. When the spring-mass-damper systems enter, the state variables correspond to proper initial states. Their behavior is produced by their type of system under the parameters listed in their definition. The type, definition, state partitioning is meant as a simple separation of model structure, model parameters, and model state.

3.1.1 UAVs

Definition 31 *Different UAV types have different data and model structures as well as fixed information that is specified by the UAV's developers during development. A specific UAV type ut is from the domain of all UAV types, $ut \in \mathcal{UT}$. The UAV types can be superscripted for identification (e.g. ut^a, ut^b, ut^c) or grouped in sets ($UT \in \mathcal{P}(\mathcal{UT})$ such as $UT^1 = \{ut^a, ut^b, ut^c\}$).*

In general lower-case letters will be used for individuals. Upper-case letters will be used for sets. Superscripts will be used for identification. Subscripts will be used for representing the evolution in time (as in $x_k \rightarrow x_{k+1}$).

The set of UAV types can change when a new type of UAV is added to the system or when an old type of UAV is retired from the system. An individual UAV type can change when physical or software upgrades significantly alter the capabilities of the UAV (e.g. an electric motor is replaced with a gas engine that increases the range/speed/duration of the UAV, or a new type of behavioral algorithm is developed to maneuver the UAV). These version changes occur off-line and are executed by the system developers. During any execution of the network, the set of UAV types is expected to remain fixed.

For C3UV the UAV types include the MLB Bat IV and Sig Rascal. The type specifies which parameters need to be filled to create a UAV definition. Chapter 7 will give details about exactly what information is in each C3UV UAV type.

The network also has a set of UAV definitions that are based on the existing set of UAV types. The UAV definitions fill in the parameters required for any UAV type.

Definition 32 *Different UAV definitions have different data and model parameters that are specified during start-up. A specific UAV definition ud is from the domain of all UAV definitions, $ud \in \mathcal{UD}$. The UAV definitions can be superscripted for identification (e.g. ud^a, ud^b, ud^c) or grouped in sets ($UD \in \mathcal{P}(\mathcal{UD})$ such as $UD^1 = \{ud^a, ud^b, ud^c\}$).*

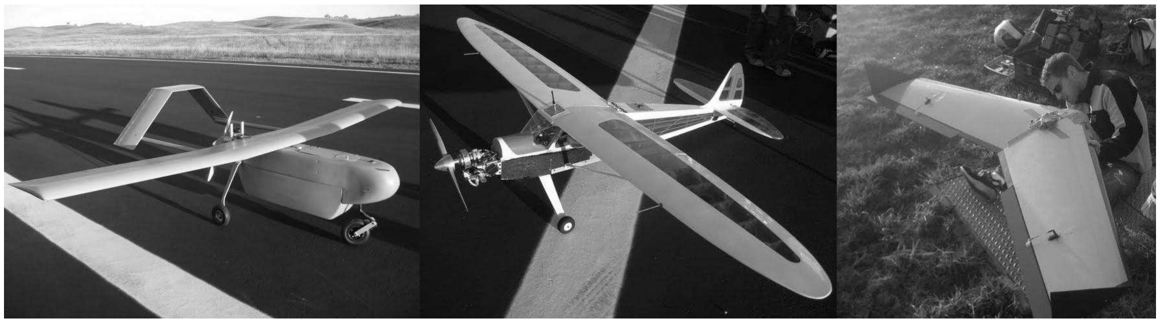


Figure 3.2: Types of aircraft in the C3UV fleet: left-MLB Bat IV, middle-Sig Rascal, right-Zagi

The UAV definition contains information such as the IP address, call sign, or sensors on board. Two Rascal UAVs are of the same type but would have different definitions with different IP addresses, call signs, etc. This information is assumed fixed during execution, but could possibly be changed on request of the ground crew or human operator (e.g. the encryption key could be changed on request and the UAV definition appropriately updated, communicated, and synchronized). As new UAVs are turned on and enter the system, they disseminate their definitions which are assumed fixed. If parameters are changed the new definitions are communicated. As old UAVs are turned off and removed from the system their definitions are removed.

Definition 33 $uavType : \mathcal{UD} \rightarrow \mathcal{UT}$*, there exists a function mapping each UAV definition to an existing UAV type.*

The network has a set of UAV states that are based on existing UAV definitions. The UAV states contain the state information that is updated automatically by the UAV. State variables may be continuous-time, discrete-time, or discrete-event and are automatically changed by the UAV's execution. This could include information about position, orientation, etc.

Definition 34 *Different UAV states have different state information that changes during execution. A specific UAV state us is from the domain of all UAV states,* $us \in \mathcal{US}$*. The UAV states can be superscripted for identification (e.g.* us^a, us^b, us^c*) or grouped in sets (*$US \in \mathcal{P}(\mathcal{US})$ *such as* $US^1 = \{us^a, us^b, us^c\}$*).*

Each physical UAV has a UAV state, a UAV definition, and a UAV type. The state is based on the definition which is based on the type.

Definition 35 $uavDef : \mathcal{US} \rightarrow \mathcal{UD}$*, there exists a function mapping each UAV state to an existing UAV definition.*

This partitioning of information illustrates how some information is specified by the developers at design time, some information is specified by the human operator or ground crew at run time, and some information is specified by the UAVs during execution. This partitioning allows subsets of information about the UAVs to be communicated as necessary. Since the UAV types do not change during execution, they can be assumed static and are not transmitted. Since the set of UAV definitions do change during execution, they can be transmitted by the UAVs during initialization and then whenever changes are made. Since the UAV states change continuously, their information can be updated, read, transmitted, and synchronized automatically by the UAVs. This allows the network to perform less communication than methods like the original C3UV Mission State Estimate (MSE) which lumped all of this information into one table that was transmitted periodically [162]. There, the type and definition information was repeated in each communication packet despite the content not changing.

3.1.2 Tasks

Tasks are similarly partitioned into task types, task definitions, and task states.

Definition 36 *Different task types have different data and model structures as well as fixed information that is specified by the task's developers during development. A specific task type tt is from the domain of all task types, $tt \in \mathcal{TT}$. The task types can be superscripted for identification (e.g. tt^a, tt^b, tt^c) or grouped in sets ($TT \in \mathcal{P}(\mathcal{TT})$ such as $TT^1 = \{tt^a, tt^b, tt^c\}$).*

The set of task types is also assumed fixed during any execution. Developers can create additional types of tasks for the system off-line. They must then update any tools such as GUIs and web servers so that these task types can be properly displayed, communicated, or used to create new task definitions. Chapter 7 contains details about the C3UV task types.

Definition 37 *Different task definitions have different data and model parameters that are initially specified during execution. A specific task definition td is from the domain of all task definitions, $td \in \mathcal{TD}$. The task definitions can be superscripted for identification (e.g. td^a, td^b, td^c) or grouped in sets ($TD \in \mathcal{P}(\mathcal{TD})$ such as $TD^1 = \{td^a, td^b, td^c\}$).*

The set of task definitions changes during execution. The human operator loads or creates new task definitions for the network. These definitions are assumed fixed unless they are manually modified by the human operator. Each task definition is based on a task type and specifies the values of the parameters for that task type. For example, two task definitions can both be of type 'visit point', while having different values for the latitudes and longitudes of the points to be visited. There can be multiple task definitions based on the same task type.

Definition 38 *taskType* $: \mathcal{TD} \rightarrow \mathcal{TT}$*, there exists a function mapping each task definition to an existing task type.*

The task's state information is changed automatically by the UAVs.

Definition 39 *Different task states have different state information that changes during execution. A specific task state* ts *is from the domain of all task states,* $ts \in \mathcal{TS}$*. The task states can be superscripted for identification (e.g.* ts^a, ts^b, ts^c*) or grouped in sets (*$TS \in \mathcal{P}(\mathcal{TS})$ *such as* $TS^1 = \{ts^a, ts^b, ts^c\}$*).*

Every task has a state, definition, and type. This information is partitioned based on how often it changes as well as who changes it. Just like for UAVs, it is possible to have task types in the system and have no task definitions based on them. It is also possible to have task definitions in the system that have no task states based on them.

Definition 40 *taskDef* $: \mathcal{TS} \rightarrow \mathcal{TD}$*, there exists a function mapping each task state to an existing task definition.*

The fixed sets of task and UAV types detail what 'kinds' of tasks and UAVs the network is equipped to handle. The definitions show possible parameter values for the tasks and UAVs that may be in the network. The states give detailed state information about the tasks and UAVs that currently do exist in the network.

Many of the different elements mentioned above will evolve in time. A logical time will be represented with subscripts. For example: $ts_k \rightarrow ts_{k+1}$ would represent a task state evolving from its value at time k to a new value at $k+1$ during an instantaneous event.

In the future the following projection operation will become convenient.

Definition 41 *The subset of task states matching a specific task definition:*
$TS|_{td} = \{ts^j \in TS \mid taskDef(ts^j) = td\}$.

Additionally, the network-level controller will assume the ability to read if a task is completed from its state.

Definition 42 *done* $: \mathcal{TS} \rightarrow \{true, false\}$*, task states can be checked for completion.*

3.2 The Concept of Network-Level Control

The separation of UAV and task (plant and controller) enables the UAV's behavior to be drastically altered during execution to adapt for unforseen circumstances by changing which task the UAV is executing. It also enables collaboration through task allocation. There can be many different methods for automatically assigning tasks to UAVs. Some are centralized while others are decentralized. They can be based on heuristics or optimization criteria. The exact details of these methods are not discussed in this dissertation, but an appropriate algorithm for assigning tasks to UAVs is assumed. The C3UV implementation operates with the sub-optimal distributed algorithm developed by Mark Godwin [162, 75].

Network-level control adds one layer of automation on top of the assumed task allocation layer. It monitors the network's progress on existing tasks and then creates new tasks while removing old completed tasks, figure 3.3. In this manner it is controlling the network through the direct manipulation of the set of tasks in the network. The network-level controller only indirectly affects the UAVs' behaviors, through task allocation.

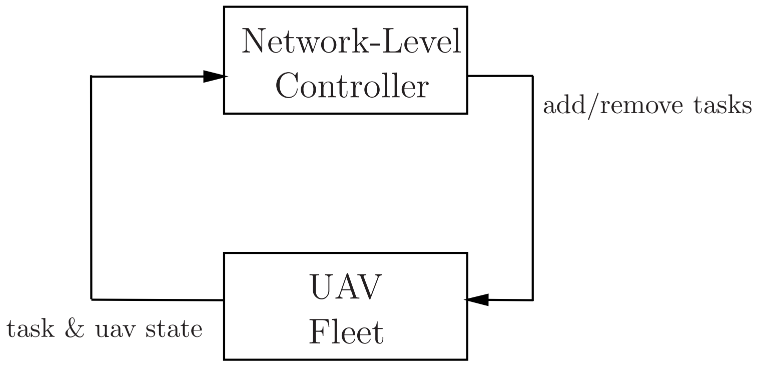


Figure 3.3: Interaction of the network-level controller and the UAVs through addition/removal of tasks.

Chapter 2 discussed several potential models for the network-level controllers. As was previously mentioned, Petri nets were chosen as the fundamental basis for network-level controllers. They provide a simple rigorous model that can be analyzed with available algorithms as well as graphically displayed in an intuitive manner. They provide just enough information in a network-focused perspective. While other more detailed models could have been chosen (e.g. Dynamic Networks of Hybrid Automata), Petri nets provide the desired functionality as well as much needed simplicity.

Task states are indirectly represented using the Petri net's tokens. Every task in the network has a state and every state is represented graphically by a token. By observing the tokens in the net, the human operator indirectly observes the number of tasks in the network.

Each task state has a corresponding task definition. These definitions are represented by the Petri net's places. If there are several tasks (each having a unique state) based on the same task definition, there will be several tokens in the associated place. Every task definition in the network will have a corresponding place.

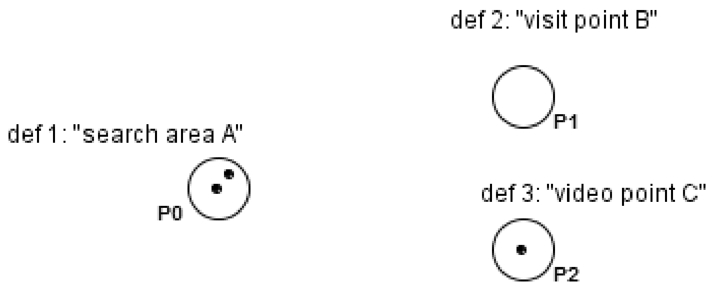


Figure 3.4: Network-level controller's graphical representation of task states (tokens) and task definitions (places).

Figure 3.4 shows an example where there are 3 tasks in the network. There are 2 tasks based on task definition 1, which is labeled "search area A". There is 1 task based on task definition 3, which is labeled "video point C". There are no tasks based on task definition 2, which is labeled "visit point B". Every task state in the network is shown with a token. Every task definition in the network is shown with a place.

The graphical representation of the network-level controller shows the existence of task definitions and states but does not display the detailed contents of either. Each task definition has parameters that define what "search area A" or "visit point B" really means. The human operator who created these definitions likely knows that they are, but the detailed information is not immediately graphically available. Similarly, the task states contain state information that can be rapidly changing. This information is also not immediately presented. The network-level controller associates a place to a task definition and a token to a task state. Showing all of the textual content of the definitions and states would clutter up the otherwise simple graphical description. A GUI implementation should allow a human operator to click on a place and open up the details for that associated definition. Likewise, one should be able to click on tokens and open up the details for the associated task states. In this way the details are easily accessible without always cluttering the display.

Figure 3.4 contained no arcs or transitions so it would be expected not to change its marking (not to automatically add/remove tasks). Figure 3.5 adds arcs and tran-

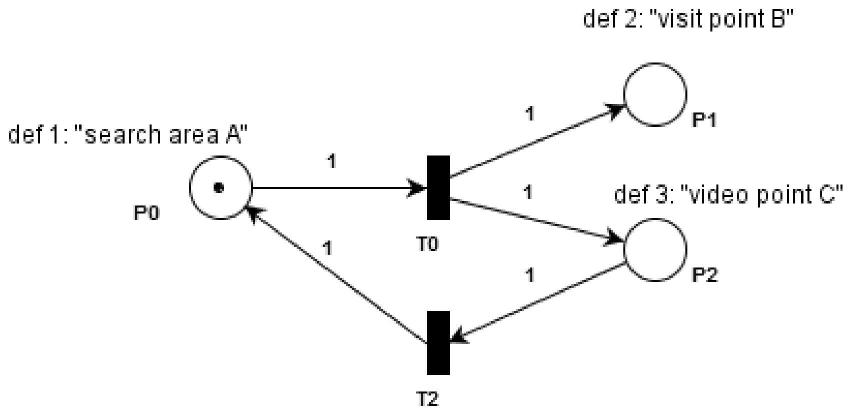


Figure 3.5: A simple network-level controller's graphical representation.

sitions. The anticipated behavior is that a "search area A" task is done followed by "visit point B" and "video point C" in parallel. Once "video point C" is done a new "search area A" is started, regardless of if "visit point B" is finished, figure 3.6. The firing of the transitions removes old tasks/tokens and creates new tasks/tokens. One additional semantic detail is the assumption that a task must be completed before it can be removed by a transition firing. Without this detail the network-level controller could create then immediately remove new tasks, never allowing the UAVs the time necessary to complete the task. The completion assumption prevents this from occurring.

Previously the completion assumption was represented with a boolean condition on each arc that would by default take the value "on done" [120]. The other options included "on to do", "on assigned", and "on canceled". Based on feedback received from the Navy SEALs during experiments at Camp Roberts these were rethought and eventually removed. The "on to do" condition would identify a task that had just been created and immediately remove it to create another task. These phantom tasks would just appear and disappear with no real affect on the system's behavior. The "on assigned" condition would identify a task that had just been assigned to be executed by a UAV and then remove it to create another task. This also had no apparent use and only added confusion and frustration. The "on canceled" condition seemed somewhat useful, but the canceling of tasks is done by the human operator and is now directly incorporated into the runtime patching used to modify the network-level controller. Runtime patching will be discussed in chapter 5.

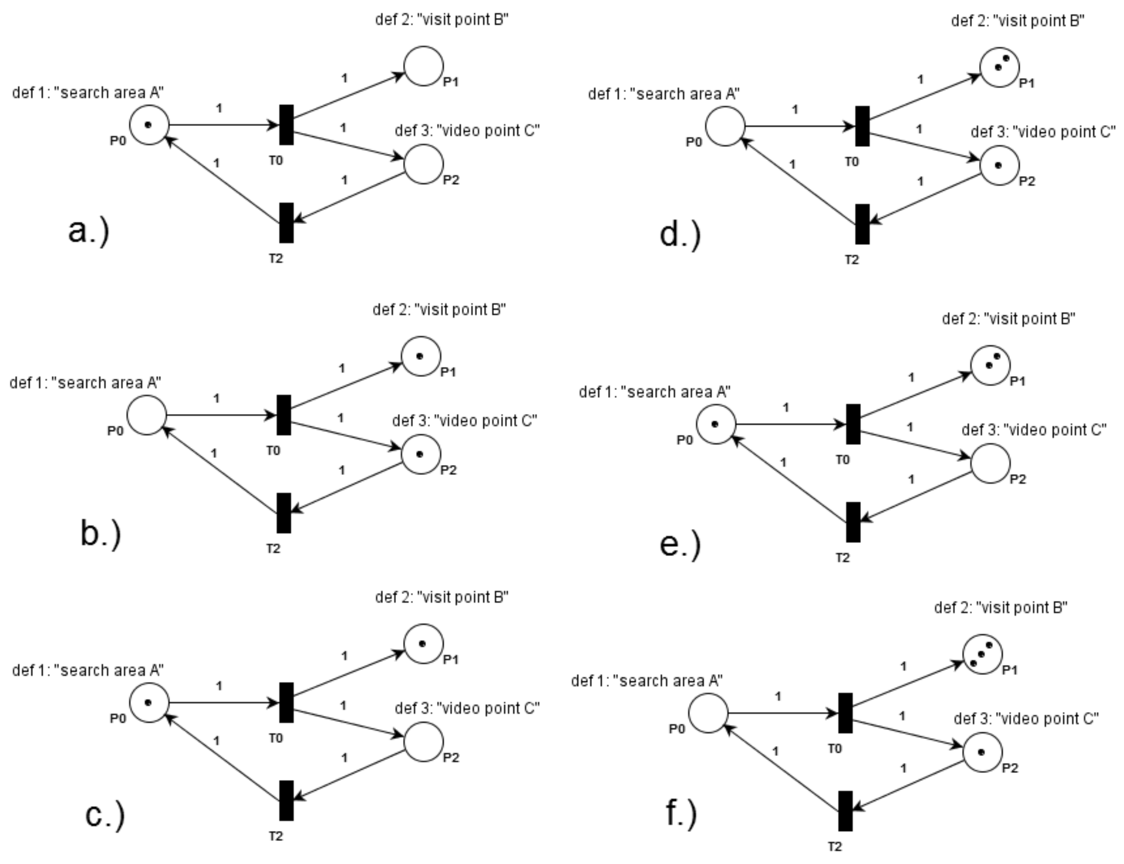


Figure 3.6: Execution of a simple network-level controller. Start as a.), when the task in definition 1 is completed fire transition $T0$ to become b).

3.3 Syntax of Network-Level Controllers

Definition 43 *A network-level controller can be modeled as a tuple* $NLC = (P, T, F, W, K, M_0, def, TD, TS_0)$ *where:*

- P *is the finite set of places,*
- T *is the finite set of transitions,*
- $F \subseteq (P \times T) \cup (T \times P)$ *is the flow relation (arcs connecting places to transitions),*
- $W : F \rightarrow \mathcal{N}$ *is the weight of each arc in* F*,*
- K*:* $P \rightarrow \mathcal{N}_+$ *is the capacity constraint for each place,*
- M_0*:* $P \rightarrow \mathcal{N}$ *is the initial marking of tokens for each place in* P*,*

- *def: $P \to TD \cup \{null\}$ associates each place to a single task definition or a special indicator null of no task definition,*
- *TD is the set of task definitions in the network,*
- *TS_0 is the initial set of task states in the network,*

the network-level controller's structure is $SNLC = (P, T, F, W, K, def, TD)$.

The first part of the network-level controller is the structure for a capacity constrained Petri net, (P, T, F, W, K). The network-level controller's structure is augmented with a set of task definitions TD and a function def associating each place in P to either a task definition or a special null symbol *null*. The places associated to *null* can be used for purely logical control conveniences and any tokens created in these places will not have actual tasks associated. This is similar to the virtual complimentary places added to assembly lines to create new place invariants. The null places will be demonstrated later in an example. As in standard Petri net structures, the network-level control structure is independent of its state/marking.

The initial marking M_0 is the number of tokens in the Petri net initially. The initial task state set TS_0 includes all task states initially in the network. The network-level controller's state is the marking and set of task states, (M_k, TS_k); these are the values expected to change during execution. The structure of the network-level controller does not automatically evolve.

A network-level controller should always be well formed. This simply means there should be no dangling arcs or places without capacity constraints.

Definition 44 *A network-level controller NLC is well formed if its structure and initial state satisfy all the criteria in definition 43.*

Additionally, it should also be fully representative. Every task definition should be represented by exactly one place. It would not be useful to have 'hidden' task definitions and 'hidden' task states that the human operator could not see in the network-level controller, and therefore could not control.

Definition 45 *A network-level controller NLC is fully representative if its structure satisfies:*

$$\forall td \in TD.(\exists! p \in P.def(p) = td).$$

A NLC's state is task-token consistent if: for each place that has a task definition, the number of tokens in that place matches the number of task states of the correct definition.

Definition 46 *A NLC state (M_k, TS_k) is task-token consistent if the following holds:*

$$\forall p \in P.[def(p) \neq null] \Rightarrow [M_k(p) = cardinality(TS_k|_{def(p)})]$$

Task-token consistency allows a human operator to observe the number and locations of tokens in places and understand the number of task states and which definitions they are based upon.

3.4 Semantics of Network-Level Controllers

For a NLC transition to be enabled the standard Petri net enabling conditions from definition 18 must be satisfied (here referred to as token enabled). As was stated earlier a "transition being enabled does not guarantee that it will ever fire; being enabled is a necessary but not sufficient condition to guarantee firing". NLC imposes the additional task completion requirement before a NLC firing can occur.

Definition 47 *A NLC transition $t \in T$ is completion enabled if:*
$\forall p \in ({}^\bullet t).[def(p) \neq null] \Rightarrow [\exists! ts^1, ...ts^{W(p,t)} \in TS_k.taskDef(ts^1) = def(p) \wedge done(ts^1) \wedge ... \wedge taskDef(ts^{W(p,t)}) = def(p) \wedge done(ts^{W(p,t)})]$.

Completion enabled requires that each place with an associated definition have a sufficient number of tasks with the correct definition that are also finished. When checking for a completion enabled transition, the task states bound are recorded in the set TS^-. If the transition ends up being both token and completion enabled it can be fired and the set of tasks TS^- will be removed from the network along with the correct number of tokens.

Similarly a set of new tasks, TS^+, must be created when a NLC transition t fires. TS^+ is the union of subsets $TS^{t,p}$, created for each outgoing arc from the transition t . If the destination of the arc is a place that is associated to a null definition, no actual tasks will be added. If the destination of the arc is a place with a non-null definition the correct number of new tasks will be added based on the arc weight.

Definition 48 *The set of tasks created after a transition $t \in T$ fires is:*
$TS^+ = \bigcup_{p \in (t^\bullet)} TS^{t,p}$

- *if $def(p) = null$ then $TS^{t,p} = \{\}$,*
- *if $def(p) \neq null$ then $TS^{t,p} = \{ts^1, ...ts^{W(t,p)}\}$*
  where $taskDef(ts^m) = def(p)$ and $done(ts^m) = false$ for each new task state.

When a NLC transition is token and completion enabled it can fire causing the tokens and the set of task states to change. The tokens change exactly as in a standard Petri net. The task states TS^- are removed and TS^+ are added.

Definition 49 *A transition $t^i \in T$ firing causes a state change $(M_k, TS_k) \xrightarrow{firing(t^i)} (M_{k+1}, TS_{k+1})$ where:*

- $M_{k+1}(p^j) = M_k(p^j) - W(p^j, t^i) + W(t^i, p^j)$,
- $TS_{k+1} = TS_k - TS^- + TS^+$.

In addition to transitions firing, the environment (the UAVs) can complete the tasks.

Assumption 1 *It is assumed that all tasks initially start not done, $done(ts) = false$ (see definition 48). Once a UAV completes the task, it sets the task to done, $done(ts) = done$. It is also assumed that this task is never set back to not done.*

From the perspective of the network-level controller the environment can cause an instantaneous change in TS_k.

Definition 50 *The environment completing a task $ts^i \in TS_k$ causes a state change $(M_k, TS_k) \xrightarrow{complete(ts^i)} (M_{k+1}, TS_{k+1})$ where:*

- $M_{k+1} = M_k$ *the marking is not affected,*
- $TS_{k+1} = TS_k - ts^i_k + ts^i_{k+1}$ *only the task state ts^i is updated, this update satisfies $done(ts^i_k) = false \land done(ts^i_{k+1}) = true$.*

The behavior of any network-level controller involves the environment completing tasks and the network-level controller firing transitions to remove completed tasks and insert new tasks.

Definition 51 *A string of states, $(M_0, TS_0), (M_1, TS_1), (M_2, TS_2)...$, is a behavior of a network-level controller NLC if:*

- (M_0, TS_0) *matches the initial state in the definition of NLC,*
- $\forall k.[(M_k, TS_k) \xrightarrow{firing(t^a)} (M_{k+1}, TS_{k+1})] \lor [(M_k, TS_k) \xrightarrow{complete(ts^b)} (M_{k+1}, TS_{k+1})]$, *the successor state is caused by a firing of a token enabled and completion enabled transition t^a, or by the completion of a task ts^b by the environment.*

The semantics of a network-level controller is given in terms of its behaviors.

Definition 52 *The language of the network-level controller NLC, $\mathcal{L}(NLC)$, is the set of all potential behaviors of NLC.*

3.5 Example Network-Level Controller

An example network-level controller will help illustrate the previous developments. Assume the network is designed to have two types of tasks: tt^1 is a visit point type of task, tt^2 is a visit line type of task. This set of task types remains fixed during execution. The human operator uses these types to create three task definitions: td^1 looks to “visit point A”, td^2 looks to “visit point B”, td^3 looks to “visit line C-D”. Obviously $taskType(td^1) = tt^1$, $taskType(td^2) = tt^1$, and $taskType(td^3) = tt^2$.

The definitions appear associated to places $P1, P2, P3$ in the network-level controller, figure 3.7. Additionally, the human operator created a place $P0$ without an associated task definition. This null place is used as a logical convenience that holds tokens associated to no task. Figure 3.7 shows a NLC that will “visit point A” in parallel with “visit point B”. When both of those tasks are completed by the UAVs, the network-level controller will create “visit line C-D” and remove the completed tasks. When “visit line C-D” is completed by the UAVs, the network-level controller will remove it and will repeat.

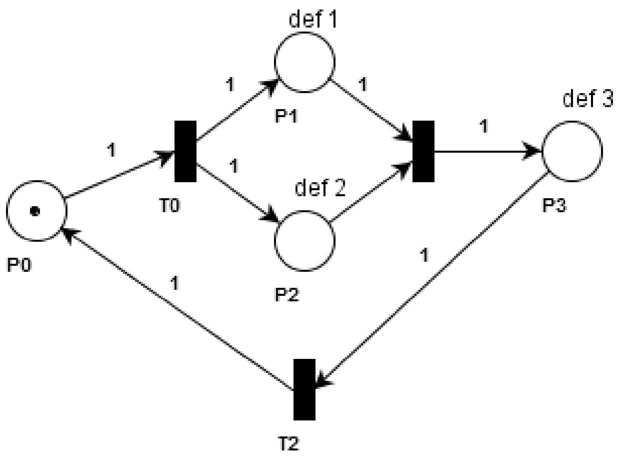
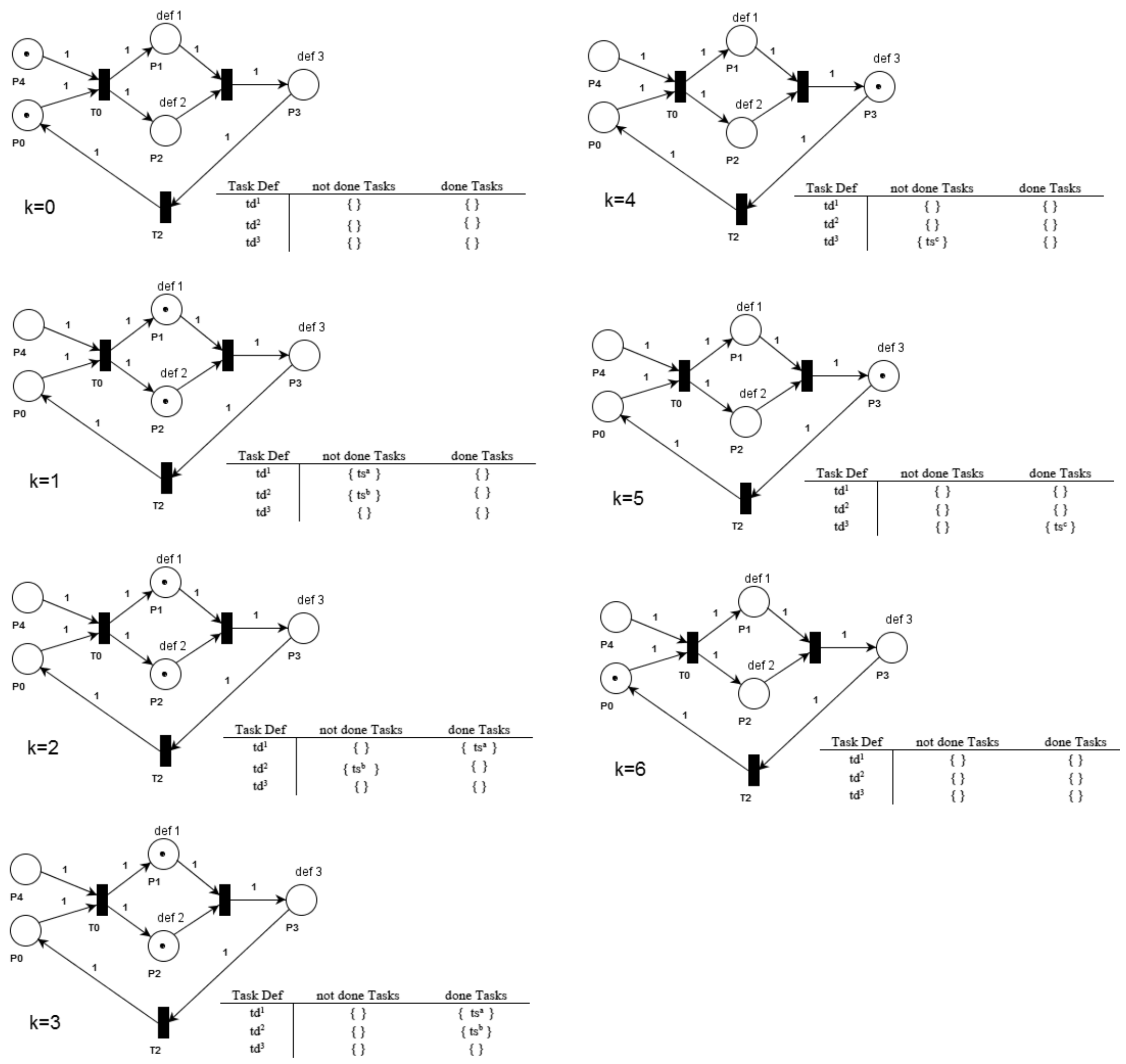


Figure 3.7: Second simple network-level controller example.

While this is a very useful and desirable behavior, it does not illustrate the benefits of the null places. If the human operator wanted to execute this loop only 1 time, a minor modification makes this constraint possible. At $k = 0$ in figure 3.8 the additional null place $P4$ has only 1 token. The tokens in $P4$ limit how many cycles can be executed. If there were x tokens, the cycle could execute up to x times. The null places give the ability to create additional logical constraints like limits and place invariants.

In figure 3.8 the NLC starts at $k = 0$ with 2 tokens in null places and no tasks



k=0

Task Def	not done Tasks	done Tasks
td^1	{ }	{ }
td^2	{ }	{ }
td^3	{ }	{ }

k=1

Task Def	not done Tasks	done Tasks
td^1	{ ts^a }	{ }
td^2	{ ts^b }	{ }
td^3	{ }	{ }

k=2

Task Def	not done Tasks	done Tasks
td^1	{ }	{ ts^a }
td^2	{ ts^b }	{ }
td^3	{ }	{ }

k=3

Task Def	not done Tasks	done Tasks
td^1	{ }	{ ts^a }
td^2	{ }	{ ts^b }
td^3	{ }	{ }

k=4

Task Def	not done Tasks	done Tasks
td^1	{ }	{ }
td^2	{ }	{ }
td^3	{ ts^c }	{ }

k=5

Task Def	not done Tasks	done Tasks
td^1	{ }	{ }
td^2	{ }	{ }
td^3	{ }	{ ts^c }

k=6

Task Def	not done Tasks	done Tasks
td^1	{ }	{ }
td^2	{ }	{ }
td^3	{ }	{ }

Figure 3.8: The behavior of the second simple network controller, but with an additional constraint of only 1 cycle iteration. The initial time is $k = 0$ and the NLC deadlocks at $k = 6$.

in the network. The transition t^0 fires causing these two tokens to be removed and 2 new tokens and 2 new tasks to be created at $k = 1$. The network completes task ts^a to move to $k = 2$ and then task ts^b to move to $k = 3$. From $k = 3$ the transition t^1 is both token and completion enabled and can fire, resulting in $k = 4$. The network completes ts^c to move to $k = 5$. Now t^2 is token and completion enabled and can fire to produce $k = 6$. At $k = 6$ the NLC deadlocks and is prevented from continuing the cyclic behavior. If there were more tokens placed in $P4$ the cycle would repeat.

Chapter 4

Properties of Network-Level Controllers

Chapter 3 described how a network-level controller is defined and how it operates. In order to understand if any given network-level controller will operate 'well', additional properties are needed. The first set of properties will be invariance properties. These properties will be true for all network-level controllers; they are based solely on the NLC semantics. The second set of properties will be analyzable properties. These will provide analysis results for each individual network-level controller to understand how the individual controller is expected to behave. These analyzable properties will be formed from the standard Petri net properties of section 2.4.3.

4.1 Invariance Properties

As the network-level controller evolves through transitions firing and the environment completing tasks, it is important that it remain well formed, fully representative, and task-token consistent. The invariance of these properties is important to guarantee that the controller remain a properly defined representation of all task definitions and task states in the network.

The structure of a network-level controller remains fixed during execution. This guarantees that if a NLC begins well formed and fully representative, it will remain well formed and fully representative. This invariance is obvious and expected. However, when runtime patching is added in chapter 5, the network-level controller's structure from definition 43 will effectively become part of the state and can be modified through runtime patching. It is then important to show that the runtime patches always produce well formed and fully representative network-level controllers.

4.1.1 Task-Token Consistency

Lemma 1 shows that for a single step task-token consistency is preserved for network-level controllers. Theorem 1 then shows that if the NLC starts task-token consistent, it will remain task-token consistent throughout execution.

Lemma 1 *If a network-level controller is in a task-token consistent state (M_k, TS_k), then every state (M_{k+1}, TS_{k+1}) immediately reachable is also task-token consistent.*

Proof: There are only two ways that (M_k, TS_k) can transition to become (M_{k+1}, TS_{k+1}), either the UAVs complete a task or the network-level controller fires a transition.

Case 1.) the environment completed a task ts^a
Changing ts^a from not done to done affects neither the number of task states nor the marking, thus this event cannot affect task-token consistency. Since (M_k, TS_k) is task-token consistent by assumption, the resulting state is also task-token consistent.

Case 2.) the network-level controller fired an enabled transition t^i
For this case to be true, all places p^j that do have an associated task definition ($def(p^j) \neq null$) must satisfy: $M_{k+1}(p^j) = cardinality(TS_{k+1}|_{def(p^j)})$.
Before the firing, $M_k(p^j) = cardinality(TS_k|_{def(p^j)}) = x$ by assumption.
According to definition 49, $W(p^j, t^i) = m$ tokens will be removed from place p^j. Also $W(t^i, p^j) = n$ tokens will be added. This means $M_{k+1}(p^j) = x - m + n$.
According to definition 47, there must be exactly $W(p^j, t^i) = m$ task states matching $def(p^j)$ in the set of task states to be removed, TS^-. Definition 48 shows that there are exactly $W(t^i, p^j) = n$ task states matching $def(p^j)$ to be added in set TS^+. The original number of task states matching definition $def(p^j)$ was x, then m were removed and n added. This produces $cardinality(TS_{k+1}|_{def(p^j)}) = x - m + n$.
Therefore, $M_{k+1}(p^j) = x - m + n = cardinality(TS_{k+1}|_{def(p^j)})$ and the resulting state is task-token consistent. $\square$

Theorem 1 *If a network-level controller starts in a task-token consistent initial state (M_0, TS_0), then it will remain task-token consistent through all reachable states (M_k, TS_k).*

Proof: The proof will proceed by an inductive argument on the length of the firing sequence required to reach the reachable state, (M_k, TS_k), where k is the length of the firing sequence.

Base Case $(k = 0)$: If the reachable state is the initial state (M_0, TS_0), it is task-token consistent by assumption.

Inductive Step ($k \neq 0$): Every reachable state can be reached through a firing sequence of some finite length k. The state (M_{k-1}, TS_{k-1}) in the behavioral sequence prior to (M_k, TS_k) is also reachable, but with a firing sequence of length $k-1$. By the inductive hypothesis (M_{k-1}, TS_{k-1}) must be task-token consistent. This with lemma 1 proves that (M_k, TS_k) is also task-token consistent. □

Assumption 2 *The network-level controller is well formed, fully representative, and has an initial state that is task-token consistent.*

Assumption 2 should be standard for all network-level controllers. Since a NLC satisfying assumption 2 starts task-token consistent, theorem 1 shows that it will also stay task-token consistent.

There can be many task-token consistent initial states. The most trivial of which is the empty state. It is obvious from definition 46 that the following proposition holds.

Proposition 1 *An empty state, one with no tokens or task states, is task-token consistent.*

It is possible that the NLC will eventually evolve into the empty state where no transitions are enabled (the NLC would become deadlocked). It is also possible that this state is used as an initial state. If the NLC started empty it would stay empty indefinitely, unless manual modifications were made to add tokens/tasks by runtime patching. Starting with a default empty state and showing that runtime patches always produce task-token consistent states from task-token consistent states will later allow the proof that the entire system, including manual modifications by the human operator, will always remain task-token consistent.

Corollary 1 *A NLC that starts at an empty state stays task-token consistent.*

Proof: By proposition 1 the empty state is task-token consistent. By theorem 1 the NLC will stay task-token consistent in all states reachable from this task-token consistent initial state, which happens to be empty. □

4.2 Analyzable Properties

The invariance of being well formed, fully representative, and task-token consistent is a result of the semantics defined for network-level controllers. The use of Petri nets within the modeling formalism allows additional properties to be proven about specific network-level controllers by analyzing the sub-Petri nets embedded within those specific network-level controllers. This allows a case-by-case analysis of different network-level controllers.

Definition 53 *The sub-Petri net for a network level controller can be extracted by* $SPN : NLC \to PN$ *where:* $NLC = (P, T, F, W, K, M_0, def, TD, TS_0)$ *and* $SPN(NLC) = (P, T, F, W, K, M_0)$ *is a standard capacity constrained Petri net.*

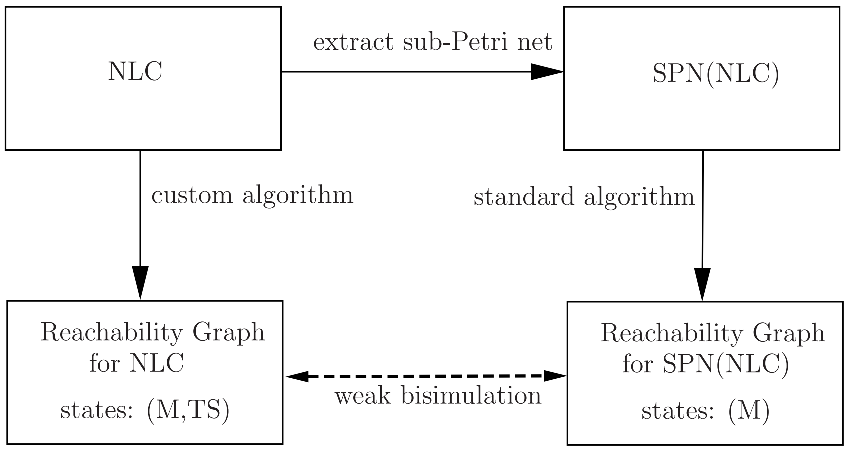


Figure 4.1: Analyze a specific network-level controller NLC by extracting the sub-Petri net, analyzing $SPN(NLC)$ with standard algorithms, then relating the results using a weak bisimulation relation.

Figure 4.1 shows that instead of developing a custom algorithm to determine the reachability graph for network-level controllers, standard Petri net algorithms can be applied to the sub-Petri net embedded within any network-level controller. The results of the sub-Petri net analysis can then be related to the network-level controller.

Since $SPN(NLC)$ is a capacity constrained Petri net, standard algorithms are guaranteed to compute all reachable states and determine properties like boundedness, liveness, and the potential for deadlock. In order for the results about $SPN(NLC)$ to be useful to understanding the parent NLC, there must be a well established relationship between the states of $SPN(NLC)$ and the states of NLC.

4.2.1 Weak Bisimulation

Petri nets and network-level controllers are both non-deterministic, making a bisimulation relation an obvious candidate for expressing a relationship between the two's states [131]. If a state of NLC is weakly bisimilar to a state of $SPN(NLC)$, written $(M_k, TS_k)\mathcal{B}(M_j)$, then sequences of observable events taken by NLC from state (M_k, TS_k) can be mirrored by $SPN(NLC)$ from state (M_j) resulting in states that are again weakly bisimilar. Conversely, sequences of observable events taken by

$SPN(NLC)$ from state (M_j) can be mirrored by NLC from state (M_k, TS_k) resulting in states that are again weakly bisimilar.

The proposed weak bisimulation relation $\mathcal{B}$ is: $(M_k, TS_k)\mathcal{B}(M_j)$ if $M_k = M_j$. This means that every state (M_j) of $SPN(NLC)$ is weakly bisimilar to the states (M_k, TS_k) of NLC that have the same marking.

Weak bisimilarity is used instead of strong bisimilarity because certain events occur only in NLC and not in $SPN(NLC)$. These events, the completions of tasks by UAVs, are undertaken by NLC but not $SPN(NLC)$. The network-level controller cares about tasks being completed, whereas the sub-Petri net has no notion of a task. These events are effectively unobservable to $SPN(NLC)$. Milner would replace these events with the symbol τ.

For the proofs below standard weak bisimulation notation will be used. Individual observable events (i.e. a transition firing) are represented as normal with $\xrightarrow{firing(t)}$. The unobservable events (i.e. a task being completed) are represented with $\xrightarrow{\tau}$. A series of zero or more unobservable events is represented as $\Longrightarrow$. $\overset{firing(t)}{\Longrightarrow}$ represents a sequence of unobservable tasks being completed by a single transition t firing, $\xrightarrow{\tau} \ldots \xrightarrow{\tau}\xrightarrow{firing(t)}$.

Lemma 2 *The relation $\mathcal{B}$ is a weak simulation.*

Proof: According to Milner's proposition 6.3 of [131], $\mathcal{B}$ is a weak simulation if, assuming $(M_k, TS_k)\mathcal{B}(M_j)$ holds, the following two cases hold.

Case 1.) if $(M_k, TS_k) \xrightarrow{\tau} (M_{k+1}, TS_{k+1})$ then there exists a (M_n) such that $(M_j) \Longrightarrow (M_n)$ and $(M_{k+1}, TS_{k+1})\mathcal{B}(M_n)$:
By assumption $M_k = M_j$. The τ events are UAVs completing tasks. According to definition 50 this does not change the marking of the NLC state. This means that when $(M_k, TS_k) \xrightarrow{\tau} (M_{k+1}, TS_{k+1})$ the marking does not change, $M_{k+1} = M_k$. Let the sought resultant $SPN(NLC)$ state be the initial state, $(M_j) = (M_n)$. This state can be achieved by allowing $\Longrightarrow$ to represent zero events. Since $M_{k+1} = M_k = M_j = M_n$, it is obvious that $(M_{k+1}, TS_{k+1})\mathcal{B}(M_n)$ holds.

Case 2.) if $(M_k, TS_k) \xrightarrow{firing(t)} (M_{k+1}, TS_{k+1})$ then there exists a (M_n) such that $(M_j) \overset{firing(t)}{\Longrightarrow} (M_n)$ and $(M_{k+1}, TS_{k+1})\mathcal{B}(M_n)$:
By assumption $M_k = M_j$. The semantics of network-level control guarantee that a transition t firing causes $M_{k+1}(p^i) = M_k(p^i) - W(p^i, t) + W(t, p^i)$. Since the network-level controller fired t, t was token enabled. This guarantees that at $(M_j) = (M_k)$ the sub-Petri net transition t is also enabled. Let the sought resultant $SPN(NLC)$ state (M_n) be produced by immediately firing t from (M_j), $(M_j) \xrightarrow{firing(t)} (M_n)$. The semantics of a Petri net also guarantee that $M_n(p^i) = M_j(p^i) - W(p^i, t) + W(t, p^i)$.

By assumption $M_k = M_j$, leaving $M_n(p^i) = M_k(p^i) - W(p^i, t) + W(t, p^i)$. The marking M_{k+1} of the NLC and M_n of the $SPN(NLC)$ are identical. This results in $(M_{k+1}, TS_{k+1})\mathcal{B}(M_n)$. $\square$

Lemma 3 *The converse of relation $\mathcal{B}$, noted $\mathcal{B}^-$, is also a weak simulation.*

Proof: According to Milner's proposition 6.3 of [131], $\mathcal{B}^-$ is a weak simulation if, assuming $(M_j)\mathcal{B}^-(M_k, TS_k)$ holds, the following two cases hold.

Case 1.) if $(M_j) \xrightarrow{\tau} (M_{j+1})$ then there exists a (M_n, TS_n) such that $(M_k, TS_k) \Longrightarrow (M_n, TS_n)$ and $(M_{j+1})\mathcal{B}^-(M_n, TS_n)$:
The Petri net $SPN(NLC)$ never experiences any unobservable events (completions of tasks). The Petri net only experiences transitions firing. This condition is vacuously true.

Case 2.) if $(M_j) \xrightarrow{firing(t)} (M_{j+1})$ then there exists a (M_n, TS_n) such that $(M_k, TS_k) \overset{firing(t)}{\Longrightarrow} (M_n, TS_n)$ and $(M_{j+1})\mathcal{B}^-(M_n, TS_n)$:
By assumption $M_j = M_k$. From the semantics of Petri nets, firing t causes $M_{j+1}(p^i) = M_j(p^i) - W(p^i, t) + W(t, p^i)$. Since t can fire, t is enabled. This implies that t is token enabled in NLC. In order for NLC to fire, t also needs to become completion enabled. Let (M_k, TS_k) experience only completion events until all tasks are completed becoming (M_m, TS_m). Completion of tasks does not change the marking, $M_m = M_k$. At (M_m, TS_m) transition t is both token and completion enabled. Let the state produced by firing t from (M_m, TS_m) be (M_n, TS_n):

$$(M_k, TS_k) \xrightarrow{\tau} \ldots \xrightarrow{\tau} (M_m, TS_m) \xrightarrow{firing(t)} (M_n, TS_n)$$

From the semantics of network-level controllers, $M_n(p^i) = M_m(p^i) - W(p^i, t) + W(t, p^i)$, but $M_m = M_k = M_j$ showing that, $M_n(p^i) = M_j(p^i) - W(p^i, t) + W(t, p^i)$. This proves that $M_{j+1} = M_n$, showing $(M_{j+1})\mathcal{B}^-(M_n, TS_n)$ because the markings are identical. $\square$

Theorem 2 *The relation $\mathcal{B}$ is a weak bisimulation between states of NLC and states of $SPN(NLC)$ where the markings match.*

Proof:
Lemma 2 shows that $\mathcal{B}$ is a weak simulation. Lemma 3 shows that the converse of $\mathcal{B}$ is also a weak simulation. This proves that $\mathcal{B}$ is a weak bisimulation. $\square$

Proposition 2 *The initial state of NLC and the initial state of its sub-Petri net $SPN(NLC)$ are weakly bisimilar, $(M_0, TS_0)\mathcal{B}(M_0)$. This is trivially true since $M_0 = M_0$ and by definition satisfies $\mathcal{B}$.*

4.2.2 Reachability

The weak bisimulation relation $\mathcal{B}$ between states of NLC and $SPN(NLC)$ can be exploited to understand how the reachable states of NLC are connected to the reachable states of $SPN(NLC)$.

Lemma 4 *Given a network-level controller NLC, every state (M_k, TS_k) reachable by NLC is weakly bisimilar to a state (M_j) reachable by $SPN(NLC)$.*

Proof:
The network-level controller's initial state is (M_0, TS_0). The sub-Petri net $SPN(NLC)$ has an initial state of (M_0). Proposition 2 explains why these two states are weakly bisimilar, $(M_0, TS_0)\mathcal{B}(M_0)$.

Every reachable state of NLC, (M_k, TS_k), can be reached from the initial state (M_0, TS_0) through a finite length string of events s, $(M_0, TS_0) \overset{s}{\Longrightarrow} (M_k, TS_k)$.

Since $\mathcal{B}$ is a weak bisimulation, and since $(M_0, TS_0)\mathcal{B}(M_0)$, the sub-Petri net can also execute s from (M_0) to produce a new state (M_j), $(M_0) \overset{s}{\Longrightarrow} (M_j)$. Additionally, (M_j) is guaranteed to be weakly bisimilar to (M_k, TS_k). $\Box$

Lemma 5 *Given a network-level controller NLC, every state (M_j) reachable by $SPN(NLC)$ is weakly bisimilar to a state (M_k, TS_k) reachable by NLC.*

Proof:
The network-level controller's initial state is (M_0, TS_0). The sub-Petri net $SPN(NLC)$ has an initial state of (M_0). Proposition 2 explains why these two states are weakly bisimilar, $(M_0, TS_0)\mathcal{B}(M_0)$.

Every reachable state of $SPN(NLC)$, (M_j), can be reached from the initial state (M_0) through a finite length string of events s, $(M_0) \overset{s}{\Longrightarrow} (M_j)$.

Since $\mathcal{B}$ is a weak bisimulation, and since $(M_0, TS_0)\mathcal{B}(M_0)$, the network-level controller can also execute s from (M_0, TS_0) to produce a new state (M_k, TS_k), $(M_0, TS_0) \overset{s}{\Longrightarrow} (M_k, TS_k)$. Additionally, (M_k, TS_k) is guaranteed to be weakly bisimilar to (M_j). $\Box$

Theorem 3 *Given a network-level controller NLC, a state (M_j) is reachable by $SPN(NLC)$ if and only if a weakly bisimilar state (M_k, TS_k) is reachable by NLC.*

Proof:
Case 1.) if (M_k, TS_k) is reachable then a weakly bisimilar (M_j) is reachable

Lemma 4.

Case 2.) if (M_j) is reachable then a weakly bisimilar (M_k, TS_k) is reachable

---

Lemma 5.

□

Theorem 3 shows that the reachability graph of $SPN(NLC)$ can be used to indirectly understand the entire reachability graph of NLC through the weak bisimulation $\mathcal{B}$. All of the reachable states of $SPN(NLC)$ can be explicitly calculated by standard algorithms. Each of these reachable states (M_j) is weakly bisimilar to a reachable state (M_k, TS_k) in NLC. This specific weak bisimulation guarantees that $M_j = M_k$, so the corresponding number of tokens for the NLC is exactly known. Since NLC is assumed to stay task-token consistent, knowing the marking M_k also reveals exactly how many task states exist and which task definitions they are based upon.

The Petri nets in figure 3.8 show all of the reachable states for the sub-Petri net of that network-level controller. The reachable states of $SPN(NLC)$ could be presented to a human operator as a reachability graph, but this information is normally complex and used to evaluate other simpler properties.

4.2.3 Boundedness

A Petri net place's bound is the maximum number of tokens that can exist in that place for any reachable state. For network-level controllers, a place's token bound also refers to the maximum number of tokens that exist in the place for any reachable state, definition 54. A network-level controller place also has a task bound which is the maximum number of task states that exist based on the place's associated task definition for any reachable state, definition 55.

Definition 54 *A NLC place $p^i \in P$ has a token bound of y if and only if all reachable states, (M_k, TS_k), satisfy: $M_k(p^i) \leq y$.*

Definition 55 *A NLC place $p^i \in P$ has a task bound of y if and only if all reachable states, (M_k, TS_k), satisfy: cardinality$(TS_k|_{def(p^i)}) \leq y$.*

Theorem 4 will show that the Petri net bounds for $SPN(NLC)$ provide the token bounds for NLC. Theorem 5 will then show that if the place has a null definition, the task bound is always 0. Theorem 6 will then show that if the place has a task definition and if NLC stays task-token consistent, the task bound matches the token bound.

Theorem 4 *A place p^i of NLC has a token bound of y if and only if p^i of $SPN(NLC)$ also has a bound of y.*

Proof:
Case 1.) p^i of NLC has a token bound of y if p^i of $SPN(NLC)$ has a bound of y

Theorem 3 shows that every reachable state (M_k, TS_k) of NLC will be weakly bisimilar to some reachable state (M_j) of $SPN(NLC)$, with $M_k = M_j$.
Since p^i of $SPN(NLC)$ has a bound of y, every reachable state (M_j) of $SPN(NLC)$ has a marking that satisfies $M_j(p^i) \leq y$.
So every reachable state (M_k, TS_k) satisfies: $M_k(p^i) = M_j(p^i) \leq y$.

Case 2.) p^i of $SPN(NLC)$ has a bound of y if p^i of NLC has a token bound of y
Theorem 3 shows that every reachable state (M_j) of $SPN(NLC)$ will be weakly bisimilar to some reachable state (M_k, TS_k) of NLC, with $M_j = M_k$.
Since p^i of NLC has a token bound of y, every reachable state (M_k, TS_k) of NLC has a marking that satisfies $M_k(p^i) \leq y$.
So every reachable state (M_j) satisfies: $M_j(p^i) = M_k(p^i) \leq y$.
$\square$

Theorem 5 *If a NLC place p^i has a token bound of y and an associated null task definition, $def(p^i) = null$, then p^i has a task bound of 0.*

Proof:
The place p^i has no associated task definition, therefore there can be no tasks in TS_k based upon the non-existent definition. The tokens in the marking exist, but no matching task states exist. $\square$

Theorem 6 *Assuming NLC starts task-token consistent, if a place p^i has a token bound of y and an associated task definition, $def(p^i) \neq null$, then p^i has a task bound of y.*

Proof:
Theorem 1 shows that all reachable states (M_k, TS_k) will also be task-token consistent.
By the token bound assumption every reachable state (M_k, TS_k) satisfies $M_k(p^i) \leq y$.
The task-token consistency assumption shows that $[M_k(p^i) = cardinality(TS_k|_{def(p^i)})]$.
Re-arranging and equating: $cardinality(TS_k|_{def(p^i)}) = M_k(p^i) \leq y$. $\square$

Corollary 2 *If a place p^i of $SPN(NLC)$ has a bound of y, and if p^i of NLC has an associated task definition $def(p^i) \neq null$; then p^i of NLC has a token bound of y and a task bound of y.*

Proof:
Theorem 4 shows that if a place p^i of $SPN(NLC)$ has a bound of y then p^i of NLC has a token bound of y. Theorem 6 then shows that since p^i of NLC has a token

bound of y, p^i also has a task bound of y.
□

Corollary 3 *If a place p^i of $SPN(NLC)$ has a bound of y, and if p^i of NLC has a null associated task definition $def(p^i) = null$; then p^i of NLC has a token bound of y and a task bound of 0.*

Proof:
Theorem 4 shows that if a place p^i of $SPN(NLC)$ has a bound of y then p^i of NLC has a token bound of y. Theorem 5 then shows that since p^i of NLC has a null associated task definition, p^i has a task bound of 0.
□

Standard algorithms can be used to calculate the bound for any place p^i of $SPN(NLC)$. If a place p^i in $SPN(NLC)$ has a calculated bound of y, then p^i in NLC also has a token bound of y. If p^i in NLC has a null task definition, $def(p^i) = null$, then the task bound for p^i is 0. If p^i in NLC has an associated task definition, $def(p^i) \neq null$, then due to the task-token consistency there will be at most y task states of definition $def(p^i)$ in the network at any moment.

If the token bounds or task bounds are too high then too many copies may potentially exist; the human operator may want to modify the network-level controller either structurally or by changing capacity constraints. The capacity constraints prevent the number of tokens from exceeding the constrained values. By computing the reachable states one may determine possibly tighter theoretical bounds than those guaranteed by the capacity constraints. If any place in $SPN(NLC)$ is behaviorally bounded below the constrained value, this tighter bound value is useful to know.

Figure 3.8 shows that place $P1$ has a behavioral bound of 1. Using the standard Petri net terminology this net is safe because each bound never exceeds 1. The bounds on the places informs the human operator that there will never be more than 1 identical task in the network at the same time.

4.2.4 Liveness

The reachability graph can also be used to guarantee that each transition could potentially fire. If there is a transition that will never possibly fire, there is no reason for it being included in the network-level controller. In this situation the human operator should be informed of the dead transition. Figure 3.8 shows that all of the transitions in that NLC will eventually fire, thus it is a live NLC.

4.2.5 Deadlock

Deadlock in Petri nets is often considered a bad thing. Here, it could indicate that a network-level controller is in a state that is ‘stuck’ or, as in figure 3.8 at $k = 6$, that the NLC is ‘finished’. The difference between being ‘stuck’ and ‘finished’ is merely the expectation of the human operator. If the human operator finds the lack of further progress acceptable, it is ‘finished’. If the human operator finds the lack of further progress unacceptable, it is ‘stuck’ and some modifications through runtime patching must be made.

4.2.6 Additional Properties

There are many other Petri net properties that can be analyzed. However reachability, boundedness, liveness, and deadlock are the most obvious, fundamental, and easily understandable.

Conflicts, where one transition firing disables another previously enabled transition, could be identified. Identifying them does nothing to suggest possible modifications; and if a conflict exists, it is likely that the human operator intended for the specified conflict to exist.

Similarly, mutual exclusions could be identified to show that two definitions never have tasks in the network at the same moment.

Mutual exclusions are sub-cases of the more general place invariants and transition invariants that could be identified. Understanding what these sets of inequalities signify would require a deep understanding of Petri net theory. These properties, like general LTL/CTL properties that could also be proven, are better used off-line by experienced industrial engineers evaluating automated assembly lines instead of on-line by non-experts in time-critical situations.

Reachability, boundedness, liveness, and deadlock are easy to evaluate and explain making them very useful as on-line feedback to human operators. Formal proofs are future work for both liveness and deadlock, but should also be rather obvious and straightforward.

Chapter 5

Runtime Patching Language for Network-Level Controllers

Knowledge of if a network-level controller is task bounded, token bounded, live, or deadlocked is only useful if the human operator can then manipulate the controller's structure to adjust its behavior. These manipulations are done through the runtime patching language for network-level controllers (RPL4NLC). Runtime patching allows a human operator to make a small incremental change to the network-level controller[120]. Since the human operator can now manipulate the network-level controller's structure, what was the structure becomes a part of the extended state.

Network-level controllers without runtime patching had fixed structures, (P, T, F, W, K, def, TD), and the state contained only the marking and set of task states, (M, TS). Runtime patching intends to allow these structures to be manipulated by the human operator so that the controller can be significantly altered. This extends the state to include all parts of the network-level controller, $(P, T, F, W, K, M, def, TD, TS)$. This state will often be written shorthand as (NLC), or even more concisely as σ to save space. Every piece of the network-level controller can be somehow affected through the runtime patching language.

5.1 Syntax

The syntax and semantics for the runtime patching language for network-level controllers will be presented following the example of Winskel's IMP from [185].

Table 5.1 shows the different syntactic meta-variables that will stand for arbitrary elements of the correct type. These elements will often be given superscripts for identification (e.g. td^m vs td^n). The detailed syntactic representations for integers, places, transitions, etc. is not specified; this is similar to how Winskel assumes a representation for integers and variables in IMP. These details would do little to illuminate what runtime patching does.

Table 5.1: Syntactic meta-variables for the runtime patching language

- rp ranges over runtime patches
- q ranges over positive integers
- p ranges over places
- t ranges over transitions
- f ranges over arcs
- td ranges over task definitions
- ts ranges over task states

The runtime patching language describes individual runtime patches rp. The language for rp is presented with a Backus-Naur Form (BNF). The BNF shows that rp can be one of 16 different individual runtime patches that incrementally modify the network-level controllers.

The runtime patching options are: adding a null place, adding a place with an associated task definition, deleting a place, adding a transition, deleting a transition, adding an arc (incoming/outgoing), deleting an arc (incoming/outgoing), modifying the weight of an arc (incoming/outgoing), modifying the capacity constraint of a place, adding a token, deleting a token, updating the content of a task definition, and updating the content of a task state.

Table 5.2: The runtime patching language presented as a BNF.

$$\begin{aligned} rp ::= \; & \mathbf{addPlace()} \mid \mathbf{addPlace(}td^m\mathbf{)} \mid \mathbf{deletePlace(}p^m\mathbf{)} \mid \mathbf{addTransition()} \mid \\ & \mathbf{deleteTransition(}t^m\mathbf{)} \mid \mathbf{addArc(}p^m, t^n\mathbf{)} \mid \mathbf{addArc(}t^m, p^n\mathbf{)} \mid \\ & \mathbf{deleteArc(}p^m, t^n\mathbf{)} \mid \mathbf{deleteArc(}t^m, p^n\mathbf{)} \mid \mathbf{modifyWeight(}p^m, t^n, q\mathbf{)} \mid \\ & \mathbf{modifyWeight(}t^m, p^n, q\mathbf{)} \mid \mathbf{modifyCapacity(}p^m, q\mathbf{)} \mid \mathbf{addToken(}p^m\mathbf{)} \mid \\ & \mathbf{deleteToken(}p^m\mathbf{)} \mid \mathbf{modifyTaskDefinition(}td^m, td^n\mathbf{)} \mid \\ & \mathbf{modifyTaskState(}ts^m, ts^n\mathbf{)} \end{aligned}$$

5.2 Structural Operational Semantics

The structural operational semantics presented below detail exactly how each patch is applied to modify the current state σ. The resulting state will be displayed as in IMP with $\sigma[A/B][C/D]$ representing the state that is identical to σ but with A replacing B and C replacing D.

Table 5.3: Structural operational semantics for runtime patching language.

$$\text{addPlace-null}\;\frac{}{<\mathbf{addPlace()}, \sigma> \to \sigma[P'/P][K'/K][M'/M][def'/def]}$$

$$P' = P + \{p^m\},\ p^m \notin P \text{ and } K'(p) = \begin{cases} K(p), & \text{if } p \neq p^m \\ defaultK, & \text{if } p = p^m \end{cases} \text{ and}$$

$$M'(p) = \begin{cases} M(p), & \text{if } p \neq p^m \\ 0, & \text{if } p = p^m \end{cases} \text{ and } def'(p) = \begin{cases} def(p), & \text{if } p \neq p^m \\ null, & \text{if } p = p^m \end{cases}$$

$$\text{addPlace}\;\frac{}{<\mathbf{addPlace(}td^m\mathbf{)}, \sigma> \to \sigma[P'/P][K'/K][M'/M][def'/def][TD'/TD]}$$

$$P' = P + \{p^m\},\ p^m \notin P \text{ and } K'(p) = \begin{cases} K(p), & \text{if } p \neq p^m \\ defaultK, & \text{if } p = p^m \end{cases} \text{ and}$$

$$M'(p) = \begin{cases} M(p), & \text{if } p \neq p^m \\ 0, & \text{if } p = p^m \end{cases} \text{ and } def'(p) = \begin{cases} def(p), & \text{if } p \neq p^m \\ td^m, & \text{if } p = p^m \end{cases} \text{ and } TD' = TD + \{td^m\}$$

$$\text{deletePlace}\;\frac{}{<\mathbf{deletePlace(}p^m\mathbf{)}, \sigma> \to \sigma[P'/P][F'/F][TD'/TD][TS'/TS]}$$

$$P' = P - \{p^m\} \text{ and } F' = F - \{p^m \times T\} - \{T \times p^m\} \text{ and}$$

$$TD' = \begin{cases} TD, & \text{if } def(p^m) = null \\ TD - \{def(p^m)\}, & \text{if } def(p^m) \neq null \end{cases}, TS' = \begin{cases} TS, & \text{if } def(p^m) = null \\ TS - TS|_{def(p^m)}, & \text{if } def(p^m) \neq null \end{cases}$$

$$\text{addTransition}\;\frac{}{<\mathbf{addTransition()}, \sigma> \to \sigma[T'/T]}$$

$$T' = T + \{t^m\},\ t^m \notin T$$

$$\text{deleteTransition}\;\frac{}{<\mathbf{deleteTransition(}t^m\mathbf{)}, \sigma> \to \sigma[T'/T][F'/F]}$$

$$T' = T - \{t^m\} \text{ and } F' = F - \{P \times t^m\} - \{t^m \times P\}$$

The 'addPlace-null' and 'addPlace' rules from table 5.3 specify how runtime patching can create new places. These new places always have a default capacity constraint value, $defaultK$, and zero tokens. If a task definition was provided, it is added to the set of task definitions in the system and the def function is appropriately updated. If no task definition is provided, def is given a $null$ value for the new place.

The 'deletePlace' rule removes a place, its connected arcs, and any associated task definitions and task states. These changes may shrink the domains of the other functions: W, K, def, M. These function's domains change but their values are not modified. Thus, to save space they will not be explicitly listed. For example, if $deletePlace(p^m)$ removes a place p^m from P to create the smaller P', the capacity function K' will not be explicitly listed since the function's values do not change, only its domain. In a practical implementation, if a table is used to store these function's values the rows or columns related to p^m may need to be removed to prevent the tables from growing arbitrarily large over time (this is essentially a form of garbage collection).

Rule 'addTransition' simply creates a new transition. The 'deleteTransition' rule removes the transition and any connected arcs; this may shrink the domain of W.

Table 5.4: Structural operational semantics for runtime patching language.

$$\text{addArc-incoming}\frac{}{<\mathbf{addArc}(p^m, t^n), \sigma > \rightarrow \sigma[F'/F][W'/W]}$$

$$F' = \begin{cases} F + \{(p^m, t^n)\}, & \text{if } p^m \in P, t^n \in T \\ F, & \text{otherwise} \end{cases} \text{ and}$$

$$W'(f) = \begin{cases} defaultW, & \text{if } f = (p^m, t^n), p^m \in P, t^n \in T \\ W(f), & \text{otherwise} \end{cases}$$

$$\text{addArc-outgoing}\frac{}{<\mathbf{addArc}(t^m, p^n), \sigma > \rightarrow \sigma[F'/F][W'/W]}$$

$$F' = \begin{cases} F + \{(t^m, p^n)\}, & \text{if } t^m \in T, p^n \in P \\ F, & \text{otherwise} \end{cases} \text{ and}$$

$$W'(f) = \begin{cases} defaultW, & \text{if } f = (t^m, p^n), t^m \in T, p^n \in P \\ W(f), & \text{otherwise} \end{cases}$$

$$\text{deleteArc-incoming}\frac{}{<\mathbf{deleteArc}(p^m, t^n), \sigma > \rightarrow \sigma[F'/F]}$$

$$F' = \begin{cases} F - \{(p^m, t^n)\}, & \text{if } (p^m, t^n) \in F \\ F, & \text{otherwise} \end{cases}$$

$$\text{deleteArc-outgoing}\frac{}{<\mathbf{deleteArc}(t^m, p^n), \sigma > \rightarrow \sigma[F'/F]}$$

$$F' = \begin{cases} F - \{(t^m, p^n)\}, & \text{if } (t^m, p^n) \in F \\ F, & \text{otherwise} \end{cases}$$

$$\text{modifyWeight-incoming}\frac{}{<\mathbf{modifyWeight}(p^m, t^n, q), \sigma > \rightarrow \sigma[W'/W]}$$

$$W'(f) = \begin{cases} q, & \text{if } f = (p^m, t^n), f \in F \\ W(f), & \text{otherwise} \end{cases}$$

$$\text{modifyWeight-outgoing}\frac{}{<\mathbf{modifyWeight}(t^m, p^n, q), \sigma > \rightarrow \sigma[W'/W]}$$

$$W'(f) = \begin{cases} q, & \text{if } f = (t^m, p^n), f \in F \\ W(f), & \text{otherwise} \end{cases}$$

Incoming arcs go from a place to a transition and outgoing arcs go from a transition to a place. The 'addArc-incoming' and 'addArc-outgoing' rules in table 5.4 reflect this. They simply insert a new arc with a default weight value, $defaultW$.

Similarly the 'deleteArc-incoming' and 'deleteArc-outgoing' rules remove existing arcs. Deleting arcs may shrink the domain of the weights, W.

Every arc should have a weight at all times. It does not make sense to add or delete these values, instead they are given default values at creation and can be modified as necessary. The 'modifyWeight-incoming' and 'modifyWeight-outgoing' rules allow for arc weights to be updated to new positive integer values.

Table 5.5: Structural operational semantics for runtime patching language.

$$\text{modifyCapacity}\frac{}{<\mathbf{modifyCapacity}(p^m, q), \sigma >\rightarrow \sigma[K'/K]}$$

$$K'(p) = \begin{cases} q, & \text{if } p = p^m,\ M(p^m) \leq q \\ K(p), & \text{otherwise} \end{cases}$$

$$\text{addToken}\frac{}{<\mathbf{addToken}(p^m), \sigma >\rightarrow \sigma[M'/M][TS'/TS]}$$

$$M'(p) = \begin{cases} M(p)+1, & \text{if } p = p^m,\ M(p^m) < K(p^m) \\ M(p), & \text{otherwise} \end{cases}$$

$$TS' = \begin{cases} TS + \{ts^q\}, & \text{if } M(p^m) < K(p^m), def(p^m) \neq null \\ & \quad \text{where: } def(p^m) = taskDef(ts^q), done(ts^q) = false \\ TS, & \text{otherwise} \end{cases}$$

$$\text{deleteToken}\frac{}{<\mathbf{deleteToken}(p^m), \sigma >\rightarrow \sigma[M'/M][TS'/TS]}$$

$$M'(p) = \begin{cases} M(p)-1, & \text{if } p = p^m, M(p^m) > 0 \\ M(p), & \text{otherwise} \end{cases}$$

$$TS' = \begin{cases} TS - \{ts^q\}, & \text{if } M(p^m) > 0, def(p^m) \neq null \\ & \quad \text{where: } def(p^m) = taskDef(ts^q) \\ TS, & \text{otherwise} \end{cases}$$

$$\text{modifyTaskDefinition}\frac{}{<\mathbf{modifyTaskDefinition}(td^m, td^n), \sigma >\rightarrow \sigma[TD'/TD]}$$

if $td^m \in TD$ then $details(td^n)$ replace the $details(td^m)$

$$\text{modifyTaskState}\frac{}{<\mathbf{modifyTaskState}(ts^m, ts^n), \sigma >\rightarrow \sigma[TS'/TS]}$$

if $ts^m \in TS$ then $details(ts^n)$ replace the $details(ts^m)$

Every place has a capacity constraint at all times. Again, it does not make sense to add or delete these values. Instead, during the place's creation the capacity constraint is given a default value, $defaultK$. This value can then be modified as necessary through the 'modifyCapacity' rule in table 5.5. The new capacity constraint value is required to be greater than or equal to the current marking.

The 'addToken' rule shows how individual tokens (and possibly task states) can be added. If a place has no additional capacity, no more tokens or task states will be added. If there is space, a new token will be created. If the place is associated to a task definition, a new task state of the correct definition will also be created.

The 'deleteToken' rule removes individual tokens. If a place has a token, that token can be removed. If the place is associated to a task definition, an appropriate task state will be removed as well.

The last two runtime patching rules, 'modifyTaskDefinition' and 'modifyTaskState', deal with modifying the contents of the task definitions and task states. While these details are not modeled in the network-level controller, the human operator will possibly want to update the details of the task definitions or task states. For instance, it may be useful to modify a task definition "visit point A" to more precisely locate "A". These commands do not add or remove task definitions and task states. They also do not affect any of the other functions in the network-level controller.

5.3 Conclusions

The runtime patching language allows a human operator to apply an incremental change to the network-level controller. These changes extend beyond modifying the tokens and task states to include the entire network-level controller. This extended state is represented as $(P, T, F, W, K, M, def, TD, TS) = (NLC) = \sigma$. This allows the runtime patches to modify all parts of the network-level controller, as specified by the structural operational semantics: $< rp, \sigma > \rightarrow \sigma'$. An alternative way to express this transition system would be: $< rp, \sigma > \rightarrow \sigma'$ if and only if $\sigma \xrightarrow{rp} \sigma'$, where the state σ still evolves to σ' under the application of the specific runtime patch rp.

Any runtime patch is very simple containing only 1 single command from a choice of 16. This was initially developed to mimic the interaction of a human operator with a graphical user interface. The human operator would likely click a button 'add transition' to cause an $addTransition()$ runtime patch to be applied. Similarly, the human operator would likely right-click on a place and hit 'modify capacity' to cause a $modifyCapacity(p^m, q)$ runtime patch to be applied.

There is no reason that the runtime patching language could not also be extended with sequential, conditional, and recursion constructs similar to IMP to add additional expressiveness [185]. These constructs would allow more complex programs of runtime patching commands to be formed. These programs would make several modifications at one time. While this would add expressiveness, the details of an if-statement are standard and not necessary to explain the concepts within runtime patching.

Chapter 6

Network-Level Controllers Modified By Runtime Patching

In chapter 3 the network-level controller's state (M, TS) evolved through UAVs completing tasks and transitions firing to add and remove both tokens and task states. Chapter 5 provided a simple runtime patching language that allowed any network-level controller to be modified by the human operator. These modifications can adjust what was the network-level controller's structure. This extended state now includes all parts of the network-level controller, $(P, T, F, W, K, M, def, TD, TS)$, written shorthand as (NLC). This network-level controller will evolve from a proper initial condition, $(P_0, T_0, F_0, W_0, K_0, M_0, def_0, TD_0, TS_0) = (NLC)_0$, by the interleaved completion of tasks, transitions firing, and runtime patches being applied.

6.1 Syntax

Definition 56 *A network-level controller (capable of being modified by the runtime patching language) can be modeled as a tuple*
$NLC = (P_0, T_0, F_0, W_0, K_0, M_0, def_0, TD_0, TS_0)$ *where:*

- P_0 *is the initial set of places,*
- T_0 *is the initial set of transitions,*
- $F_0 \subseteq (P_0 \times T_0) \cup (T_0 \times P_0)$ *is the initial flow relation (arcs connecting places to transitions),*
- $W_0 : F_0 \rightarrow \mathcal{N}$ *is the initial weight of each arc in* F_0*,*
- K_0*:* $P_0 \rightarrow \mathcal{N}_+$ *is the initial capacity constraint for each place,*
- M_0*:* $P_0 \rightarrow \mathcal{N}$ *is the initial marking of tokens for each place in* P_0*,*

- *def_0: $P_0 \to TD_0 \cup \{null\}$ is the initial association of each place to a single task definition or a special indicator null of no task definition,*
- *TD_0 is the initial set of task definitions in the network,*
- *TS_0 is the initial set of task states in the network.*

A network-level controller should always be well formed. In section 3.3 this was a property of the network-level controller's structure. Here, since that structure is now part of the state, being well formed will be a property of a network-level controller's state.

Definition 57 *A network-level controller state $(NLC)_k = (P_k, T_k, F_k, W_k, K_k, M_k, def_k, TD_k, TS_k)$ is well formed if:*

- *$F_k \subseteq (P_k \times T_k) \cup (T_k \times P_k)$ all of the currently existing arcs go from currently existing places to currently existing transitions or from currently existing transitions to currently existing places,*
- *$W_k : F_k \to \mathcal{N}$ each currently existing arc has a non-negative integer weight,*
- *K_k: $P_k \to \mathcal{N}_+$ each currently existing place has a positive integer capacity constraint,*
- *M_k: $P_k \to \mathcal{N}$ each currently existing place has a non-negative integer number of tokens,*
- *def_k: $P_k \to TD_k \cup \{null\}$ each currently existing place is associated to a single currently existing task definition or a special indicator null of no task definition.*

A state being well formed guarantees: there are no dangling arcs connected to something that no longer exists, there are no arcs with incorrectly defined weights, there are no places with incorrectly defined capacity constraints, there are no places with incorrectly defined markings, and that there are no places with incorrectly defined task definition associations. As the NLC evolves through transitions firing, tasks becoming completed, or runtime patches being applied; each resultant state should be well formed. The invariance of being well formed will be proven in section 6.4.

Well formed asserts the 'correctness' of the Petri net portion of the network-level controller's state. Fully representative does the same for the set of task definitions, TD_k. Every NLC state should be fully representative, meaning every task definition should always be represented by exactly one place.

Definition 58 *A network-level controller state $(NLC)_k = (P_k, T_k, F_k, W_k, K_k, M_k, def_k, TD_k, TS_k)$ is fully representative if:*

$$\forall td \in TD_k.(\exists! p \in P_k.def_k(p) = td).$$

The final form of 'correctness' is that a NLC's tokens and tasks 'match'. A NLC state is task-token consistent if: for each place that has a task definition, the number of tokens in that place matches the number of task states of that definition.

Definition 59 *A NLC state $(P_k, T_k, F_k, W_k, K_k, M_k, def_k, TD_k, TS_k)$ is task-token consistent if the following holds:*

$$\forall p \in P_k.[def_k(p) \neq null] \Rightarrow [M_k(p) = cardinality(TS_k|_{def_k(p)})].$$

After the semantics of network-level controllers (capable of being modified by the runtime patching language) is presented, the invariance of being well formed, fully representative, and task-token consist will be proven. These properties could have been lumped into one larger property, but they are easier to express, explain, and prove individually.

6.2 Semantics

Network-level controllers evolve from a state $(P_k, T_k, F_k, W_k, K_k, M_k, def_k, TD_k, TS_k)$ to a state $(P_{k+1}, T_{k+1}, F_{k+1}, W_{k+1}, K_{k+1}, M_{k+1}, def_{k+1}, TD_{k+1}, TS_{k+1})$ based on either a task being completed, a transition being fired, or a runtime patch being applied.

The environment (the UAVs) can complete the tasks just as in section 3.4. From the perspective of the network-level controller the environment only causes an instantaneous change in TS_k. Since the state was expanded, the new semantics for a task being completed are listed, despite being essentially the same.

Definition 60 *The environment completing a task $ts^i \in TS_k$ causes a state change $(P_k, T_k, F_k, W_k, K_k, M_k, def_k, TD_k, TS_k) \xrightarrow{complete(ts^i)}$ $(P_{k+1}, T_{k+1}, F_{k+1}, W_{k+1}, K_{k+1}, M_{k+1}, def_{k+1}, TD_{k+1}, TS_{k+1})$ where:*

- $P_{k+1} = P_k$,
- $T_{k+1} = T_k$,
- $F_{k+1} = F_k$,
- $W_{k+1} = W_k$,
- $K_{k+1} = K_k$,
- $M_{k+1} = M_k$,

- $def_{k+1} = def_k$,
- $TD_{k+1} = TD_k$,
- $TS_{k+1} = TS_k - ts^i_k + ts^i_{k+1}$ *only the task state* ts^i *is updated, this update satisfies* $done(ts^i_k) = false \wedge done(ts^i_{k+1}) = true$.

Obviously there are many components of the network-level controller's state that remain unaffected.

For a NLC transition to be enabled in state $(P_k, T_k, F_k, W_k, K_k, M_k, def_k, TD_k, TS_k)$, it must be token enabled and completion enabled. Token enabled is the standard Petri net condition applied to the current sub-Petri net.

Definition 61 *At time* k *a NLC transition* $t \in T_k$ *is token enabled if:*

$$\forall p \in (^\bullet t) \mid M_k(p) \geq W_k(p, t)$$

Completion enabled is also very similar to before.

Definition 62 *At time* k *a NLC transition* $t \in T_k$ *is completion enabled if:*
$\forall p \in (^\bullet t).[def_k(p) \neq null] \Rightarrow [\exists! ts^1_k, ...ts^{W_k(p,t)}_k \in TS_k.taskDef(ts^1_k) = def_k(p) \wedge done(ts^1_k) \wedge ... \wedge taskDef(ts^{W_k(p,t)}_k) = def_k(p) \wedge done(ts^{W_k(p,t)}_k)]$.

Completion enabled requires that at the current moment, each place with an arc going to t and an associated task definition have a sufficient number of tasks of the correct definition that are also currently finished. When checking for a completion enabled transition, the task states bound are recorded in the set TS^-. If the transition ends up being both token and completion enabled it can be fired and the set of tasks TS^- will be removed from the network along with the correct tokens.

Again, a set of new tasks, TS^+, must be created when a NLC transition t fires.

Definition 63 *The set of tasks to be created after a transition* $t \in T_k$ *fires is:*
$TS^+ = \bigcup_{p \in (t^\bullet)} TS^{t,p}$

- *if* $def_k(p) = null$ *then* $TS^{t,p} = \{\}$,
- *if* $def_k(p) \neq null$ *then* $TS^{t,p} = \{ts^1_k, ...ts^{W_k(t,p)}_k\}$ *where* $taskDef(ts^m_k) = def_k(p)$ *and* $done(ts^m_k) = false$ *for each new task state.*

When a NLC transition is token and completion enabled it can fire causing the tokens and the set of task states to change just as in chapter 3. No other part of the NLC state is affected.

Definition 64 *A transition $t \in T_k$ firing at time k causes a state change* $(P_k, T_k, F_k, W_k, K_k, M_k, def_k, TD_k, TS_k) \xrightarrow{firing(t)}$ $(P_{k+1}, T_{k+1}, F_{k+1}, W_{k+1}, K_{k+1}, M_{k+1}, def_{k+1}, TD_{k+1}, TS_{k+1})$ *where:*

- $P_{k+1} = P_k$,
- $T_{k+1} = T_k$,
- $F_{k+1} = F_k$,
- $W_{k+1} = W_k$,
- $K_{k+1} = K_k$,
- $M_{k+1}(p) = M_k(p) - W_k(p, t) + W_k(t, p)$,
- $def_{k+1} = def_k$,
- $TD_{k+1} = TD_k$,
- $TS_{k+1} = TS_k - TS^- + TS^+$.

Tasks being completed and transitions firing can evolve the M_k and TS_k portions of the state just as in section 3.4. If the human operator chooses not to apply runtime patches this will be the emergent behavior as the rest of the state will remain constant while the network-level controller fires transitions and the UAVs complete tasks.

Alternatively, a runtime patch rp can be applied by the human operator to drastically alter the network-level controller's state and consequently its future behavior. Chapter 5 gave a BNF syntax and structural operational semantics that describe how a runtime patch rp affects the state $(NLC)_k$. Those semantic rules show how $< rp, (NLC)_k > \rightarrow (NLC)_{k+1}$.

The representations for the structural operational semantics of runtime patches and the set-theoretic semantics of transitions firing and tasks being completed do not match. To consistently represent a behavior involving both, a runtime patch rp being applied can also be represented as $(NLC)_k \xrightarrow{rp} (NLC)_{k+1}$, but this is understood to occur if and only if $< rp, (NLC)_k > \rightarrow (NLC)_{k+1}$.

Any behavior of a network-level controller is an interleaving of tasks being completed, transitions firing, and runtime patches being applied.

Definition 65 *A string of states, $(NLC)_0, (NLC)_1, (NLC)_2 ...$, is a behavior of a network-level controller if:*

- $(P_0, T_0, F_0, W_0, K_0, M_0, def_0, TD_0, TS_0)$ *matches the initial state in the definition of the NLC,*

- $\forall k.[(NLC)_k \xrightarrow{complete(ts)} (NLC)_{k+1}] \ \vee \ [(NLC)_k \xrightarrow{firing(t)} (NLC)_{k+1}]$
  $\vee \ [< rp, (NLC)_k > \rightarrow (NLC)_{k+1}]$,
  the successor state is caused by a task being completed, a transition being fired, or a runtime patch being applied.

Definition 66 *The language of the network-level controller, $\mathcal{L}(NLC)$, is the set of all potential behaviors.*

6.3 Example

Figure 6.1 shows an example behavior with runtime patching. The network-level controller initializes at $k = 0$ to the empty controller and then is modified through a sequence of runtime patches by the human operator until a token (and a task state ts^1 based on td^1) is added at $k = 9$. At this point the transition t^0 is token enabled, but the new task is not completed. The UAVs execute and complete the task moving the controller to $k = 10$ where t^0 is token and completion enabled. The network-level controller then automatically fires transition t^0, creating and removing tokens and task states appropriately.

The behavior shown in figure 6.1 is:
$(NLC)_0 \xrightarrow{addPlace(td^1)} (NLC)_1 \xrightarrow{addPlace(td^2)} (NLC)_2 \xrightarrow{addTransition()} (NLC)_3 \xrightarrow{addTransition()}$
$(NLC)_4 \xrightarrow{addArc(p^0,t^0)} (NLC)_5 \xrightarrow{addArc(t^0,p^1)} (NLC)_6 \xrightarrow{addArc(p^1,t^1)} (NLC)_7 \xrightarrow{addArc(t^1,p^0)}$
$(NLC)_8 \xrightarrow{addToken(p^0)} (NLC)_9 \xrightarrow{completion(ts^1)} (NLC)_{10} \xrightarrow{firing(t^0)} (NLC)_{11} \xrightarrow{completion(ts^2)}$
$(NLC)_{12} \xrightarrow{firing(t^1)} (NLC)_{13}$.

A total of 3 different tokens/tasks are created in the figure. The first one is manually inserted by using runtime patching at $k = 9$, the next two are automatically generated by the NLC. The network-level controller would experience this cyclic behavior until another runtime patch is applied.

It again seems appropriate to point out that the graphical representation in figure 6.1 does not show all of the details of the network's state. The simplified representation hides the details of the task definitions and task states. All of the existing task definitions are represented by the places, assuming the state is fully representative. Also all of the existing task states are represented by the tokens, assuming the state is task-token consistent. At any moment k, viewing the Petri net portion of the network-level controller's state tells the human operator what tasks currently exist based on what definitions.

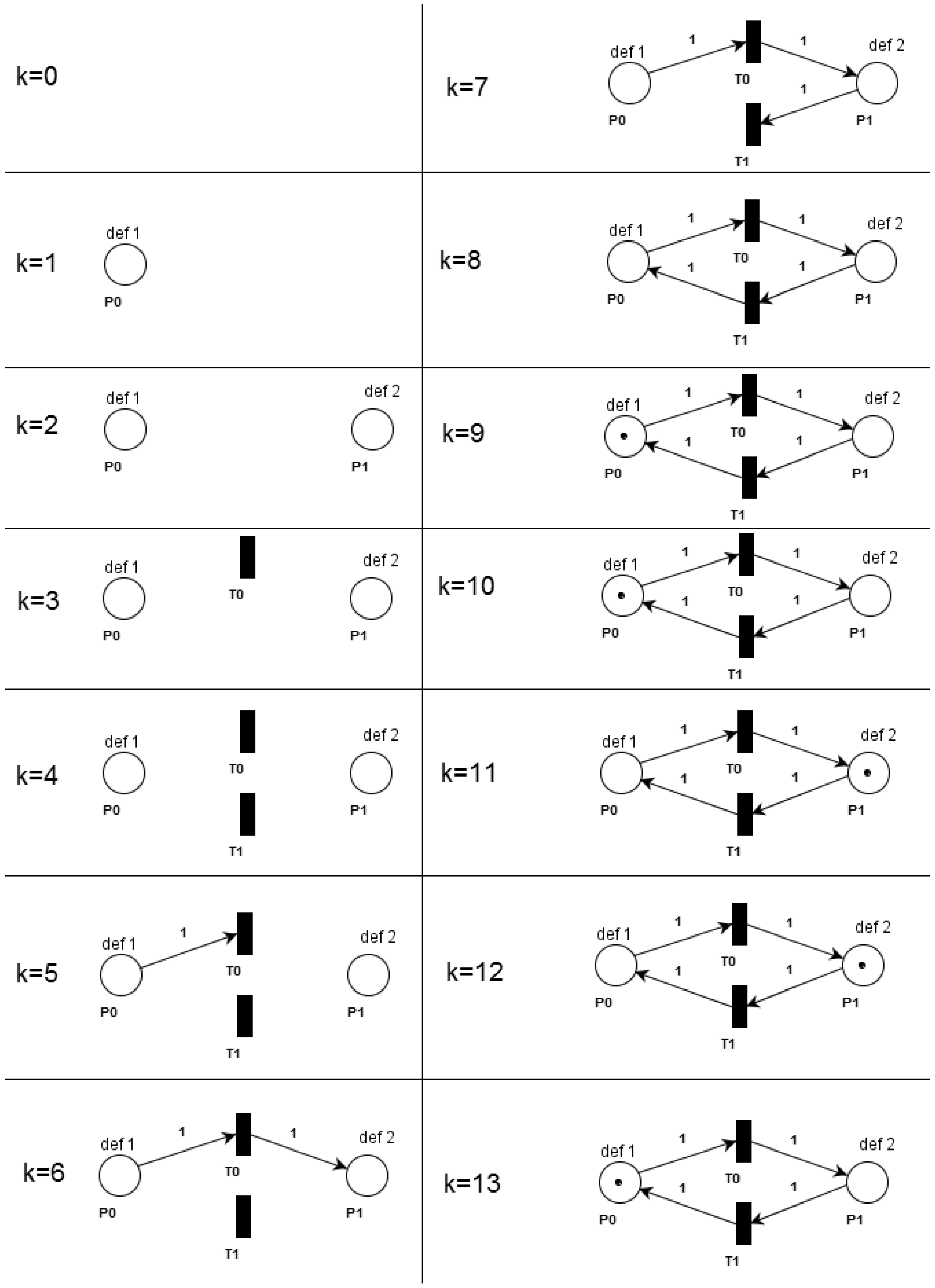


Figure 6.1: Example of runtime patching to modify a network-level controller.

6.4 Invariance Theorems

In order for the network-level controller to correctly operate, the state must remain well formed, fully representative, and task-token consistent. These are conditions on the state of the network level controller $(P_k, T_k, F_k, W_k, K_k, M_k, def_k, TD_k, TS_k)$, but the invariance of these properties is a result of the semantics.

6.4.1 Well Formed

Lemma 6 will show that any single step from a well formed state will result in a well formed state. Theorem 7 will extend this to show that all states reachable from a well formed initial state are guaranteed to be well formed.

Lemma 6 *If a network-level controller is in a well formed state $(P_k, T_k, F_k, W_k, K_k, M_k, def_k, TD_k, TS_k)$, then every state immediately reachable, $(P_{k+1}, T_{k+1}, F_{k+1}, W_{k+1}, K_{k+1}, M_{k+1}, def_{k+1}, TD_{k+1}, TS_{k+1})$, is also well formed.*

Proof: There are several ways that $(P_k, T_k, F_k, W_k, K_k, M_k, def_k, TD_k, TS_k)$ can transition to become $(P_{k+1}, T_{k+1}, F_{k+1}, W_{k+1}, K_{k+1}, M_{k+1}, def_{k+1}, TD_{k+1}, TS_{k+1})$, this proof will show that well formedness is preserved for all possible cases: a task being completed, a transition firing, and any runtime patch being applied.

Due to the numerous cases, the proof is included in appendix A. □

Theorem 7 *If a network-level controller starts in a well formed initial state $(P_0, T_0, F_0, W_0, K_0, M_0, def_0, TD_0, TS_0)$, then it will remain well formed through all reachable states $(P_k, T_k, F_k, W_k, K_k, M_k, def_k, TD_k, TS_k)$.*

Proof: The proof will proceed by an inductive argument on the length of the firing sequence required to reach the reachable state, $(P_k, T_k, F_k, W_k, K_k, M_k, def_k, TD_k, TS_k)$, where k is the length of the firing sequence.

Base Case $(k = 0)$: If the reachable state is the initial state $(P_0, T_0, F_0, W_0, K_0, M_0, def_0, TD_0, TS_0)$, it is well formed by assumption.

Inductive Step $(k \neq 0)$: Every reachable state can be reached through a firing sequence of some finite length k. The state $(NLC)_{k-1}$ in the behavioral sequence prior to $(NLC)_k$ is also reachable, but with a firing sequence of length $k-1$. By the inductive hypothesis $(NLC)_{k-1}$ must be well formed. This with lemma 6 proves that $(NLC)_k$ is also well formed. □

Theorem 7 proves the invariance of being well formed. Proposition 3 gives the most trivial well formed state. The empty state trivially satisfies all conditions of definition 57.

Proposition 3 *An empty state, one with no places, transitions, arcs, arc weights, capacity constraints, tokens, task definitions, or task states, is well formed.*

Corollary 4 *A NLC that starts in an empty state stays well formed indefinitely.*

Proof: By proposition 3 the empty state is well formed. By theorem 7 the NLC will stay well formed in all states reachable from this well formed initial state, which happens to be empty. □

Corollary 4 proves that if the network-level controller is initialized to the empty state, then it is guaranteed to remain well formed.

6.4.2 Fully Representative

Lemma 7 will show that any single step from a fully representative state will result in a fully representative state. Theorem 8 will extend this to show that all states reachable from a fully representative initial state are guaranteed to be fully representative.

Lemma 7 *If a network-level controller is in a fully representative state* $(P_k, T_k, F_k, W_k, K_k, M_k, def_k, TD_k, TS_k)$*, then every state immediately reachable,* $(P_{k+1}, T_{k+1}, F_{k+1}, W_{k+1}, K_{k+1}, M_{k+1}, def_{k+1}, TD_{k+1}, TS_{k+1})$*, is also fully representative.*

Proof: There are several ways that $(P_k, T_k, F_k, W_k, K_k, M_k, def_k, TD_k, TS_k)$ *can transition to become* $(P_{k+1}, T_{k+1}, F_{k+1}, W_{k+1}, K_{k+1}, M_{k+1}, def_{k+1}, TD_{k+1}, TS_{k+1})$*, this proof will show that being fully representative is preserved for all possible cases: a task being completed, a transition firing, and any runtime patch being applied.*

Due to the numerous cases, the proof is included in appendix B. □

Theorem 8 *If a network-level controller starts in a fully representative initial state* $(P_0, T_0, F_0, W_0, K_0, M_0, def_0, TD_0, TS_0)$*, then it will remain fully representative through all reachable states* $(P_k, T_k, F_k, W_k, K_k, M_k, def_k, TD_k, TS_k)$*.*

Proof: The proof will proceed by an inductive argument on the length of the firing sequence required to reach the reachable state, $(P_k, T_k, F_k, W_k, K_k, M_k, def_k, TD_k, TS_k)$*, where* k *is the length of the firing sequence.*

Base Case ($k = 0$): If the reachable state is the initial state $(P_0, T_0, F_0, W_0, K_0, M_0, def_0, TD_0, TS_0)$, it is fully representative by assumption.

Inductive Step ($k \neq 0$): Every reachable state can be reached through a firing sequence of some finite length k. The state $(NLC)_{k-1}$ in the behavioral sequence prior to $(NLC)_k$ is also reachable, but with a firing sequence of length $k - 1$. By the inductive hypothesis $(NLC)_{k-1}$ must be fully representative. This with lemma 7 proves that $(NLC)_k$ is also fully representative. □

Proposition 4 gives the most trivial fully representative state. The empty state trivially satisfies all conditions of definition 58.

Proposition 4 *An empty state, one with no places, transitions, arcs, arc weights, capacity constraints, tokens, task definitions, or task states, is fully representative.*

Corollary 5 *A NLC that starts in an empty state stays fully representative indefinitely.*

Proof: By proposition 4 the empty state is fully representative. By theorem 8 the NLC will stay fully representative in all states reachable from this fully representative initial state, which happens to be empty. □

Corollary 5 proves that if the network-level controller is initialized to the empty state, then it is guaranteed to remain fully representative.

6.4.3 Task-Token Consistent

Lemma 8 will show that any single step from a task-token consistent state will result in a task-token consistent state. Theorem 9 will extend this to show that all states reachable from a task-token consistent initial state are guaranteed to be task-token consistent.

Lemma 8 *If a network-level controller is in a task-token consistent state $(P_k, T_k, F_k, W_k, K_k, M_k, def_k, TD_k, TS_k)$, then every state immediately reachable, $(P_{k+1}, T_{k+1}, F_{k+1}, W_{k+1}, K_{k+1}, M_{k+1}, def_{k+1}, TD_{k+1}, TS_{k+1})$, is also task-token consistent.*

Proof: There are several ways that $(P_k, T_k, F_k, W_k, K_k, M_k, def_k, TD_k, TS_k)$ can transition to become $(P_{k+1}, T_{k+1}, F_{k+1}, W_{k+1}, K_{k+1}, M_{k+1}, def_{k+1}, TD_{k+1}, TS_{k+1})$, this proof will show that being task-token consistent is preserved for all possible cases: a task being completed, a transition firing, and any runtime patch being applied.

Due to the numerous cases, the proof is included in appendix C. □

Theorem 9 *If a network-level controller starts in a task-token consistent initial state $(P_0, T_0, F_0, W_0, K_0, M_0, def_0, TD_0, TS_0)$, then it will remain task-token consistent through all reachable states $(P_k, T_k, F_k, W_k, K_k, M_k, def_k, TD_k, TS_k)$.*

Proof: The proof will proceed by an inductive argument on the length of the firing sequence required to reach the reachable state, $(P_k, T_k, F_k, W_k, K_k, M_k, def_k, TD_k, TS_k)$, where k is the length of the firing sequence.

Base Case $(k = 0)$: If the reachable state is the initial state $(P_0, T_0, F_0, W_0, K_0, M_0, def_0, TD_0, TS_0)$, it is task-token consistent by assumption.

Inductive Step $(k \neq 0)$: Every reachable state can be reached through a firing sequence of some finite length k. The state $(NLC)_{k-1}$ in the behavioral sequence prior to $(NLC)_k$ is also reachable, but with a firing sequence of length $k - 1$. By the inductive hypothesis $(NLC)_{k-1}$ must be task-token consistent. This with lemma 8 proves that $(NLC)_k$ is also task-token consistent. $\square$

Proposition 5 gives the most trivial fully representative state. The empty state trivially satisfies all conditions of definition 59.

Proposition 5 *An empty state, one with no places, transitions, arcs, arc weights, capacity constraints, tokens, task definitions, or task states, is task-token consistent.*

Corollary 6 *A NLC that starts in an empty state stays task-token consistent indefinitely.*

Proof: By proposition 5 the empty state is task-token consistent. By theorem 9 the NLC will stay task-token consistent in all states reachable from this task-token consistent initial state, which happens to be empty. $\square$

Corollary 6 proves that if the network-level controller is initialized to the empty state, then it is guaranteed to remain task-token consistent.

If the network-level controller is started in a well formed, fully representative, and task-token consistent initial state; then theorems 7, 8, and 9 guarantee that all reachable states are also well formed, fully representative, and task-token consistent. By encapsulating the manipulations of the network-level controller's structure with runtime patching, the invariance of these properties can be guaranteed. This allows human operators to only create correct-by-construction network-level controllers.

Corollaries 4, 5, and 6 suggest that the network-level controller can begin at the trivially empty initial state and be guaranteed well formed, fully representative, and task-token consistent. The human operator can then create the desired network-level controller through runtime patches.

6.5 Analyzable Properties

If a network-level controller state $(P_k, T_k, F_k, W_k, K_k, M_k, def_k, TD_k, TS_k)$ is constructed through runtime patching, the sub-Petri net at this state can be evaluated as in section 4.2 under the assumption that this structure will remain fixed. If the NLC evolves only through tasks being completed and transitions firing, this assumption is valid and the network-level controller with runtime patching will behave like a network-level controller without runtime patching. If a runtime patch is applied, the new network-level controller state will be structurally different than the preceding state and any properties that were evaluated should be re-evaluated.

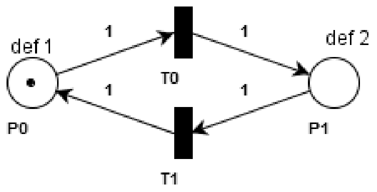


Figure 6.2: Simple example of a network-level controller that will automatically experience a cyclic behavior until a runtime patch occurs.

Figure 6.2 shows the simple network-level controller that was constructed in figure 6.1. If left alone, meaning no more runtime patches are applied by the human operator, it will experience the cyclic creation and completion of tasks based on td^1 and td^2. All of the transitions are live, the bounds on all places are 1, it cannot deadlock, and it currently is not deadlocked.

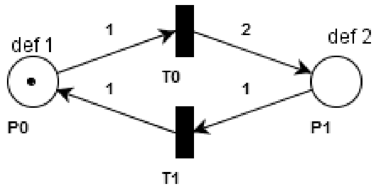


Figure 6.3: The network-level controller from figure 6.2 after a simple runtime patch modifying an arc weight, $modifyWeight(t^0, p^1, 2)$.

A simple runtime patch, $modifyWeight(t^0, p^1, 2)$, would result in figure 6.3. This state has drastically different potential behaviors than those in figure 6.2. If the network-level controller where left alone after this state, meaning no more runtime patches: all of the transitions would still be live, the bounds on the places would increase to their capacity constrained values, and it still would not be deadlocked.

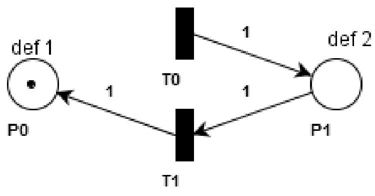


Figure 6.4: The network-level controller from figure 6.2 after a simple arc deletion, $deleteArc(p^0, t^0)$.

If instead a runtime patch $deleteArc(p^0, t^0)$ were applied to figure 6.2, figure 6.4 would result. Again, the potential behaviors of the network-level controller, if left alone, would drastically change. Neither of the transitions are live because the state is currently deadlocked.

6.6 Conclusions

Network-level controllers modified by the runtime patching language provide a novel method of controlling and automating the insertion and removal of tasks in a collaborative system. The NLC operates at a high level of abstraction and has a simple graphical depiction that illustrates the existence of tasks as well as the ordering constraints currently imposed on future tasks. The model has a simple graphical depiction, a network focus, handles a varying number of tasks, and can be affected by the human operator on-line. It satisfies all of the criteria sought by this dissertation.

However, the developments presented in this dissertation assumed a synchronous network. It was implicitly assumed that at any moment there was one consistent set of task definitions, TD_k, and one consistent set of task states, TS_k. It was also assumed that a transition firing could instantaneously remove or add task states to the network. Similarly, it was assumed that a runtime patch could instantaneously add, delete, or modify task definitions and task states. The synchronous network assumption simplifies the expected behavior of the network. Using a synchronous network allows this dissertation to develop and present the concepts of network-level control independent of the complicating issues that arise in any asynchronous distributed network, such as UAVs.

Developments for an asynchronous network would require assuming a specific method of propagating information among the many distributed network members. The specifics of this method would affect how quickly and with what guarantees the task definition and task state changes could occur. The network-level controller can make changes to its local copy, but the details of this asynchronous communication scheme would determine how and if those changes are fully propagated.

Future work includes selecting a specific asynchronous communication method and showing that network-level controllers operating in this environment, under the assumption that their local state estimate is the actual state, still execute 'correctly'. Several additional assumptions on the behavior of the asynchronous communication scheme will likely be necessary.

Chapter 7

The Collaborative Sensing Language

The Collaborative Sensing Language (CSL) is an XML-based language that enables interaction with a network-level controller, called the 'Publisher', at the Center for Collaborative Control of Unmanned Vehicles (C_3UV) [120]. CSL can be used to retrieve the network's state from this network-level controller or to send runtime patches that modify the network-level controller. This Publisher performs the high-level control of C_3UV's task-based collaborative UAV network for Intelligence, Surveillance, and Reconnaissance (ISR) tasks.

CSL provides the network's state from the perspective of the Publisher. Meaning, that when the state is requested it is formed from the Publisher's local estimate. The Petri net information only exists in the Publisher; however, the task definitions, task states, UAV definitions, UAV states are distributed throughout the network. The local Publisher estimate of these values is what is provided by CSL.

The other form of CSL interactions are the runtime patches that modify the behavior of the Publisher (network-level controller). In this manner, the intended network behavior can be drastically modified and controlled on-line through CSL.

The concepts presented in previous chapters are formalizations of ideas originally developed for CSL and the Publisher. The Publisher is a specific implementation of a network-level controller. CSL is a specific format to output the network-level controller's state as well as to input runtime patches. Network-level control was created to both formalize the model of computation and generalize the ideas behind CSL and the Publisher, so that they could potentially apply to other applications. This may cause some confusion with inconsistent terminology such as: Publisher = network-level controller. Where appropriate, these relationships will be identified and clarified.

7.1 Implementation

The Publisher component is an implementation of network-level control that uses CSL as a 'language independent' data exchange format to communicate with GUIs or web servers. The publisher is written in C++ and uses the Xerces-2.7.0 library for XML support. Xerces is an Apache XML Project product that contains a DOMparser, DOMwriter, and a DOMvalidator (for checking against the CSL schema). This allows the Publisher to read, write, and verify that the XML is correctly defined CSL.

Internally, the Publisher interacts with C_3UV's Collaborative Sensing Middleware (CSM) and Collaborative Control System (CCS). CSM's purpose is to transparently communicate and synchronize all of the information within the UAV network. When the Publisher fires a transition, it may create/remove tasks in a local estimate of the network state. It is the CSM sub-system that disseminates and synchronizes this new information with the UAVs throughout the network. The details of the CSM implementation determine how far from reality the synchronous network assumption is. If CSM was developed poorly, information would never or only very slowly synchronize. Luckily, in simulations and demonstrations the current ad-hoc CSM has worked rather well. The future work listed in section 6.6 will involve incorporating a model, such as one for CSM, directly into the semantics of network-level control. The expanded semantics will give a more complete view of the network-level controller's interaction with the entire network.

As new tasks (e.g. ts^a) are added by the Publisher and distributed by CSM, the Collaborative Control System (CCS) determines which UAVs will be assigned to which tasks and then executes those tasks. This is done in a decentralized sub-optimal manner that can be executed on-line [75]. CCS causes these tasks to become 'done'. This information is sent from the UAVs back to the Publisher by CSM. When the Publisher's local estimate is updated to contain the completed tasks, a transition may become enabled and fire, creating more tasks to be disseminated by CSM and then completed by CCS.

The Publisher executes as a network-level controller, but it assumes that its local state estimate is the 'real' state of the network. The Publisher periodically (10 Hz) performs a loop that: checks for and applies up to 1 runtime patch, checks for and fires up to 1 enabled transition, waits and synchronizes information with the rest of the network. When it checks for any runtime patches, if one is waiting, it is applied and the response is issued. Next, the list of transitions is searched for any enabled transition. If one is found, it is fired. Only one transition is fired in any cycle. The list of transitions is cycled through in a deterministic top-to-bottom manner so the implementation may give preference to transitions listed higher. Lastly, the Publisher sits and waits while CSM synchronizes its state with the network. This should disseminate out any new tasks that were created while pulling in any information about old tasks that have been completed. So during any individual cycle it is possible that no events occur, causing the internal Publisher state to remain unaffected. It is also possible that a runtime patch, a transition firing, and task states being completed all occur causing several state transitions.

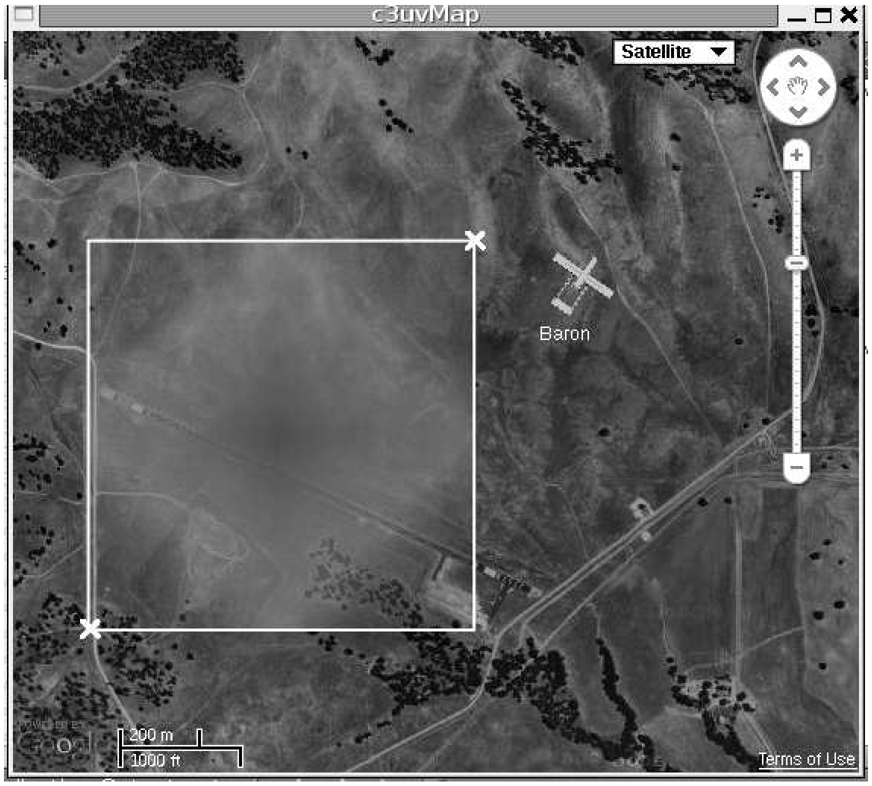


Figure 7.1: Screenshot of the Google Maps based graphical user interface (GUI).

The Publisher is connected to either web services or a graphical user interface (called the 'Commander GUI'). The Commander GUI displays the CSL information in a graphical manner. It is a Qt 4.0 GUI that uses an embedded Google Map display to show geographic data, figure 7.1. The GUI was developed to interact with a touchscreen monitor. All of the task definitions are drawn on the map and can be 'dragged' to modify their points. Once the task definition is modified, it can be 'sent' to the CSM and distributed throughout the network.

The web server implementation exchanged the XML-based CSL, but a web services based GUI has not yet been created. An iPhone/web server demonstration was performed in the Spring of 2008, but the iPhone textually specified the CSL instead of having a well developed GUI. This was much more of a proof-of-concept demonstration than a fully developed product.

In the different demonstrations imagery, such as figure 7.2, is collected and stored by the UAVs when executing tasks. These images can then be pulled down and

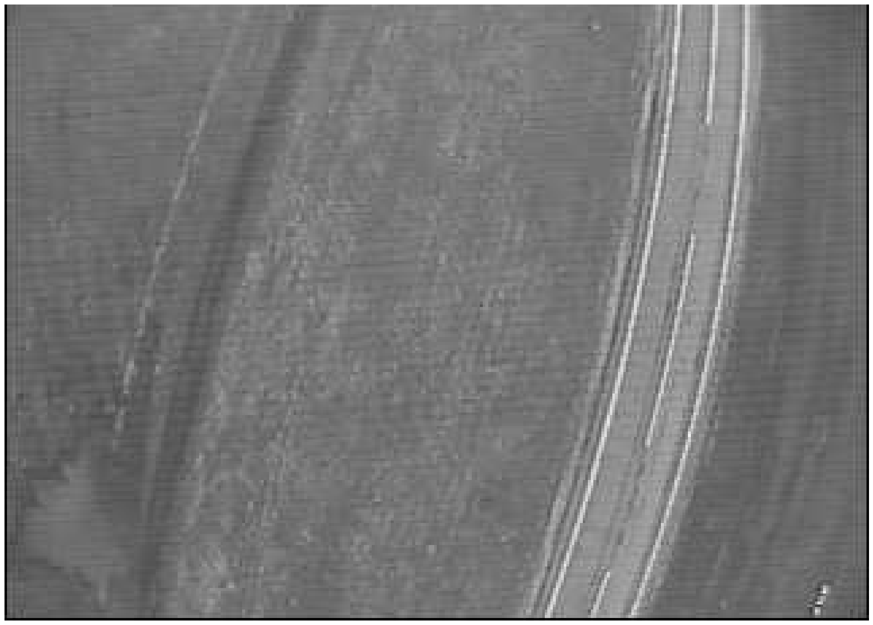


Figure 7.2: Real image collected from Camp Roberts, CA.

viewed. Alternatively, video streams are often recorded and streamed over RTP to human operators on the ground using the Video Lan Client software (VLC).

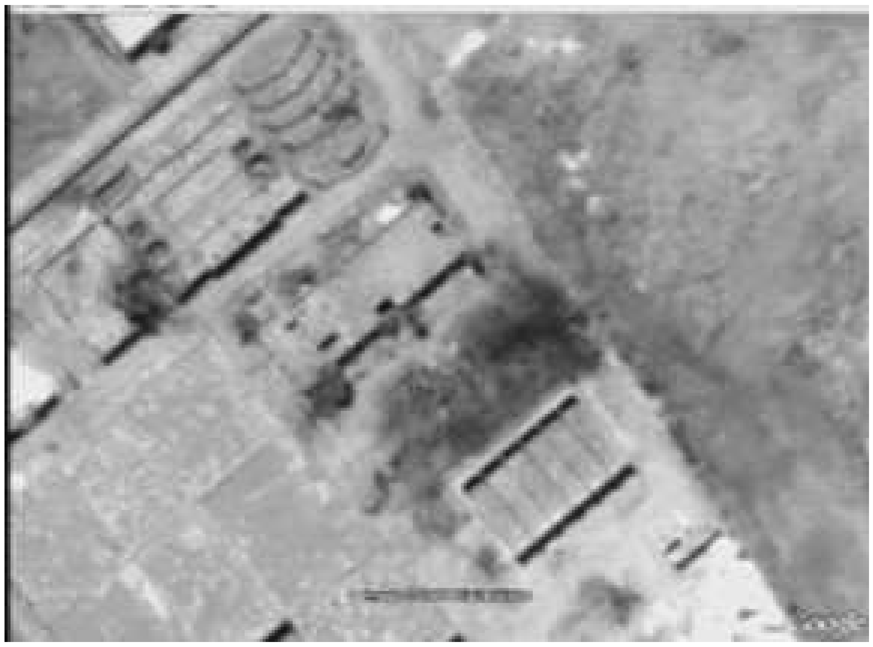


Figure 7.3: Simulated camera imagery using Google Earth.

Since it is not always possible to go out and physically fly, hardware-in-the-loop (HIL) simulations also allow experiments to be performed. HIL simulations use all of the real hardware, but fake the sensor inputs (GPS, gyros, etc.) to allow the UAV to believe it is flying and respond appropriately. A Google Earth based camera view simulator was created to fake the video stream as well, figure 7.3. This software took in the position and orientation of the UAV as well as relative location of the camera to appropriately position the Google Earth simulated camera. This allowed a 10 Hz fake video to be produced as if it was real video being generated by the UAV. This was used in an extensive simulation at Quantico, VA when a real demonstration was not possible due to security concerns.

7.2 C_3UV Platforms

The Publisher and CSL were developed to help control the UAV fleet at C_3UV. This fleet contains MLB Bat IVs, figure 7.4, and Sig Rascal 110s, figure 7.5.

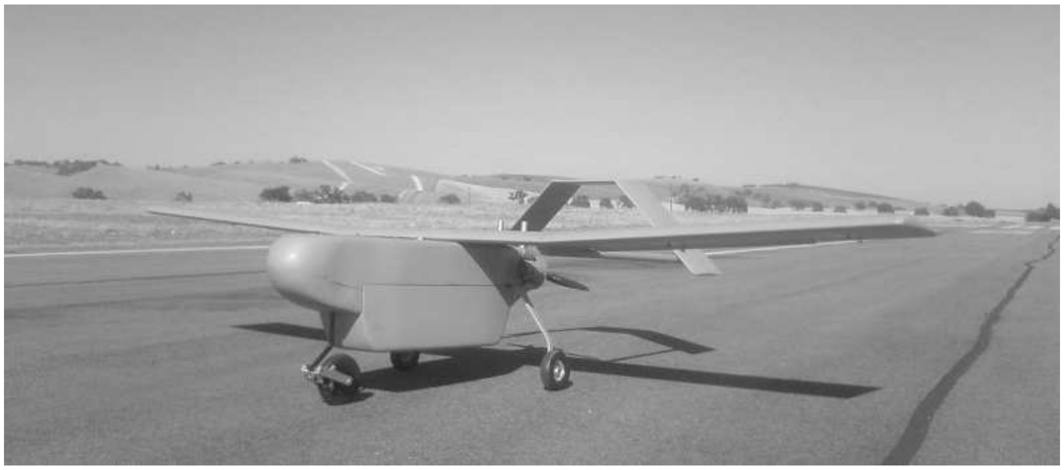


Figure 7.4: MLB Bat IV UAV

The Bat IV is the larger of the two UAV platforms. It has a 13 foot wingspan and a payload of approximately 50 pounds. Its normal cruising speed is around 60 mph. It has the potential for an integrated on-board generator to charge the batteries which power the electronics. With the generator functioning, a flight time of up to 8 hours could be achieved. Issues with the generator have lead to most experiments being performed without it, limiting the battery life and flight time to around 2 hours. The Bat's servos are controlled by the commercial Cloud Cap Piccolo II autopilot, figure 7.6. The Piccolo II performs all of the low-level autopilot control and estimation. The Bat also has 2 PC-104 stacks performing the CCS and CSM functions. The main benefit of the Bat IV during experiments has been its additional payload which can carry an extra PC-104 stack as well as a heavier gimbaled camera.

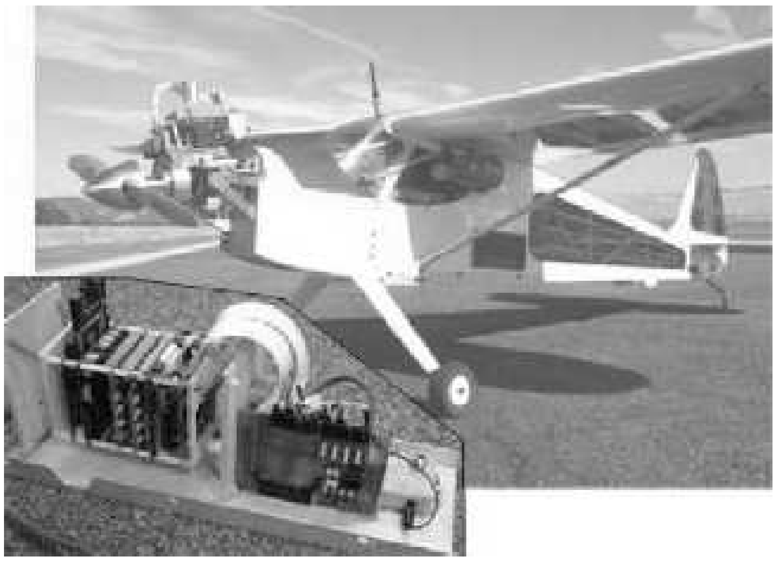


Figure 7.5: Sig Rascal 110 UAV and a single PC-104 stack

The Sig Rascal is the smaller, but more frequently used UAV platform. It is based on a hobbyist kit, making replacement parts cheaper and easier to acquire. It also requires significantly less preparation than the Bat IV as well as being easier to fly. Its 9 foot wingspan, 12 pound payload, 50 mph cruise speed, and 90 minute flight time are more than sufficient for most experiments. It also utilizes a Piccolo II autopilot that is integrated with a PC-104 stack. The Rascal is typically flown with either a fixed camera or a side-mounted gimbaled camera which has a restricted range of motion.

Figure 7.6: Cloud Cap Piccolo II autopilot

The PC-104 stack, shown in figure 7.5, is a computer of a special form factor that is easy to embed in the UAVs. Each PC-104 stack has a dual-core processor, most of which are 2.0 GHz Pentium processors. They also feature solid state flash memory hard drives (typically 4 GB or 8 GB) and a 802.11 b/g wireless connection boosted by a 1 watt amplifier. The 802.11 wireless connection is used by C_3UV to transmit all data, but the Piccolo II also has a 900 MHz link that allows for manual control of the UAV to be regained for take-off and landing.

7.3 XML-based Syntax

XML was chosen for specifying CSL based on it being 'language independent'. XML is itself a markup language, a language that is formed from standard ASCII text and used to create a rooted tree of nested tags. Most programming languages (C, C++, Java) can understand standard ASCII text and this allows XML parsers to be developed for use by these programming languages. This makes an XML representation usable by almost every programming language, thus XML is a 'language independent' information exchange format.

The CSL XML grammar can be specified using either an XML DTD or an XML schema. Both DTDs and schemas can specify the same XML, they are just different methods for doing so. Since DTDs are allegedly being deprecated, and since schemas allow for values to be typed, a schema will be used to specify CSL (example benefit of schema's typing: an altitude can be forced to be a positive number with a schema, but with a DTD it could have a value 'John'). The schema will be presented in chunks interleaved with a few CSL XML examples. The schema and the examples are both XML, so the schema will be presented with 'figures' and the examples with 'examples'. The typefaces should be distinctive enough to easily distinguish between the two.

The two fundamental parts of CSL are reading the network's state from the network-level controller and sending runtime patches to the network-level controller.

Figure 7.7 shows that the base CSL message can contain exactly one of: a "get_state" message, the corresponding "state_results" message, a "send_runtime_patch" message, or the corresponding "runtime_patch_results" message. Every CSL message is also given a timestamp to record when it was created.

A "get_state" message can be sent to the Publisher, which then reads the internal network-level controller's state and transforms it into a CSL "state_results" reply. This does not affect the Publisher's state, it simply transforms and outputs the network's current state. However, this state can be quite complex and lengthy. Example 8 shows a "get_state" CSL message. The first tag indicates that the message is XML. The second tag indicates that the message is CSL; it also has several attributes which are standard for all CSL messages noting which schema to use for validation. The attributes of the CSL tag lastly contain the timestamp for when the message was

```
<xs:element name="CSL">
  <xs:complexType>
    <xs:choice>
      <xs:element name="get_state"/>
      <xs:element name="state_results"
            type="state_results_type"/>
      <xs:element name="send_runtime_patch"
            type="send_runtime_patch_type"/>
      <xs:element name="runtime_patch_results"
            type="runtime_patch_results_type"/>
    </xs:choice>
    <xs:attribute name="timestamp" type="xs:dateTime"/>
  </xs:complexType>
</xs:element>
```

Figure 7.7: The root CSL element.

created. The content of the CSL message is the extremely simple "get_state" tag. Example 9 shows a portion of the response. The "..." would be filled in with several additional tags describing the current state. Figure 7.8 shows the content that replaces the "...". The XML and CSL tags take up a good deal of space and will be omitted from future examples, however all CSL messages do have these enclosing tags.

Example 8 *Get State Message Example.*

```
<?xml version="1.0" encoding="utf-8"?>
<CSL xmlns="http://www.c3uv.berkeley.edu"
xsi:schemaLocation="http://www.c3uv.berkeley.edu
file:///J:/Files/Thoughts/Thesis/XML%20Schema/CSL_schema.xsd"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
timestamp="2011-11-07T10:10:10">
      <get_state/>
</CSL>
```

Example 9 *State Results Message Example.*

```
<?xml version="1.0" encoding="utf-8"?>
<CSL xmlns="http://www.c3uv.berkeley.edu"
xsi:schemaLocation="http://www.c3uv.berkeley.edu
```

```
file:///J:/Files/Thoughts/Thesis/XML%20Schema/CSL_schema.xsd"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
timestamp="2011-11-07T10:10:10">
      <state_results>
         ...
      </state_results>
</CSL>
```

Continuing with figure 7.7, if a "send_runtime_patch" message is sent to the Publisher, the Publisher responds with a "runtime_patch_results" message. The runtime patches are applied and do update the internal network-level controller state. The CSL syntax for specifying runtime patches will be discussed in section 7.3.2; first the CSL state representation will be presented.

7.3.1 CSL's State Representation

```
<xs:complexType name="state_results_type">
  <xs:sequence>
    <xs:element name="petri_net" type="petri_net_type"/>
    <xs:element name="task_definitions" type="task_definitions_type"/>
    <xs:element name="task_states" type="task_states_type"/>
    <xs:element name="uav_definitions" type="uav_definitions_type"/>
    <xs:element name="uav_states" type="uav_states_type"/>
  </xs:sequence>
</xs:complexType>
```

Figure 7.8: The state output.

Figure 7.8 shows that the state results always include information about the Petri net, task definitions, task states, UAV definitions, and UAV states. The next subsections will discuss these elements in that order.

Petri net Representation

Figure 7.9 shows that the Petri net element contains only a timestamp for when the last runtime patch was applied, a list of places, a list of transitions, and a list of arcs. From definition 56 it is apparent that the arc weights, capacity constraints, associated task definitions, and markings are missing. They are not actually missing, but are embedded within the individual places and arcs. Every arc is supposed to be assigned a weight, and figure 7.12 shows that every arc does indeed have a weight assigned.

```
<xs:complexType name="petri_net_type">
  <xs:sequence>
    <xs:element name="last_update" type="xs:dateTime"/>
    <xs:element name="places" type="places_type"/>
    <xs:element name="transitions" type="transitions_type"/>
    <xs:element name="arcs" type="arcs_type"/>
  </xs:sequence>
</xs:complexType>
```

Figure 7.9: The Petri net representation.

Likewise, every place is supposed to have a capacity constraint, an associated task definition, and a marking. These are shown to be present in figure 7.10.

```
<xs:complexType name="places_type">
  <xs:sequence minOccurs="0" maxOccurs="unbounded">
    <xs:element name="place" type="place_type"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="place_type">
  <xs:sequence>
    <xs:element name="place_id" type="xs:string"/>
    <xs:element name="capacity" type="xs:positiveInteger"/>
    <xs:element name="associated_task_definition_id" type="xs:string"/>
    <xs:element name="marking" type="xs:nonNegativeInteger"/>
  </xs:sequence>
</xs:complexType>
```

Figure 7.10: The places representation.

The top of figure 7.10 shows that the set of places in the Petri net can contain zero or more individual places. This construct will appear often to show that a list of several individuals can occur.

The benefit of using a schema over a DTD can be seen by forcing the capacity constraint element in figure 7.10 to have a positive integer value. A DTD would only state that there must be a capacity element with some value, it could not type that value. An unfortunate drawback of this additional typing feature is that the grammar becomes even lengthier.

Since no information was associated to transitions, definition 56, it should not be

```
<xs:complexType name="transitions_type">
  <xs:sequence minOccurs="0" maxOccurs="unbounded">
    <xs:element name="transition" type="transition_type"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="transition_type">
  <xs:sequence>
    <xs:element name="transition_id" type="xs:string"/>
  </xs:sequence>
</xs:complexType>
```

Figure 7.11: The transitions representation.

surprising that CSL has no information other than an identifier for each transition.

```
<xs:complexType name="arcs_type">
  <xs:choice minOccurs="0" maxOccurs="unbounded">
    <xs:element name="arc" type="arc_type"/>
  </xs:choice>
</xs:complexType>

<xs:simpleType name="arc_direction_type">
  <xs:restriction base="xs:string">
    <xs:enumeration value="incoming"/>
    <xs:enumeration value="outgoing"/>
  </xs:restriction>
</xs:simpleType>

<xs:complexType name="arc_type">
  <xs:sequence>
    <xs:element name="arc_id" type="xs:string"/>
    <xs:element name="direction" type="arc_direction_type"/>
    <xs:element name="from_id" type="xs:string"/>
    <xs:element name="to_id" type="xs:string"/>
    <xs:element name="weight" type="xs:positiveInteger"/>
  </xs:sequence>
</xs:complexType>
```

Figure 7.12: The arcs representation.

Arcs contain both the direction they are pointing (incoming to a transition, or outgoing from a transition), a weight, and which elements they are connected to.

The CSL Petri net description for figure 6.4 would look like:

Example 10 *Petri net Example from figure 6.4.*

```
<petri_net>
  <last_update>2011-11-07T06:24:15</last_update>
  <places>
    <place>
      <place_id>P0</place_id>
      <capacity>10</capacity>
      <associated_task_definition_id>TD1
       </associated_task_definition_id>
      <marking>1</marking>
    </place>
    <place>
      <place_id>P1</place_id>
      <capacity>10</capacity>
      <associated_task_definition_id>TD2
       </associated_task_definition_id>
      <marking>0</marking>
    </place>
  </places>
  <transitions>
    <transition>
      <transition_id>T0</transition_id>
    </transition>
    <transition>
      <transition_id>T1</transition_id>
    </transition>
  </transitions>
  <arcs>
    <arc>
      <arc_id>A1</arc_id>
      <direction>outgoing</direction>
      <from_id>T0</from_id>
      <to_id>P1</to_id>
      <weight>1</weight>
    </arc>
    <arc>
      <arc_id>A2</arc_id>
      <direction>incoming</direction>
      <from_id>P1</from_id>
      <to_id>T1</to_id>
      <weight>1</weight>
    </arc>
    <arc>
      <arc_id>A3</arc_id>
      <direction>outgoing</direction>
      <from_id>T1</from_id>
      <to_id>P0</to_id>
      <weight>1</weight>
    </arc>
  </arcs>
</petri_net>
```

As is evident, the CSL description for even a simple Petri net becomes rather lengthy. This example Petri net has lists of 2 places, 2 transitions, and 3 arcs. It also contains the details for the places, transitions, and arcs.

Frequently Occurring Types

There are several types of values that will be used by CSL's task definitions, task states, UAV definitions, and UAV states. Creation of the types allows them to be specified once and used repeatedly.

Figure 7.13 shows three types that will be used for representing longitude, latitude, and altitude, respectively. Again, a benefit of schemas over DTDs is the ability to restrict the ranges to acceptable values.

Figure 7.14 uses the types from figure 7.13 to create common types for points and rotations. Any point will contain a latitude from -90.00 through 90.00, a longitude from -180.00 through 180.00, and an altitude that is positive. Since Euler angles are used by C_3UV (partially due to their use by the Cloudcap Piccolo autopilot) the yaw, pitch, and roll of the UAV are all expressed from -180.00 through 180.00.

Finally, figure 7.15 shows the representation of the probability grid that had to be integrated for the Summer 2009 and 2010 demonstrations to include distributed data fusion (DDF). A probability grid is made up of a list of individual cells. Each cell has a (x, y) grid coordinate as well as a probability value.

```
<xs:simpleType name="symmetric180_double">
  <xs:restriction base="xs:double">
    <xs:maxInclusive value="180.0000000"/>
    <xs:minInclusive value="-180.000000"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="symmetric90_double">
  <xs:restriction base="xs:double">
    <xs:maxInclusive value="90.0000000"/>
    <xs:minInclusive value="-90.000000"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="positive_double">
  <xs:restriction base="xs:double">
    <xs:minInclusive value="0.000000"/>
  </xs:restriction>
</xs:simpleType>
```

Figure 7.13: Common number types used by C_3UV.

```
<xs:complexType name="point">
  <xs:sequence>
    <xs:element name="latitude" type="symmetric90_double"/>
    <xs:element name="longitude" type="symmetric180_double"/>
    <xs:element name="altitude" type="positive_double"/>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="rotation">
  <xs:sequence>
    <xs:element name="yaw" type="symmetric180_double"/>
    <xs:element name="pitch" type="symmetric180_double"/>
    <xs:element name="roll" type="symmetric180_double"/>
  </xs:sequence>
</xs:complexType>
```

Figure 7.14: Common GPS points and rotations.

Task Definition Representation

The chapter 3 presentation of network-level control abstracted away most details about task types, task definitions, task states, UAV types, UAV definitions, and

```
<xs:complexType name="probability_grid_type">
  <xs:sequence>
    <xs:element name="cell" type="cell_type" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="cell_type">
  <xs:sequence>
    <xs:element name="x" type="xs:integer"/>
    <xs:element name="y" type="xs:integer"/>
    <xs:element name="probability" type="positive_double"/>
  </xs:sequence>
</xs:complexType>
```

Figure 7.15: Probability grid representation used for Distributed Data Fusion.

UAV states. CSL must contain all of this detail. This information is vital to and specifically tailored for C_3UV's ISR application. The task and UAV types determine the grammar for the task definitions, task states, UAV definitions, and UAV states. If CSL were to be altered for a different application, this content would need be modified appropriately.

```
<xs:simpleType name="task_types">
  <xs:restriction base="xs:string">
    <xs:enumeration value="visit_point"/>
    <xs:enumeration value="visit_line"/>
    <xs:enumeration value="visit_area"/>
    <xs:enumeration value="watch_point"/>
    <xs:enumeration value="watch_line"/>
    <xs:enumeration value="watch_area"/>
  </xs:restriction>
</xs:simpleType>
```

Figure 7.16: The list of task types.

Two fundamental types of tasks were identified early at C_3UV: visit and watch. They have also been called travel and track, respectively. The visit types of tasks are done once and completed automatically by the UAVs. The watch types of tasks are done indefinitely until the human operator determines that the task is complete. Both have come in 0, 1, and 2 dimensional variations, figure 7.16.

A visit point type of task results in a UAV simply flying over the point. This is the most commonly used task type in demonstrations. A visit line type of task causes a UAV to fly along a line segment once. A visit area type of task results in a lawn-mower pattern being flown to cover the area once.

A watch point type of task results in a UAV circling the point indefinitely. A watch line type of task causes a UAV to patrol a line segment repeatedly. A watch area type of task is connected to the recent distributed data fusion (DDF) developments. It was only recently integrated into the Summer 2009 and Summer 2010 demonstrations. The task is given an initial probability grid distribution (prior). The UAVs are flown based on an optimization algorithm that is seeking to minimize the entropy of the current probability grid distribution. As the UAVs observe the area, the probability grid evolves, changing the optimal trajectory for the UAV. The optimal trajectories are not always intuitively obvious. Again, the task is completed by the human operator, if left alone the UAVs would continue to re-localize the target.

Figure 7.17 shows the list of zero or more task definitions that is a part of the CSL state results from figure 7.8.

```
<xs:complexType name="task_definitions_type">
  <xs:choice minOccurs="0" maxOccurs="unbounded">
    <xs:group ref="task_definition_group"/>
  </xs:choice>
</xs:complexType>

<xs:group name="task_definition_group">
  <xs:choice>
    <xs:element name="visit_point_definition"
                type="visit_point_definition_type"/>
    <xs:element name="visit_line_definition"
                type="visit_line_definition_type"/>
    <xs:element name="visit_area_definition"
                type="visit_area_definition_type"/>
    <xs:element name="watch_point_definition"
                type="watch_point_definition_type"/>
    <xs:element name="watch_line_definition"
                type="watch_line_definition_type"/>
    <xs:element name="watch_area_definition"
                type="watch_area_definition_type"/>
  </xs:choice>
</xs:group>
```

Figure 7.17: The task definitions.

Figure 7.18 shows the details of the visit task definitions. A visit point requires a single point while a visit line requires two points. A visit area task requires the upper left and lower right corners of the rectangular area to be swept.

All of the task definitions have identifiers, descriptions, priorities, and timestamps for when they were last modified. They also have which task type they were based upon (providing the information represented by $taskType(td^1) = tt^2$ in chapter 3).

Figure 7.19 shows that the content for watch tasks is mostly the same as for visit tasks. However, since the watch area task is the CSL integration of DDF, it requires the number of cells in the grid as well as an initial prior. During demonstrations these probability grids were been kept small to reduce the amount of computation and communication required to keep them synchronized and optimized over.

Example 11 *Task Definition Example.*

```
<task_definitions>
  <visit_point_definition>
    <task_definition_id>TD1</task_definition_id>
    <task_type>visit_point</task_type>
    <description>Fly over the tree near the road.</description>
    <point>
      <latitude>35.734374</latitude>
      <longitude>-122.715324</longitude>
      <altitude>60.0</altitude>
    </point>
    <completion_radius>40.0</completion_radius>
    <priority>3</priority>
    <last_update>2010-01-04T06:13:44</last_update>
  </visit_point_definition>
</task_definitions>
```

Example 11 shows a simple task definitions state that contains only one task definition, a visit point task definition. The definition has all of the content required in figure 7.18.

Task State Representation

Similarly to how the CSL "state_results" contains a list of currently existing task definitions (TD from chapter 3), figure 7.20 shows that there can be zero or more currently existing task states as part of the overall CSL state (TS_k from chapter 3).

Figure 7.21 shows the visit task states corresponding to the definitions in figure 7.18. The visit task states contain an identifier, which UAVs are executing the task,

```
<xs:complexType name="visit_point_definition_type">
  <xs:sequence>
    <xs:element name="task_definition_id" type="xs:string"/>
    <xs:element name="task_type" type="task_types"
          fixed="visit_point"/>
    <xs:element name="description" type="xs:string"/>
    <xs:element name="point" type="point"/>
    <xs:element name="completion_radius" type="xs:positiveInteger"/>
    <xs:element name="priority" type="xs:positiveInteger"/>
    <xs:element name="last_update" type="xs:dateTime"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="visit_line_definition_type">
  <xs:sequence>
    <xs:element name="task_definition_id" type="xs:string"/>
    <xs:element name="task_type" type="task_types"
          fixed="visit_line"/>
    <xs:element name="description" type="xs:string"/>
    <xs:element name="point_a" type="point"/>
    <xs:element name="point_b" type="point"/>
    <xs:element name="priority" type="xs:positiveInteger"/>
    <xs:element name="last_update" type="xs:dateTime"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="visit_area_definition_type">
  <xs:sequence>
    <xs:element name="task_definition_id" type="xs:string"/>
    <xs:element name="task_type" type="task_types"
          fixed="visit_area"/>
    <xs:element name="description" type="xs:string"/>
    <xs:element name="upper_left_point" type="point"/>
    <xs:element name="bottom_right_point" type="point"/>
    <xs:element name="priority" type="xs:positiveInteger"/>
    <xs:element name="last_update" type="xs:dateTime"/>
  </xs:sequence>
</xs:complexType>
```

Figure 7.18: The visit tasks definitions.

```
<xs:complexType name="watch_point_definition_type">
  <xs:sequence>
    <xs:element name="task_definition_id" type="xs:string"/>
    <xs:element name="task_type" type="task_types"
          fixed="watch_point"/>
    <xs:element name="description" type="xs:string"/>
    <xs:element name="point" type="point"/>
    <xs:element name="priority" type="xs:positiveInteger"/>
    <xs:element name="last_update" type="xs:dateTime"/>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="watch_line_definition_type">
  <xs:sequence>
    <xs:element name="task_definition_id" type="xs:string"/>
    <xs:element name="task_type" type="task_types"
          fixed="watch_line"/>
    <xs:element name="description" type="xs:string"/>
    <xs:element name="point_a" type="point"/>
    <xs:element name="point_b" type="point"/>
    <xs:element name="priority" type="xs:positiveInteger"/>
    <xs:element name="last_update" type="xs:dateTime"/>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="watch_area_definition_type">
  <xs:sequence>
    <xs:element name="task_definition_id" type="xs:string"/>
    <xs:element name="task_type" type="task_types"
          fixed="watch_area"/>
    <xs:element name="description" type="xs:string"/>
    <xs:element name="upper_left_point" type="point"/>
    <xs:element name="lower_right_point" type="point"/>
    <xs:element name="number_horizontal_cells"
          type="xs:positiveInteger"/>
    <xs:element name="number_veritcal_cells"
          type="xs:positiveInteger"/>
    <xs:element name="initial_prior_distribution"
          type="probability_grid_type"/>
    <xs:element name="priority" type="xs:positiveInteger"/>
    <xs:element name="last_update" type="xs:dateTime"/>
  </xs:sequence>
</xs:complexType>
```

Figure 7.19: The watch tasks definitions.

```
<xs:complexType name="task_states_type">
  <xs:choice minOccurs="0" maxOccurs="unbounded">
    <xs:group ref="task_state_group"/>
  </xs:choice>
</xs:complexType>
<xs:group name="task_state_group">
  <xs:choice>
    <xs:element name="visit_point_state"
                type="visit_point_state_type"/>
    <xs:element name="visit_line_state"
                type="visit_line_state_type"/>
    <xs:element name="visit_area_state"
                type="visit_area_state_type"/>
    <xs:element name="watch_point_state"
                type="watch_point_state_type"/>
    <xs:element name="watch_line_state"
                type="watch_line_state_type"/>
    <xs:element name="watch_area_state"
                type="watch_area_state_type"/>
  </xs:choice>
</xs:group>
```

Figure 7.20: The task states.

its current status (mode that should eventually become 'done'), and a timestamp for the last update. All task states also record the task definition the task state is based upon (providing the information represented by $taskDef(ts^1) = td^3$ in chapter 3).

Figure 7.22 shows the content of the watch task states. They contain information similar to the visit task states, except the watch area state also contains a current probability distribution. If this distribution contains a large number of cells it will take up a very significant portion of the CSL message. For example a 20x20 gird will produce 400 cells resulting in a long, long list like:

Example 12 *Probability Distribution Cells.*

```
...
<cell>
  <x>14</x>
  <y>5</y>
  <probability>0.025</probability>
</cell>
...
```

```
<xs:complexType name="visit_point_state_type">
  <xs:sequence>
    <xs:element name="task_state_id" type="xs:string"/>
    <xs:element name="task_definition_id" type="xs:string"/>
    <xs:element name="assigned_uav_state_id" type="xs:string"
                minOccurs="0" maxOccurs="unbounded"/>
    <xs:element name="current_status" type="xs:string"/>
    <xs:element name="last_update" type="xs:dateTime"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="visit_line_state_type">
  <xs:sequence>
    <xs:element name="task_state_id" type="xs:string"/>
    <xs:element name="task_definition_id" type="xs:string"/>
    <xs:element name="assigned_uav_state_id" type="xs:string"
                minOccurs="0" maxOccurs="unbounded"/>
    <xs:element name="current_status" type="xs:string"/>
    <xs:element name="last_update" type="xs:dateTime"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="visit_area_state_type">
  <xs:sequence>
    <xs:element name="task_state_id" type="xs:string"/>
    <xs:element name="task_definition_id" type="xs:string"/>
    <xs:element name="assigned_uav_state_id" type="xs:string"
                minOccurs="0" maxOccurs="unbounded"/>
    <xs:element name="current_status" type="xs:string"/>
    <xs:element name="last_update" type="xs:dateTime"/>
  </xs:sequence>
</xs:complexType>
```

Figure 7.21: The visit tasks states.

```
<xs:complexType name="watch_point_state_type">
  <xs:sequence>
    <xs:element name="task_state_id" type="xs:string"/>
    <xs:element name="task_definition_id" type="xs:string"/>
    <xs:element name="assigned_uav_state_id" type="xs:string"
                minOccurs="0" maxOccurs="unbounded"/>
    <xs:element name="current_status" type="xs:string"/>
    <xs:element name="last_update" type="xs:dateTime"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="watch_line_state_type">
  <xs:sequence>
    <xs:element name="task_state_id" type="xs:string"/>
    <xs:element name="task_definition_id" type="xs:string"/>
    <xs:element name="assigned_uav_state_id" type="xs:string"
                minOccurs="0" maxOccurs="unbounded"/>
    <xs:element name="current_status" type="xs:string"/>
    <xs:element name="last_update" type="xs:dateTime"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="watch_area_state_type">
  <xs:sequence>
    <xs:element name="task_state_id" type="xs:string"/>
    <xs:element name="task_definition_id" type="xs:string"/>
    <xs:element name="assigned_uav_state_id" type="xs:string"
                minOccurs="0" maxOccurs="unbounded"/>
    <xs:element name="current_probability_distribution"
                type="probability_grid_type"/>
    <xs:element name="current_status" type="xs:string"/>
    <xs:element name="last_update" type="xs:dateTime"/>
  </xs:sequence>
</xs:complexType>
```

Figure 7.22: The watch tasks states.

UAV Definition Representation

The UAV definitions and UAV states are not directly used in the Publisher (UD and US were not part of $(NLC)_k$ in chapter 3). Although the UAV information is not needed to fire transitions, it is included in the CSL state. The human operator will generally want this information to better understand the behavior of the network.

```
<xs:simpleType name="uav_types">
  <xs:restriction base="xs:string">
    <xs:enumeration value="bat"/>
    <xs:enumeration value="rascal"/>
  </xs:restriction>
</xs:simpleType>
```

Figure 7.23: The list of UAV types.

Figure 7.23 lists the two types of integrated C_3UV UAVs: the MLB Bat IV, and the Sig Rascal. A Zagi delta-wing has been in development for a few years, but has never been integrated into the larger C_3UV system. The Zagi has not yet demonstrated a fully functioning autopilot and integrated on-board computing. Eventually these requirements may be fulfilled and CSL may need to be extended to include the Zagi platform.

```
<xs:complexType name="uav_definitions_type">
  <xs:choice minOccurs="0" maxOccurs="unbounded">
    <xs:group ref="uav_definition_group"/>
  </xs:choice>
</xs:complexType>

<xs:group name="uav_definition_group">
  <xs:choice>
   <xs:element name="bat_definition" type="bat_definition_type"/>
   <xs:element name="rascal_definition"type="rascal_definition_type"/>
  </xs:choice>
</xs:group>
```

Figure 7.24: The list of UAV definitions.

Figure 7.24 again shows that there can be zero or more UAV definitions as part of the overall CSL state.

The Sig Rascal and Bat IV aircraft have been used interchangably with the main benefit of the Bat IV being the additional computing power on-board and longer

```
<xs:complexType name="bat_definition_type">
  <xs:sequence>
    <xs:element name="uav_definition_id" type="xs:string"/>
    <xs:element name="uav_type" type="uav_types" fixed="bat"/>
    <xs:element name="callsign" type="xs:string"/>
    <xs:element name="color" type="xs:string" minOccurs="0"/>
    <xs:element name="ip_address" type="xs:string"
     minOccurs="0" maxOccurs="unbounded"/>
    <xs:element name="video_port" type="xs:string" minOccurs="0"/>
    <xs:element name="lost_communication_waypoint" type="point"
     minOccurs="0"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="rascal_definition_type">
  <xs:sequence>
    <xs:element name="uav_definition_id" type="xs:string"/>
    <xs:element name="uav_type" type="uav_types" fixed="rascal"/>
    <xs:element name="callsign" type="xs:string"/>
    <xs:element name="color" type="xs:string" minOccurs="0"/>
    <xs:element name="ip_address" type="xs:string"
     minOccurs="0" maxOccurs="unbounded"/>
    <xs:element name="video_port" type="xs:string" minOccurs="0"/>
    <xs:element name="lost_communication_waypoint" type="point"
     minOccurs="0"/>
  </xs:sequence>
</xs:complexType>
```

Figure 7.25: The UAV definitions for a bat and a rascal.

duration. Their definitions contain identifiers, callsigns, the color of the platform (optional), the IP addresses (optional), the port for video streaming (optional), and potentially a lost communication waypoint if one was set. Additionally they list their UAV type (providing the information represented by $uavType(ud^3) = ut^1$ in chapter 3).

UAV State Representation

As was mentioned, the CSL state may include a list of several currently existing UAV states (US_k from chapter 3). These are the states for zero or more Bat IV and Sig Rascal UAVs, figure 7.26.

```
<xs:complexType name="uav_states_type">
  <xs:choice minOccurs="0" maxOccurs="unbounded">
    <xs:group ref="uav_state_group"/>
  </xs:choice>
</xs:complexType>

<xs:group name="uav_state_group">
  <xs:choice>
    <xs:element name="bat_state" type="bat_state_type"/>
    <xs:element name="rascal_state" type="rascal_state_type"/>
  </xs:choice>
</xs:group>
```

Figure 7.26: The list of UAV states.

The individual UAV states contain an identifier, the associated UAV definition (to represent $uavDef(us^1) = ud^6$ from chapter 3), the position of the UAV, the orientation of the UAV, the airspeed, the task currently assigned, and a timestamp recording how fresh the data is, figure 7.27.

Looking back at figure 7.8, all of the information presented so far is a part of the CSL “state_results”. A single “get_state” request will result in the entire network’s state being returned. The CSL “state_results” message provides a ’language independent’ translation of the network-level controller’s internal state.

7.3.2 CSL’s Runtime Patches

The previous CSL content was used to communicate the internal state of the Publisher (network-level controller). The CSL “send_runtime_patch” message is used to apply patches which do modify the Publisher’s internal state according to chapter 5.

Figure 7.28 shows all of the runtime patches defined in chapter 5. Only 12 are listed, while there were 16 in chapter 5. The difference is that adding places, adding arcs, deleting arcs, and modifying arc weights came in two varieties in 6.2. Here the added place can optionally have a task definition inside, the add arc patch works for both types of arcs (incoming and outgoing), the delete arc patch works for both types of arcs (incoming and outgoing), and the modify weights patch works for both types of arcs (incoming and outgoing). Including these ’dual’ options, all 16 runtime patches are accounted for.

Figure 7.29 shows that the “add_place” patch can be optionally given a new task definition to add. To delete a place, only the place_id is required. Adding a new

```
<xs:complexType name="bat_state_type">
  <xs:sequence>
    <xs:element name="uav_state_id" type="xs:string"/>
    <xs:element name="uav_definition_id" type="xs:string"/>
    <xs:element name="position" type="point"/>
    <xs:element name="orientation" type="rotation"/>
    <xs:element name="air_speed" type="positive_double"/>
    <xs:element name="assigned_task_state_id" type="xs:string"
          minOccurs="0"/>
    <xs:element name="last_update" type="xs:dateTime"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="rascal_state_type">
  <xs:sequence>
    <xs:element name="uav_state_id" type="xs:string"/>
    <xs:element name="uav_definition_id" type="xs:string"/>
    <xs:element name="position" type="point"/>
    <xs:element name="orientation" type="rotation"/>
    <xs:element name="air_speed" type="positive_double"/>
    <xs:element name="assigned_task_state_id" type="xs:string"
          minOccurs="0"/>
    <xs:element name="last_update" type="xs:dateTime"/>
  </xs:sequence>
</xs:complexType>
```

Figure 7.27: The states for the Bat IV and Sig Rascal.

transition requires no information. While deleting a transition on requires a transition identifier.

Figure 7.30 illustrates that to add an arc the direction must be specified along with the identifiers for which place and transition the arc is connecting. To delete the arc only an arc identifier is required. To modify the arc weight an identifier and a desired weight are necessary.

Figure 7.31 shows the CSL for modifying a capacity constraint, adding a token, and deleting a token. All are very straightforward.

The "modify_task_definition" patch allows a task definition, specified by its identifier, to have its content manually replaced. This is what happens when a human operator 'moves' a visit point task definition. Figure 7.32 also shows how task states are similarly manually updated.

```
<xs:complexType name="send_runtime_patch_type">
  <xs:choice>
   <xs:element name="add_place" type="add_place_type"/>
   <xs:element name="delete_place" type="delete_place_type"/>
   <xs:element name="add_transition" type="add_transition_type"/>
   <xs:element name="delete_transition"
    type="delete_transition_type"/>
   <xs:element name="add_arc" type="add_arc_type"/>
   <xs:element name="delete_arc" type="delete_arc_type"/>
   <xs:element name="modify_weight" type="xs:positiveInteger"/>
   <xs:element name="modify_capacity" type="xs:positiveInteger"/>
   <xs:element name="add_token" type="add_token_type"/>
   <xs:element name="delete_token" type="delete_token_type"/>
   <xs:element name="modify_task_definition"
     type="modify_task_definition_type"/>
    <xs:element name="modify_task_state"
     type="modify_task_state_type"/>
  </xs:choice>
</xs:complexType>
```

Figure 7.28: The runtime patches to send.

```
<xs:complexType name="add_place_type">
  <xs:sequence minOccurs="0" maxOccurs="1">
    <xs:group ref="task_definition_group"/>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="delete_place_type">
  <xs:sequence>
    <xs:element name="place_id" type="xs:string"/>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="add_transition_type">
</xs:complexType>
<xs:complexType name="delete_transition_type">
  <xs:sequence>
    <xs:element name="transition_id" type="xs:string"/>
  </xs:sequence>
</xs:complexType>
```

Figure 7.29: The runtime patches for adding/deleting places and transitions.

```
<xs:complexType name="add_arc_type">
  <xs:sequence>
    <xs:element name="direction" type="arc_direction_type"/>
    <xs:element name="from_id" type="xs:string"/>
    <xs:element name="to_id" type="xs:string"/>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="delete_arc_type">
  <xs:sequence>
    <xs:element name="arc_id" type="xs:string"/>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="modify_weight_type">
  <xs:sequence>
    <xs:element name="arc_id" type="xs:string"/>
    <xs:element name="desired_weight" type="xs:positiveInteger"/>
  </xs:sequence>
</xs:complexType>
```

Figure 7.30: The runtime patches for adding/deleting/modifying arcs.

```
<xs:complexType name="modify_capacity_type">
  <xs:sequence>
    <xs:element name="place_id" type="xs:string"/>
    <xs:element name="desired_capacity" type="xs:positiveInteger"/>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="add_token_type">
  <xs:sequence>
    <xs:element name="place_id" type="xs:string"/>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="delete_token_type">
  <xs:sequence>
    <xs:element name="place_id" type="xs:string"/>
  </xs:sequence>
</xs:complexType>
```

Figure 7.31: The runtime patches for modifying capacity constraints and adding/deleting tokens.

```
<xs:complexType name="modify_task_definition_type">
  <xs:sequence>
    <xs:element name="task_definition_id" type="xs:string"/>
    <xs:group ref="task_definition_group"/>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="modify_task_state_type">
  <xs:sequence>
    <xs:element name="task_state_id" type="xs:string"/>
    <xs:group ref="task_state_group"/>
  </xs:sequence>
</xs:complexType>
```

Figure 7.32: The runtime patches for modifying task definitions and task states.

When a runtime patch is applied, a response is issued to indicate that the patch has been successfully processed. Figure 7.33 shows what that response could entail. When a new place is added, the place_id is the response. Similarly for transitions, arcs, task definitions, and task states being added. When an arc weight or capacity constraint is updated, the new value is the response. There is also a default "error" message, which currently contains no content, but indicates that something went wrong.

Compared to the CSL state, runtime patches are rather small and simple messages. Example 13 shows the entirety of CSL message to add a transition. Example 14 shows the simple result returning the transition identifier for the new transition.

Example 13 *A CSL add transition runtime patch.*

```
<?xml version="1.0" encoding="utf-8"?>
<CSL xmlns="http://www.c3uv.berkeley.edu"
xsi:schemaLocation="http://www.c3uv.berkeley.edu
file:///J:/Files/Thoughts/Thesis/XML%20Schema/CSL_schema.xsd"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
timestamp="2011-11-07T10:10:10">
  <send_runtime_patch>
      <add_transition/>
  </send_runtime_patch>
</CSL>
```

Example 14 *Results of adding a transition.*

```
<?xml version="1.0" encoding="utf-8"?>
<CSL xmlns="http://www.c3uv.berkeley.edu"
xsi:schemaLocation="http://www.c3uv.berkeley.edu
file:///J:/Files/Thoughts/Thesis/XML%20Schema/CSL_schema.xsd"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
timestamp="2011-11-07T10:10:10">
  <runtime_patch_results>
      <transition_id>T3</transition_id>
    </runtime_patch_results>
</CSL>
```

```
<xs:complexType name="runtime_patch_results_type">
  <xs:sequence>
    <xs:choice minOccurs="0" maxOccurs="unbounded">
      <xs:element name="place_id" type="xs:string"/>
      <xs:element name="transition_id" type="xs:string"/>
      <xs:element name="arc_id" type="xs:string"/>
      <xs:element name="updated_weight" type="xs:positiveInteger"/>
      <xs:element name="updated_capacity" type="xs:positiveInteger"/>
      <xs:element name="task_definition_id" type="xs:string"/>
      <xs:element name="task_state_id" type="xs:string"/>
      <xs:element name="error" type="xs:string"/>
    </xs:choice>
  </xs:sequence>
</xs:complexType>
```

Figure 7.33: The confirmation for a runtime patch.

Chapter 8

Conclusions

This dissertation has developed the concept of network-level controllers. Network-level controllers are intended to interact with a task-based collaborative control system at a very high level of abstraction. The network-level controller monitors the tasks in the system. It also orders and automates the insertion and removal of tasks. When tasks are completed they may be automatically removed and new tasks may be automatically created. The network-level controller allows a human operator to specify the desired network behavior, thus the name 'network-level control'.

The network-level controller model of computation is an extension of Petri nets. Petri nets provide a concise and network-focused representation. This allows an intuitive graphical representation of the network's state. This representation shows which tasks currently exist, which task definitions they are based upon, and what ordering constraints apply for all current and future tasks.

As a human operator observes the network's behavior, he/she might decide to modify the intended network behavior. Runtime patching allows modifications to be made to the network-level controller in a manner that guarantees that the resulting network-level controller is well formed, fully representative, and task-token consistent.

The Collaborative Sensing Language (CSL) is an XML-based implementation of network-level control for Intelligence Surveillance and Reconnaissance (ISR) applications by UAVs. It enabled network-level controllers for the Center for Collaborative Control of Unmanned Vehicles (C_3UV).

8.1 Contributions

The syntax and semantics for network-level controllers were presented and are themselves novel contributions. Petri nets have had extremely limited use in mobile robotics and their application to the high-level control of multiple UAVs is entirely original.

Additionally, several invariant properties of network-level controllers were identified and proven. These invariant properties only depend upon the semantics of network-level control. They are true for all network-level controllers. Other provable properties like boundedness, liveness, and deadlock were identified. These properties depend upon the specifics of any given network-level controller.

The syntax and semantics for runtime patching are also contributions. The runtime patching language allows a human operator to manipulate the network-level controller during execution.

Several invariant properties for network-level controllers with runtime patching were also proven. These guarantee that the network-level controller always remains correctly defined after all possible runtime patches.

The Collaborative Sensing Language (CSL) and its implementation are also contributions. CSL was developed as an XML-based specification of network-level controllers for Intelligence Surveillance and Reconnaissance (ISR) applications by UAVs.

8.2 Critiques

One of the most obvious assumptions made in this dissertation is the existence of an underlying task-based collaborative control system. Tasks are not the only method for collaboration. The concept of network-level control does not apply to networks with other methods of collaboration. Swarm-based or emergent behaviors are common examples of collaborative networks that are not based upon tasks and a network-level controller would have no well defined method for interacting with these networks.

In section 6.6 the synchronous network assumption was mentioned. In this dissertation it was assumed that there was a set of task definitions TD and task states TS that could be read from and written to instantaneously. In a 'real' UAV network this assumption is unrealistic. The system is inherently asynchronous and distributed. Every physical element of the system may have a local estimate for what these sets and their values are, but it takes a well developed method of communication and synchronization to guarantee that any changes made by the network-level controller will be fully propagated to all elements of the network. Assuming a synchronous network prevented this dissertation from having to address these additional complicating factors that are themselves areas of on-going research. In this way, network-level control could be the focus of the discussion.

8.3 Directions for Future Work

Network-level controllers could be created for non-ISR applications or for non-UAV applications. This would not involve drastic modifications to the concepts of network-level control, but merely the investigation into which applications it may be appropriate for. As a specific example that diverges greatly from UAVs, the use of network-level control could be useful within traditional workflow management. This would allow a business manager to create, monitor, and modify the workflow process in an on-line manner. As the work force changed or new important orders were placed, the network-level controller could be modified to 'adjust' the company's behavior to deal with a changing business environment.

More importantly, the synchronous network assumption should be further investigated. A specific asynchronous communication mechanism and scheme could be assumed. These assumptions' impacts on the guarantees and operation of a network-level controller could then be investigated. It is likely that additional assumptions must be made on the behavior of the asynchronous network in order for the network-level controller to operate correctly. For instance, if the network-level controller creates a new task state, but the communication infrastructure never disseminates the information, there is no possibility for a UAV to ever execute it. This direction of research would seek to clarify the additional assumptions necessary so that any asynchronous communication schemes satisfying these assumptions could be used in conjunction with a network-level controller.

Bibliography

[1] PATH Website. http://www.path.berkeley.edu/, October 2010.

[2] SHIFT Website. http://path.berkeley.edu/SHIFT/, October 2010.

[3] DESUMA Website. http://www.eecs.umich.edu/umdes/toolboxes.html, May 2011.

[4] The Edinburgh Concurrency Workbench - Website. http://homepages.inf.ed.ac.uk/perdita/cwb/, January 2011.

[5] Formal Systems - Website. http://www.fsel.com/index.html, January 2011.

[6] Integrated Net Analyzer, September 2011.

[7] LoLA - A Low Level Petri Net Analyser, September 2011.

[8] mCRL2: Analyzing System Behaviour, April 2011.

[9] Model Checking @CMU, June 2011.

[10] ON-THE-FLY, LTL MODEL CHECKING with SPIN, June 2011.

[11] The Place for Communicating Processes. http://www.wotug.org/occam/, February 2011.

[12] Platform Independent Petri net Editor 2, September 2011.

[13] A Spatial Logic Model Checker for Concurrency, Distribution and Mobility. http://ctp.di.fct.unl.pt/SLMC/, January 2011.

[14] Supremica- A Tool for Verification and Synthesis of Discrete Event Supervisors. http://academic.research.microsoft.com/Publication/5501279/supremica-a-tool-for-verification-and-synthesis-of-discrete-event-supervisors, May 2011.

[15] TIme petri Net Analyzer, September 2011.

[16] UMDES Website. http://www.eecs.umich.edu/umdes/toolboxes.html, May 2011.

[17] UPPAAL Home, August 2011.

[18] U.S. Military Unmanned Aerial Vehicles (UAVs) Market Forecast 2010-2015. Website, March 2011.

[19] Knut Akesson, Martin Fabian, and Hugo Flordal. Supremica in a Nutshell. October 2007.

[20] Robert Alexander, Martin Hall-May, and Tim Kelly. Certification of Autonomous Systems. In *2nd SEAS DTC Technical Conference*, 2007.

[21] R. Alur, T.A. Henzinger, and Pei-Hsin Ho. Automatic symbolic verification of embedded systems. *IEEE Transactions on Software Engineering*, 22(3):181 –201, mar 1996.

[22] R. Alur, T.A. Henzinger, G. Lafferriere, and G.J. Pappas. Discrete abstractions of hybrid systems. *Proceedings of the IEEE*, 88(7):971 –984, jul 2000.

[23] Rajeev Alur, T. Dang, Joel Esposito, Rafael Fierro, Yerang Hur, Franjo Ivancic, Vijay Kumar, Insup Lee, P. Mishra, George Pappas, and Oleg Sokolsky. Hierarchical Hybrid Modeling of Embedded Systems. In *Lecture Notes in Computer Science*. Springer-Verlap, 2001.

[24] Rajeev Alur, Radu Grosu, Yerang Hur, Vijay Kumar, and Insup 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 Berlin / Heidelberg, 2000.

[25] Rajeev Alur, Franjo Ivancic, Jesung Kim, Insup Lee, and Oleg Sokolsky. Generating Embedded Software from Hierarchical Hybrid Models. In *LCTES '03*. ACM, June 2003.

[26] Paul C. Attie and Nancy A. Lynch. Dynamic Input/Output Automata: A Formal Model for Dynamic Systems. In Kim Larsen and Mogens Nielsen, editors, *CONCUR 2001 Concurrency Theory*, volume 2154 of *Lecture Notes in Computer Science*, pages 137–151. Springer Berlin / Heidelberg, 2001.

[27] Andrea Balluchi, Antonio Bicchi, and Philippe Soueres. Path-Following with a Bounded-Curvature Vehicle: a Hybrid Control Approach. *International Journal of Control*, 78:1228–1247, 2005.

[28] Andrea Balluchi, Philippe Soures, and Antonio Bicchi. Hybrid Feedback Control for Path Tracking by a BoundedCurbature Vehicle. In Maria Di Benedetto and Alberto Sangiovanni-Vincentelli, editors, *Hybrid Systems: Computation and Control*, volume 2034 of *Lecture Notes in Computer Science*, pages 133–146. Springer Berlin / Heidelberg, 2001.

[29] Samary Baranov. Synthesis of Control Units for Mobile Robots. In *Proceedings., Second EUROMICRO workshop on Advanced Mobile Robots*, pages 80 –86, oct 1997.

[30] Magali Barbier and Elodie Chanthery. Autonomous mission management for unmanned aerial vehicles. *Aerospace Science and Technology*, 8(4):359 – 368, 2004.

[31] John Bastian, Arvind Savargaonkar, S. Venkateswaren, K.Ramoji Rao, T.V.V.S. Nagesh, and K.S. Raghunthan. State Machine Design-An Interactive Approach. In *Proceedings of the Fourth CSI/IEEE International Symposium on VLSI Design*, pages 41 –44, January 1991.

[32] S. Bayraktar, G.E. Fainekos, and G.J. Pappas. Experimental cooperative control of fixed-wing unmanned aerial vehicles. In *Decision and Control, 2004. CDC. 43rd IEEE Conference on*, volume 4, pages 4292 – 4298 Vol.4, dec. 2004.

[33] Gerd Behrmann, Alexandre David, and Kim Larsen. *A Tutorial on Uppaal.* Department of Computer Science Aalborg University, Denmark, 2011.

[34] Calin Belta, Antonio Bicchi, Magnus Egerstedt, Emilio Frazzoli, Eric Klavins, and George Pappas. Symbolic planning and control of robot motion [Grand Challenges of Robotics]. *Robotics Automation Magazine, IEEE*, 14(1):61 –70, March 2007.

[35] J. A. Bergstra and J. W. Klop. Algebra of communicating processes with abstraction. *Theoretical Computer Science*, 37:77 – 121, 1985.

[36] J. A. Bergstra and C. A. Middelburg. Process algebra for hybrid systems. *Theoretical Computer Science*, 335(2-3):215 – 280, 2005. Process Algebra.

[37] J.A. Bergstra and J.W. Klop. Process Algebra for Synchronous Communication. *Information and Control*, 60(1-3):109 – 137, 1984.

[38] Christopher Bolkcom. Homeland Security: Unmanned Aerial Vehicles and Border Surveillance. Technical Report ADA477712, Library of Congress, Washington DC Congressional Research Service, http://handle.dtic.mil/100.2/ADA477712, June 2004.

[39] Joao Borges de Sousa and G. Goncalves. Mixed Initiative Control of Unmanned Air and Ocean Going Vehicles: Models, Tools and Experimentation. Ada478701, UNIVERSIDADE FACULDADE DE ENGENHARIA DO PORTO (PORTUGAL) DEPARTAMENTO DE ENGENHARIA MECANICA E GESTAO INDUSTRIAL, http://handle.dtic.mil/100.2/ADA478701, November 2007.

[40] Joao Borges de Sousa, G. Goncalves, A. Costa, and J. Morgado. Mixed-initiative Control of Unmanned Air Vehicle Systems: the PITVANT R&D UAV Program.

[41] Joo Borges de Sousa, Karl H. Johansson, Jorge Silva, and Alberto Speranzon. A Verified Hierarchical Control Architecture for Co-ordinated Multi-vehicle Operations. *International Journal of Adaptive Control and Signal Processing*, 21(2-3):159–188, 2007.

[42] Julian Bradfield and Colin Stirling. Modal Logics and mu-Calculi: An Introduction. Technical report, Laboratory for Foundations of Computer Science, University of Edinburgh, 2001.

[43] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A Theory of Communicating Sequential Processes. *J. ACM*, 31:560–599, June 1984.

[44] Lus Caires. Behavioral and Spatial Observations in a Logic for the pi-Calculus. In Igor Walukiewicz, editor, *Foundations of Software Science and Computation Structures*, volume 2987 of *Lecture Notes in Computer Science*, pages 72–89. Springer Berlin / Heidelberg, 2004.

[45] Lus Caires and Luca Cardelli. A Spatial Logic for Concurrency (Part II). In Lubo Brim, Mojmr Kretnsk, Antonn Kucera, and Petr Jancar, editors, *CONCUR 2002 Concurrency Theory*, volume 2421 of *Lecture Notes in Computer Science*, pages 163–199. Springer Berlin / Heidelberg, 2002.

[46] Lus Caires and Luca Cardelli. A spatial logic for concurrency (part I). *Information and Computation*, 186(2):194 – 235, 2003. Theoretical Aspects of Computer Software (TACS 2001).

[47] Christos Cassandras and Stephane Lafortune. *Introduction to Discrete Event Systems*. Springer, second edition, 2008.

[48] Walid Chainbi, Chihab Hanachi, and Christophe Sibertin-Blanc. The Multi-agent Prey/Predator problem: A Petri net solution. In *IMACS Multiconference Computational Engineering in Systems Applications (CESA)*, 1996.

[49] Randy Cieslak, C. Desclaux, A.S. Fawaz, and Pravin Varaiya. Supervisory Control of Discrete-Event Processes with Partial Observations. *IEEE Transactions on Automatic Control*, 33(3):249 –260, mar 1988.

[50] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic Verification of Finite-State Concurrent Systems using Temporal Logic Specifications. *ACM Trans. Program. Lang. Syst.*, 8:244–263, April 1986.

[51] David T. Cole, Salah Sukkarieh, and Ali Haydar Gktogan. System development and demonstration of a UAV control architecture for information gathering missions. *Journal of Field Robotics*, 23(6-7):417–440, 2006.

[52] K.L.B. Cook. The Silent Force Multiplier: The History and Role of UAVs in Warfare. In *2007 IEEE Aerospace Conference*, pages 1 –7, March 2007.

[53] Petru Corts, Luis Alejandro andEles and Zebo Peng. Modeling and formal verification of embedded systems based on a Petri net representation. *Journal of Systems Architecture*, 49(12-15):571 – 598, 2003. Synthesis and Verification.

[54] M.A Cuijpers, P.J.L. and.Reniers. Hybrid process algebra. *Journal of Logic and Algebraic Programming*, 62(2):191 – 245, 2005.

[55] Akash Deshpande. AHS Components in SHIFT. Technical report, California Partners for Automated Transit and Highways, 1997.

[56] Akash Deshpande, Aleks Gollu, and Luigi Semenzato. The SHIFT Programming Language and Run-Time System for Dynamic Networks of Hybrid Automata. Technical report, California PATH Program, 1997.

[57] Akash Deshpande, Aleks Gollu, and Luigi Semenzato. *Shift Reference Manual*. California PATH Program, January 1997.

[58] Akash Deshpande, Aleks Gollu, and Luigi Semenzato. The SHIFT Programming Langauge for Dynamic Networks of Hybrid Automata. In *IEEE Transactions on Automatic Control*, volume 43. April 1998.

[59] Akash Deshpande, Aleks Gollu, and Pravin Varaiya. SHIFT: A Formalism and a Programming Language for Dynamic Networks of Hybrid Automata. In *Hybrid Systems IV*, volume 1273 of *Lecture Notes in Computer Science*, pages 113–133. Springer-Verlag, 1997.

[60] Srinivas Devadas. Approaches to Multi-Level Sequential Logic Synthesis. In *Proceedings of the 26th ACM/IEEE Design Automation Conference*, DAC '89, pages 270–276, New York, NY, USA, 1989. ACM.

[61] Ekaterina Dolginova and Nancy Lynch. Safety verification for automated platoon maneuvers: A case study. In Oded Maler, editor, *Hybrid and Real-Time Systems*, volume 1201 of *Lecture Notes in Computer Science*, pages 154–170. Springer Berlin / Heidelberg, 1997. 10.1007/BFb0014723.

[62] Haiqiang Dun, Haiying Xu, and Lifu Wang. Transformation of BPEL Processes to Petri Nets. *Theoretical Aspects of Software Engineering, Joint IEEE/IFIP Symposium on*, 0:166–173, 2008.

[63] Abdulla Eid. Finite ω-Automata and Buchi Automata. University of Illinois, CS475 Project, May 2009.

[64] Yoichiro Endo, Douglas C. MacKenzie, and Ronald C. Arkin. Usability Evaluation of High-Level User Assistance for Robot Mission Specification. *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews*, 34(2):168 –180, may 2004.

[65] Joaquin Ezpeleta, Jose Manuel Colom, and Javier Martinez. A Petri net based deadlock prevention policy for flexible manufacturing systems. *Robotics and Automation, IEEE Transactions on*, 11(2):173 –184, apr 1995.

[66] xianwen Fang, zhicai Xu, and zhixiang Yin. Distributed Processing Based on Timed Petri Nets. In *Third International Conference on Natural Computation*, volume 5, pages 287 –291, aug. 2007.

[67] Rafael Fierro, Aveek Das, John Spletzer, Joel Esposito, Vijay Kumar, James Ostrowski, Georgre Pappas, Camillo Taylor, Yerang Hur, Rajeev Alur, Insup Lee, Greg Grudic, and Ben Southall. A Framework and Architecture for Multi-Robot Coordination. *The International Journal of Robotics Research*, 21(10-11):977–995, 2002.

[68] Formal Systems (Europe) Ltd. and Oxford University Computing Laboratory. *Failures-Divergence Refinement: FDR2 User Manual*, October 2010.

[69] Gene Franklin, J. David Powell, and Abbas Emami-Naeini. *Feedback Control of Dynamic Systems*. Pearson Prentice Hall, fifth edition, 2006.

[70] Emilio Frazzoli, Munther A. Dahleh, and Eric Feron. Maneuver-Based Motion Planning for Nonlinear Systems With Symmetries. *IEEE Transactions on Robotics*, 21(6):1077 – 1091, dec. 2005.

[71] A. R. Girard, J. Borges de Sousa, and J. K. Herdrick. A selection of recent advances in networked multivehicle systems. *Proceedings of the Institution of Mechanical Engineers, Part I: Journal of Systems and Control Engineering*, 219:1–14, 2005.

[72] Anouck Girard, Karl Hedrick, and Joo Tasso de Figueiredo Borges de Sousa. A hierarchical control architecture for mobile offshore bases. *Marine Structures*, 13(4-5):459 – 476, 2000.

[73] Anouck R. Girard and J. Karl Hedrick. Formation control of multiple vehicles using dynamic surface control and hybrid systems. *International Journal of Control*, 76:913–923(11), 1 June 2003.

[74] Alain Girault, Bilung Lee, and Edward A. Lee. Hierarchical Finite State Machines with Multiple Concurrency Models. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(6):742 –760, jun 1999.

[75] M.F. Godwin, S. Spry, and J.K. Hedrick. Distributed collaboration with limited communication using mission state estimates. In *American Control Conference, 2006*, page 7 pp., june 2006.

[76] C. H. Golaszewski and P. J. Ramadge. Control of Discrete Event Processes with Forced Events. In *26th IEEE Conference on Decision and Control*, volume 26, pages 247 –251, dec. 1987.

[77] Aleks Gollu and Mikhail Kourjanski. Object-oriented design of automated highway simulations using the SHIFT programming language. In *Proc. IEEE Conf. Intelligent Transportation System ITSC '97*, pages 141–146, 1997.

[78] B. Grocholsky, J. Keller, V. Kumar, and G. Pappas. Cooperative air and ground surveillance. *Robotics Automation Magazine, IEEE*, 13(3):16 –25, sept. 2006.

[79] Ben Grocholsky, Alexei Makarenko, Tobias Kaupp, and Hugh Durrant-Whyte. Scalable Control of Decentralised Sensor Platforms. In Feng Zhao and Leonidas Guibas, editors, *Information Processing in Sensor Networks*, volume 2634 of *Lecture Notes in Computer Science*, pages 551–551. Springer Berlin / Heidelberg, 2003.

[80] David Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8(3):231 – 274, 1987.

[81] David Harel and Amnon Naamad. The STATEMATE Semantics of Statecharts. *ACM Transactions on Software Engineering Methodology*, 5:293–333, October 1996.

[82] Thomas Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. *A User Guide to HyTech*, October 1996.

[83] Tom Henzinger. HyTech: The HYbrid TECHnology tool, August 2011.

[84] Katrina Herrick. Development of the unmanned aerial vehicle market: forecasts and trends. *Air & Space Europe*, 2(2):25–27, 2000.

[85] M. G. Hinchey, C. A. Rouff, J. L. Rash, and W. F. Truszkowski. Requirements of an integrated formal method for intelligent swarms. In *Proceedings of the 10th international workshop on Formal methods for industrial critical systems*, FMICS '05, pages 125–133, New York, NY, USA, 2005. ACM.

[86] Sebastian Hinz, Karsten Schmidt, and Christian Stahl. Transforming BPEL to Petri Nets. In *Business Process Management*, volume 3649 of *Lecture Notes in Computer Science*, pages 220–235. Springer Berlin / Heidelberg, 2005.

[87] Kunihiko Hiraishi. A Petri-net-based model for the mathematical analysis of multi-agent systems. In *Systems, Man, and Cybernetics, 2000 IEEE International Conference on*, volume 4, pages 3009 –3014 vol.4, 2000.

[88] Kunihiko Hiraishi. A formalism for decentralized control of discrete event systems. In *SICE 2002. Proceedings of the 41st SICE Annual Conference*, volume 1, pages 272 – 277 vol.1, aug. 2002.

[89] C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[90] C.A.R. Hoare. *Communicating Sequential Processes.* International Series in Computer Science. Prentice Hall International, 1985.

[91] L. E. Holloway, B. H. Krogh, and A. Giua. A Survey of Petri Net Methods for Controlled Discrete Event Systems. *Discrete Event Dynamic Systems*, 7:151–190, 1997. 10.1023/A:1008271916548.

[92] M. Ani Hsieh, Anthony Cowley, James F. Keller, Luiz Chaimowicz, Ben Grocholsky, Vijay Kumar, Camillo J. Taylor, Yoichiro Endo, Ronald C. Arkin, Boyoon Jung, Denis F. Wolf, Gaurav S. Sukhatme, and Douglas C. MacKenzie. Adaptive teams of autonomous aerial and ground robots for situational awareness. *Journal of Field Robotics*, 24(11-12):991–1014, 2007.

[93] Yerang Hur, Rafael Fierro, and Insup Lee. Modeling distributed autonomous robots using CHARON: formation control case study. In *Proc. Sixth IEEE Int Object-Oriented Real-Time Distributed Computing Symp*, pages 93–96, 2003.

[94] M. Iordache and P. Antsaklis. Supervision Based on Place Invariants: A Survey. *Discrete Event Dynamic Systems*, 16:451–492, 2006. 10.1007/s10626-006-0021-9.

[95] M.V. Iordache and P.J. Antsaklis. Decentralized supervision of Petri nets. *Automatic Control, IEEE Transactions on*, 51(2):376 – 381, feb. 2006.

[96] Kurt Jensen. Coloured petri nets and the invariant-method. *Theoretical Computer Science*, 14(3):317 – 336, 1981.

[97] Kurt Jensen. Coloured Petri nets. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Petri Nets: Central Models and Their Properties*, volume 254 of *Lecture Notes in Computer Science*, pages 248–299. Springer Berlin / Heidelberg, 1987. 10.1007/BFb0046842.

[98] Kurt Jensen. A brief introduction to coloured Petri Nets. In Ed Brinksma, editor, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1217 of *Lecture Notes in Computer Science*, pages 203–208. Springer Berlin / Heidelberg, 1997. 10.1007/BFb0035389.

[99] Kurt Jensen, Lars Kristensen, and Lisa Wells. Coloured Petri Nets and CPN Tools for modelling and validation of concurrent systems. *International Journal on Software Tools for Technology Transfer (STTT)*, 9:213–254, 2007. 10.1007/s10009-007-0038-x.

[100] Yan Jin, Ali A. Minai, and Marios M. Polycarpou. Cooperative Real-Time Search and Task Allocation in UAV Teams. In *Proceedings of 42nd IEEE Conference on Decision and Control*, volume 1, pages 7 – 12 Vol.1, dec. 2003.

[101] K.H. Johansson, J. Lygeros, S. Sastry, and M. Egerstedt. Simulation of Zeno hybrid automata. In *Proceedings of the 38th IEEE Conference on Decision and Control*, volume 4, pages 3538 –3543 vol.4, 1999.

[102] Daniel Karlsson, Petru Eles, and Zebo Peng. Formal verification of systemc designs using a petri-net based representation. In *Proceedings of the conference on Design, automation and test in Europe: Proceedings*, DATE '06, pages 1228–1233, 3001 Leuven, Belgium, Belgium, 2006. European Design and Automation Association.

[103] Yonit Kestin, Amir Pnueli, and Li-on Raviv. Algorithmic Verification of Linear Temporal Logic Specifications. In Kim Larsen, Sven Skyum, and Glynn Winskel, editors, *Automata, Languages and Programming*, volume 1443 of *Lecture Notes in Computer Science*, pages –. Springer Berlin / Heidelberg, 1998.

[104] YoungWoo Kim, Tatsuya Kato, Shigeru Okuma, and Tatsuo Narikiyo. Traffic Network Control Based on Hybrid Dynamical System Modeling and Mixed Integer Nonlinear Programming With Convexity Analysis. *IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and Humans*, 38(2):346 –357, march 2008.

[105] Eric Klavins, Robert Ghrist, and David Lipsky. A grammatical approach to self-organizing robotic systems. *IEEE Transactions on Automatic Control*, 51(6):949 – 962, june 2006.

[106] Michael Kohler, Daniel Moldt, and Heiko Rolke. Modelling the Structure and Behaviour of Petri Net Agents. In Jos-Manuel Colom and Maciej Koutny, editors, *Applications and Theory of Petri Nets 2001*, volume 2075 of *Lecture Notes in Computer Science*, pages 224–241. Springer Berlin / Heidelberg, 2001.

[107] Konstantinos Koutroumpas and John Lygeros. Modeling and verification of stochastic hybrid systems using HIOA: a case study on DNA replication. In *Proceedings of the 13th ACM international conference on Hybrid systems: computation and control*, HSCC '10, pages 263–272, New York, NY, USA, 2010. ACM.

[108] M.A. Kovacina, D. Palmer, Guang Yang, and R. Vaidyanathan. Multi-agent control algorithms for chemical cloud detection and mapping using unmanned air vehicles. In *Intelligent Robots and Systems, 2002. IEEE/RSJ International Conference on*, volume 3, pages 2782 – 2788 vol.3, 2002.

[109] Fabian Kratz, Oleg Sokolsky, George Pappas, and Insup Lee. R-Charon, a Modeling Language for Reconfigurable Hybrid Systems. In *Lecture Notes in Computer Science*, pages 392–406. Springer-Verlag, 2006.

[110] Hadas Kress-Gazit, Georgios E. Fainekos, and George J. Pappas. Temporal-Logic-Based Reactive Mission and Motion Planning. *IEEE Transactions on Robotics*, 25(6):1370 –1381, dec. 2009.

[111] Stephane Lafortune and Demosthenis Teneketzis. *UMDES-LIB: Library of Commands for Discrete Event Systems Modeled by Finite State Machines.* DES Group, University of Michigan, August 2000.

[112] Timo Latvala. Model Checking LTL Properties of High-Level Petri Nets with Fairness Constraints. In Josa-Manuel Colom and Maciej Koutny, editors, *Applications and Theory of Petri Nets 2001*, volume 2075 of *Lecture Notes in Computer Science*, pages 242–262. Springer Berlin / Heidelberg, 2001.

[113] Bilung Lee and Edward A. Lee. Interaction of Finite State Machines and Concurrency Models. In *Signals, Systems Computers, 1998. Conference Record of the Thirty-Second Asilomar Conference on*, volume 2, pages 1715 –1719 vol.2, nov 1998.

[114] David Lee and Mihalis Yannakakis. Principles and Methods of Testing Finite State Machines-A Survey. *Proceedings of the IEEE*, 84(8):1090 –1123, aug 1996.

[115] Edward A. Lee. What's ahead for embedded software? *Computer*, 33(9):18 –26, sep 2000.

[116] Jing Liu and Houshang Darabi. Ramadge-Wonham Supervisory Control of Mobile Robots: Lessons from Practice. In *IEEE International Conference on Robotics and Automation (ICRA)*, volume 1, pages 670 – 675 vol.1, 2002.

[117] Lennart Ljung. *System Identification: Theory for the User*. Prentice Hall, second edition, 99.

[118] Niels Lohmann. A Feature-Complete Petri Net Semantics for WS-BPEL2.0. In Marlon Dumas and Reiko Heckel, editors, *Web Services and Formal Methods*, volume 4937 of *Lecture Notes in Computer Science*, pages 77–91. Springer Berlin / Heidelberg, 2008.

[119] Panagiotis Louridas. Orchestrating Web Services with BPEL. *Software, IEEE*, 25(2):85 –87, march-april 2008.

[120] Joshua Love, Jerry Jariyasunant, Eloi Pereira, Marco Zennaro, Karl Hedrick, Christoph Kirsch, and Raja Sengupta. CSL: A Language to Specify and Re-specify Mobile Sensor Network Behaviors. *Real-Time and Embedded Technology and Applications Symposium, IEEE*, 0:67–76, 2009.

[121] Martin Lundell, Jingpeng Tang, and Kendall Nygard. Fuzzy Petri net for UAV decision making. In *Proceedings of the 2005 International Symposium on Collaborative Technologies and Systems*, pages 347 –352, may 2005.

[122] John Lygeros and Nancy Lynch. Conditions for Safe Deceleration of Strings of Vehicles. Institute of transportation studies, research reports, working papers, proceedings, Institute of Transportation Studies, UC Berkeley, 2000.

[123] John Lygeros, Claire Tomlin, and Shankar Sastry. *Hytbrid Systems: Modeling, Analysis and Control*. 2008.

[124] Nancy Lynch, Roberto Segala, and Frits Vaandrager. Hybrid I/O Automata Revisited. In Maria Di Benedetto and Alberto Sangiovanni-Vincentelli, editors, *Hybrid Systems: Computation and Control*, volume 2034 of *Lecture Notes in Computer Science*, pages 403–417. Springer Berlin / Heidelberg, 2001.

[125] Nancy Lynch, Roberto Segala, Frits Vaandrager, and H. Weinberg. Hybrid I/O automata. In Rajeev Alur, Thomas Henzinger, and Eduardo Sontag, editors, *Hybrid Systems III*, volume 1066 of *Lecture Notes in Computer Science*, pages 496–510. Springer Berlin / Heidelberg, 1996. 10.1007/BFb0020971.

[126] Nancy A. Lynch and Mark R. Tuttle. An Introduction to Input/Output Automata. Technical Report MIT/LCS/TM-373, Massachusetts Institute of Technology, September 1988.

[127] Douglas C. MacKenzie, Ronald Arkin, and Jonathan M. Cameron. Multiagent Mission Specification and Execution. *Autonomous Robots*, 4:29–52, 1997. 10.1023/A:1008807102993.

[128] Ernst W. Mayr. An algorithm for the general Petri net reachability problem. In *Proceedings of the thirteenth annual ACM symposium on Theory of computing*, STOC '81, pages 238–246, New York, NY, USA, 1981. ACM.

[129] John-Michael McNew and Eric Klavins. Locally Interacting Hybrid Systems with Embedded Graph Grammars. In *45th IEEE Conference on Decision and Control*, pages 6080 –6087, dec. 2006.

[130] John-Michael McNew and Eric Klavins. A Grammatical Approach to Cooperative Control. In Don Grundel, Robert Murphey, Panos Pardalos, and Oleg Prokopyev, editors, *Cooperative Systems*, volume 588 of *Lecture Notes in Economics and Mathematical Systems*, pages 117–138. Springer Berlin Heidelberg, 2007.

[131] Robin Milner. A Calculus of Communicating Systems. *Lecture Notes in Computer Science*, 92, 1980.

[132] Robin Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, 25(3):267 – 310, 1983.

[133] Robin Milner. *Communicating and Mobile Systems: the pi-Calculus.* Cambridge University Press, 1999.

[134] Robin Milner, Joachim Parrow, and David Walker. A Calculus of Mobile Processes, Part II. Technical report, University of Edinburgh, 1990.

[135] Robin Milner, Joachim Parrow, and David Walker. A Calculus of Mobile Processes, I. *Information and Computation*, 100(1):1 – 40, 1992.

[136] Ian Mitchell, August 2011.

[137] Tadao Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541 –580, apr 1989.

[138] Nicholas G. Odrey and Gonzalo Meja. A re-configurable multi-agent system architecture for error recovery in production systems. *Robotics and Computer-Integrated Manufacturing*, 19(1-2):35 – 43, 2003.

[139] Katsuhiko Ogata. *Discrete-Time Control Systems.* Prentice Hall, second edition, 1995.

[140] Katsuhiko Ogata. *System Dynamics.* Pearson Prentice Hall, fourth edition, 2004.

[141] P. Oliveira, A. Pascoal, V. Silva, and C. Silvestre. Design, development, and testing at sea of the mission control system for the MARIUS autonomous underwater vehicle. In *Conference Proceedings OCEANS MTS/IEEE. 'Prospects for the 21st Century'.*, volume 1, pages 401 –406 vol.1, sep 1996.

[142] P. Oliverira, C. Silvestre, P. Aguiar, and A. Pascoal. Guidance and control of the SIRENE underwater vehicle: from system design to tests at sea. In *OCEANS '98 Conference Proceedings*, volume 2, pages 1043 –1048 vol.2, sep-1 oct 1998.

[143] Chun Ouyang, Eric Verbeek, Wil van der Aalst, Stephan Breutel, Marlon Dumas, and Arthur ter Hofstede. WofBPEL: A Tool for Automated Analysis of BPEL Processes. In *Service-Oriented Computing - ICSOC 2005*, volume 3826 of *Lecture Notes in Computer Science*, pages 484–489. Springer Berlin / Heidelberg, 2005.

[144] Pier Palamara, Vittorio Ziparo, Luca Iocchi, Daniele Nardi, and Pedro Lima. Teamwork Design Based on Petri Net Plans. In Luca Iocchi, Hitoshi Matsubara, Alfredo Weitzenfeld, and Changjiu Zhou, editors, *RoboCup 2008: Robot Soccer World Cup XII*, volume 5399 of *Lecture Notes in Computer Science*, pages 200–211. Springer Berlin / Heidelberg, 2009.

[145] Narcis Palomeras, Marc Carreras, Pere Ridao, and Emili Hernandez. Mission control system for dam inspection with an AUV. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2551 –2556, oct. 2006.

[146] Joachim Parrow. *Handbook of Process Algebra*, chapter 8: An Introduction to the pi-Calculus, pages 479 – 544. Elsevier Science, 2001.

[147] James Pasley. How BPEL and SOA are changing Web services development. *Internet Computing, IEEE*, 9(3):60 – 67, may-june 2005.

[148] M. C. L. Patterson, A. Mulligan, J. Douglas, J. Robinson, and J. S. Pallister. Volcano Surveillance by ACR Silver Fox. In *AIAA infotech@aerospace.* AIAA, Sept 2005.

[149] Carl Petri. Communication with Automata. Technical Report 1, Griffiss Air Force Base, September 1966.

[150] Orjan Pettersen. A configuration tool for process oriented UAV programming. Master's thesis, University of Tromso, Department of Computer Science, June 2010.

[151] Amir Pnueli. The Temporal Logic of Programs. *Annual IEEE Symposium on Foundations of Computer Science*, 0:46–57, 1977.

[152] Wendy Pyper. Population survey pilots an unmanned aircraft. *Australian Antarctic Magazine*, 2008.

[153] P. J. Ramadge and W. M. Wonham. Supervisory Control of a Class of Discrete Event Processes. In A. Bensoussan and J. Lions, editors, *Analysis and Optimization of Systems*, volume 63 of *Lecture Notes in Control and Information Sciences*, pages 475–498. Springer Berlin / Heidelberg, 1984. 10.1007/BFb0006306.

[154] Peter J. Ramadge. Observability of discrete event systems. In *25th IEEE Conference on Decision and Control*, volume 25, pages 1108 –1112, dec. 1986.

[155] P.J.G. Ramadge and W.M. Wonham. The Control of Discrete Event Systems. *Proceedings of the IEEE*, 77(1):81 –98, jan 1989.

[156] A. Ramirez-Serrano, S.C. Zhu, and B. Benhabib. Moore Automata for the Supervisory Control of Robotic Manufacturing Workcells. *Autonomous Robots*, 9:59–69, 2000. 10.1023/A:1008976319182.

[157] A. Rango, A. Laliberte, C. Steele, J. E. Herrick, B. Bestelmeyer, T. Schmugge, A. Roanhorse, and V. Jenkins. Using Unmanned Aerial Vehicles for Rangelands: Current Applications and Future Proposals. *Environmental Practice*, 8:159–168, 2006.

[158] L. Ricker, S. Lafortune, and S. Gene. DESUMA: A Tool Integrating GIDDES and UMDES. In *Discrete Event Systems, 2006 8th International Workshop on*, pages 392 –393, july 2006.

[159] William Rounds and Hosung Song. The -Calculus: A Language for Distributed Control of Reconfigurable Embedded Systems. In Oded Maler and Amir Pnueli, editors, *Hybrid Systems: Computation and Control*, volume 2623 of *Lecture Notes in Computer Science*, pages 435–449. Springer Berlin / Heidelberg, 2003.

[160] I. Rubin, A. Behzad, Huei-Jiun Ju, R. Zhang, X. Huang, Y. Liu, and R. Khalaf. Ad Hoc Wireless Networks with Mobile Backbones. In *15th IEEE International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC), 2004.*, volume 1, pages 566 – 573 Vol.1, 2004.

[161] Karen Rudie and W.M. Wonham. Think Globally, Act Locally: Decentralized Supervisory Control. *IEEE Transactions on Automatic Control*, 37(11):1692 –1708, nov 1992.

[162] A. Ryan, J. Tisdale, M. Godwin, D. Coatta, D. Nguyen, S. Spry, R. Sengupta, and J.K. Hedrick. Decentralized Control of Unmanned Aerial Vehicle Collaborative Sensing Missions. In *American Control Conference, 2007. ACC '07*, pages 4672 –4677, july 2007.

[163] Allison Ryan, David Nguyen, and Karl Hedrick. Hybrid Control for UAV-Assisted Search and Rescue. In *ASME 2005 International Mechanical Engineering Congress and Exposition*. ASME, November 2005.

[164] Khodakaram Salimifard and Mike Wright. Petri net-based modelling of workflow systems: An overview. *European Journal of Operational Research*, 134(3):664 – 676, 2001.

[165] Rupa Sampath, Houshang Darabi, Ugo Buy, and Liu Jing. Control Reconfiguration of Discrete Event Systems With Dynamic Control Specifications. *Automation Science and Engineering, IEEE Transactions on*, 5(1):84 –100, jan. 2008.

[166] Alan C. Shaw. Communicating Real-Time State Machines. *IEEE Transactions on Software Engineering*, 18(9):805 –816, sep 1992.

[167] Michael Sipser. *Introduction to the Theory of Computation*. Course Technology, 2nd edition, 2006.

[168] Oleg Sokolsky and George Pappas. Platform-Independent Autonomy Modeling. Technical report, Department of Computer and Information Science, University of Pennsylvania, 2004.

[169] Joao Sousa, Tunc Simsek, and Pravin Varaiya. Task Planning and Execution for UAV Teams. In *43rd IEEE Conference on Decision and Control (CDC)*, volume 4, pages 3804 – 3810 Vol.4, dec. 2004.

[170] Suman Srinivasan, Haniph Latchman, John Shea, Tan Wong, and Janice McNair. Airborne traffic surveillance systems: video surveillance of highway traffic. In *Proceedings of the ACM 2nd international workshop on Video surveillance & sensor networks*, VSSN '04, pages 131–135, New York, NY, USA, 2004. ACM.

[171] Perdita Stevens and Faron Moller. *The Edinburgh Concurrency Workbench user manual (Version 7.1)*. Laboratory for Foundations of Computer Science, University of Edinburgh, 07 1999.

[172] Colin Stirling and David Walker. Local model checking in the modal mu-calculus. In Josep Daz and Fernando Orejas, editors, *TAPSOFT '89*, volume 351 of *Lecture Notes in Computer Science*, pages 369–383. Springer Berlin Heidelberg, 1989.

[173] Technische Universiteit Eindhoven. *mCRL2 User Manual*, July 2010.

[174] Dave Thomas and Andy Hunt. State Machines. *IEEE Software*, 19(6):10 – 12, nov/dec 2002.

[175] Bernardo Toninho and Lus Caires. A Spatial-Epistemic Logic and Tool for Reasoning about Security Protocols. Technical report, CITI and Faculdade de Ciencias e Tecnologia, Universidade Nova de Lisboa, 2010.

[176] W. van der Aalst. Workflow Verification: Finding Control-Flow Errors Using Petri-Net-Based Techniques. In Wil van der Aalst, Jarg Desel, and Andreas Oberweis, editors, *Business Process Management*, volume 1806 of *Lecture Notes in Computer Science*, pages 19–128. Springer Berlin / Heidelberg, 2000.

[177] Hans Vangheluwe and Juan de Lara. Computer automated multi-paradigm modelling for analysis and design of traffic networks. In *Proceedings of the 36th conference on Winter simulation*, WSC '04, pages 249–258. Winter Simulation Conference, 2004.

[178] H. M. W. Verbeek and W. M. P. van der Aalst. Analyzing BPEL processes using Petri nets. In *Florida International University*, pages 59–78, 2005.

[179] Hugo Vieira and Luis Caires. *Spatial Logic Model Checker User's Guide version 1.15.* Departamento de Informatica, FCT/UNL, June 2009.

[180] N. Viswanadham and Y. Narahari. Coloured Petri net models for automated manufacturing systems. In *Robotics and Automation. Proceedings. 1987 IEEE International Conference on*, volume 4, pages 1985 – 1990, mar 1987.

[181] V. Volovoi. Modeling of system reliability Petri nets with aging tokens. *Reliability Engineering & System Safety*, 84(2):149 – 161, 2004.

[182] Adam Watts, Scott Bowman, Amr Abd-Elrahman, Ahmed Mohamed, Benjamin Wilkinson, John Perry, Youssef Kaddoura, and Kyuho Lee. Unmanned Aircraft Systems (UASs) for Ecological Research and Natural-Resource Monitoring. *Ecological Restoration*, 26:13–14, 2008.

[183] Johannes Weidl, Rene Klosch, Georg Trausmuth, and Harald Gall. Facilitating Program Comprehension via Generic Components for State Machines. In *Proceedings of the 5th International Workshop on Program Comprehension (WPC '97)*, pages 118–, Washington, DC, USA, 1997. IEEE Computer Society.

[184] Gera Weiss and Rajeev Alur. Automata Based Interfaces for Control and Scheduling. In Alberto Bemporad, Antonio Bicchi, and Giorgio Buttazzo, editors, *Hybrid Systems: Computation and Control*, volume 4416 of *Lecture Notes in Computer Science*, pages 601–613. Springer Berlin / Heidelberg, 2007.

[185] Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction*. Foundations of Computing. The MIT Press, 1993.

[186] W. Wonham and P. Ramadge. Modular Supervisory Control of Discrete-Event Systems. *Mathematics of Control, Signals, and Systems (MCSS)*, 1:13–30, 1988. 10.1007/BF02551233.

[187] W. M. Wonham and P. J. Ramadge. On the Supremal Controllable Sublanguage of a Given Language. *SIAM Journal of Control and Optimization*, 25(3):637–659, 1987.

[188] Naiqi Wu. Necessary and sufficient conditions for deadlock-free operation in flexible manufacturing systems using a colored Petri net model. *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, 29(2):192 –204, may 1999.

[189] Yali Wu, Weimin Wu, Jianchao Zeng, Guoji Sun, Hongye Su, and Jian Chu. Modeling and simulation of hybrid dynamical systems with generalized differential Petri nets. In *Intelligent Control, 2002. Proceedings of the 2002 IEEE International Symposium on*, pages 789 – 794, 2002.

[190] V. Ziparo, L. Iocchi, Pedro Lima, D. Nardi, and P. Palamara. Petri Net Plans. *Autonomous Agents and Multi-Agent Systems*, 23:344–383, 2011. 10.1007/s10458-010-9146-1.

Appendix A

Proof of Lemma 6

If a network-level controller is in a well formed state $(NLC)_k = (P_k, T_k, F_k, W_k, K_k, M_k, def_k, TD_k, TS_k)$, then every state immediately reachable, $(NLC)_{k+1} = (P_{k+1}, T_{k+1}, F_{k+1}, W_{k+1}, K_{k+1}, M_{k+1}, def_{k+1}, TD_{k+1}, TS_{k+1})$, is also well formed.

Proof: There are several ways that $(P_k, T_k, F_k, W_k, K_k, M_k, def_k, TD_k, TS_k)$ can transition to become $(P_{k+1}, T_{k+1}, F_{k+1}, W_{k+1}, K_{k+1}, M_{k+1}, def_{k+1}, TD_{k+1}, TS_{k+1})$, this proof will show that well formedness is preserved for all possible cases: a task being completed, a transition firing, or any runtime patch being applied. In all there are 18 different cases.

According to definition 57, for a state $(NLC)_k$ to be well formed the following five criteria must be met:

1. $F_k \subseteq (P_k \times T_k) \cup (T_k \times P_k)$,
2. $W_k : F_k \rightarrow \mathcal{N}$,
3. K_k: $P_k \rightarrow \mathcal{N}_+$,
4. M_k: $P_k \rightarrow \mathcal{N}$,
5. def_k: $P_k \rightarrow TD_k \cup \{null\}$.

For each case below, it is assumed that $(NLC)_k$ meets these five criteria. Then, $(NLC)_{k+1}$ is shown to meet the same five criteria. This proves that the case preserves being well formed.

For several of the cases, some of the criteria will be trivially true because all of the portions of the state that they potentially depend upon where not modified and are identical at k and $k + 1$. These trivial criteria will be marked with a $\bigotimes$.

- $\underline{complete(ts^i)}$: When a UAV completes a task $ts^i \in TS_k$, the task states TS_k changes. All other parts of the NLC state remain the same: $P_{k+1} = P_k$, $T_{k+1} = T_k$, $F_{k+1} = F_k$, $W_{k+1} = W_k$, $K_{k+1} = K_k$, $M_{k+1} = M_k$, $def_{k+1} = def_k$, $TD_{k+1} = TD_k$.

  1. Show that $F_{k+1} \subseteq (P_{k+1} \times T_{k+1}) \cup (T_{k+1} \times P_{k+1})$. $\bigotimes$
  2. Show that $W_{k+1} : F_{k+1} \to \mathcal{N}$. $\bigotimes$
  3. Show that K_{k+1}: $P_{k+1} \to \mathcal{N}_+$. $\bigotimes$
  4. Show that M_{k+1}: $P_{k+1} \to \mathcal{N}$. $\bigotimes$
  5. Show that def_{k+1}: $P_{k+1} \to TD_{k+1} \cup \{null\}$. $\bigotimes$

  All five criteria are met and this case preserves being well formed.

- $\underline{firing(t)}$: When a network-level controller transition $t \in T_k$ is fired, the marking M_k and the task states TS_k change. All other parts of the NLC state remain the same: $P_{k+1} = P_k$, $T_{k+1} = T_k$, $F_{k+1} = F_k$, $W_{k+1} = W_k$, $K_{k+1} = K_k$, $def_{k+1} = def_k$, $TD_{k+1} = TD_k$.

  1. Show that $F_{k+1} \subseteq (P_{k+1} \times T_{k+1}) \cup (T_{k+1} \times P_{k+1})$. $\bigotimes$
  2. Show that $W_{k+1} : F_{k+1} \to \mathcal{N}$. $\bigotimes$
  3. Show that K_{k+1}: $P_{k+1} \to \mathcal{N}_+$. $\bigotimes$
  4. Show that M_{k+1}: $P_{k+1} \to \mathcal{N}$.
     By assumption, M_k: $P_k \to \mathcal{N}$, or every place at time k has a non-negative integer marking. The set of places does not change, but the marking's values do. The semantics of a NLC transition firing show that $M_{k+1}(p) = M_k(p) - W_k(p,t) + W_k(t,p)$. Since t can only fire if token enabled, definition 61 guarantees that $M_k(p) \geq W_k(p,t)$. This shows that $M_k(p) - W_k(p,t) \geq 0$. Since weights are all non-negative integers, $M_{k+1}(p) = M_k(p) - W_k(p,t) + W_k(t,p) \geq M_k(p) - W_k(p,t) \geq 0$. This shows that M_{k+1} is a non-negative integer for all places.
  5. Show that def_{k+1}: $P_{k+1} \to TD_{k+1} \cup \{null\}$. $\bigotimes$

  All five criteria are met and this case preserves being well formed.

- $\underline{addPlace()}$: When a null place is added, the structural operational semantics of table 5.3 show that P_k, K_k, M_k, and def_k change. All other parts of the NLC state remain the same: $T_{k+1} = T_k$, $F_{k+1} = F_k$, $W_{k+1} = W_k$, $TD_{k+1} = TD_k$, $TS_{k+1} = TS_k$.

1. Show that $F_{k+1} \subseteq (P_{k+1} \times T_{k+1}) \cup (T_{k+1} \times P_{k+1})$.
   By assumption, $F_k \subseteq (P_k \times T_k) \cup (T_k \times P_k)$, or every arc that exists at time k is connected to a place and transition that exist at time k. The addition of the new place p^m to P_k to form P_{k+1} does not automatically create any new arcs. The set of arcs and transitions did not change. Every arc in F_{k+1} was in F_k and all of the places and transitions that existed at k still exist at $k+1$. This guarantees that $F_{k+1} \subseteq (P_{k+1} \times T_{k+1}) \cup (T_{k+1} \times P_{k+1})$.
2. Show that $W_{k+1} : F_{k+1} \to \mathcal{N}$. $\bigotimes$
3. Show that K_{k+1}: $P_{k+1} \to \mathcal{N}_+$.
   By assumption, K_k: $P_k \to \mathcal{N}_+$, or every place that exists at time k has a positive integer capacity constraint. The capacity constraints for all places existing at time k are not changed. The new place p^m is given a default positive integer capacity constraint by definition. This guarantees that all places in P_{k+1} still have positive integer capacity constraints.
4. Show that M_{k+1}: $P_{k+1} \to \mathcal{N}$.
   By assumption, M_k: $P_k \to \mathcal{N}$, or every place that exists at time k has a non-negative integer marking. The marking for all places existing at time k is not changed. The new place p^m is given 0 tokens by definition. This guarantees that all places in P_{k+1} have a natural number marking.
5. Show that def_{k+1}: $P_{k+1} \to TD_{k+1} \cup \{null\}$.
   By assumption, def_k: $P_k \to TD_k \cup \{null\}$, or all places existing at time k have either an associated task definition or a value of *null* associated. The associated task definitions for all places existing at time k are not changed. The new place p^m is given an associated task definition *null*. This guarantees that all places in P_{k+1} have either an associated task definition or *null*.

All five criteria are met and this case preserves being well formed.

- $\underline{addPlace(td^m)}$: When a place with an associated task definition is added, the structural operational semantics of table 5.3 show that P_k, K_k, M_k, def_k, and TD_k change. All other parts of the NLC state remain the same: $T_{k+1} = T_k$, $F_{k+1} = F_k$, $W_{k+1} = W_k$, $TS_{k+1} = TS_k$.

1. Show that $F_{k+1} \subseteq (P_{k+1} \times T_{k+1}) \cup (T_{k+1} \times P_{k+1})$.
   By assumption, $F_k \subseteq (P_k \times T_k) \cup (T_k \times P_k)$, or every arc that exists at time k is connected to a place and transition that exist at time k. The addition of the new place p^m to P_k to form P_{k+1} does not automatically create any new arcs. The set of arcs and transitions did not change. Every arc in F_{k+1} was in F_k and all of the places and transitions that existed at k still exist at $k+1$. This guarantees that $F_{k+1} \subseteq (P_{k+1} \times T_{k+1}) \cup (T_{k+1} \times P_{k+1})$.

2. Show that $W_{k+1} : F_{k+1} \rightarrow \mathcal{N}$. $\bigotimes$

3. Show that K_{k+1}: $P_{k+1} \rightarrow \mathcal{N}_+$.
   By assumption, K_k: $P_k \rightarrow \mathcal{N}_+$, or every place that exists at time k has a positive integer capacity constraint. The capacity constraints for all places existing at time k are not changed. The new place p^m is given a default positive integer capacity constraint by definition. This guarantees that all places in P_{k+1} still have positive integer capacity constraints.

4. Show that M_{k+1}: $P_{k+1} \rightarrow \mathcal{N}$.
   By assumption, M_k: $P_k \rightarrow \mathcal{N}$, or every place that exists at time k has a non-negative integer marking. The marking for all places existing at time k is not changed. The new place p^m is given 0 tokens by definition. This guarantees that all places in P_{k+1} have a natural number marking.

5. Show that def_{k+1}: $P_{k+1} \rightarrow TD_{k+1} \cup \{null\}$.
   By assumption, def_k: $P_k \rightarrow TD_k \cup \{null\}$, or all places existing at time k have either an associated task definition or a value of *null* associated. The associated task definitions for all places existing at time k are not changed. The new place p^m is given an associated task definition td^m. This guarantees that all places in P_{k+1} have either an associated task definition or *null*.

All five criteria are met and this case preserves being well formed.

- $\underline{deletePlace(p^m)}$: When a place is deleted, P_k, F_k, TD_k, and TS_k change. All other parts of the NLC state remain the same: $T_{k+1} = T_k$, $W_{k+1} = W_k$, $K_{k+1} = K_k$, $M_{k+1} = M_k$, $def_{k+1} = def_k$.

  1. Show that $F_{k+1} \subseteq (P_{k+1} \times T_{k+1}) \cup (T_{k+1} \times P_{k+1})$.
     By assumption, $F_k \subseteq (P_k \times T_k) \cup (T_k \times P_k)$, or all of the existing arcs at time k go from places to transitions or transitions to places. No new arcs are added. The set of transitions does not change. The set of places only shrinks by the removal of p^m, $P_{k+1} = P_k - p^m$. By definition, every arc in $(p^m \times T_k) \cup (T_k \times p^m)$ is removed from F_k to form F_{k+1}. So every arc left in F_{k+1} does not go to or come from p^m. So for all remaining arcs, $F_{k+1} \subseteq (P_{k+1} \times T_{k+1}) \cup (T_{k+1} \times P_{k+1})$.

  2. Show that $W_{k+1} : F_{k+1} \rightarrow \mathcal{N}$.
     By assumption, $W_k : F_k \rightarrow \mathcal{N}$, or every arc has a non-negative integer weight. The weight values do not change, $W_{k+1} = W_k$. All arcs connected to p^m are removed, shrinking the domain of W_{k+1}, but not changing any values. All arcs that still exist have the same weights as before, showing $W_{k+1} : F_{k+1} \rightarrow \mathcal{N}$.

3. Show that K_{k+1}: $P_{k+1} \to \mathcal{N}_+$.
   By assumption, K_k: $P_k \to \mathcal{N}_+$, or every place existing at time k has a positive integer capacity constraint. The capacity constraints do not change value, $K_{k+1} = K_k$. Only the place p^m is removed, shrinking the domain of K_{k+1}, but not changing any values. All places that still exist have the same capacity constraints as before, showing K_{k+1}: $P_{k+1} \to \mathcal{N}_+$.
4. Show that M_{k+1}: $P_{k+1} \to \mathcal{N}$.
   By assumption, M_k: $P_k \to \mathcal{N}$, or that every place at time k has a non-negative integer marking. The marking's values do not change, $M_{k+1} = M_k$, but its domain is shrunk by the removal of place p^m. All places that still exist have the same marking as before, showing M_{k+1}: $P_{k+1} \to \mathcal{N}$.
5. Show that def_{k+1}: $P_{k+1} \to TD_{k+1} \cup \{null\}$.
   By assumption, def_k: $P_k \to TD_k \cup \{null\}$, or that every place at time k is associated to either *null* or a task definition. The associated task definitions function has its domain shrunk by the removal of place p^m. All other places and task definitions still exist and have not changed value, showing def_{k+1}: $P_{k+1} \to TD_{k+1} \cup \{null\}$.

All five criteria are met and this case preserves being well formed.

- $\underline{addTransition()}$: When a transition is added, T_k changes. All other parts of the NLC state remain the same: $P_{k+1} = P_k$, $F_{k+1} = F_k$, $W_{k+1} = W_k$, $K_{k+1} = K_k$, $M_{k+1} = M_k$, $def_{k+1} = def_k$, $TD_{k+1} = TD_k$, $TS_{k+1} = TS_k$.

1. Show that $F_{k+1} \subseteq (P_{k+1} \times T_{k+1}) \cup (T_{k+1} \times P_{k+1})$.
   By assumption, $F_k \subseteq (P_k \times T_k) \cup (T_k \times P_k)$, or that all arcs existing at time k go from a place to a transition or a transition to a place at time k. The sets of arcs and places do not change. The set of transitions is augmented with a single new transition t^m, $(T_{k+1} = T_k \cup t^m)$. Every arc that had existed, still exists. No new arcs were added. Every place and transition that the existing arcs connected also still exist. Consequently, $F_{k+1} \subseteq (P_{k+1} \times T_{k+1}) \cup (T_{k+1} \times P_{k+1})$.
2. Show that $W_{k+1} : F_{k+1} \to \mathcal{N}$. $\otimes$
3. Show that K_{k+1}: $P_{k+1} \to \mathcal{N}_+$. $\otimes$
4. Show that M_{k+1}: $P_{k+1} \to \mathcal{N}$. $\otimes$
5. Show that def_{k+1}: $P_{k+1} \to TD_{k+1} \cup \{null\}$. $\otimes$

All five criteria are met and this case preserves being well formed.

- *deleteTransition*(t^m): When a transition $t^m \in T_k$ is deleted, T_k and F_k change. All other parts of the NLC state remain the same: $P_{k+1} = P_k$, $W_{k+1} = W_k$, $K_{k+1} = K_k$, $M_{k+1} = M_k$, $def_{k+1} = def_k$, $TD_{k+1} = TD_k$, $TS_{k+1} = TS_k$.

  1. Show that $F_{k+1} \subseteq (P_{k+1} \times T_{k+1}) \cup (T_{k+1} \times P_{k+1})$.
     By assumption, $F_k \subseteq (P_k \times T_k) \cup (T_k \times P_k)$, or that all arcs existing at time k go from a place to a transition or a transition to a place at time k. The set of places does not change. No new arcs are added. A single transition t^m is removed. By definition, all arcs connected to t^m are removed. All of the remaining arcs connect places in $P_{k+1} = P_k$ to transitions in $T_{k+1} = T_k - \{t^m\}$.
  2. Show that $W_{k+1} : F_{k+1} \rightarrow \mathcal{N}$.
     By assumption, $W_k : F_k \rightarrow \mathcal{N}$, or that all arcs existing at time k have a non-negative integer weight. The weights do not change. All arcs connected to t^m are removed, shrinking the domain of W_{k+1}, but not changing any values. All arcs that still exist have the same weights as before, showing $W_{k+1} : F_{k+1} \rightarrow \mathcal{N}$.
  3. Show that K_{k+1}: $P_{k+1} \rightarrow \mathcal{N}_+$. $\bigotimes$
  4. Show that M_{k+1}: $P_{k+1} \rightarrow \mathcal{N}$. $\bigotimes$
  5. Show that def_{k+1}: $P_{k+1} \rightarrow TD_{k+1} \cup \{null\}$. $\bigotimes$

  All five criteria are met and this case preserves being well formed.

- *addArc*(p^m, t^n): When an arc between $p^m \in P_k$ and $t^n \in T_k$ is added, F_k and W_k change. All other parts of the NLC state remain the same: $P_{k+1} = P_k$, $T_{k+1} = T_k$, $K_{k+1} = K_k$, $M_{k+1} = M_k$, $def_{k+1} = def_k$, $TD_{k+1} = TD_k$, $TS_{k+1} = TS_k$.

  1. Show that $F_{k+1} \subseteq (P_{k+1} \times T_{k+1}) \cup (T_{k+1} \times P_{k+1})$. By assumption, $F_k \subseteq (P_k \times T_k) \cup (T_k \times P_k)$, or that all arcs existing at time k go from a place to a transition or a transition to a place at time k. The sets of places and transitions do not change. Only one new arc is added. This arc is added between a place and a transition that existed at k and still exist at $k+1$. All of the arcs in F_{k+1} connect places in $P_{k+1} = P_k$ to transitions in $T_{k+1} = T_k$.
  2. Show that $W_{k+1} : F_{k+1} \rightarrow \mathcal{N}$. By assumption, $W_k : F_k \rightarrow \mathcal{N}$, or that all arcs existing at time k have a non-negative integer weight. The weights on these previously existing arcs do not change. The single new arc is given a default non-negative integer weight value by definition. All arcs, those existing at k and the one new at $k+1$, have non-negative integer weights, showing $W_{k+1} : F_{k+1} \rightarrow \mathcal{N}$.

3. Show that K_{k+1}: $P_{k+1} \rightarrow \mathcal{N}_+$. $\bigotimes$
4. Show that M_{k+1}: $P_{k+1} \rightarrow \mathcal{N}$. $\bigotimes$
5. Show that def_{k+1}: $P_{k+1} \rightarrow TD_{k+1} \cup \{null\}$. $\bigotimes$

All five criteria are met and this case preserves being well formed.

- $\underline{addArc(t^m, p^n)}$: When an arc between $t^m \in T_k$ and $p^n \in P_k$ is added, F_k and W_k change. All other parts of the NLC state remain the same: $P_{k+1} = P_k$, $T_{k+1} = T_k$, $K_{k+1} = K_k$, $M_{k+1} = M_k$, $def_{k+1} = def_k$, $TD_{k+1} = TD_k$, $TS_{k+1} = TS_k$.

    1. Show that $F_{k+1} \subseteq (P_{k+1} \times T_{k+1}) \cup (T_{k+1} \times P_{k+1})$. By assumption, $F_k \subseteq (P_k \times T_k) \cup (T_k \times P_k)$, or that all arcs existing at time k go from a place to a transition or a transition to a place at time k. The sets of places and transitions do not change. Only one new arc is added. This arc is added between a transition and a place that existed at k and still exist at $k+1$. All of the arcs in F_{k+1} connect places in $P_{k+1} = P_k$ to transitions in $T_{k+1} = T_k$.
    2. Show that $W_{k+1} : F_{k+1} \rightarrow \mathcal{N}$. By assumption, $W_k : F_k \rightarrow \mathcal{N}$, or that all arcs existing at time k have a non-negative integer weight. The weights on these previously existing arcs do not change. The single new arc is given a default non-negative integer weight value by definition. All arcs, those existing at k and the one new at $k+1$, have non-negative integer weights, showing $W_{k+1} : F_{k+1} \rightarrow \mathcal{N}$.
    3. Show that K_{k+1}: $P_{k+1} \rightarrow \mathcal{N}_+$. $\bigotimes$
    4. Show that M_{k+1}: $P_{k+1} \rightarrow \mathcal{N}$. $\bigotimes$
    5. Show that def_{k+1}: $P_{k+1} \rightarrow TD_{k+1} \cup \{null\}$. $\bigotimes$

    All five criteria are met and this case preserves being well formed.

- $\underline{deleteArc(p^m, t^n)}$: When an arc $(p^m, t^n) \in F_k$ is removed, F_k changes. All other parts of the NLC state remain the same: $P_{k+1} = P_k$, $T_{k+1} = T_k$, $W_{k+1} = W_k$, $K_{k+1} = K_k$, $M_{k+1} = M_k$, $def_{k+1} = def_k$, $TD_{k+1} = TD_k$, $TS_{k+1} = TS_k$.

    1. Show that $F_{k+1} \subseteq (P_{k+1} \times T_{k+1}) \cup (T_{k+1} \times P_{k+1})$. By assumption, $F_k \subseteq (P_k \times T_k) \cup (T_k \times P_k)$, or that all arcs existing at time k go from a place to a transition or a transition to a place at time k. The sets of places and transitions do not change. Only one existing arc is removed, (p^m, t^n). All of the remaining arcs in F_{k+1} still connect places in $P_{k+1} = P_k$ to transitions in $T_{k+1} = T_k$.

2. Show that $W_{k+1} : F_{k+1} \to \mathcal{N}$. By assumption, $W_k : F_k \to \mathcal{N}$, or that all arcs existing at time k have a non-negative integer weight. The weights do not change. Only the arc (p^m, t^n) is removed, shrinking the domain of W_{k+1}, but not changing any values. All arcs that still exist have the same weights as before, showing $W_{k+1} : F_{k+1} \to \mathcal{N}$.
3. Show that K_{k+1}: $P_{k+1} \to \mathcal{N}_+$. $\otimes$
4. Show that M_{k+1}: $P_{k+1} \to \mathcal{N}$. $\otimes$
5. Show that def_{k+1}: $P_{k+1} \to TD_{k+1} \cup \{null\}$. $\otimes$

All five criteria are met and this case preserves being well formed.

- <u>$deleteArc(t^m, p^n)$</u>: When an arc $(t^m, p^n) \in F_k$ is removed, F_k changes. All other parts of the NLC state remain the same: $P_{k+1} = P_k$, $T_{k+1} = T_k$, $W_{k+1} = W_k$, $K_{k+1} = K_k$, $M_{k+1} = M_k$, $def_{k+1} = def_k$, $TD_{k+1} = TD_k$, $TS_{k+1} = TS_k$.

1. Show that $F_{k+1} \subseteq (P_{k+1} \times T_{k+1}) \cup (T_{k+1} \times P_{k+1})$. By assumption, $F_k \subseteq (P_k \times T_k) \cup (T_k \times P_k)$, or that all arcs existing at time k go from a place to a transition or a transition to a place at time k. The sets of places and transitions do not change. Only one existing arc is removed, (t^m, p^n). All of the remaining arcs in F_{k+1} still connect places in $P_{k+1} = P_k$ to transitions in $T_{k+1} = T_k$.
2. Show that $W_{k+1} : F_{k+1} \to \mathcal{N}$. By assumption, $W_k : F_k \to \mathcal{N}$, or that all arcs existing at time k have a non-negative integer weight. The weights do not change. Only the arc (t^m, p^n) is removed, shrinking the domain of W_{k+1}, but not changing any values. All arcs that still exist have the same weights as before, showing $W_{k+1} : F_{k+1} \to \mathcal{N}$.
3. Show that K_{k+1}: $P_{k+1} \to \mathcal{N}_+$. $\otimes$
4. Show that M_{k+1}: $P_{k+1} \to \mathcal{N}$. $\otimes$
5. Show that def_{k+1}: $P_{k+1} \to TD_{k+1} \cup \{null\}$. $\otimes$

All five criteria are met and this case preserves being well formed.

- <u>$modifyWeight(p^m, t^n, q)$</u>: When the weight of the arc $(p^m, t^n) \in F_k$ is modified to q, W_k changes. All other parts of the NLC state remain the same: $P_{k+1} = P_k$, $T_{k+1} = T_k$, $F_{k+1} = F_k$, $K_{k+1} = K_k$, $M_{k+1} = M_k$, $def_{k+1} = def_k$, $TD_{k+1} = TD_k$, $TS_{k+1} = TS_k$.

1. Show that $F_{k+1} \subseteq (P_{k+1} \times T_{k+1}) \cup (T_{k+1} \times P_{k+1})$. $\otimes$

2. Show that $W_{k+1} : F_{k+1} \to \mathcal{N}$. By assumption, $W_k : F_k \to \mathcal{N}$, or that all arcs existing at time k have a non-negative integer weight at k. Only the weight for arc (p^m, t^n) is changed. It is changed to the value q which is also a non-negative integer value by definition. All arcs still have non-negative integer weights, showing $W_{k+1} : F_{k+1} \to \mathcal{N}$.
3. Show that K_{k+1}: $P_{k+1} \to \mathcal{N}_+$. $\bigotimes$
4. Show that M_{k+1}: $P_{k+1} \to \mathcal{N}$. $\bigotimes$
5. Show that def_{k+1}: $P_{k+1} \to TD_{k+1} \cup \{null\}$. $\bigotimes$

All five criteria are met and this case preserves being well formed.

- $\underline{modifyWeight(t^m, p^n, q)}$: When the weight of the arc $(t^m, p^n) \in F_k$ is modified to q, W_k changes. All other parts of the NLC state remain the same: $P_{k+1} = P_k$, $T_{k+1} = T_k$, $F_{k+1} = F_k$, $K_{k+1} = K_k$, $M_{k+1} = M_k$, $def_{k+1} = def_k$, $TD_{k+1} = TD_k$, $TS_{k+1} = TS_k$.

  1. Show that $F_{k+1} \subseteq (P_{k+1} \times T_{k+1}) \cup (T_{k+1} \times P_{k+1})$. $\bigotimes$
  2. Show that $W_{k+1} : F_{k+1} \to \mathcal{N}$. By assumption, $W_k : F_k \to \mathcal{N}$, or that all arcs existing at time k have a non-negative integer weight at k. Only the weight for arc (t^m, p^n) is changed. It is changed to the value q which is also a non-negative integer value by definition. All arcs still have non-negative integer weights, showing $W_{k+1} : F_{k+1} \to \mathcal{N}$.
  3. Show that K_{k+1}: $P_{k+1} \to \mathcal{N}_+$. $\bigotimes$
  4. Show that M_{k+1}: $P_{k+1} \to \mathcal{N}$. $\bigotimes$
  5. Show that def_{k+1}: $P_{k+1} \to TD_{k+1} \cup \{null\}$. $\bigotimes$

  All five criteria are met and this case preserves being well formed.

- $\underline{modifyCapacity(p^m, q)}$: When the capacity constraint on $p^m \in P_k$ is modified to q, K_k changes. All other parts of the NLC state remain the same: $P_{k+1} = P_k$, $T_{k+1} = T_k$, $F_{k+1} = F_k$, $W_{k+1} = W_k$, $M_{k+1} = M_k$, $def_{k+1} = def_k$, $TD_{k+1} = TD_k$, $TS_{k+1} = TS_k$.

  1. Show that $F_{k+1} \subseteq (P_{k+1} \times T_{k+1}) \cup (T_{k+1} \times P_{k+1})$. $\bigotimes$
  2. Show that $W_{k+1} : F_{k+1} \to \mathcal{N}$. $\bigotimes$
  3. Show that K_{k+1}: $P_{k+1} \to \mathcal{N}_+$. By assumption, $K_k : P_k \to \mathcal{N}_+$, or that all places existing at time k have a positive integer capacity constraint at k. Only the capacity constraint for p^m is changed. It is changed to the value q which is also a positive integer value by definition. All places still have positive integer capacity constraints, showing $K_{k+1} : P_{k+1} \to \mathcal{N}_+$.

4. Show that M_{k+1}: $P_{k+1} \rightarrow \mathcal{N}$. $\bigotimes$
5. Show that def_{k+1}: $P_{k+1} \rightarrow TD_{k+1} \cup \{null\}$. $\bigotimes$

All five criteria are met and this case preserves being well formed.

- $\underline{addToken(p^m)}$: When a token is added to $p^m \in P_k$, M_k and TS_k change. All other parts of the NLC state remain the same: $P_{k+1} = P_k$, $T_{k+1} = T_k$, $F_{k+1} = F_k$, $W_{k+1} = W_k$, $K_{k+1} = K_k$, $def_{k+1} = def_k$, $TD_{k+1} = TD_k$.

  1. Show that $F_{k+1} \subseteq (P_{k+1} \times T_{k+1}) \cup (T_{k+1} \times P_{k+1})$. $\bigotimes$
  2. Show that $W_{k+1} : F_{k+1} \rightarrow \mathcal{N}$. $\bigotimes$
  3. Show that K_{k+1}: $P_{k+1} \rightarrow \mathcal{N}_+$. $\bigotimes$
  4. Show that M_{k+1}: $P_{k+1} \rightarrow \mathcal{N}$. By assumption, M_k: $P_k \rightarrow \mathcal{N}$, or every place at time k has a non-negative integer marking. The set of places does not change. Adding a token to p^m increases the non-negative integer marking of p^m by 1, resulting in a guaranteed positive integer marking. All other places' markings are not affected. Every place in P_{k+1} has a non-negative integer marking.
  5. Show that def_{k+1}: $P_{k+1} \rightarrow TD_{k+1} \cup \{null\}$. $\bigotimes$

  All five criteria are met and this case preserves being well formed.

- $\underline{deleteToken(p^m)}$: When a token is deleted from $p^m \in P_k$, M_k and TS_k change. All other parts of the NLC state remain the same: $P_{k+1} = P_k$, $T_{k+1} = T_k$, $F_{k+1} = F_k$, $W_{k+1} = W_k$, $K_{k+1} = K_k$, $def_{k+1} = def_k$, $TD_{k+1} = TD_k$.

  1. Show that $F_{k+1} \subseteq (P_{k+1} \times T_{k+1}) \cup (T_{k+1} \times P_{k+1})$. $\bigotimes$
  2. Show that $W_{k+1} : F_{k+1} \rightarrow \mathcal{N}$. $\bigotimes$
  3. Show that K_{k+1}: $P_{k+1} \rightarrow \mathcal{N}_+$. $\bigotimes$
  4. Show that M_{k+1}: $P_{k+1} \rightarrow \mathcal{N}$. By assumption, M_k: $P_k \rightarrow \mathcal{N}$, or every place at time k has a non-negative integer marking. All places that are not p^m retain the same marking. By rule 'deleteToken' in table 5.5, changes are only made if there is at least 1 token in $M_k(p^m)$. If this is the case, there is a single token that can be removed from p^m resulting in a new marking that is also at least 0. Every place in P_{k+1} has a non-negative integer marking.
  5. Show that def_{k+1}: $P_{k+1} \rightarrow TD_{k+1} \cup \{null\}$. $\bigotimes$

  All five criteria are met and this case preserves being well formed.

- $\underline{modifyTaskDefinition(td^p, td^q)}$: When task definition td^p at time k has its content replaced with td^q for time $k+1$, only details of TD_k change. All other parts of the NLC state remain the same: $P_{k+1} = P_k$, $T_{k+1} = T_k$, $F_{k+1} = F_k$, $W_{k+1} = W_k$, $K_{k+1} = K_k$, $M_{k+1} = M_k$, $def_{k+1} = def_k$, $TS_{k+1} = TS_k$.

  1. Show that $F_{k+1} \subseteq (P_{k+1} \times T_{k+1}) \cup (T_{k+1} \times P_{k+1})$. $\bigotimes$
  2. Show that $W_{k+1} : F_{k+1} \to \mathcal{N}$. $\bigotimes$
  3. Show that K_{k+1}: $P_{k+1} \to \mathcal{N}_+$. $\bigotimes$
  4. Show that M_{k+1}: $P_{k+1} \to \mathcal{N}$. $\bigotimes$
  5. Show that def_{k+1}: $P_{k+1} \to TD_{k+1} \cup \{null\}$. The set of task definitions TD_k does not change. The contents of the specific task definition td^p at time k are updated so that $details(td^p_{k+1}) = details(td^q)$. The places and associated task definitions also do not change.

  All five criteria are met and this case preserves being well formed.

- $\underline{modifyTaskState(ts^p, ts^q)}$: When task state ts^p at time k has its content replaced with ts^q for time $k+1$, only details of TS_k change. All other parts of the NLC state remain the same: $P_{k+1} = P_k$, $T_{k+1} = T_k$, $F_{k+1} = F_k$, $W_{k+1} = W_k$, $K_{k+1} = K_k$, $M_{k+1} = M_k$, $def_{k+1} = def_k$, $TD_{k+1} = TD_k$.

  1. Show that $F_{k+1} \subseteq (P_{k+1} \times T_{k+1}) \cup (T_{k+1} \times P_{k+1})$. $\bigotimes$
  2. Show that $W_{k+1} : F_{k+1} \to \mathcal{N}$. $\bigotimes$
  3. Show that K_{k+1}: $P_{k+1} \to \mathcal{N}_+$. $\bigotimes$
  4. Show that M_{k+1}: $P_{k+1} \to \mathcal{N}$. $\bigotimes$
  5. Show that def_{k+1}: $P_{k+1} \to TD_{k+1} \cup \{null\}$. $\bigotimes$

  All five criteria are met and this case preserves being well formed.

□

Appendix B

Proof of Lemma 7

If a network-level controller is in a fully representative state $(NLC)_k = (P_k, T_k, F_k, W_k, K_k, M_k, def_k, TD_k, TS_k)$, then every state immediately reachable, $(NLC)_{k+1} = (P_{k+1}, T_{k+1}, F_{k+1}, W_{k+1}, K_{k+1}, M_{k+1}, def_{k+1}, TD_{k+1}, TS_{k+1})$, is also fully representative.

Proof: There are several ways that $(P_k, T_k, F_k, W_k, K_k, M_k, def_k, TD_k, TS_k)$ can transition to become $(P_{k+1}, T_{k+1}, F_{k+1}, W_{k+1}, K_{k+1}, M_{k+1}, def_{k+1}, TD_{k+1}, TS_{k+1})$, this proof will show that being fully representative is preserved for all possible cases: a task being completed, a transition firing, or any runtime patch being applied. In all there are 18 different cases.

According to definition 58, for a state $(NLC)_k$ to be fully representative:

$$\forall td \in TD_k.(\exists! p \in P_k.def_k(p) = td).$$

For each case below, it is assumed that $(NLC)_k$ is fully representative. Then, $(NLC)_{k+1}$ is shown to also be fully representative. This proves that the case preserves being fully representative.

In several of the cases below the places, associated task definitions, and set of task definitions will not change from k to $k+1$. Since $(NLC)_k$ is fully representative by assumption, and since the parts of the state that being fully representative depend upon are not modified, $(NLC)_{k+1}$ is also guaranteed to be fully representative. These trivial cases will be marked with a $\bigotimes$.

- $\underline{complete(ts^i)}$: When a UAV completes a task $ts^i \in TS_k$, the task states TS_k changes. All other parts of the NLC state remain the same: $P_{k+1} = P_k$, $T_{k+1} = T_k$, $F_{k+1} = F_k$, $W_{k+1} = W_k$, $K_{k+1} = K_k$, $M_{k+1} = M_k$, $def_{k+1} = def_k$, $TD_{k+1} = TD_k$.

  $\bigotimes$

- $\underline{firing(t)}$: When a network-level controller transition $t \in T_k$ is fired, the marking M_k and the task states TS_k change. All other parts of the NLC state remain the same: $P_{k+1} = P_k$, $T_{k+1} = T_k$, $F_{k+1} = F_k$, $W_{k+1} = W_k$, $K_{k+1} = K_k$, $def_{k+1} = def_k$, $TD_{k+1} = TD_k$.

  $\otimes$

- $\underline{addPlace()}$: When a null place is added, the structural operational semantics of table 5.3 show that P_k, K_k, M_k, and def_k change. All other parts of the NLC state remain the same: $T_{k+1} = T_k$, $F_{k+1} = F_k$, $W_{k+1} = W_k$, $TD_{k+1} = TD_k$, $TS_{k+1} = TS_k$.

  By assumption every task definition $td \in TD_k$ is connected to a unique place $p \in P_k$ by def_k. Adding a null place does not create any new task definitions. The function def_k only has its domain expanded by the new place p^m which points to null, $def_{k+1}(p^m) = null$. Every task definition that exists at $k+1$ existed at k and is still connected to the same unique place p by def_{k+1}, which has the same values at $k+1$ as k except for place p^m.

- $\underline{addPlace(td^m)}$: When a place with an associated task definition is added, the structural operational semantics of table 5.3 show that P_k, K_k, M_k, def_k, and TD_k change. All other parts of the NLC state remain the same: $T_{k+1} = T_k$, $F_{k+1} = F_k$, $W_{k+1} = W_k$, $TS_{k+1} = TS_k$.

  By assumption every task definition $td \in TD_k$ is connected to a unique place $p \in P_k$ by def_k. The set of task definitions expanded by the one new element td^m, $TD_{k+1} = TD_k \cup \{td^m\}$. The set of places expanded by the one new element p^m, $P_{k+1} = P_k \cup \{p^m\}$. By definition, $def_{k+1}(p^m) = td^m$ and all other def_{k+1} values stay the same as def_k. All previously existing task definitions stay connected to the same unique previously existing places by def_{k+1}. The single new task definition td^m is connected to a new place p^m by def_{k+1}. Since p^m is new it is guaranteed that this association is unique. All task definitions in TD_{k+1}, the previously existing and the new td^m, have a unique place in P_{k+1} that they are associated with through def_{k+1}.

- $\underline{deletePlace(p^m)}$: When a place is deleted, the structural operational semantics of table 5.3 show that P_k, F_k, TD_k, and TS_k change. All other parts of the NLC state remain the same: $T_{k+1} = T_k$, $W_{k+1} = W_k$, $K_{k+1} = K_k$, $M_{k+1} = M_k$, $def_{k+1} = def_k$.

  By assumption every task definition $td \in TD_k$ is connected to a unique place $p \in P_k$ by def_k. When a place p^m is deleted, if there is and associated task definition, $def_k(p^m) = td^m$, it is also removed. The function def_k has its domain shrunk by the removal of p^m to become def_{k+1}, but all other values remain the same. All of the remaining task definitions are still connected to the same unique places by def_{k+1}.

- $\underline{addTransition()}$: When a transition is added, T_k changes. All other parts of the NLC state remain the same: $P_{k+1} = P_k$, $F_{k+1} = F_k$, $W_{k+1} = W_k$, $K_{k+1} = K_k$, $M_{k+1} = M_k$, $def_{k+1} = def_k$, $TD_{k+1} = TD_k$, $TS_{k+1} = TS_k$.

  $\bigotimes$

- $\underline{deleteTransition(t^m)}$: When a transition $t^m \in T_k$ is deleted, T_k and F_k change. All other parts of the NLC state remain the same: $P_{k+1} = P_k$, $W_{k+1} = W_k$, $K_{k+1} = K_k$, $M_{k+1} = M_k$, $def_{k+1} = def_k$, $TD_{k+1} = TD_k$, $TS_{k+1} = TS_k$.

  $\bigotimes$

- $\underline{addArc(p^m, t^n)}$: When an arc between $p^m \in P_k$ and $t^n \in T_k$ is added, F_k and W_k change. All other parts of the NLC state remain the same: $P_{k+1} = P_k$, $T_{k+1} = T_k$, $K_{k+1} = K_k$, $M_{k+1} = M_k$, $def_{k+1} = def_k$, $TD_{k+1} = TD_k$, $TS_{k+1} = TS_k$.

  $\bigotimes$

- $\underline{addArc(t^m, p^n)}$: When an arc between $t^m \in T_k$ and $p^n \in P_k$ is added, F_k and W_k change. All other parts of the NLC state remain the same: $P_{k+1} = P_k$, $T_{k+1} = T_k$, $K_{k+1} = K_k$, $M_{k+1} = M_k$, $def_{k+1} = def_k$, $TD_{k+1} = TD_k$, $TS_{k+1} = TS_k$.

  $\bigotimes$

- $\underline{deleteArc(p^m, t^n)}$: When an arc $(p^m, t^n) \in F_k$ is removed, F_k changes. All other parts of the NLC state remain the same: $P_{k+1} = P_k$, $T_{k+1} = T_k$, $W_{k+1} = W_k$, $K_{k+1} = K_k$, $M_{k+1} = M_k$, $def_{k+1} = def_k$, $TD_{k+1} = TD_k$, $TS_{k+1} = TS_k$.

  $\bigotimes$

- $\underline{deleteArc(t^m, p^n)}$: When an arc $(t^m, p^n) \in F_k$ is removed, F_k changes. All other parts of the NLC state remain the same: $P_{k+1} = P_k$, $T_{k+1} = T_k$, $W_{k+1} = W_k$, $K_{k+1} = K_k$, $M_{k+1} = M_k$, $def_{k+1} = def_k$, $TD_{k+1} = TD_k$, $TS_{k+1} = TS_k$.

  $\bigotimes$

- $\underline{modifyWeight(p^m, t^n, q)}$: When the weight of the arc $(p^m, t^n) \in F_k$ is modified to q, W_k changes. All other parts of the NLC state remain the same: $P_{k+1} = P_k$, $T_{k+1} = T_k$, $F_{k+1} = F_k$, $K_{k+1} = K_k$, $M_{k+1} = M_k$, $def_{k+1} = def_k$, $TD_{k+1} = TD_k$, $TS_{k+1} = TS_k$.

  $\bigotimes$

- $\underline{modifyWeight(t^m, p^n, q)}$: When the weight of the arc $(t^m, p^n) \in F_k$ is modified to q, W_k changes. All other parts of the NLC state remain the same: $P_{k+1} = P_k$, $T_{k+1} = T_k$, $F_{k+1} = F_k$, $K_{k+1} = K_k$, $M_{k+1} = M_k$, $def_{k+1} = def_k$, $TD_{k+1} = TD_k$, $TS_{k+1} = TS_k$.

  $\otimes$

- $\underline{modifyCapacity(p^m, q)}$: When the capacity constraint on $p^m \in P_k$ is modified to q, K_k changes. All other parts of the NLC state remain the same: $P_{k+1} = P_k$, $T_{k+1} = T_k$, $F_{k+1} = F_k$, $W_{k+1} = W_k$, $M_{k+1} = M_k$, $def_{k+1} = def_k$, $TD_{k+1} = TD_k$, $TS_{k+1} = TS_k$.

  $\otimes$

- $\underline{addToken(p^m)}$: When a token is added to $p^m \in P_k$, M_k and TS_k change. All other parts of the NLC state remain the same: $P_{k+1} = P_k$, $T_{k+1} = T_k$, $F_{k+1} = F_k$, $W_{k+1} = W_k$, $K_{k+1} = K_k$, $def_{k+1} = def_k$, $TD_{k+1} = TD_k$.

  $\otimes$

- $\underline{deleteToken(p^m)}$: When a token is deleted from $p^m \in P_k$, M_k and TS_k change. All other parts of the NLC state remain the same: $P_{k+1} = P_k$, $T_{k+1} = T_k$, $F_{k+1} = F_k$, $W_{k+1} = W_k$, $K_{k+1} = K_k$, $def_{k+1} = def_k$, $TD_{k+1} = TD_k$.

  $\otimes$

- $\underline{modifyTaskDefinition(td^p, td^q)}$: When task definition td^p at time k has its content replaced with td^q for time $k+1$, only details of TD_k change. All other parts of the NLC state remain the same: $P_{k+1} = P_k$, $T_{k+1} = T_k$, $F_{k+1} = F_k$, $W_{k+1} = W_k$, $K_{k+1} = K_k$, $M_{k+1} = M_k$, $def_{k+1} = def_k$, $TS_{k+1} = TS_k$.

  Only the content of the task definition is changed. The set of task definitions, the set of places, and the function def_k are not modified.

- $\underline{modifyTaskInstance(ts^p, ts^q)}$: When task state ts^p at time k has its content replaced with ts^q for time $k+1$, only details of TS_k change. All other parts of the NLC state remain the same: $P_{k+1} = P_k$, $T_{k+1} = T_k$, $F_{k+1} = F_k$, $W_{k+1} = W_k$, $K_{k+1} = K_k$, $M_{k+1} = M_k$, $def_{k+1} = def_k$, $TD_{k+1} = TD_k$.

  $\otimes$

$\square$

Appendix C

Proof of Lemma 8

If a network-level controller is in a task-token consistent state $(NLC)_k = (P_k, T_k, F_k, W_k, K_k, M_k, def_k, TD_k, TS_k)$, then states immediately reachable, $(NLC)_{k+1} = (P_{k+1}, T_{k+1}, F_{k+1}, W_{k+1}, K_{k+1}, M_{k+1}, def_{k+1}, TD_{k+1}, TS_{k+1})$, are also task-token consistent.

Proof: There are several ways that $(P_k, T_k, F_k, W_k, K_k, M_k, def_k, TD_k, TS_k)$ can transition to become $(P_{k+1}, T_{k+1}, F_{k+1}, W_{k+1}, K_{k+1}, M_{k+1}, def_{k+1}, TD_{k+1}, TS_{k+1})$, this proof will show that being task-token consistent is preserved for all possible cases: a task being completed, a transition firing, or any runtime patch being applied. In all there are 18 different cases.

According to definition 59, for a state $(NLC)_k$ to be task-token consistent:

$$\forall p \in P_k.[def_k(p) \neq null] \Rightarrow [M_k(p) = cardinality(TS_k|_{def_k(p)})].$$

For each case below, it is assumed that $(NLC)_k$ is task-token consistent. Then, $(NLC)_{k+1}$ is shown to also be task-token consistent. This proves that the case preserves being task-token consistent.

In several of the cases below the places, markings, associated task definitions, task definitions, and task states will not change from k to $k+1$. Since $(NLC)_k$ is task-token consistent by assumption, and since the parts of the state that being task-token consistent depend upon are not modified, $(NLC)_{k+1}$ is also guaranteed to be task-token consistent. These trivial cases will be marked with a $\bigotimes$.

- <u>$complete(ts^i)$</u>: When a UAV completes a task $ts^i \in TS_k$, the task states TS_k changes. All other parts of the NLC state remain the same: $P_{k+1} = P_k$, $T_{k+1} = T_k$, $F_{k+1} = F_k$, $W_{k+1} = W_k$, $K_{k+1} = K_k$, $M_{k+1} = M_k$, $def_{k+1} = def_k$, $TD_{k+1} = TD_k$.

  Changing ts^i from not done to done affects neither the number of task states nor the marking, thus this event cannot affect task-token consistency.

- $\underline{firing(t^i)}$: When a network-level controller transition $t^i \in T_k$ is fired, the marking M_k and the task states TS_k change. All other parts of the NLC state remain the same: $P_{k+1} = P_k$, $T_{k+1} = T_k$, $F_{k+1} = F_k$, $W_{k+1} = W_k$, $K_{k+1} = K_k$, $def_{k+1} = def_k$, $TD_{k+1} = TD_k$.

  For this case to be true, all places p^j that do have an associated task definition ($def(p^j) \neq null$) must satisfy: $M_{k+1}(p^j) = cardinality(TS_{k+1}|_{def(p^j)})$.

  Before the firing, $M_k(p^j) = cardinality(TS_k|_{def_k(p^j)}) = x$ by the task-token consistency assumption at k.

  When t^i is fired, according to definition 64, $W_k(p^j, t^i) = m$ tokens will be removed from place p^j. Also $W_k(t^i, p^j) = n$ tokens will be added. This means $M_{k+1}(p^j) = x - m + n$.

  According to definition 62, there must be exactly $W_k(p^j, t^i) = m$ task states matching $def_k(p^j)$ in the set of task states to be removed, TS^-. Definition 63 shows that there are exactly $W_k(t^i, p^j) = n$ task states matching $def_k(p^j)$ to be added in set TS^+.

  The original number of task states matching $def_k(p^j)$ was x, then m were removed and n added. This produces $cardinality(TS_{k+1}|_{def_{k+1}(p^j)}) = x - m + n$.

  Therefore, $M_{k+1}(p^j) = x - m + n = cardinality(TS_{k+1}|_{def_{k+1}(p^j)})$ and the resulting state is task-token consistent.

- $\underline{addPlace()}$: When a null place is added, the structural operational semantics of table 5.3 show that P_k, K_k, M_k, and def_k change. All other parts of the NLC state remain the same: $T_{k+1} = T_k$, $F_{k+1} = F_k$, $W_{k+1} = W_k$, $TD_{k+1} = TD_k$, $TS_{k+1} = TS_k$.

  By assumption, at time k all places $p \in P_k$ are either associated to $null$ or have a marking that is equal to the number of task states based on the associated task definition.

  The set of places is expanded by the single new place p^m, $P_{k+1} = P_k + \{p^m\}$. By definition this place is given a default capacity constraint, zero tokens, and is associated to $null$. Since $def_{k+1}(p^m) = null$, this new place trivially satisfies the criteria.

  All other places $p \in P_{k+1}$ existed at k and retain the same capacity constraint, marking, and task definition association. Since these values satisfied the criteria at k by assumption, and since they are the same at $k+1$, they also satisfy the criteria at $k+1$.

- $\underline{addPlace(td^m)}$: When a place with an associated task definition is added, the structural operational semantics of table 5.3 show that P_k, K_k, M_k, def_k, and

TD_k change. All other parts of the NLC state remain the same: $T_{k+1} = T_k$, $F_{k+1} = F_k$, $W_{k+1} = W_k$, $TS_{k+1} = TS_k$.

By assumption, at time k all places $p \in P_k$ are either associated to *null* or have a marking that is equal to the number of task states based on the associated task definition.

The set of task definitions is expanded by td^m, $TD_{k+1} = TD_k + \{td^m\}$. The set of places is expanded by the single new place p^m, $P_{k+1} = P_k + \{p^m\}$. By definition this place is given a default capacity constraint, zero tokens, and is associated to td^m.

Since $def_{k+1}(p^m) = td^m$, and since no task states were created based on this new task definition, $cardinality(TS_{k+1}|_{def_{k+1}(p^m)}) = 0$. Since the marking for p^m is zero by definition, $M_{k+1}(p^m) = 0$. The marking and number of task states based on the associated definition match, both being 0, and the new place p^m satisfies the criteria.

All other places $p \in P_{k+1}$ existed at k and retain the same capacity constraint, marking, and task definition association. Since these values satisfied the criteria at k by assumption, and since they are the same at $k+1$, they also satisfy the criteria at $k+1$.

- $\underline{deletePlace(p^m)}$: When a place is deleted, P_k, F_k, TD_k, and TS_k change. All other parts of the NLC state remain the same: $T_{k+1} = T_k$, $W_{k+1} = W_k$, $K_{k+1} = K_k$, $M_{k+1} = M_k$, $def_{k+1} = def_k$.

  When p^m is removed, any connected arcs, the associated task definition $td^m = def_k(p^m)$, and any task states based upon it, $TS_k|_{def_k(p^m)}$, are also removed.

  All of the other places that existed at k still exist at $k+1$, and still have the same markings, the same task definition associations, and the same task states. All of these remaining places were unaffected and still satisfy the criteria.

- $\underline{addTransition()}$: When a transition is added, T_k changes. All other parts of the NLC state remain the same: $P_{k+1} = P_k$, $F_{k+1} = F_k$, $W_{k+1} = W_k$, $K_{k+1} = K_k$, $M_{k+1} = M_k$, $def_{k+1} = def_k$, $TD_{k+1} = TD_k$, $TS_{k+1} = TS_k$.

  $\otimes$

- $\underline{deleteTransition(t^m)}$: When a transition $t^m \in T_k$ is deleted, T_k and F_k change. All other parts of the NLC state remain the same: $P_{k+1} = P_k$, $W_{k+1} = W_k$, $K_{k+1} = K_k$, $M_{k+1} = M_k$, $def_{k+1} = def_k$, $TD_{k+1} = TD_k$, $TS_{k+1} = TS_k$.

  $\otimes$

- $\underline{addArc(p^m, t^n)}$: When an arc between $p^m \in P_k$ and $t^n \in T_k$ is added, F_k and W_k change. All other parts of the NLC state remain the same: $P_{k+1} = P_k$,

$T_{k+1} = T_k$, $K_{k+1} = K_k$, $M_{k+1} = M_k$, $def_{k+1} = def_k$, $TD_{k+1} = TD_k$, $TS_{k+1} = TS_k$.

$\otimes$

- $\underline{addArc(t^m, p^n)}$: When an arc between $t^m \in T_k$ and $p^n \in P_k$ is added, F_k and W_k change. All other parts of the NLC state remain the same: $P_{k+1} = P_k$, $T_{k+1} = T_k$, $K_{k+1} = K_k$, $M_{k+1} = M_k$, $def_{k+1} = def_k$, $TD_{k+1} = TD_k$, $TS_{k+1} = TS_k$.

  $\otimes$

- $\underline{deleteArc(p^m, t^n)}$: When an arc $(p^m, t^n) \in F_k$ is removed, F_k changes. All other parts of the NLC state remain the same: $P_{k+1} = P_k$, $T_{k+1} = T_k$, $W_{k+1} = W_k$, $K_{k+1} = K_k$, $M_{k+1} = M_k$, $def_{k+1} = def_k$, $TD_{k+1} = TD_k$, $TS_{k+1} = TS_k$.

  $\otimes$

- $\underline{deleteArc(t^m, p^n)}$: When an arc $(t^m, p^n) \in F_k$ is removed, F_k changes. All other parts of the NLC state remain the same: $P_{k+1} = P_k$, $T_{k+1} = T_k$, $W_{k+1} = W_k$, $K_{k+1} = K_k$, $M_{k+1} = M_k$, $def_{k+1} = def_k$, $TD_{k+1} = TD_k$, $TS_{k+1} = TS_k$.

  $\otimes$

- $\underline{modifyWeight(p^m, t^n, q)}$: When the weight of the arc $(p^m, t^n) \in F_k$ is modified to q, W_k changes. All other parts of the NLC state remain the same: $P_{k+1} = P_k$, $T_{k+1} = T_k$, $F_{k+1} = F_k$, $K_{k+1} = K_k$, $M_{k+1} = M_k$, $def_{k+1} = def_k$, $TD_{k+1} = TD_k$, $TS_{k+1} = TS_k$.

  $\otimes$

- $\underline{modifyWeight(t^m, p^n, q)}$: When the weight of the arc $(t^m, p^n) \in F_k$ is modified to q, W_k changes. All other parts of the NLC state remain the same: $P_{k+1} = P_k$, $T_{k+1} = T_k$, $F_{k+1} = F_k$, $K_{k+1} = K_k$, $M_{k+1} = M_k$, $def_{k+1} = def_k$, $TD_{k+1} = TD_k$, $TS_{k+1} = TS_k$.

  $\otimes$

- $\underline{modifyCapacity(p^m, q)}$: When the capacity constraint on $p^m \in P_k$ is modified to q, K_k changes. All other parts of the NLC state remain the same: $P_{k+1} = P_k$, $T_{k+1} = T_k$, $F_{k+1} = F_k$, $W_{k+1} = W_k$, $M_{k+1} = M_k$, $def_{k+1} = def_k$, $TD_{k+1} = TD_k$, $TS_{k+1} = TS_k$.

  $\otimes$

- $\underline{addToken(p^m)}$: When a token is added to $p^m \in P_k$, M_k and TS_k change. All other parts of the NLC state remain the same: $P_{k+1} = P_k$, $T_{k+1} = T_k$, $F_{k+1} = F_k$, $W_{k+1} = W_k$, $K_{k+1} = K_k$, $def_{k+1} = def_k$, $TD_{k+1} = TD_k$.

By assumption, at time k all places $p \in P_k$ are either associated to $null$ or have a marking that is equal to the number of task states based on the associated task definition.

If p^m is associated to $null$, $[def_{k+1}(p^m) \neq null] \Rightarrow$ $[M_{k+1}(p^m) = cardinality(TS_{k+1}|_{def_{k+1}(p^m)})]$ is trivially true and p^m's change in tokens has no affect on task-token consistency.

If p^m is associated to td^m by $def_k(p^m) = td^m$, and if a token is added, it increases the marking of p^m by 1, $M_{k+1}(p^m) = M_k(p^m) + 1$. By definition one new task state is also created. Since the marking and the number of task states both increase by exactly 1, if they were equal to x at time k, they are both equal to $x + 1$ at $k + 1$, and task-token consistency is also preserved.

- $\underline{deleteToken(p^m)}$: When a token is deleted from $p^m \in P_k$, M_k and TS_k change. All other parts of the NLC state remain the same: $P_{k+1} = P_k$, $T_{k+1} = T_k$, $F_{k+1} = F_k$, $W_{k+1} = W_k$, $K_{k+1} = K_k$, $def_{k+1} = def_k$, $TD_{k+1} = TD_k$.

  By assumption, at time k all places $p \in P_k$ are either associated to $null$ or have a marking that is equal to the number of task states based on the associated task definition.

  If the marking was zero, $M_k(p^m) = 0$, the deletion has no affect on the state and the state at $k + 1$ is identical to k, thus preserving task-token consistency.

  If there is at least 1 token and $def_k(p^m) = null$, the token can be removed with no affect on task-token consistency because $[def_{k+1}(p^m) \neq null] \Rightarrow$ $[M_{k+1}(p^m) = cardinality(TS_{k+1}|_{def_{k+1}(p^m)})]$ will be trivially true.

  If there is at least 1 token and $def_k(p^m) = td^m$, the token and a task state based on td^m will be removed by definition. The task state is guaranteed to exist because of the task-token consistency assumption at k. Since the marking and the number of task states both decrease by exactly 1, if they were equal to x at time k, they are both equal to $x - 1$ at $k + 1$, and task-token consistency is again preserved.

- $\underline{modifyTaskDefinition(td^p, td^q)}$: When task definition td^p at time k has its content replaced with td^q for time $k + 1$, only details of TD_k change. All other parts of the NLC state remain the same: $P_{k+1} = P_k$, $T_{k+1} = T_k$, $F_{k+1} = F_k$, $W_{k+1} = W_k$, $K_{k+1} = K_k$, $M_{k+1} = M_k$, $def_{k+1} = def_k$, $TS_{k+1} = TS_k$.

  $\otimes$

- $\underline{modifyTaskInstance(ts^p, ts^q)}$: When task state ts^p at time k has its content replaced with ts^q for time $k + 1$, only details of TS_k change. All other parts of the NLC state remain the same: $P_{k+1} = P_k$, $T_{k+1} = T_k$, $F_{k+1} = F_k$, $W_{k+1} = W_k$, $K_{k+1} = K_k$, $M_{k+1} = M_k$, $def_{k+1} = def_k$, $TD_{k+1} = TD_k$.

$\otimes$

$\square$